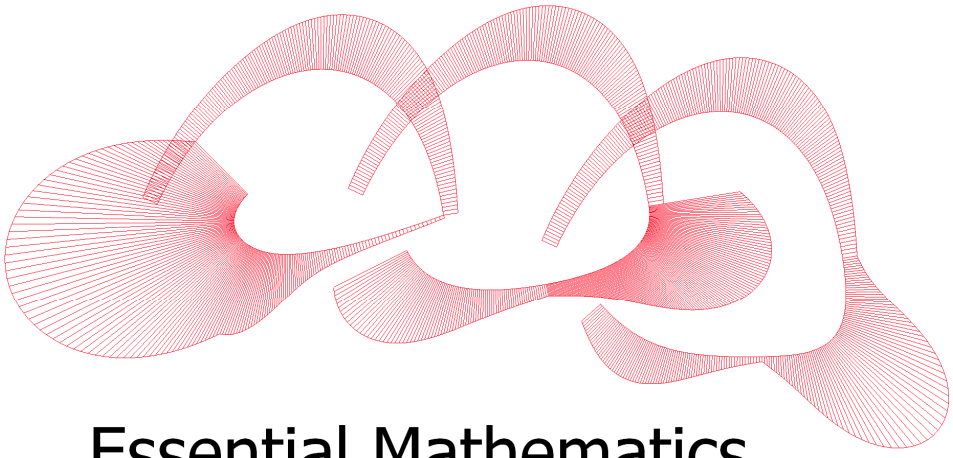


Essential Mathematics

for computational design - Third Edition

Rajaa Issa

Robert McNeel & Associates

Preface

Essential Mathematics for Computational Design introduces to design professionals the foundation mathematical concepts that are necessary for effective development of computational methods for 3-D modeling and computer graphics. This is not meant to be a complete and comprehensive resource, but rather an overview of the basic and most commonly used concepts.

The material is directed towards designers who have little or no background in mathematics beyond high school. All concepts are explained visually using Grasshopper® (GH), the generative modeling environment for Rhinoceros® (Rhino). For more information, go to www.rhino3d.com and www.grasshopper3d.com.

The content is divided into three chapters. Chapter 1 discusses vector math including vector representation, vector operation, and line and plane equations. Chapter 2 reviews matrix operations and transformations. Chapter 3 includes an in-depth review of parametric curves with special focus on NURBS curves and the concepts of continuity and curvature. It also reviews NURBS surfaces and polysurfaces.

I would like to acknowledge the excellent and thorough technical review by Dr. Dale Lear of Robert McNeel & Associates. His valuable comments were instrumental in producing this third edition. I would also like to acknowledge Ms. Margaret Becker of Robert McNeel & Associates for reviewing the technical writing and formatting the document.

Rajaa Issa
Robert McNeel & Associates

Table of Contents

1 Vector Mathematics

A *vector* indicates a quantity, such as velocity or force, that has *direction* and *length*. Vectors in 3-D coordinate systems are represented with an ordered set of three real numbers and look like:

v = <a1, a2, a3>

Vector representation

In this document, lower case bold letters will notate vectors. Vector components are also enclosed in angle brackets. Upper case letters will notate points. Point coordinates will always be enclosed by parentheses.

Using a coordinate system and any set of anchor points in that system, we can represent or visualize these vectors using a line-segment representation. An arrowhead shows the vector direction.

For example, if we have a vector that has a direction parallel to the x-axis of a given 3-D coordinate system and a length of 5 units, we can write the vector as follows:

v = <5, 0, 0>

To represent that vector, we need an anchor point in the coordinate system. For example, all of the arrows in the following figure are equal representations of the same vector despite the fact that they are anchored at different locations.

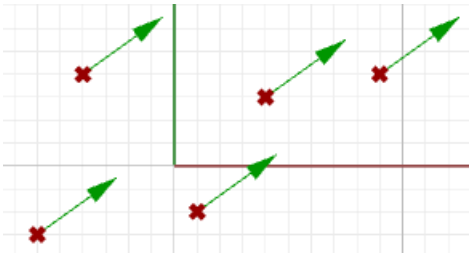


Figure (1): Vector representation in the 3-D coordinate system.

> Given a 3-D vector ***v*** = < a1, a2, a3 >, all vector components a1, a2, a3 are real numbers. Also all line segments from a point A(x,y,z) to point B(x+a1, y+a2, z+a3) are equivalent representations of vector ***v***.

So, how do we define the end points of a line segment that represents a given vector?

Let us define an anchor point (A) so that:

A = (1, 2, 3)

And a vector:

v = <5, 6, 7>

The tip point (B) of the vector is calculated by adding the corresponding components from anchor point and vector **v**:

B = A + **v**

B = (1+5, 2+6, 3+7)

B = (6, 8, 10)

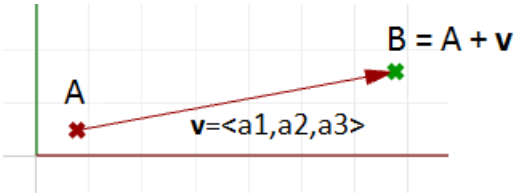


Figure (2): The relationship between a vector, the vector anchor point, and the point coinciding with the vector tip location.

Position vector

One special vector representation uses the origin point (0,0,0) as the vector anchor point. The position vector **v** = <a1,a2,a3> is represented with a line segment between two points, the origin and B, so that:

Origin point = (0,0,0)

B = (a1,a2,a3)

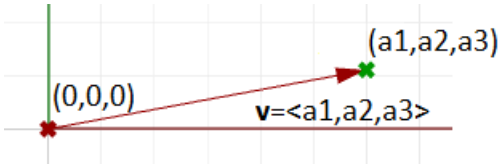


Figure (3): Position vector. The tip point coordinates equal the corresponding vector components.

> A position vector for a given vector ***v***= < a1,a2,a3 > is a special line segment representation from the origin point (0,0,0) to point (a1,a2,a3).

Vectors vs. points

Do not confuse vectors and points. They are very different concepts. Vectors, as we mentioned, represent a quantity that has direction and length, while points indicate a location. For example, the North direction is a vector, while the North Pole is a location (point).

If we have a vector and a point that have the same components, such as:

v = <3,1,0>

P = (3,1,0)

We can draw the vector and the point as follows:

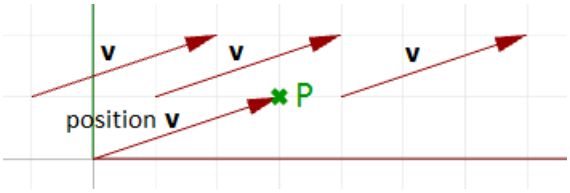


Figure (4): A vector defines a direction and length. A point defines a location.

Vector length

As mentioned before, vectors have length. We will use |**a**| to notate the length of a given vector **a**. For example:

a = <4, 3, 0>

$|\mathbf{a}| = \sqrt{(4^2 + 3^2 + 0^2)}$

|**a**| = 5

In general, the **length** of a vector **a**<a1,a2,a3> is calculated as follows:

$|\mathbf{a}| = \sqrt{(a1^2 + a2^2 + a3^2)}$

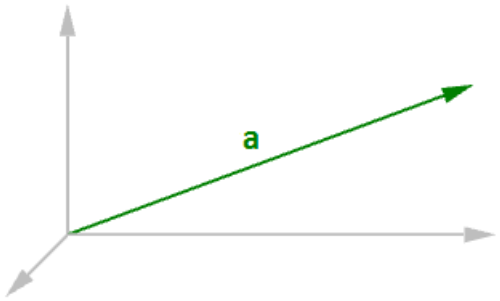


Figure (5): Vector length.

Unit vector

A unit vector is a vector with a length equal to one unit. Unit vectors are commonly used to compare the directions of vectors.

> A unit vector is a vector whose length is equal to one unit.

To calculate a unit vector, we need to find the length of the given vector, and then divide the vector components by the length. For example:

a = <4, 3, 0>

$|\mathbf{a}| = \sqrt{(4^2 + 3^2 + 0^2)}$

|**a**| = 5 unit length

If **b** = unit vector of **a**, then:

b = <4/5, 3/5, 0/5>

b = <0.8, 0.6, 0>

$|\mathbf{b}| = \sqrt{(0.8^2 + 0.6^2 + 0^2)}$

$|\mathbf{b}| = \sqrt{(0.64 + 0.36 + 0)}$

$|\mathbf{b}| = \sqrt{(1)}$ = 1 unit length

In general:

a = <a1, a2, a3>

The unit vector of **a** = <a1/|**a**|, a2/|**a**|, a3/|**a**|>

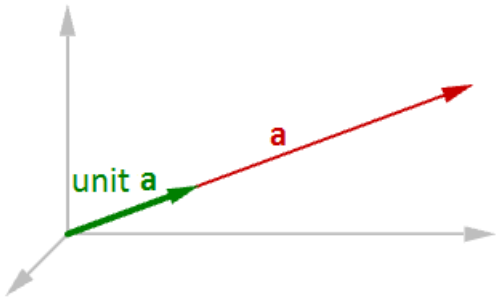


Figure (6): Unit vector equals one-unit length of the vector.

Vector operations

Vector scalar operation

Vector scalar operation involves multiplying a vector by a number. For example:

a = <4, 3, 0>

2***a*** = <2*4, 2*3, 2*0>

2***a*** = <8, 6, 0>

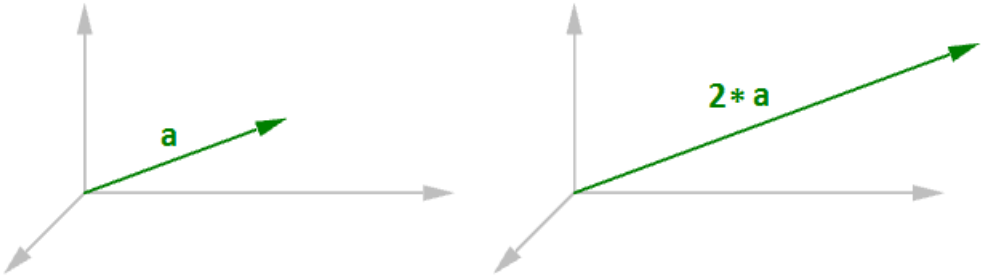


Figure (7): Vector scalar operation

In general, given vector **a** = <a1, a2, a3>, and a real number *t*

*t****a*** = < *t**a1, *t**a2, *t**a3 >

Vector addition

Vector addition takes two vectors and produces a third vector. We add vectors by adding their components.

> Vectors are added by adding their components.

For example, if we have two vectors:

a<1, 2, 0>

b<4, 1, 3>

a+**b** = <1+4, 2+1, 0+3>

a+**b** = <5, 3, 3>

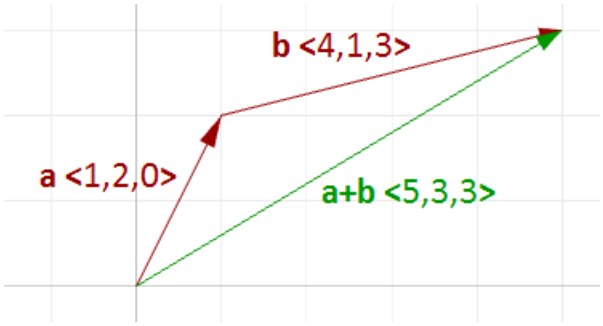


Figure (8): Vector addition.

In general, vector addition of the two vectors **a** and **b** is calculated as follows:

a = <a1, a2, a3>

b = <b1, b2, b3>

a+**b** = <a1+b1, a2+b2, a3+b3>

Vector addition is useful for finding the average direction of two or more vectors. In this case, we usually use same-length vectors. Here is an example that shows the difference between using same-length vectors and different-length vectors on the resulting vector addition:

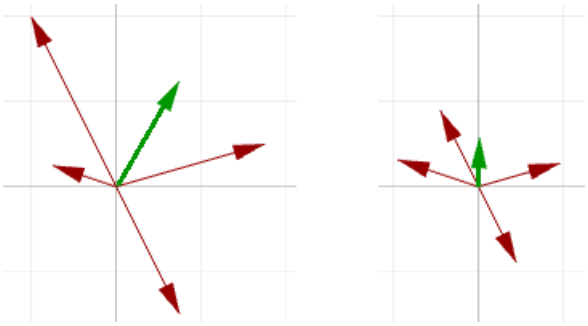


Figure (9): Adding various length vectors (left). Adding same length vectors (right) to get the average direction.

Input vectors are not likely to be same length. In order to find the average direction, you need to use the unit vector of input vectors. As mentioned before, the unit vector is a vector of that has a length equal to 1.

Vector subtraction

Vector subtraction takes two vectors and produces a third vector. We subtract two vectors by subtracting corresponding components. For example, if we have two vectors **a** and **b** and we subtract **b** from **a**, then:

a<1, 2, 0>

b<4, 1, 4>

a-**b** = <1-4, 2-1, 0-4>

a-**b** = <-3, 1, -4>

If we subtract **b** from **a**, we get a different result:

b - **a** = <4-1, 1-2, 4-0>

b - **a** = <3, -1, 4>

Note that the vector **b - a** has the same length as the vector **a - b**, but goes in the opposite direction.

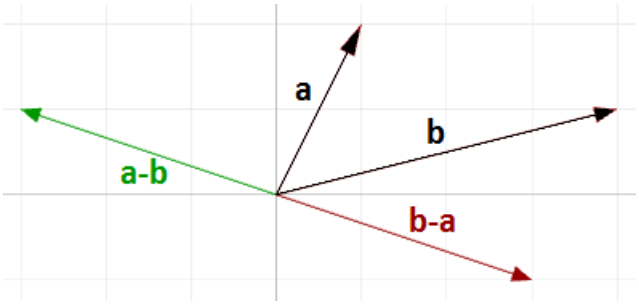


Figure (10): Vector subtraction.

In general, if we have two vectors, **a** and **b**, then **a** - **b** is a vector that is calculated as follows:

a = <a1, a2, a3>

b = <b1, b2, b3>

a - **b** = <a1 - b1, a2 - b2, a3 - b3>

Vector subtraction is commonly used to find vectors between points. So if we need to find a vector that goes from the tip point of the position vector **b** to the tip point of the position vector **a**, then we use vector subtraction (**a-b**) as shown in Figure (11).

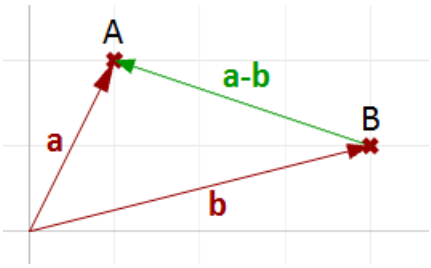


Figure (11): Use vector subtraction to find a vector between two points.

Vector properties

There are eight properties of vectors. If **a**, **b**, and **c** are vectors, and *s* and *t* are numbers, then:

$\mathbf{a} + \mathbf{b} = \mathbf{b} + \mathbf{a}$

$\mathbf{a} + 0 = \mathbf{a}$

$s * (\mathbf{a} + \mathbf{b}) = s * \mathbf{a} + s * \mathbf{b}$

$s * t * (\mathbf{a}) = s * (t * \mathbf{a})$

$\mathbf{a} + (\mathbf{b} + \mathbf{c}) = (\mathbf{a} + \mathbf{b}) + \mathbf{c}$

$\mathbf{a} + (-\mathbf{a}) = 0$

$(s + t) * \mathbf{a} = s * \mathbf{a} + t * \mathbf{a}$

$1 * \mathbf{a} = \mathbf{a}$

Vector dot product

The dot product takes two vectors and produces a number.

For example, if we have the two vectors **a** and **b** so that:

a = <1, 2, 3>

b = <5, 6, 7>

Then the dot product is the sum of multiplying the components as follows:

$\mathbf{a} \cdot \mathbf{b} = 1 * 5 + 2 * 6 + 3 * 7$

$\mathbf{a} \cdot \mathbf{b} = 38$

In general, given the two vectors **a** and **b**:

a = <a1, a2, a3>

b = <b1, b2, b3>

$\mathbf{a} \cdot \mathbf{b} = a1 * b1 + a2 * b2 + a3 * b3$

We always get a positive number for the dot product between two vectors when they go in the same general direction. A negative dot product between two vectors means that the two vectors go in the opposite general direction.

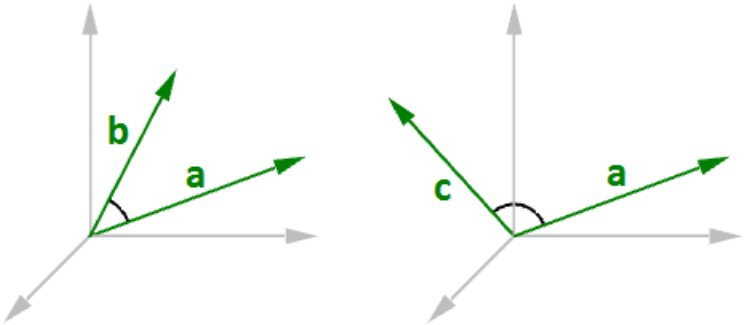


Figure (12): When the two vectors go in the same direction (left), the result is a positive dot product. When the two vectors go in the opposite direction (right), the result is a negative dot product.

When calculating the dot product of two unit vectors, the result is always between -1 and +1. For example:

a = <1, 0, 0>

b = <0.6, 0.8, 0>

$\mathbf{a} \cdot \mathbf{b} = (1 * 0.6, 0 * 0.8, 0 * 0) = 0.6$

In addition, the dot product of a vector with itself is equal to that vector's length to the power of two. For example:

a = <0, 3, 4>

$\mathbf{a} \cdot \mathbf{a} = 0 * 0 + 3 * 3 + 4 * 4$

$\mathbf{a} \cdot \mathbf{a} = 25$

Calculating the square length of vector **a**:

$|\mathbf{a}| = \sqrt{(4^2 + 3^2 + 0^2)}$

$|\mathbf{a}| = 5$

$|\mathbf{a}|^2 = 25$

Vector dot product, lengths, and angles

There is a relationship between the dot product of two vectors and the angle between them.

> The dot product of two non-zero unit vectors equals the cosine of the angle between them.

In general:

$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| * |\mathbf{b}| * \cos(\theta)$, or

$\mathbf{a} \cdot \mathbf{b} / (|\mathbf{a}| * |\mathbf{b}|) = \cos(\theta)$

Where:

θ is the angle included between the vectors.

If vectors **a** and **b** are unit vectors, we can simply say:

$\mathbf{a} \cdot \mathbf{b} = \cos(\theta)$

And since the cosine of a 90-degree angle is equal to 0, we can say:

> Vectors *a* and *b* are orthogonal if, and only if, $a \cdot b = 0$.

For example, if we calculate the dot product of the two orthogonal vectors, World x-axis and y-axis, the result will equal zero.

$\mathbf{x} = \langle 1, 0, 0 \rangle$

$\mathbf{y} = \langle 0, 1, 0 \rangle$

$\mathbf{x} \cdot \mathbf{y} = (1 * 0) + (0 * 1) + (0 * 0)$

$\mathbf{x} \cdot \mathbf{y} = 0$

There is also a relationship between the dot product and the projection length of one vector onto another. For example:

$\mathbf{a} = \langle 5, 2, 0 \rangle$

$\mathbf{b} = \langle 9, 0, 0 \rangle$

$\text{unit}(\mathbf{b}) = \langle 1, 0, 0 \rangle$

$\mathbf{a} \cdot \text{unit}(\mathbf{b}) = (4 * 1) + (3 * 0) + (0 * 0)$

$\mathbf{a} \cdot \text{unit}(\mathbf{b}) = 4$ (which is equal to the projection length of **a** onto **b**)

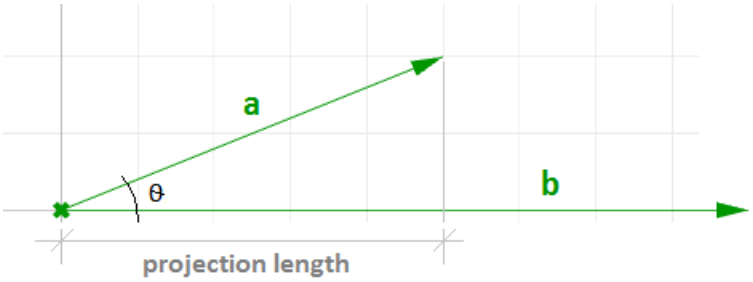


Figure (13): The dot product equals the projection length of one vector onto a non-zero unit vector.

In general, given a vector **a** and a non-zero vector **b**, we can calculate the projection length *pL* of vector **a** onto vector **b** using the dot product.

$pL = |\mathbf{a}| * \cos(\theta)$

$pL = \mathbf{a} \cdot \text{unit}(\mathbf{b})$

Dot product properties

If **a**, **b**, and **c** are vectors and *s* is a number, then:

$\mathbf{a} \cdot \mathbf{a} = |\mathbf{a}|^2$

$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$

$0 \cdot \mathbf{a} = 0$

$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$

$(s * \mathbf{a}) \cdot \mathbf{b} = s * (\mathbf{a} \cdot \mathbf{b}) = \mathbf{a} \cdot (s * \mathbf{b})$

Vector cross product

The cross product takes two vectors and produces a third vector that is orthogonal to both.

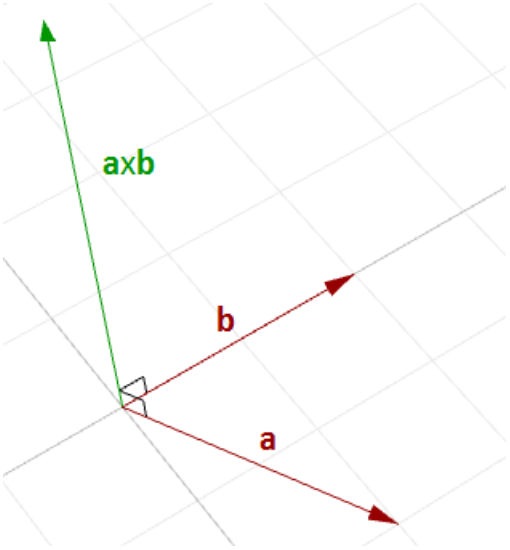


Figure (14): Calculating the cross product of two vectors.

For example, if you have two vectors lying on the World xy-plane, then their cross product is a vector perpendicular to the xy-plane going either in the positive or negative World z-axis direction. For example:

a = <3, 1, 0>

b = <1, 2, 0>

$\mathbf{a} \times \mathbf{b} = < (1 * 0 - 0 * 2), (0 * 1 - 3 * 0), (3 * 2 - 1 * 1) >$

$\mathbf{a} \times \mathbf{b} = <0, 0, 5>$

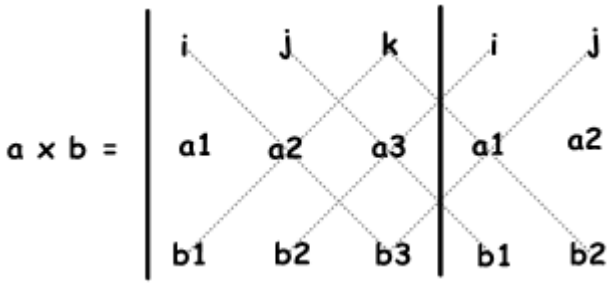
The vector *a* x *b* is orthogonal to both *a* and *b.*

You will probably never need to calculate a cross product of two vectors by hand, but if you are curious about how it is done, continue reading; otherwise you can safely skip this section. The cross product $\mathbf{a} \times \mathbf{b}$ is defined using *determinants*. Here is a simple illustration of how to calculate a determinant using the standard basis vectors:

i = <1, 0, 0>

j = <0,1, 0>

k = <0, 0, 1>



The cross product of the two vectors **a**<a1, a2, a3> and **b**<b1, b2, b3> is calculated as follows using the above diagram:

a × **b** = **i** (a2 * b3) + **j** (a3 * b1) + **k** (a1 * b2) - **k** (a2 * b1) - **i** (a3 * b2) - **j** (a1 * b3)

a × **b** = **i** (a2 * b3 - a3 * b2) + **j** (a3 * b1 - a1 * b3) + **k** (a1 * b2 - a2 * b1)

a × **b** = <a2 * b3 – a3 * b2, a3 * b1 - a1 * b3, a1 * b2 - a2 * b1 >

Cross product and angle between vectors

There is a relationship between the angle between two vectors and the length of their cross product vector. The smaller the angle (smaller sine); the shorter the cross product vector will be. The order of operands is important in vectors cross product. For example:

a = <1, 0, 0>

b = <0, 1, 0>

a × **b** = <0, 0, 1>

b × **a** = <0, 0, -1>

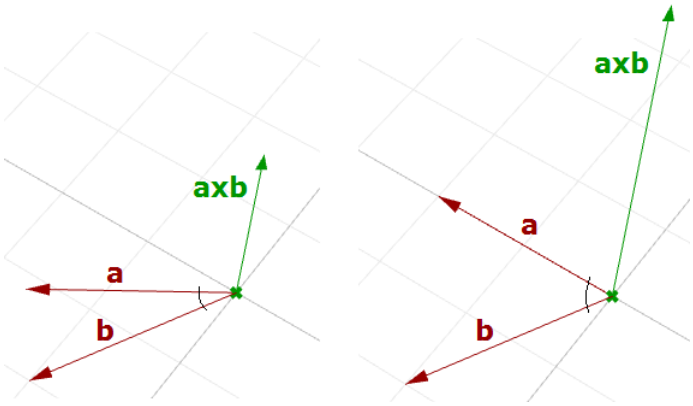
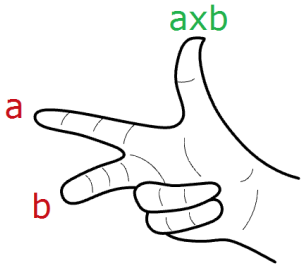


Figure (15): The relationship between the sine of the angle between two vectors and the length of their cross product vector.

In Rhino's right-handed system, the direction of **a** × **b** is given by the right-hand rule (where **a** = index finger, **b** = middle finger, and **a** × **b** = thumb).



In general, for any pair of 3-D vectors **a** and **b**:

$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin(\theta)$

Where:

θ is the angle included between the position vectors of **a** and **b**

If **a** and **b** are unit vectors, then we can simply say that the length of their cross product equals the sine of the angle between them. In other words:

$|\mathbf{a} \times \mathbf{b}| = \sin(\theta)$

The cross product between two vectors helps us determine if two vectors are parallel. This is because the result is always a zero vector.

> Vectors *a* and *b* are parallel if, and only if, $a \times b = 0$.

Cross product properties

If **a**, **b**, and **c** are vectors, and *s* is a number, then:

$\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$

$(s * \mathbf{a}) \times \mathbf{b} = s * (\mathbf{a} \times \mathbf{b}) = \mathbf{a} \times (s * \mathbf{b})$

$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$

$(\mathbf{a} + \mathbf{b}) \times \mathbf{c} = \mathbf{a} \times \mathbf{c} + \mathbf{b} \times \mathbf{c}$

$\mathbf{a} \cdot (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) \cdot \mathbf{c}$

$\mathbf{a} \times (\mathbf{b} \times \mathbf{c}) = (\mathbf{a} \cdot \mathbf{c}) * \mathbf{b} - (\mathbf{a} \cdot \mathbf{b}) * \mathbf{c}$

Vector equation of line

The vector line equation is used in 3-D modeling applications and computer graphics.

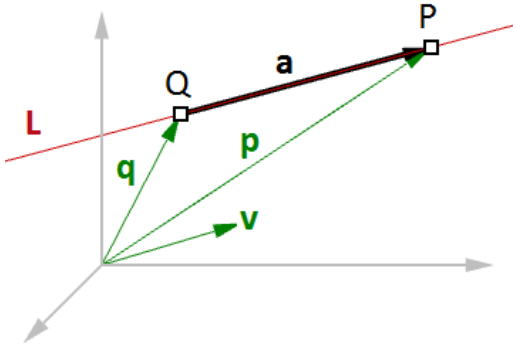


Figure (16): Vector equation of a line.

For example, if we know the direction of a line and a point on that line, then we can find any other point on the line using vectors, as in the following:

L = line

v = <a, b, c> line direction unit vector

Q = (x0, y0, z0) line position point

P = (x, y, z) any point on the line

We know that:

$\mathbf{a} = t * \mathbf{v}$ --- (2)

$\mathbf{p} = \mathbf{q} + \mathbf{a}$ --- (1)

From 1 and 2:

$\mathbf{p} = \mathbf{q} + t * \mathbf{v}$ --- (3)

However, we can write (3) as follows:

$\langle x, y, z \rangle = \langle x0, y0, z0 \rangle + \langle t * a, t * b, t * c \rangle$

$\langle x, y, z \rangle = \langle x0 + t * a, y0 + t * b, z0 + t * c \rangle$

Therefore:

$x = x0 + t * a$
$y = y0 + t * b$
$z = z0 + t * c$

Which is the same as:

$P = Q + t * \mathbf{v}$

> Given a point Q and a direction v on a line, any point P on that line can be calculated using the vector equation of a line P = Q + t * v where *t* is a number.

Another common example is to find the midpoint between two points. The following shows how to find the midpoint using the vector equation of a line:

q is the position vector for point Q

p is the position vector for point P

a is the vector going from Q to P

From vector subtraction, we know that:

$\mathbf{a} = \mathbf{p} - \mathbf{q}$

From the line equation, we know that:

$M = Q + t * \mathbf{a}$

And since we need to find midpoint, then:

$t = 0.5$

Hence we can say:

$M = Q + 0.5 * \mathbf{a}$

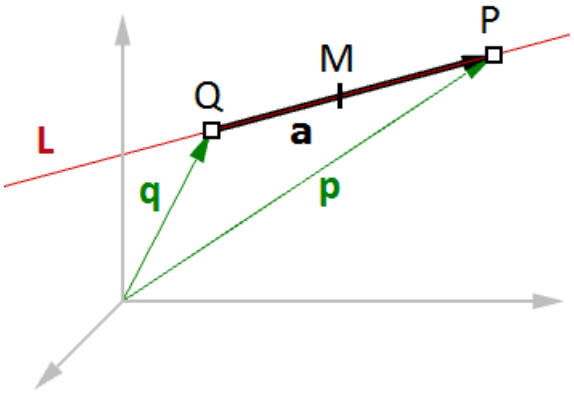


Figure (17): Find the midpoint between two input points.

In general, you can find any point between Q and P by changing the *t* value between 0 and 1 using the general equation:

$M = Q + t * (P - Q)$

Given two points Q and P, any point M between the two points is calculated using the equation M = Q + *t* * (P - Q) where *t* is a number between 0 and 1.

Vector equation of a plane

One way to define a plane is when you have a point and a vector that is perpendicular to the plane. That vector is usually referred to as *normal* to the plane. The normal points in the direction above the plane.

One example of how to calculate a plane normal is when we know three non-linear points on the plane.

In Figure (16), given:

A = the first point on the plane

B = the second point on the plane

C = the third point on the plane

And:

a = a position vector of point A

b = a position vector of point B

c = a position vector of point C

We can find the normal vector **n** as follows:

$\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$

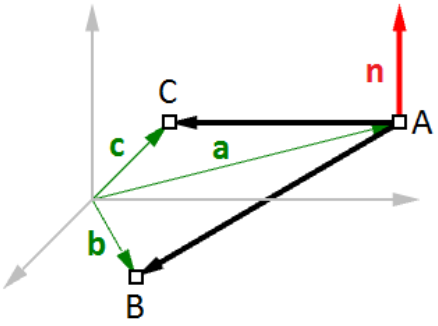


Figure (18): Vectors and planes

We can also derive the scalar equation of the plane using the vector dot product:

$\mathbf{n} \cdot (\mathbf{b} - \mathbf{a}) = 0$

If:

n = <a, b, c>

b = <x, y, z>

a = <x0, y0, z0>

Then we can expand the above:

<a, b, c> · <x-x0, y-y0, z-z0 > = 0

Solving the dot product gives the general scalar equation of a plane:

a * (x - x0) + b * (y - y0) + c * (z - z0) = 0

Tutorials

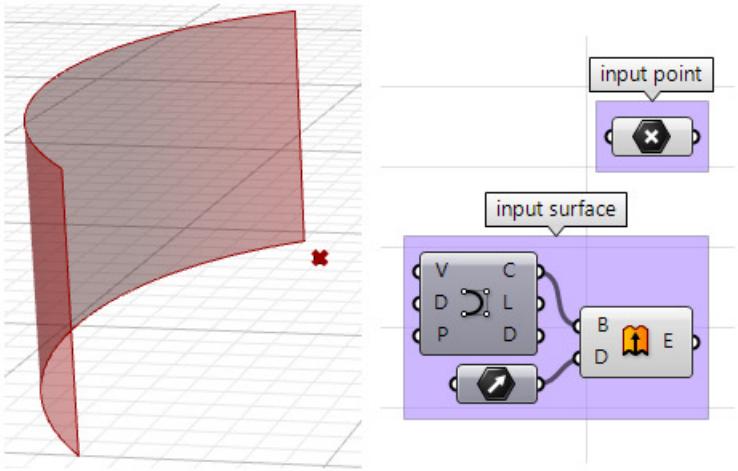
All the concepts we reviewed in this chapter have a direct application to solving common geometry problems encountered when modeling. The following are step-by-step tutorials that use the concepts learned in this chapter using Rhinoceros and Grasshopper (GH).

Face direction

Given a point and a surface, how can we determine whether the point is facing the front or back side of that surface?

Input:

1. a surface
2. a point



Parameters:

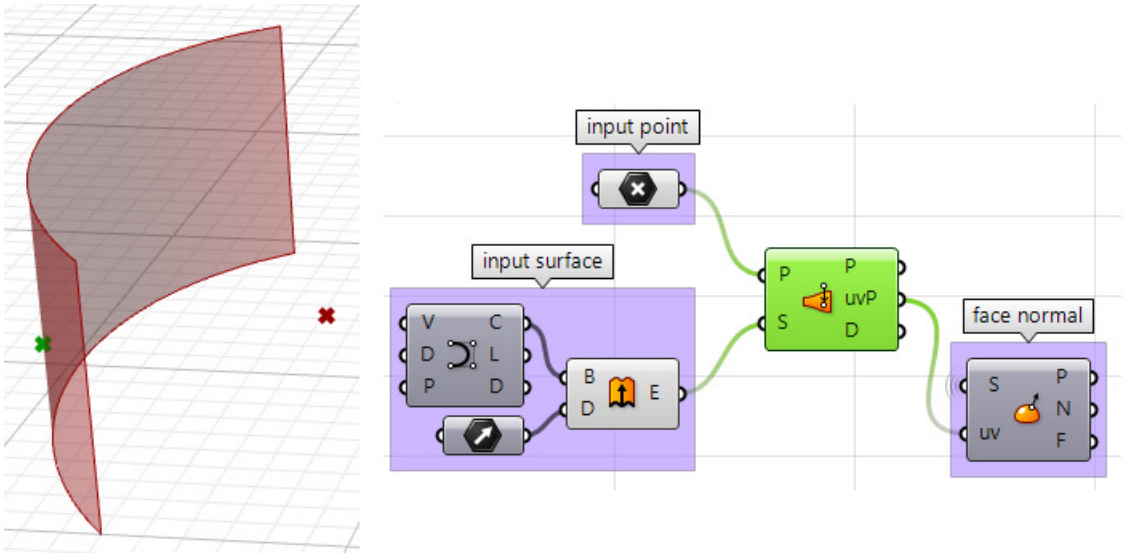
The face direction is defined by the surface normal direction. We will need the following information:

- The surface normal direction at a surface location closest to the input point.
- A vector direction from the closest point to the input point.

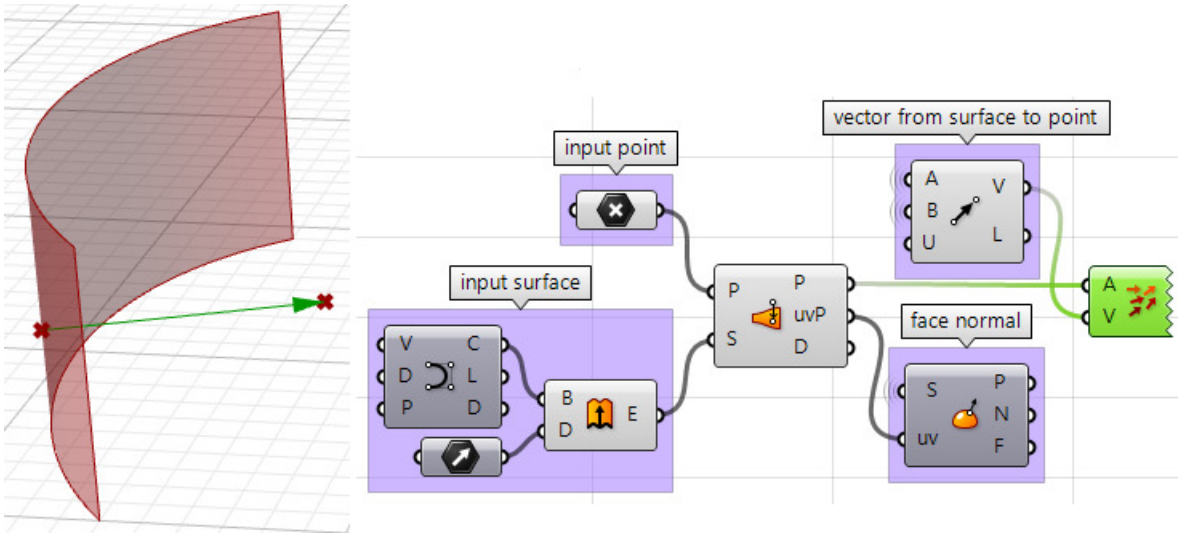
Compare the above two directions, if going the same direction, the point is facing the front side, otherwise it is facing the back.

Solution:

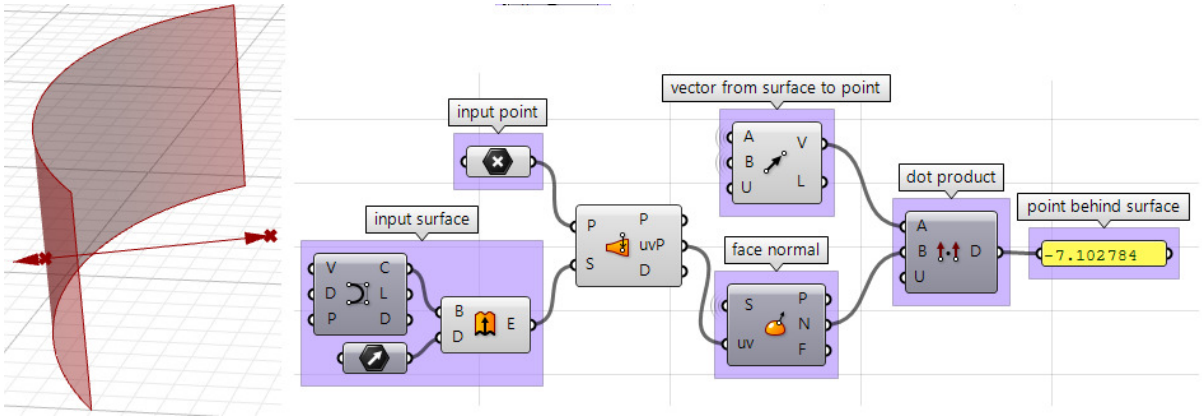
1. Find the closest point location on the surface relative to the input point using the **Pull** component. This will give us the uv location of the closest point, which we can then use to evaluate the surface and find its normal direction.



2. We can now use the closest point to draw a vector going towards the input point. We can also draw:

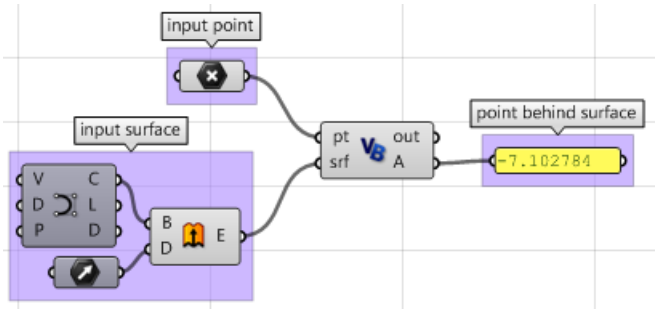


3. We can compare the two vectors using the dot product. If the result is positive, the point is in front of the surface. If the result is negative, the point is behind the surface.



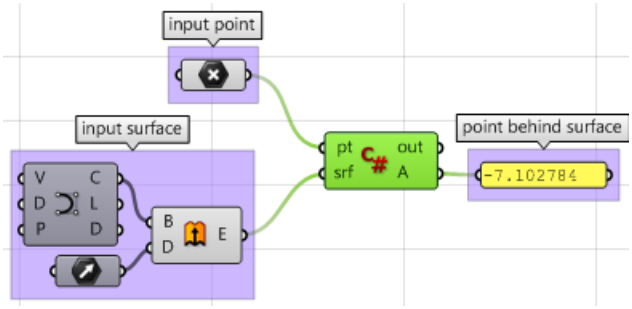
The above steps can also be solved using other scripting languages.

Using the Grasshopper VB component:



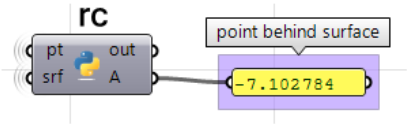

```
Private Sub RunScript(ByVal pt As Point3d, ByVal srf As Surface, ByRef A As Object)

  'Decalre variables
  Dim u, v As Double
  Dim closest_pt As Point3d

  'get closest point u, v
  srf.ClosestPoint(pt, u, v)

  'get closest point
  closest_pt = srf.PointAt(u, v)

  'calculate direction from closest point to test point
  Dim dir As New Vector3d(pt - closest_pt)

  'calculate surface normal
  Dim normal = srf.NormalAt(u, v)

  'compare the two directions using the dot product
  A = dir * normal

End Sub
```

Using the Grasshopper C# component:



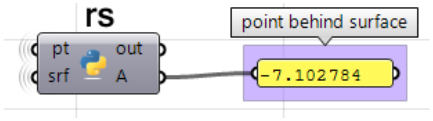

```
private void RunScript(Point3d pt, Surface srf, ref object A)
{
  //Decalre variables
  double u, v;
  Point3d closest_pt;

  //get closest point u, v
  srf.ClosestPoint(pt, out u, out v);

  //get closest point
  closest_pt = srf.PointAt(u, v);

  //calculate direction from closest point to test point
  Vector3d dir = pt - closest_pt;

  //calculate surface normal
  Vector3d normal = srf.NormalAt(u, v);

  //compare the two directions using the dot product
  A = dir * normal;
}
```

Using the Grasshopper Python component and the RhinoCommon SDK:




```
#find the closest point
found, u, v = srf.ClosestPoint(pt)

if found:

    #get closest point
    closest_pt = srf.PointAt(u, v)

    #calculate direction from closest point to test point
    dir = pt - closest_pt

    #calculate surface normal
    normal = srf.NormalAt(u, v)

    #compare the two directions using the dot product
    A = dir * normal
```

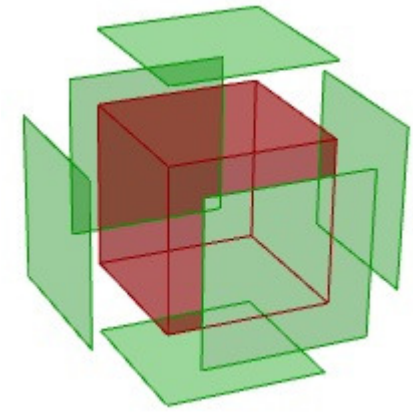
Using the Grasshopper Python component and the RhinoScriptSyntax Library:




```
#import RhinoScript library
import rhinoscriptsyntax as rs

#find the closest point
u, v = rs.SurfaceClosestPoint(srf, pt)

#get closest point
closest_pt = rs.EvaluateSurface(srf, u, v)

#calculate direction from closest point to test point
dir = rs.PointCoordinates(pt) - closest_pt

#calculate surface normal
normal = rs.SurfaceNormal(srf, [u, v])

#compare the two directions using the dot product
A = dir * normal
```

Exploded box

The following tutorial shows how to explode a polysurface. This is what the final exploded box looks like:



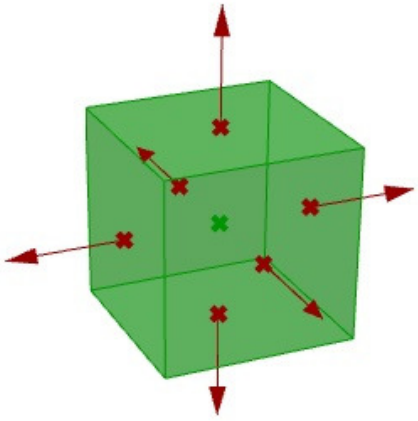
Input:

Identify the input, which is a box. We will use the **Box** parameter in GH:

Parameters:

Think of all the parameters we need to know in order to solve this tutorial.

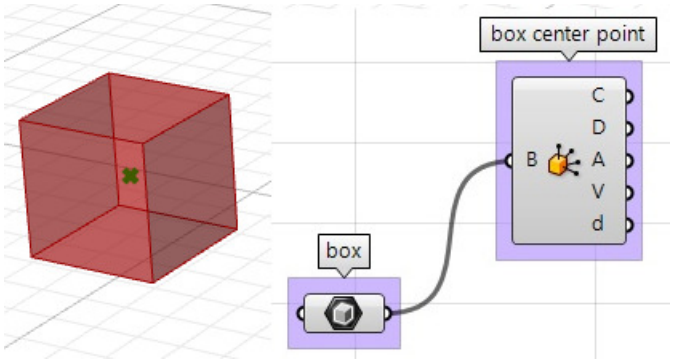
- The center of explosion.
- The box faces we are exploding.
- The direction in which each face is moving.



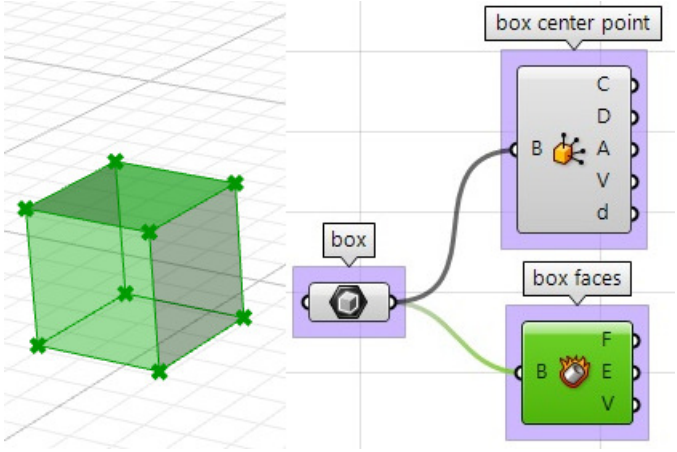
Once we have identified the parameters, it is a matter of putting it together in a solution by piecing together the logical steps to reach an answer.

Solution:

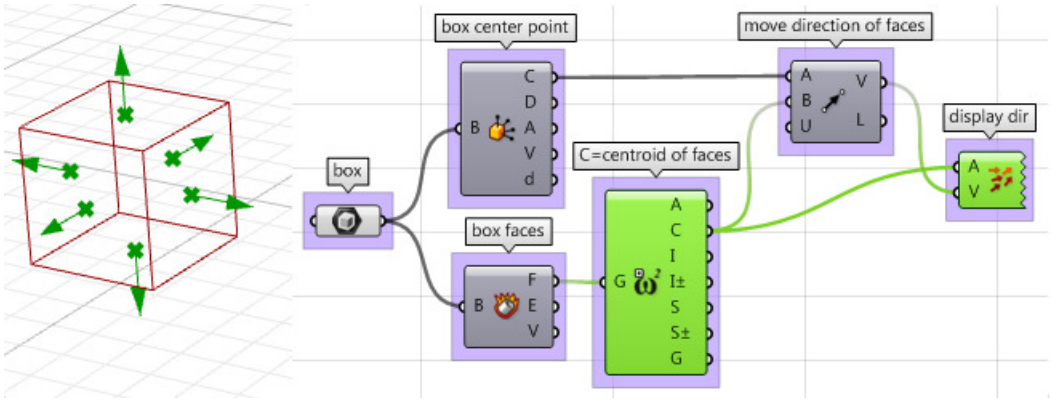
1. Find the center of the box using the **Box Properties** component in GH:



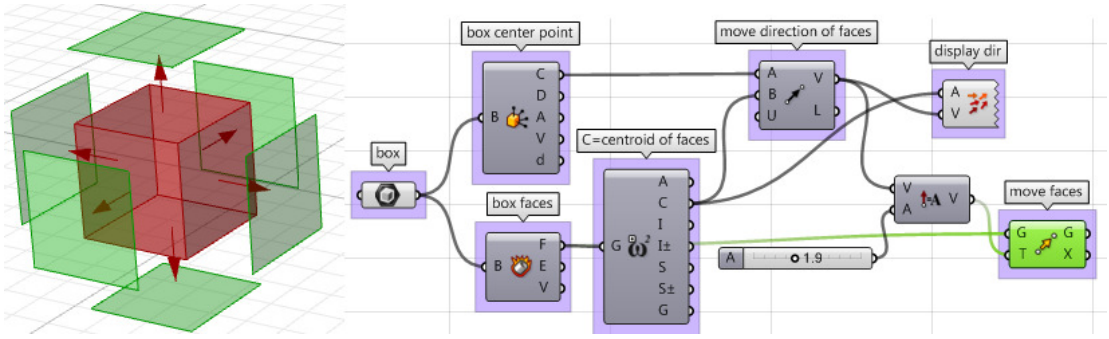
2. Extract the box faces with the **Deconstruct Brep** component:



3. The direction we move the faces is the tricky part. We need to first find the center of each face, and then define the direction from the center of the box towards the center of each face as follows:

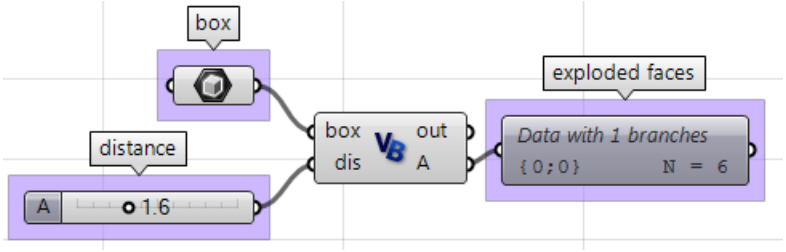


4. Once we have all the parameters scripted, we can use the **Move** component to move the faces in the appropriate direction. Just make sure to set the vectors to the desired amplitude, and you will be good to go.



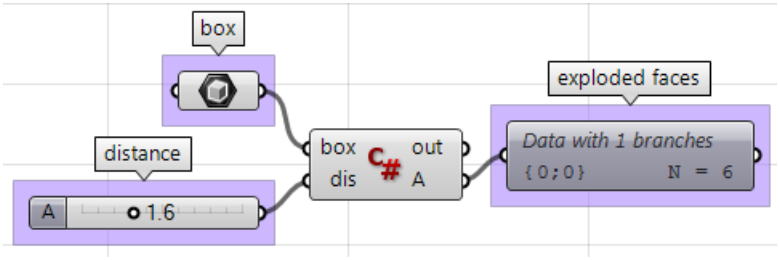
The above steps can also be solved using VB script, C# or Python. Following is the solution using these scripting languages.

Using the Grasshopper VB component:



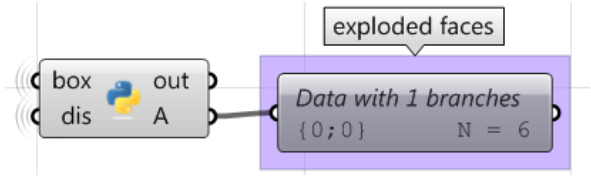
```
Private Sub RunScript(ByVal box As Brep, ByVal dis As Double, ByRef A As Object)

  'get the brep center
  Dim area As Rhino.Geometry.AreaMassProperties
  area = Rhino.Geometry.AreaMassProperties.Compute(box)

  Dim box_center As Point3d
  box_center = area.Centroid

  'get a list of faces
  Dim faces As Rhino.Geometry.Collections.BrepFaceList = box.Faces

  'decalre variables
  Dim center As Point3d
  Dim dir As Vector3d
  Dim exploded_faces As New List( Of Rhino.Geometry.Brep )
  Dim i As Int32
  'loop through all faces

  For i = 0 To faces.Count() - 1
    'extract each of the face
    Dim extracted_face As Rhino.Geometry.Brep = box.Faces.ExtractFace(i)

    'get the center of each face
    area = Rhino.Geometry.AreaMassProperties.Compute(extracted_face)
    center = area.Centroid

    'calculate move direction (from box centroid to face center)
    dir = center - box_center
    dir.Unitize()
    dir *= dis

    'move the extracted face
    extracted_face.Transform(Transform.Translation(dir))

    'add to exploded_faces list
    exploded_faces.Add(extracted_face)
  Next

  'assign exploded list of faces to output
  A = exploded_faces
End Sub
```

Using the Grasshopper C# component:




```
private void RunScript(Brep box, double dis, ref object A)
{

  //get the brep center
  Rhino.Geometry.AreaMassProperties area = Rhino.Geometry.AreaMassProperties.Compute(box);
  Point3d box_center = area.Centroid;

  //get a list of faces
  Rhino.Geometry.Collections.BrepFaceList faces = box.Faces;

  //decalre variables
  Point3d center;
  Vector3d dir;
  List<Rhino.Geometry.Brep> exploded_faces = new List<Rhino.Geometry.Brep>();

  //loop through all faces
  for( int i = 0; i < faces.Count(); i++ )
  {
    //extract each of the face
    Rhino.Geometry.Brep extracted_face = box.Faces.ExtractFace(i);

    //get the center of each face
    area = Rhino.Geometry.AreaMassProperties.Compute(extracted_face);
    center = area.Centroid;

    //calculate move direction (from box centroid to face center)
    dir = center - box_center;
    dir.Unitize();
    dir *= dis;

    //move the extracted face
    extracted_face.Transform(Transform.Translation(dir));

    //add to exploded_faces list
    exploded_faces.Add(extracted_face);
  }

  //assign exploded list of faces to output
  A = exploded_faces;
}
```

Using the Grasshopper Python component:

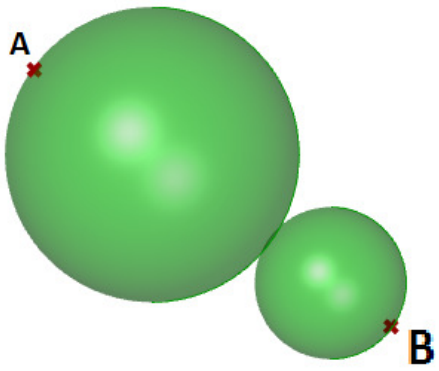



```
import Rhino

#get the brep center
area = Rhino.Geometry.AreaMassProperties.Compute(box)
box_center = area.Centroid

#get a list of faces
faces = box.Faces

#decalre variables
exploded_faces = []

#loop through all faces
for i, face in enumerate(faces):

    #get a duplicate of the face
    extracted_face = faces.ExtractFace(i)

    #get the center of each face
    area = Rhino.Geometry.AreaMassProperties.Compute(extracted_face)
    center = area.Centroid

    #calculate move direction (from box centroid to face center)
    dir = center - box_center
    dir.Unitize()
    dir *= dis

    #move the extracted face
    move = Rhino.Geometry.Transform.Translation(dir)
    extracted_face.Transform(move)

    #add to exploded_faces list
    exploded_faces.append(extracted_face)

#assign exploded list of faces to output
A = exploded_faces
```

Tangent spheres

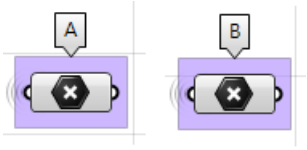
This tutorial will show how to create two tangent spheres between two input points.

This is what the result looks like:



Input:

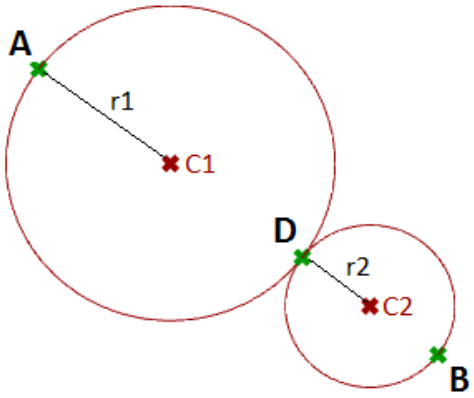
Two points (A and B) in the 3-D coordinate system.



Parameters:

The following is a diagram of the parameters that we will need in order to solve the problem:

- A tangent point D between the two spheres, at some *t* parameter (0-1) between points A and B.
- The center of the first sphere or the midpoint C1 between A and D.
- The center of the second sphere or the midpoint C2 between D and B.
- The radius of the first sphere (r1) or the distance between A and C1.
- The radius of the second sphere (r2) or the distance between D and C2.



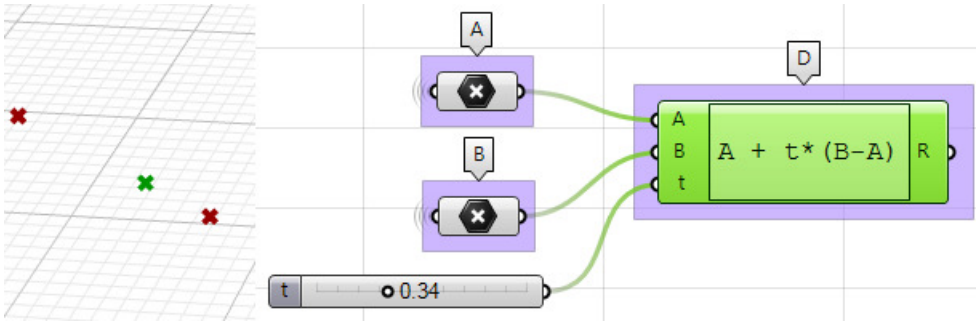
Solution:

1. Use the **Expression** component to define point **D** between **A** and **B** at some parameter **t**. The expression we will use is based on the vector equation of a line: D = A + t*(B-A).

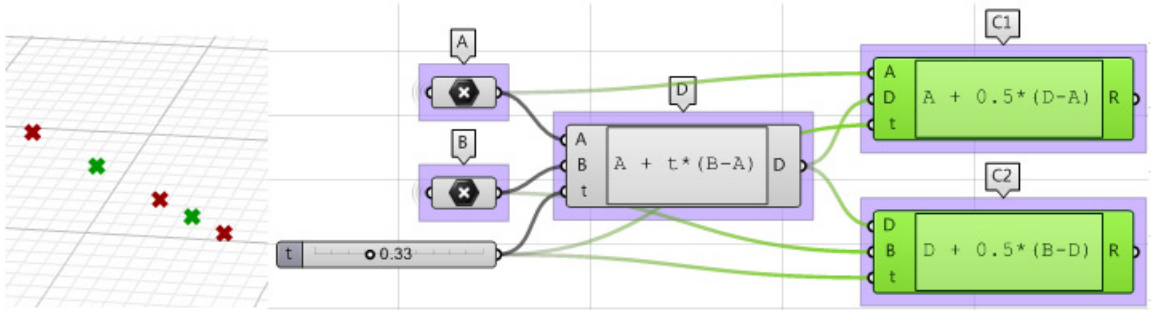
   B-A: is the vector that goes from B to A using the vector subtraction operation.

   t*(B-A): where *t* is between 0 and 1 to get us a location on the vector.

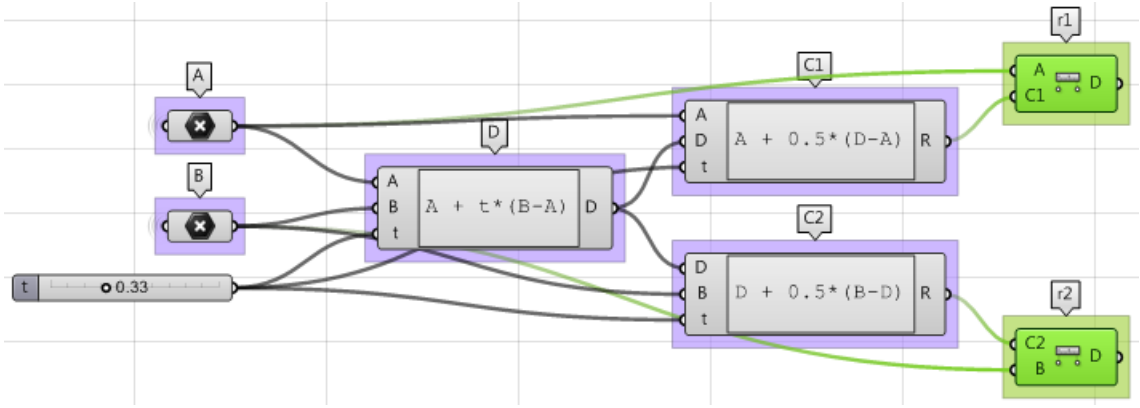
   A+t*(B-A): gets a point on the vector between A and B.



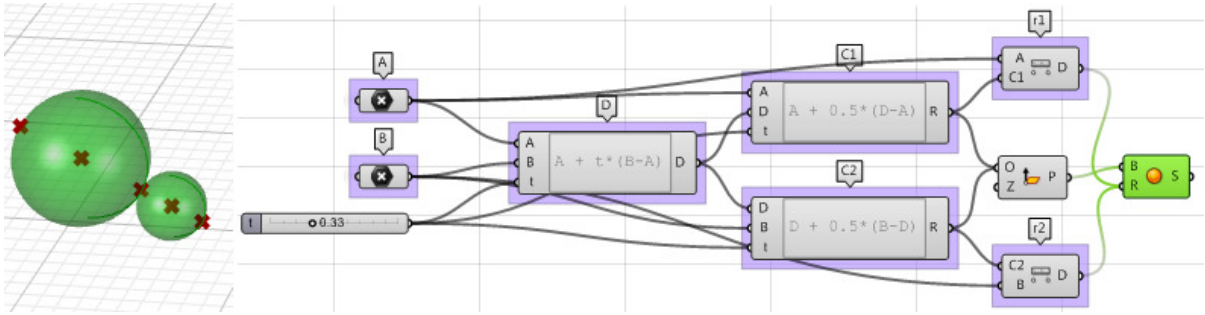
2. Use the **Expression** component to also define the mid points **C1** and **C2**.



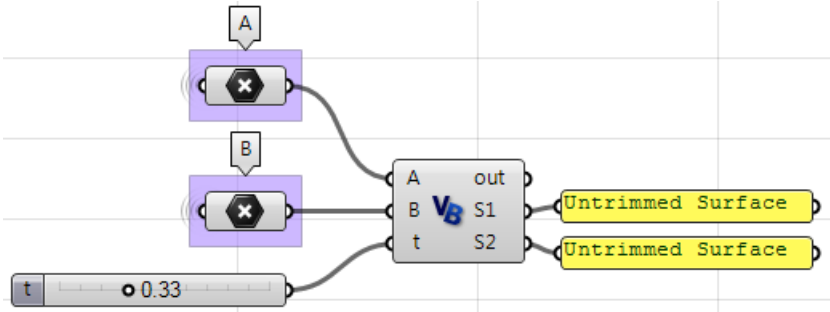
3. The first sphere radius (**r1**) and the second sphere radius (**r2**) can be calculated using the **Distance** component.



4. The final step involves creating the sphere from a base plane and radius. We need to make sure the origins are hooked to **C1** and **C2** and the radius from **r1** and **r2**.

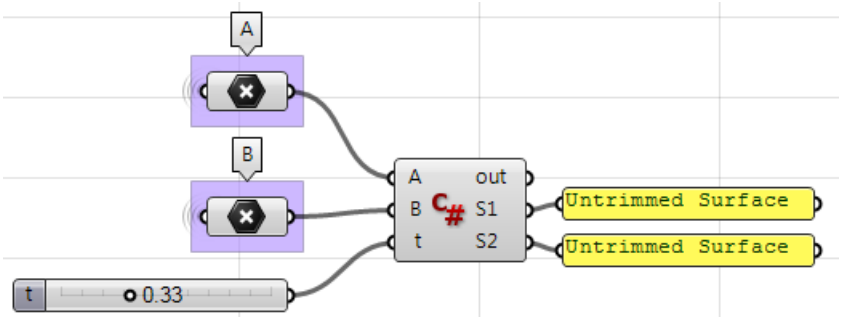


Using the Grasshopper VB component:



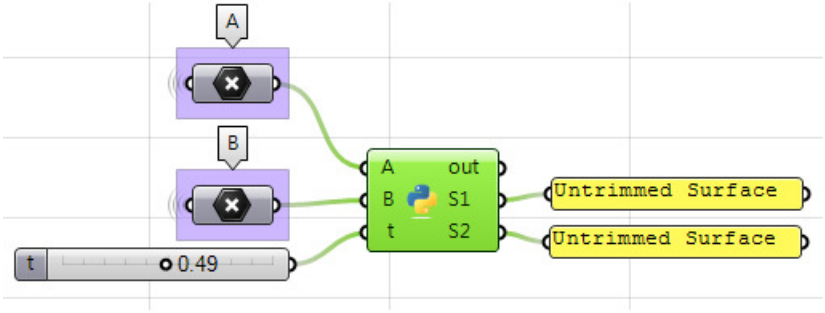

```
Private Sub RunScript(ByVal A As Point3d, ByVal B As Point3d, ByVal t As Double, ByRef S1 As Object, ByRef S2 As Object)

  'declare variables
  Dim D, C1, C2 As Rhino.Geometry.Point3d
  Dim r1, r2 As Double

  'find a point between A and B
  D = A + t * (B - A)

  'find mid point between A and D
  C1 = A + 0.5 * (D - A)

  'find mid point between D and B
  C2 = D + 0.5 * (B - D)

  'find spheres radius
  r1 = A.DistanceTo(C1)
  r2 = B.DistanceTo(C2)

  'create spheres and assign to output
  S1 = New Rhino.Geometry.Sphere(C1, r1)
  S2 = New Rhino.Geometry.Sphere(C2, r2)

End Sub
```

Using the Grasshopper C# component:



```
private void RunScript(Point3d A, Point3d B, double t, ref object S1, ref object S2)
{
  //declare variables
  Rhino.Geometry.Point3d D, C1, C2;
  double r1, r2;

  //find a point between A and B
  D = A + t * (B - A);

  //find mid point between A and D
  C1 = A + 0.5 * (D - A);

  //find mid point between D and B
  C2 = D + 0.5 * (B - D);

  //find spheres radius
  r1 = A.DistanceTo(C1);
  r2 = B.DistanceTo(C2);

  //create spheres and assign to output
  S1 = new Rhino.Geometry.Sphere(C1, r1);
  S2 = new Rhino.Geometry.Sphere(C2, r2);
}
```

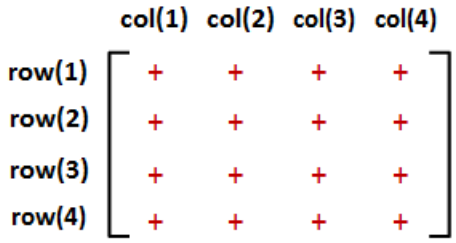
Using the Grasshopper Python component:




```
import Rhino

#find a point between A and B
D = A + t * (B - A)

#find mid point between A and D
C1 = A + 0.5 * (D - A)

#find mid point between D and B
C2 = D + 0.5 * (B - D)

#find spheres radius
r1 = A.DistanceTo(C1)
r2 = B.DistanceTo(C2)

#create spheres and assign to output
S1 = Rhino.Geometry.Sphere(C1, r1)
S2 = Rhino.Geometry.Sphere(C2, r2)
```

2 Matrices and Transformations

Transformations refer to operations such as moving (also called *translating*), rotating, and scaling objects. They are stored in 3-D programming using matrices, which are nothing but rectangular arrays of numbers. Multiple transformations can be performed very quickly using matrices. It turns out that a [4x4] matrix can represent all transformations. Having a unified matrix dimension for all transformations saves calculation time.

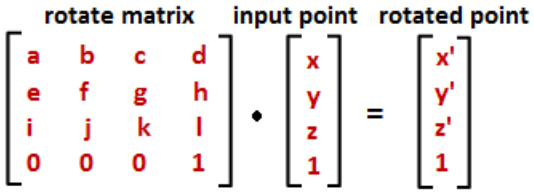


Matrix operations

The one operation that is most relevant in computer graphics is *matrix multiplication*. We will explain it with some detail.

Matrix multiplication

Matrix multiplication is used to apply transformations to geometry. For example if we have a point and would like to rotate it around some axis, we use a rotation matrix and multiply it by the point to get the new rotated location.



Most of the time, we need to perform multiple transformations on the same geometry. For example, if we need to move and rotate a thousand points, we can use either of the following methods.

Method 1

1. Multiply the move matrix by 1000 points to move the points.
2. Multiply the rotate matrix by the resulting 1000 points to rotate the moved points.

   Number of operations = **2000**.

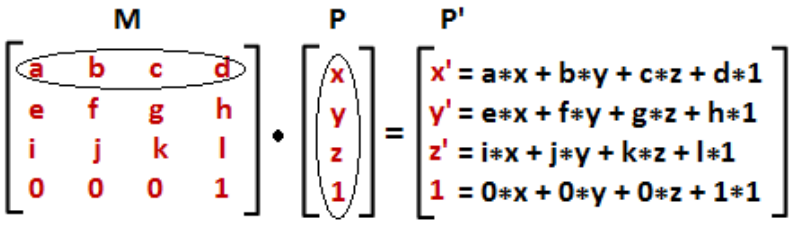
Method 2

1. Multiply the rotate and move matrices to create a combined transformation matrix.
2. Multiply the combined matrix by 1000 points to move and rotate in one step.

   Number of operations = **1001**.

Notice that method 1 takes almost twice the number of operations to achieve the same result. While method 2 is very efficient, it is only possible if both the move and rotate matrices are [4x4]. This is why in computer graphics a [4x4] matrix is used to represent all transformations, and a [4x1] matrix is used to represent points.

Three-dimensional modeling applications provide tools to apply transformations and multiply matrices, but if you are curious about how to mathematically multiply matrices, we will explain a simple example. In order to multiply two matrices, they have to have matching dimensions. That means the number of columns in the first matrix must equal the number of rows of the second matrix. The resulting matrix has a size equal to the number of rows from the first matrix and the number of columns from the second matrix. For example, if we have two matrices, **M** and **P**, with dimensions equal to [4x4] and [4x1] respectively, then there resulting multiplication matrix **M · P** has a dimension equal to [4x1] as shown in the following illustration:



Identity matrix

The *identity matrix* is a special matrix where all diagonal components equal 1 and the rest equal 0.

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

The main property of the identity matrix is that if it is multiplied by any other matrix, the values multiplied by zero do not change.

1.0	0.0	0.0	0.0
0.0	1.0	0.0	0.0
0.0	0.0	1.0	0.0
0.0	0.0	0.0	1.0

X

2.0
3.0
1.0
1.0

=

1.0x2.0 + 0.0x3.0 + 0.0x1.0 + 0.0x1.0
0.0x2.0 + 1.0x3.0 + 0.0x1.0 + 0.0x1.0
0.0x2.0 + 0.0x3.0 + 1.0x1.0 + 0.0x1.0
0.0x2.0 + 0.0x3.0 + 0.0x1.0 + 1.0x1.0

=

2.0
3.0
1.0
1.0

Transformation operations

Most transformations preserve the parallel relationship among the parts of the geometry. For example collinear points remain collinear after the transformation. Also points on one plane stay coplanar after transformation. This type of transformation is called an *affine transformation*.

Translation (move) transformation

Moving a point from a starting position by certain a vector can be calculated as follows:

P' = P + V

Suppose:

P(x,y,z) is a given point

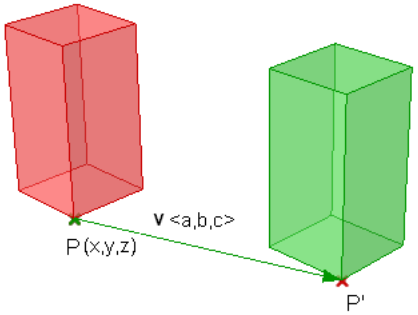
v<a,b,c> is a translation vector

Then:

P'(x) = x + a

P'(y) = y + b

P'(z) = z + c



Points are represented in a matrix format using a [4x1] matrix with a 1 inserted in the last row. For example the point P(x,y,z) is represented as follows:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Using a [4x4] matrix for transformations (what is called a homogenous coordinate system), instead of a [3x3] matrices, allows representing all transformations including translation. The general format for a translation matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & a1 \\ 0 & 1 & 0 & a2 \\ 0 & 0 & 1 & a3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example, to move point P(2,3,1) by vector **v**<2,2,2>, the new point location is:

P’ = P + **v** = (2+2, 3+2, 1+2) = (4, 5, 3)

If we use the matrix form and multiply the translation matrix by the input point, we get the new point location as in the following:

$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} (1*2 + 0*3 + 0*1 + 2*1) \\ (0*2 + 1*3 + 0*1 + 2*1) \\ (0*2 + 0*3 + 1*1 + 2*1) \\ (0*2 + 0*3 + 0*1 + 1*1) \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 3 \\ 1 \end{bmatrix}$$

Similarly, any geometry is translated by multiplying its construction points by the translation matrix. For example, if we have a box that is defined by eight corner points, and we want to move it 4 units in the x-direction, 5 units in the y-direction and 3 units in the z- direction, we must multiply each of the eight box corner points by the following translation matrix to get the new box.

$$\begin{bmatrix} 1 & 0 & 0 & 4 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

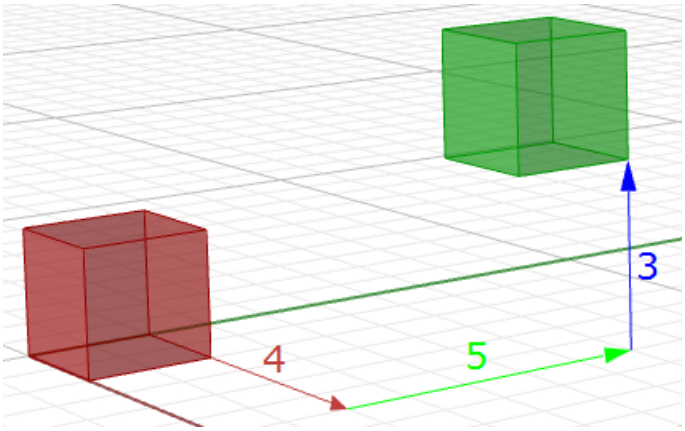


Figure (19): Translate all box corner points.

Rotation transformation

This section shows how to calculate rotation around the z-axis and the origin point using trigonometry, and then to deduce the general matrix format for the rotation.

Take a point on x,y plane P(x,y) and rotate it by angle(b).

From the figure, we can say the following:

$x = d \cos(a)$ ---(1)

$y = d \sin(a)$ ---(2)

$x' = d \cos(b+a)$ ---(3)

$y' = d \sin(b+a)$ --- (4)

Expanding x' and y' using trigonometric identities for the sine and cosine of the sum of angles:

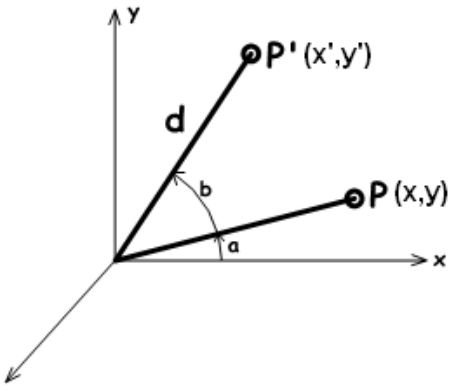
$x' = d \cos(a)\cos(b) - d \sin(a)\sin(b)$ ---(5)

$y' = d \cos(a)\sin(b) + d \sin(a)\cos(b)$ ---(6)

Using Eq 1 and 2:

$x' = x \cos(b) - y \sin(b)$

$y' = x \sin(b) + y \cos(b)$



The rotation matrix around the **z-axis** looks like:

$$\begin{bmatrix} \cos(b) & -\sin(b) & 0 & 0 \\ \sin(b) & \cos(b) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix around the **x-axis** by angle **b** looks like:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(b) & -\sin(b) & 0 \\ 0 & \sin(b) & \cos(b) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The rotation matrix around the **y-axis** by angle **b** looks like:

$$\begin{bmatrix} \cos(b) & 0 & \sin(b) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(b) & 0 & \cos(b) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example, if we have a box and would like to rotate it 30 degrees, we need the following:

1. Construct the 30-degree rotation matrix. Using the generic form and the cos and sin values of 30-degree angle, the rotation matrix will look like the following:

$$\begin{bmatrix} 0.87 & -0.5 & 0 & 0 \\ 0.5 & 0.87 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. Multiply the rotation matrix by the input geometry, or in the case of a box, multiply by each of the corner points to find the box's new location.

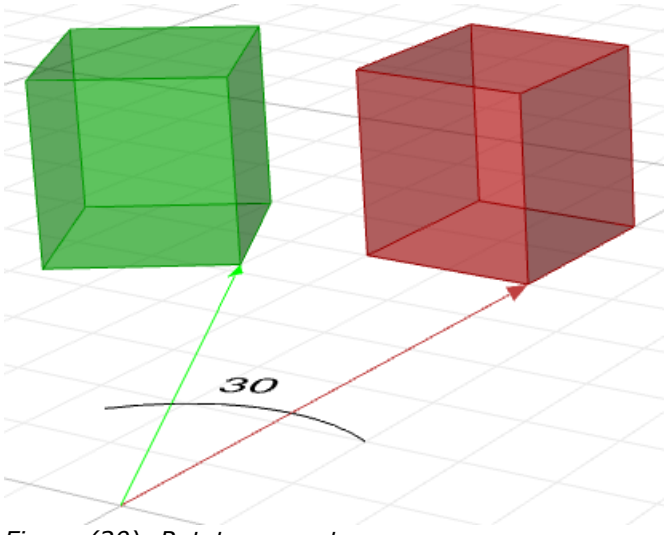


Figure (20): Rotate geometry.

Scale transformation

In order to scale geometry, we need a scale factor and a center of scale. The scale factor can be uniform scaling equally in x-, y-, and z-directions, or can be unique for each dimension. Scaling a point can use the following equation:

P' = ScaleFactor(S) * P

Or:

$P'.x = S_x * P.x$

$P'.y = S_y * P.y$

$P'.z = S_z * P.z$

This is the matrix format for scale transformation, assuming that the center of scale is the World origin point (0,0,0).

$$\begin{bmatrix} Scale\text{-}x & 0 & 0 & 0 \\ 0 & Scale\text{-}y & 0 & 0 \\ 0 & 0 & Scale\text{-}z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

For example, if we would like to scale a box by 0.25 relative to the World origin, the scale matrix will look like the following:

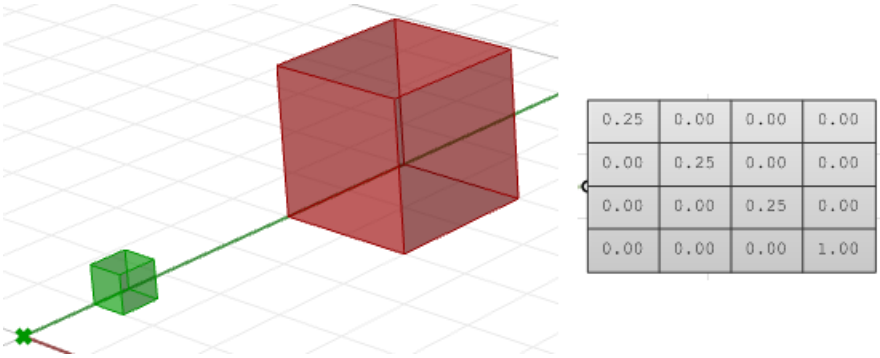


Figure (21): Scale geometry

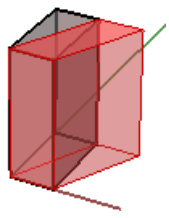
Shear transformation

Shear in 3-D is measured along a pair of axes relative to a third axis. For example, a shear along a z-axis will not change geometry along that axis, but will alter it along x and y. Here are few examples of shear matrices:

1. Shear in x and z, keeping the y-coordinate fixed:

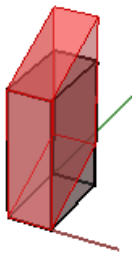
Shear x-axis

$$\begin{bmatrix} 1.0 & 0.5 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



Shear z-axis

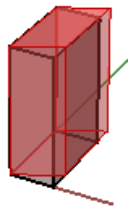
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



2. Shear in y and z, keeping the x-coordinate fixed:

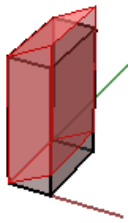
Shear y-axis

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.5 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



Shear z-axis

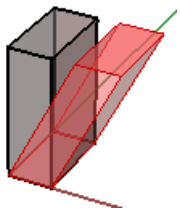
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.5 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



3. Shear in x and y, keeping the z-coordinate fixed:

Shear x-axis

$$\begin{bmatrix} 1.0 & 0.0 & 0.5 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



Shear y-axis

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.5 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

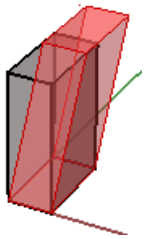
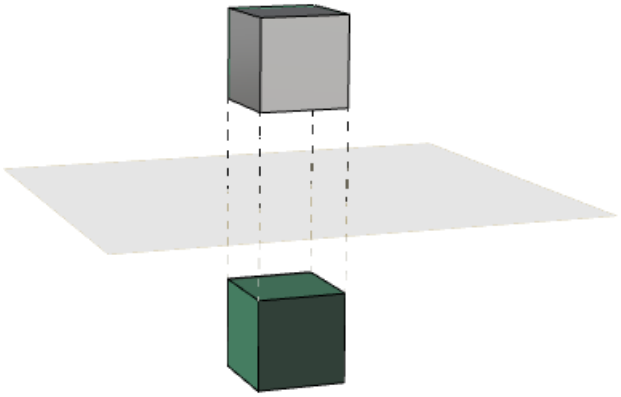


Figure (22): Shear Matrices.

Mirror or reflection transformation

The mirror transformation creates a reflection of an object across a line or a plane. 2-D objects are mirrored across a line, while 3-D objects are mirrored across a plane. Keep in mind that the mirror transformation flips the normal direction of the geometry faces.



1.000	0.000	0.000	0.000
0.000	1.000	0.000	0.000
0.000	0.000	-1.000	0.000
0.000	0.000	0.000	1.000

Figure (23): Mirror matrix across World xy-plane. Face directions are flipped.

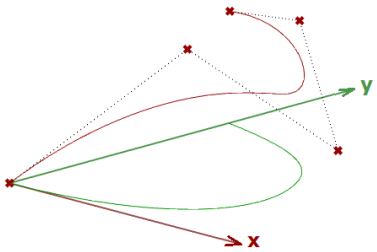
Planar Projection transformation

Intuitively, the projection point of a given 3-D point P(x,y,z) on the world xy-plane equals $P_{xy}(x,y,0)$ setting the z value to zero. Similarly, a projection to xz-plane of point P is $P_{xz}(x,0,z)$. When projecting to yz-plane, P_{xz} = (0,y,z). Those are called orthogonal projections[i].

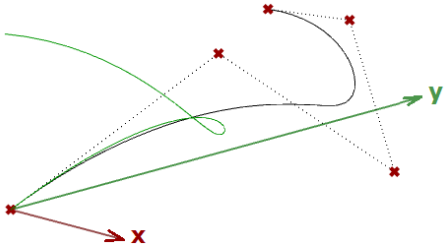
If we have a curve as an input, and we apply the planar projection transformation, we get a curve projected to that plane. The following shows an example of a curve projected to xy-plane with the matrix format.

Note: NURBS curves (explained in the next chapter) use control points to define curves. Projecting a curve amounts to projecting its control points.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



$$\begin{bmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

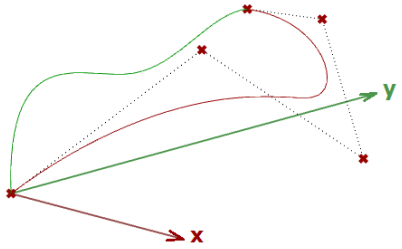


Figure (24): Projection matrices.

3 Parametric Curves and Surfaces

Suppose you travel every weekday from your house to your work. You leave at 8:00 in the morning and arrive at 9:00. At each point in time between 8:00 and 9:00, you would be at some location along the way. If you plot your location every minute during your trip, you can define the path between home and work by connecting the 60 points you plotted. Assuming you travel the exact same speed every day, at 8:00 you would be at home (start location), at 9:00 you would be at work (end location) and at 8:40 you would at the exact same location on the path as the 40th plot point. Congratulations, you have just defined your first parametric curve! You have used *time* as a *parameter* to define your path, and hence you can call your path curve a *parametric curve*. The time interval you spend from start to end (8 to 9) is called the *curve domain* or *interval*.

In general, we can describe the x, y, and z location of a parametric curve in terms of some parameter *t* as follows:

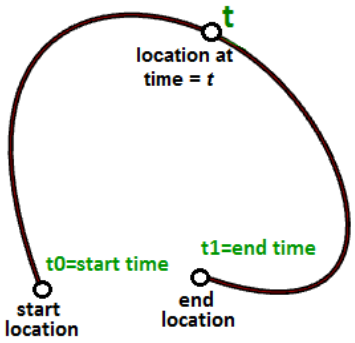
$x = x(t)$

$y = y(t)$

$z = z(t)$

Where:

t is a range of real numbers



We saw earlier that the parametric equation of a line in terms of parameter *t* is defined as:

$x = x' + t * a$
$y = y' + t * b$
$z = z' + t * c$

Where:

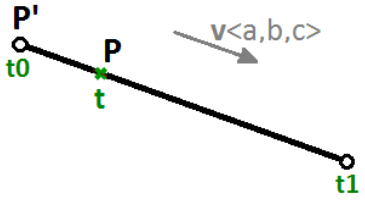
x, y, and z are functions of *t* where *t* represents a range of real numbers.

X′, y′, and z′ are the coordinates of a point on the line segment.

a, b, and c define the slope of the line, such that the vector **v**<a, b, c> is parallel to the line.

We can therefore write the parametric equation of a line segment using a *t* parameter that ranges between two real number values *t0*, *t1* and a unit vector **v** that is in the direction of the line as follows:

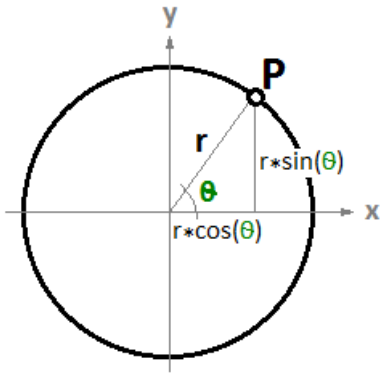
$P = P' + t * \mathbf{v}$



Another example is a circle. The parametric equation of the circle on the xy-plane with a center at the origin (0,0) and an angle parameter *t* ranging between 0 and 2π radians is:

$x = r \cos(t)$

$y = r \sin(t)$



We can derive the general equation of a circle for the parametric one as follows:

$x/r = \cos(t)$

$y/r = \sin(t)$

And since:

$\cos(t)^2 + \sin(t)^2 = 1$ (Pythagorean identity)

Then:

$(x/r)^2 + (y/r)^2 = 1$, or

$x^2 + y^2 = r^2$

Parametric curves

Curve parameter

A parameter on a curve represents the address of a point on that curve. As mentioned before, you can think of a parametric curve as a path traveled between two points in a certain amount of time, traveling at a fixed or variable speed. If traveling takes T amount of time, then the parameter *t* represents a time within T that translates to a location (point) on the curve.

If you have a straight path (line segment) between the two points A and B, and **v** were a vector from A to B (**v** = B - A), then you can use the parametric line equation to find all points M between A and B as follows:

$M = A + t*(B-A)$

Where:

t is a value between 0 and 1.

The range of *t* values, 0 to 1 in this case, is referred to as the *curve domain* or *interval*. If *t* was a value outside the domain (less that 0 or more than 1), then the resulting point M will be outside the linear curve AB.

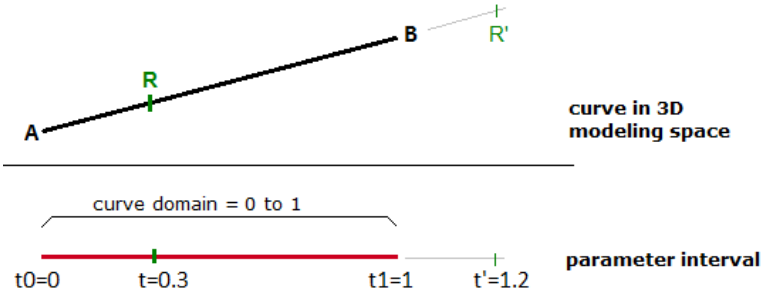


Figure (25): Parametric linear curve in 3-D space and parameter interval.

The same principle applies for any parametric curve. Any point on the curve can be calculated using the parameter *t* within the interval or domain of values that define the limits of the curve. The start parameter of the domain is usually referred to as t0 and the end of the domain as t1.

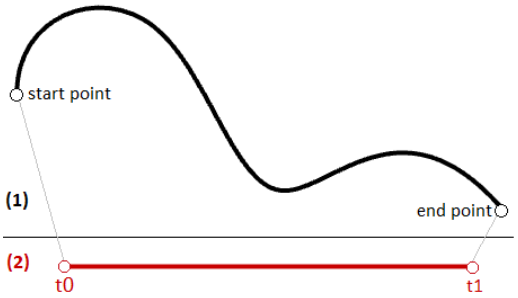


Figure (26): Curve in 3-D space (1). Curve domain (2).

Curve domain or interval

A curve *domain* or *interval* is defined as the range of parameters that evaluate into a point within that curve. The domain is usually described with two real numbers defining the domain limits expressed in the form (min to max) or (min, max). The domain limits can be any two values that may or may not be related to the actual length of the curve. In an increasing domain, the domain min parameter evaluates to the start point of the curve and the domain max evaluates to the end point of the curve.

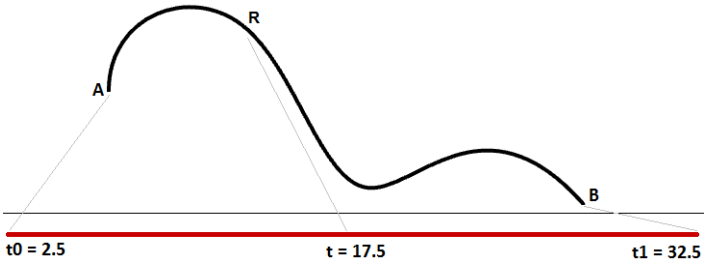


Figure (27): Curve domain or interval can be between any two numbers.

Changing a curve domain is referred to as the process of *reparameterizing* the curve. For example, it is very common to change the domain to be (0 to 1). Reparameterizing a curve does not affect the shape of the 3-D curve. It is like changing the travel time on a path by running instead of walking, which does not change the shape of the path.

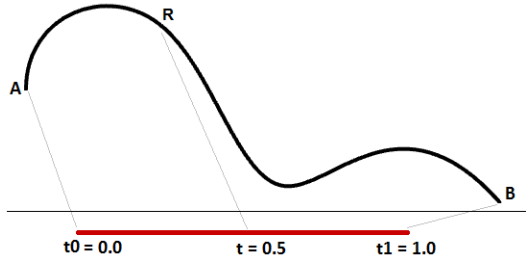


Figure (28): Normalized curve domain to be 0 to 1.

An increasing domain means that the minimum value of the domain points to the start of the curve. Domains are typically increasing, but not always.

Curve evaluation

We learned that a curve interval is the range of all parameter values that evaluate to points within the 3-D curve. There is, however, no guarantee that evaluating at the middle of the domain, for example, will give a point that is in the middle of the curve, as shown in Figure (25).

We can think of uniform parameterization of a curve as traveling a path with constant speed. A degree-1 line between two points is one example where equal intervals or parameters translate into equal intervals of arc length on the line. This is a special case where equal intervals of parameters evaluate to equal intervals on the 3-D curve.

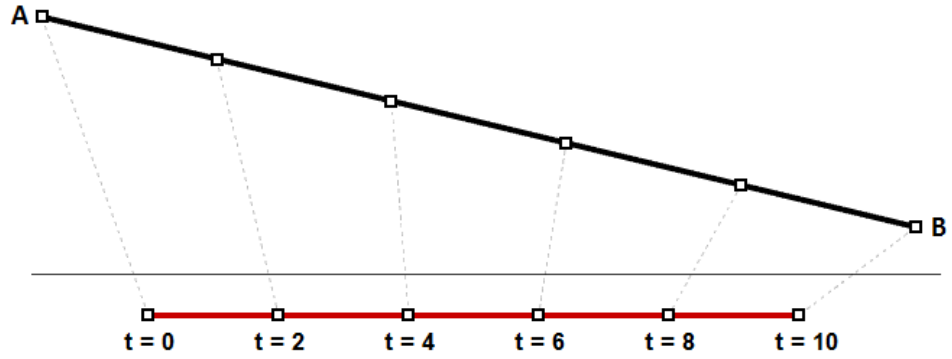


Figure (29): Equal parameter intervals in a degree-1 line evaluate to equal curve lengths.

It is, however, more likely that the speed decreases or increases along the path. For example, if it takes 30 minutes to travel a road, it is unlikely that you will be exactly half way through at minute 15. Figure (27) shows a typical case where equal parameter intervals evaluate to variable length on the 3-D curve.

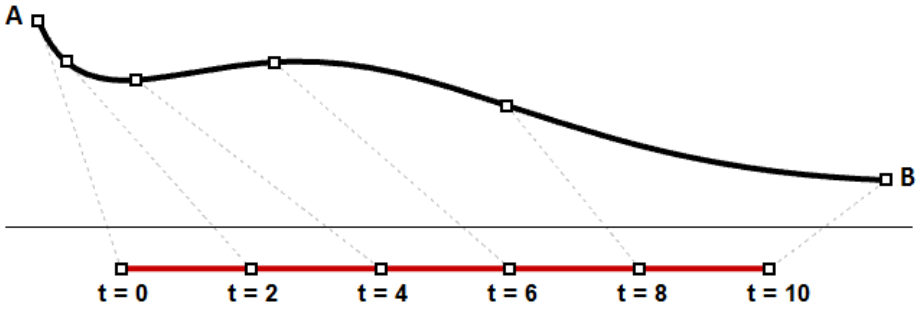


Figure (30): Equal parameter intervals do not usually translate into equal distances on a curve.

You may need to evaluate points on a 3-D curve that are at a defined percentage of the curve length. For example, you might need to divide the curve into equal lengths. Typically, 3-D modelers provide tools to evaluate curves relative to arc length.

Tangent vector to a curve

A tangent to a curve at any parameter (or point on a curve) is the vector that touches the curve at that point, but does not cross over. The slope of the tangent vector equals the slope of the curve at the same point. The following example evaluates the tangent to a curve at two different parameters.

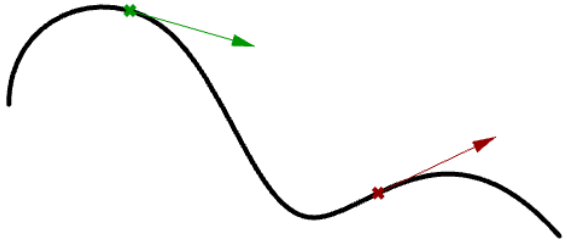


Figure (31): Tangents to a curve.

Cubic polynomial curves

Hermite[ii] and Bézier[iii] curves are two examples of cubic polynomial curves that are determined by four parameters. A Hermite curve is determined by two end points and two tangent vectors at these points, while a Bézier curve is defined by four points. While they differ mathematically, they share similar characteristics and limitations.

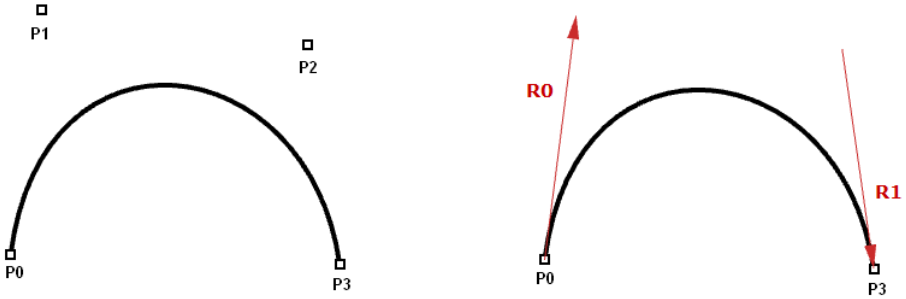


Figure (32): Cubic polynomial curves. The Bézier curve (left) is defined by four points. The Hermite curve (right) is defined by two points and two tangents.

In most cases, curves are made out of multiple segments. This requires making what is called a *piecewise cubic curve*. Here is an illustration of a piecewise Bézier curve that uses 7 storage points to create a two-segment cubic curve. Note that although the final curve is joined, it does not look smooth or continuous.

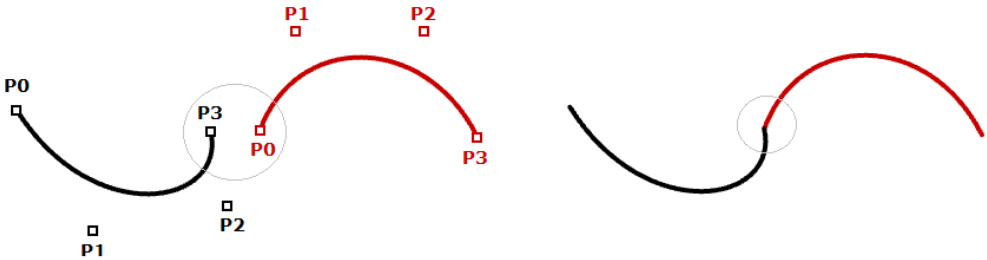


Figure (33): Two Bezier spans share one point.

Although Hermite curves use the same number of parameters as Bézier curves (four parameters to define one curve), they offer the additional information of the tangent curve that can also be shared with the next piece to create a smoother looking curve with less total storage, as shown in the following.

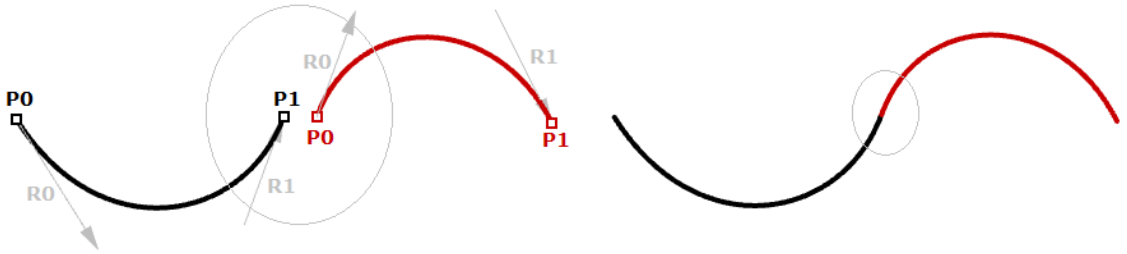


Figure (34): Two Hermite spans share one point and a tangent.

The non-uniform rational B-spline[iv] (NURBS) is a powerful curve representation that maintains even smoother and more continuous curves. Segments share more control points to achieve even smoother curves with less storage.

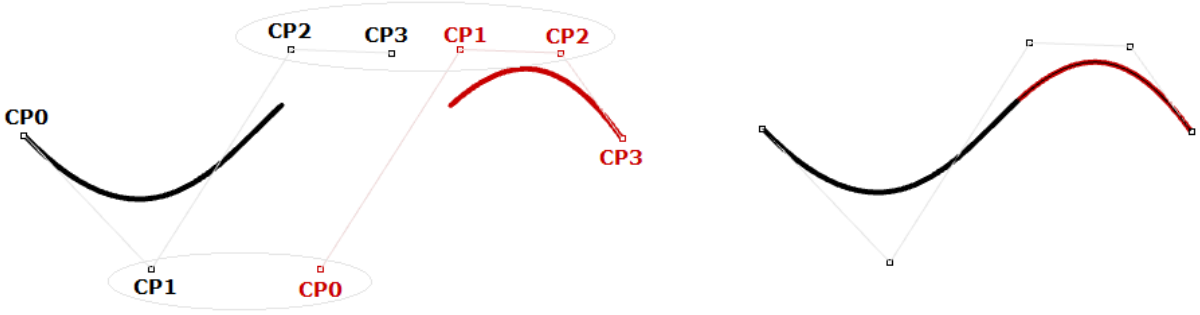


Figure (35): Two degree-3 NURBS spans share three control points.

NURBS curves and surfaces are the main mathematical representation used by Rhino to represent geometry. NURBS curve characteristics and components will be covered with some detail later in this chapter.

Evaluating cubic Bézier curves

Named after its inventor, Paul de Casteljau, the de Casteljau algorithm[v] evaluates Bézier curves using a recursive method. The algorithm steps can be summarized as follows:

Input:

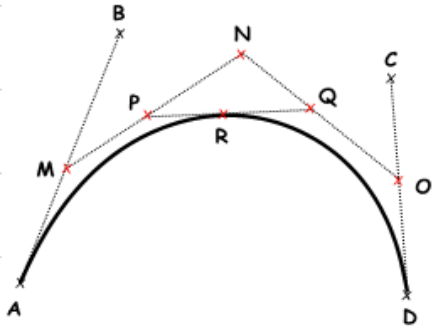
Four points A, B, C, D define a curve *t*, is any parameter within curve domain

Output:

Point R on curve that is at parameter *t*.

Solution:

1. Find point M at *t* parameter on line AB.
2. Find point N at *t* parameter on line BC.
3. Find point O at *t* parameter on line CD.
4. Find point P at *t* parameter on line MN.
5. Find point Q at *t* parameter on line NO.
6. Find point R at *t* parameter on line PQ.



NURBS curves

NURBS is an accurate mathematical representation of curves that is highly intuitive to edit. It is easy to represent free-form curves using NURBS and the control structure makes it easy and predictable to edit.

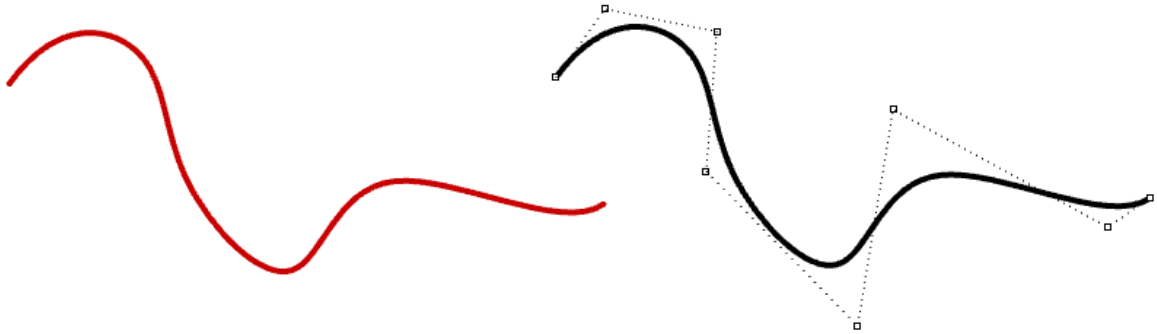


Figure (36): Non-uniform rational B-splines and their control structure.

There are many books and references for those of you interested in an in-depth reading about NURBS[vi]. A basic understanding of NURBS is however necessary to help use a NURBS modeler more effectively. There are four main attributes define the NURBS curve: *degree*, *control points*, *knots*, and *evaluation rules*.

Degree

Curve degree is a whole positive number. Rhino allows working with any degree curve starting with 1. Degrees 1, 2, 3, and 5 are the most useful but the degrees 4 and those above 5 are not used much in the real world. Following are a few examples of curves and their degree:

Lines and **polylines** are degree-1 NURBS curves.

Circles and **ellipses** are examples of degree-2 NURBS curves.

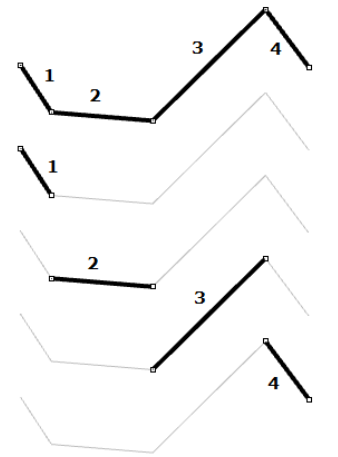
Free-form **curves** are usually represented as degree-3 or 5 NURBS curves.

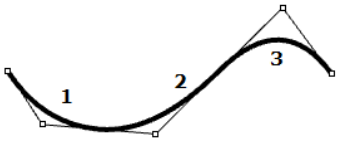
Control points

The control points of a NURBS curve is a list of at least (degree+1) points. The most intuitive way to change the shape of a NURBS curve is through moving its control points.

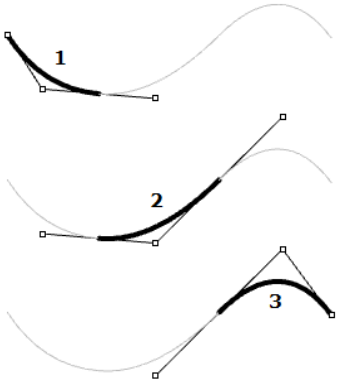
The number of control points that affect each span in a NURBS curve is defined by the degree of the curve. For example, each span in a degree-1 curve is affected only by the two end control points. In a degree-2 curve, three control points affect each span and so on.

Control points of degree-1 curves go through all curve control points. In a degree-1 NURBS curve, two (degree+1) control points define each span. Using five control points, the curve has four spans.

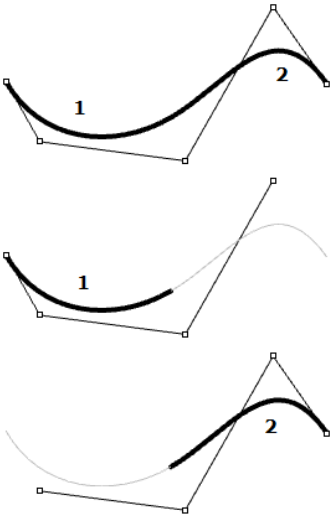




Circles and ellipses are examples of degree two curves. In a degree-2 NURBS curve, three (degree+1) control points define each span. Using five control points, the curve has three spans.



Control points of degree-3 curves do not usually touch the curve, except at end points in open curves. In a degree-3 NURBS curve, four (degree+1) control points define each span. Using five control points, the curve has two spans



Weights of control points

Each control point has an associated number called *weight*. With a few exceptions, weights are positive numbers. When all control points have the same weight, usually 1, the curve is called non-rational. Intuitively, you can think of weights as the amount of gravity each control point has. The higher the relative weight a control point has, the closer the curve is pulled towards that control point.

It is worth noting that it is best to avoid changing curve weights. Changing weights rarely gives desired result while it introduces a lot of calculation challenges in operations such as intersections. The only good reason for using rational curves is to represent curves that cannot otherwise be drawn, such as circles and ellipses.

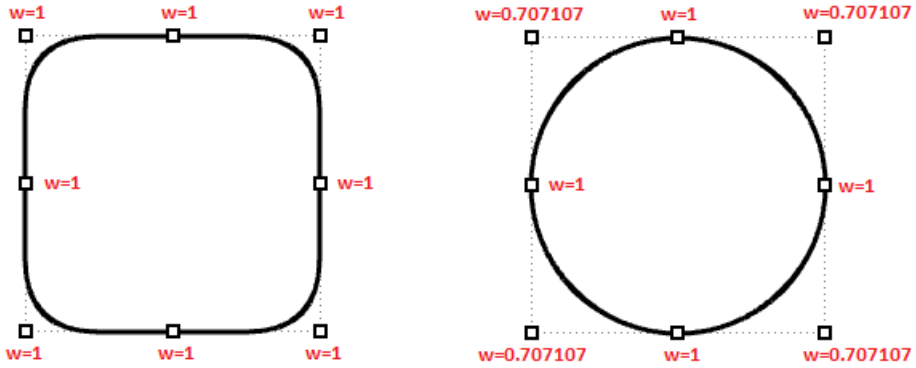


Figure (37): The effect of varying weights of control points on the result curve.
The left curve is non-rational with uniform control point weights.
The circle on the right is a rational curve with corner control points having weights less than 1.

Knots

Each NURBS curve has a list of numbers associated with it called a *list of knots* (sometimes referred to as *knot vector*). Knots are a little harder to understand and set. While using a 3-D modeler, you will not need to manually set the knots for each curve you create; a few things will be useful to learn about knots.

Knots are parameter values

Knots are a non-decreasing list of parameter values. There is degree+1 more knots than control points. Usually, for non-periodic curves, the first degree many knots are the same and the last degree many are the same. The domain of the curve is between these two extreme knot values at start and end.

For example, the knots of an open degree-3 NURBS curve with seven control points and a domain between 0 and 6 may look like the following:

Knots= <0, 0, 0, 1, 2, 3, 4, 5, 6, 6, 6>

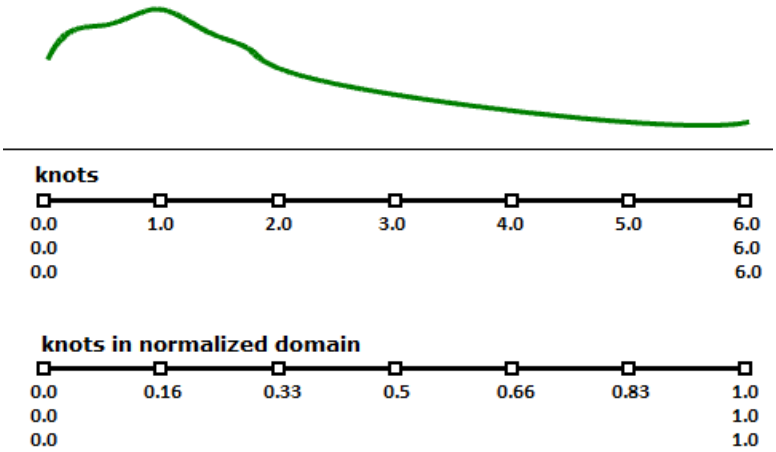


Figure (38): Knots are calculated within the curve domain.
If the curve domain is between 0 and 6, the list of knots will have values between 0 and 6.
If the domain is normalized to 0 to 1, the list of knots will range between 0 and 1 as well.

Knot multiplicity

The multiplicity of a knot is the number of times it is listed in the list of knots. The multiplicity of a knot cannot be more than the degree of the curve. Knot multiplicity is used to control continuity at the corresponding curve point.

Fully-multiple knots

A fully multiple knot has multiplicity equal to the curve degree. At a fully multiple knot there is a corresponding control point, and the curve goes through this point.

For example, clamped or open curves have knots with full multiplicity at the ends of the curve. This is why the end control points coincide with the curve end points. Interior fully multiple knots allow a kink in the curve at the corresponding point.

Simple knot

A simple knot is one with value appearing only once. Interior knots are typically simple.

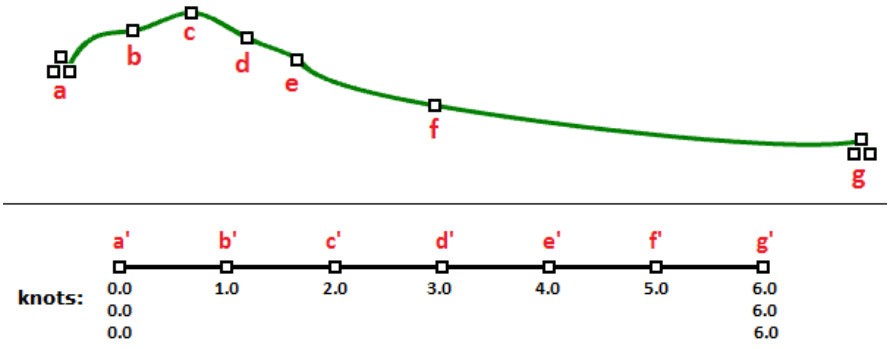


Figure (39): Clamped curves have fully-multiple knots at their start and end. The rest of the knots are simple.

Uniform knots

A uniform list of knots satisfies the following condition:

Knots start with a fully-multiple knot, are followed by simple knots, and terminate with a fully-multiple knot. The values are increasing and equally spaced. This is typical of clamped or open curves. Periodic curves work differently as we will see later.

Here are two curves that have the same number and location of control points, and yet have different knots and curve shape:

Degree = 3
Number of control points = 7
knots = <0,0,0,1,2,3,4,4,4> = 9 knots
Domain (0 to 4)

Degree = 3
Number of control points = 7
knots = <0,0,0,1,1,1,4,4,4> = 9 knots
Domain (0 to 4)
Note: A fully multiple knot in the middle creates a kink and the curve is forced to go through the associated control point.

Evaluation rule

The evaluation rule uses a mathematical formula that takes a number within the curve domain and assigns a point. The formula takes into account the degree, control points, and knots.

Using this formula, specialized curve functions can take a curve parameter and produce the corresponding point on that curve. A parameter is a number that lies within the curve domain. Domains are usually increasing and consist of two numbers: the minimum domain parameter t(0) that evaluates to the start point of the curve and the maximum t(1) that evaluates to the end point of the curve.

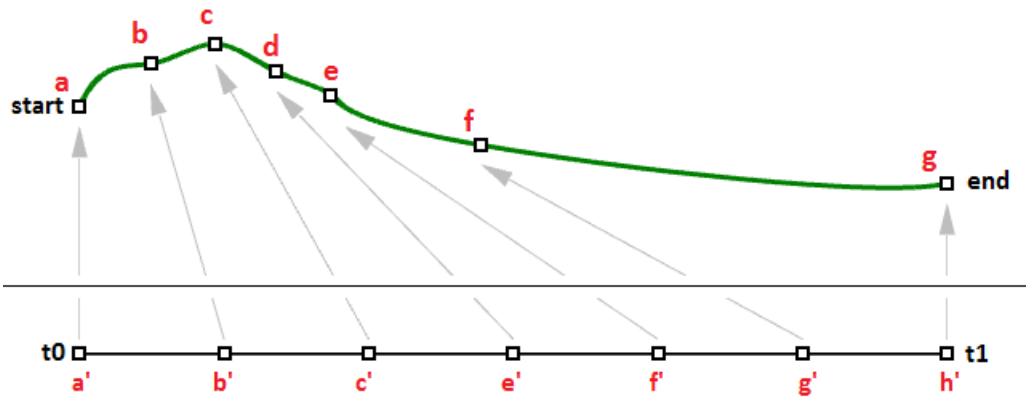


Figure (40): Evaluate parameters to points on curve.

Characteristics of NURBS curves

In order to create a NURBS curve, you will need to provide the following information:

- Dimension, typically 3

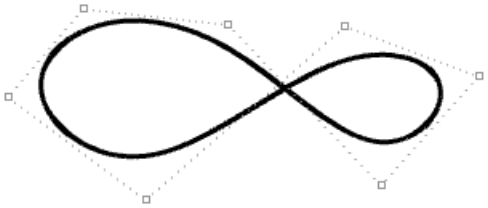
- Degree, (sometimes use the *order*, which is equal to degree+1)
- Control points (list of points)
- Weight of the control point (list of numbers)
- Knots (list of numbers)

When you create a curve, you need to at least define the degree and locations of the control points. The rest of the information necessary to construct NURBS curves can be generated automatically. Selecting an end point to coincide with the start point would typically create a periodic smooth closed curve. The following table shows examples of open and closed curves:

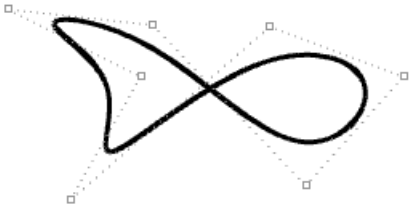
Degree-1 open curve.
The curve goes through all control points.

Degree-3 open curve.
Both curve ends coincide with end control points.

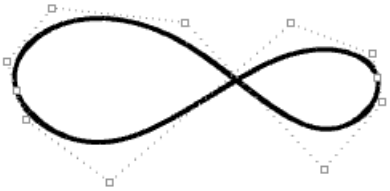
Degree-3 closed periodic curve.
The curve seam does not go through a control point.



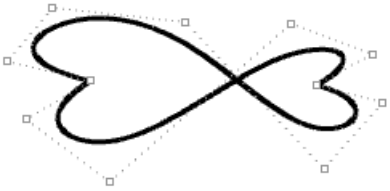
Moving control points of a periodic curve does not affect curve smoothness.



Kinks are created when the curve is forced through some control points.



Moving the control points of a non-periodic curve does not guarantee smooth continuity of the curve, but enables more control over the outcome.



Clamped vs. periodic NURBS curves

The end points of closed clamped curves coincide with end control points. Periodic curves are smooth closed curves. The best way to understand the differences between the two is through comparing control points and knots.

The following is an example of an open, clamped non-rational NURBS curve. This curve has four control points, uniform knots with full-multiplicity at the start and end knots and the weights of all control points equal to 1.

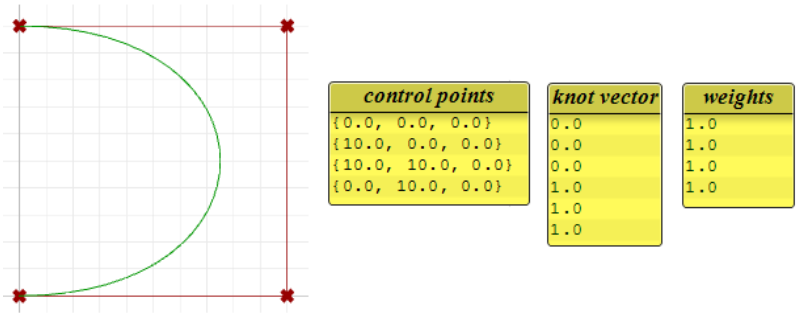


Figure (41): Analyze degree-3 open non-rational NURBS curve.

The following circular curve is an example of a degree-3 closed periodic NURBS curve. It is also non-rational because all weights are equal. Note that periodic curves require more control points with few overlapping. Also the knots are simple.

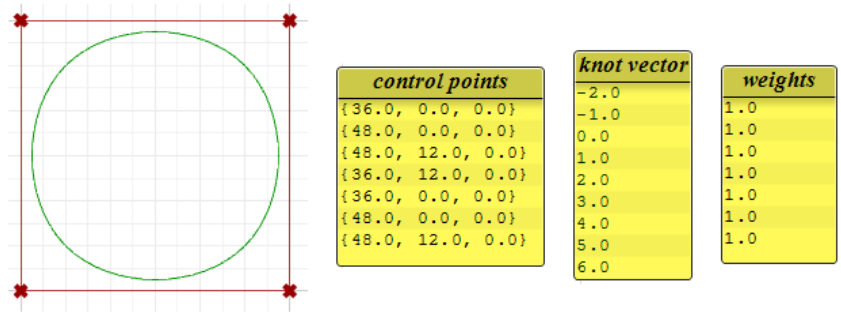


Figure (42): Analyze degree-3 closed (periodic) NURBS curve.

Notice that the periodic curve turned the four input points into seven control points (degree+4), while the clamped curve used only the four control points. The knots of the periodic curve uses only simple knots, while the clamped curve start and end knots have full multiplicity forcing the curve to go through the start and end control points.

If we set the degree of the previous examples to 2 instead of 3, the knots become smaller, and the number of control points of periodic curves changes.

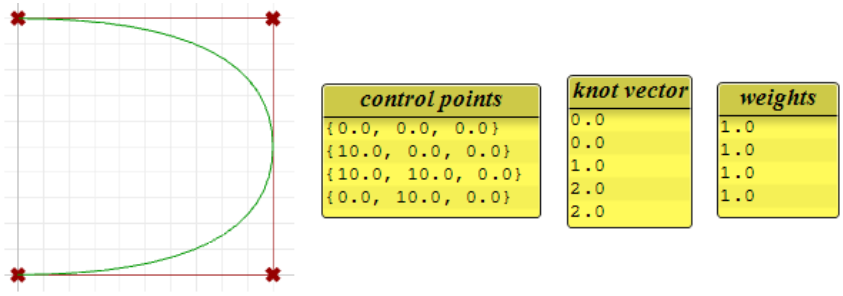
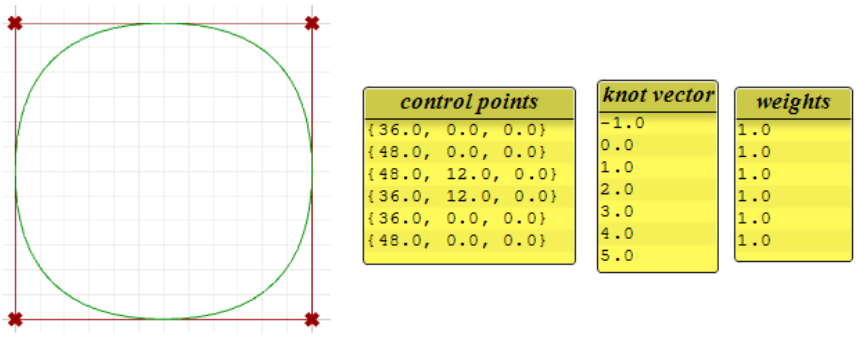


Figure (43): Analyze degree-2 open NURBS curve.



control points
{36.0, 0.0, 0.0}
{48.0, 0.0, 0.0}
{48.0, 12.0, 0.0}
{36.0, 12.0, 0.0}
{36.0, 0.0, 0.0}
{48.0, 0.0, 0.0}

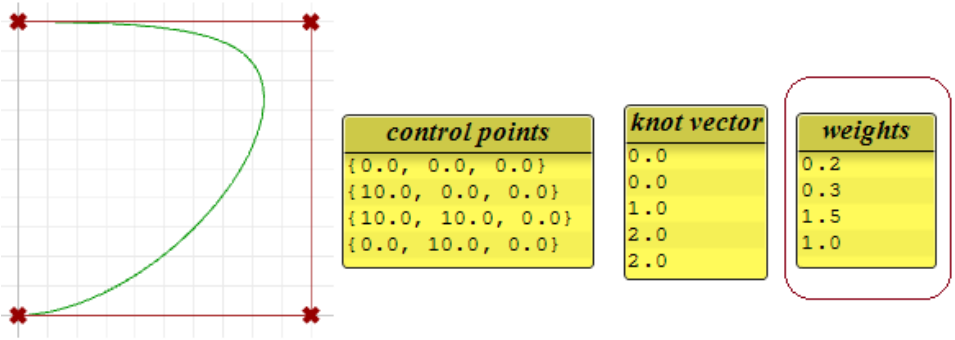
knot vector
-1.0
0.0
1.0
2.0
3.0
4.0
5.0

weights
1.0
1.0
1.0
1.0
1.0
1.0

Figure (44): Analyze degree-2 closed (periodic) NURBS curve.

Weights

Weights of control points in a uniform NURBS curve are set to 1, but this number can vary in rational NURBS curves. The following example shows the effect of varying the weights of control points.

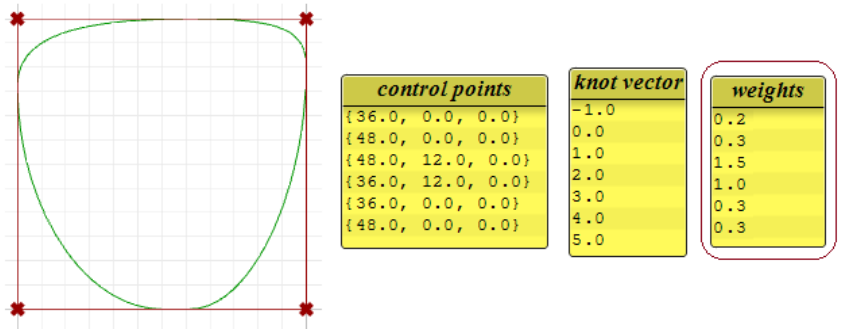


control points
{0.0, 0.0, 0.0}
{10.0, 0.0, 0.0}
{10.0, 10.0, 0.0}
{0.0, 10.0, 0.0}

knot vector
0.0
0.0
1.0
2.0
2.0

weights
0.2
0.3
1.5
1.0

Figure (45): Analyze weights in open NURBS curve.



control points
{36.0, 0.0, 0.0}
{48.0, 0.0, 0.0}
{48.0, 12.0, 0.0}
{36.0, 12.0, 0.0}
{36.0, 0.0, 0.0}
{48.0, 0.0, 0.0}

knot vector
-1.0
0.0
1.0
2.0
3.0
4.0
5.0

weights
0.2
0.3
1.5
1.0
0.3
0.3

Figure (46): Analyze weights in closed NURBS curve.

Evaluating NURBS curves

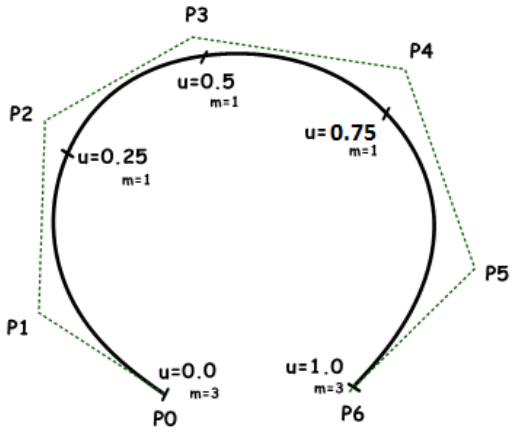
Named after its inventor, Carl de Boor, the de Boor's algorithm[vii] is a generalization of the de Casteljau algorithm for Bézier curves. It is numerically stable and is widely used to evaluate points on NURBS curves in 3-D applications. The following is an example for evaluating a point on a degree-3 NURBS curve using de Boor's algorithm.[viii]

Input:

Seven control points P_0 to P_6

Knots:

u0 = 0.0
u1 = 0.0
u2 = 0.0
u3= 0.25
u4 = 0.5
u5 = 0.75
u6 = 1.0
u7 = 1.0
u8 = 1.0



Output:

Point on curve that is at u=0.4

Solution:

1. Calculate coefficients for the first iteration:

   $Ac = (u - u_1) / (u_{1+3} - u_1) = 0.8$

   $Bc = (u - u_2) / (u_{2+3} - u_2) = 0.53$

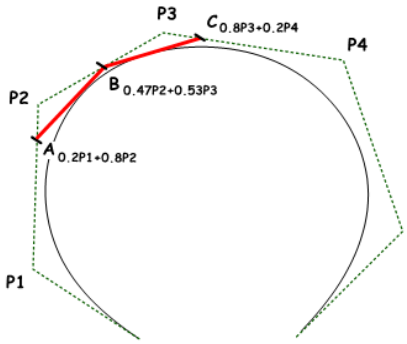
   $Cc = (u - u_3) / (u_{3+3} - u_3) = 0.2$

2. Calculate points using coefficient data:

   $\mathbf{A} = 0.2\mathbf{P_1} + 0.8\mathbf{P_2}$

   $\mathbf{B} = 0.47\ \mathbf{P_2} + 0.53\ \mathbf{P_3}$

   $\mathbf{C} = 0.8\ \mathbf{P_3} + 0.2\ \mathbf{P_4}$



3. Calculate coefficients for the second iteration:

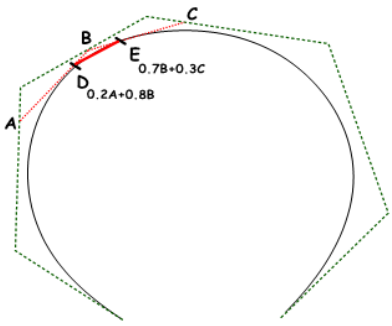
   $Dc = (u - u_2) / (u_{2+3-1} - u_2) = 0.8$

   $Ec = (u - u_3) / (u_{3+3-1} - u_3) = 0.3$

4. Calculate points using coefficient data:

   $\mathbf{D} = 0.2\mathbf{A} + 0.8\mathbf{B}$

   $\mathbf{E} = 0.7\mathbf{B} + 0.3\mathbf{C}$

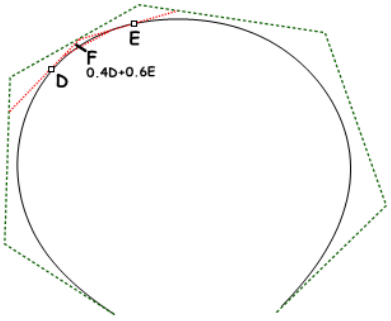


5. Calculate the last coefficient:

   $Fc = (u - u_3)/ (u_{3+3-2} - u_3) = 0.6$

   Find the point on curve at u=0.4 parameter:

   $F = 0.4\mathbf{D} + 0.6\mathbf{E}$



Curve geometric continuity

Continuity is an important concept in 3-D modeling. Continuity is important for achieving visual smoothness and for obtaining smooth light and airflow.

The following table shows various continuities and their definitions:

G0 (Position continuous)	Two curve segments joined together
G1 (Tangent continuous)	Direction of tangent at joint point is the same for both curve segments
G2 (Curvature Continuous)	Curvatures as well as tangents agree for both curve segments at the common endpoint
GN	The curves agree to higher order

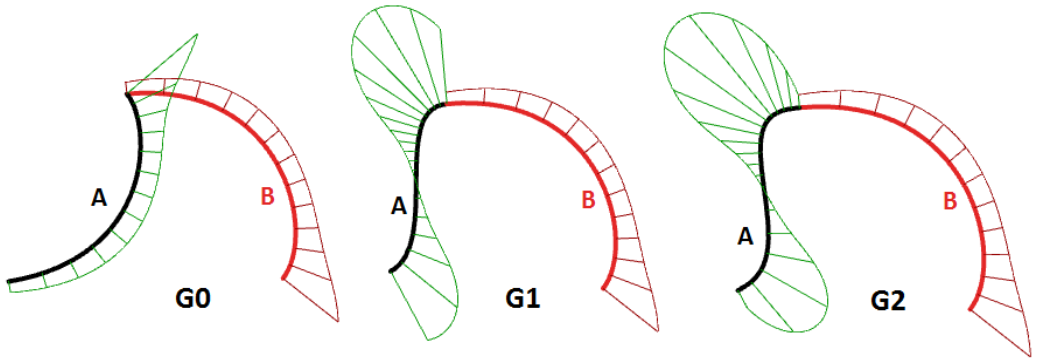


Figure (47): Examining curve continuity with curvature graph analysis.

Curve curvature

Curvature is a widely used concept in modeling 3-D curves and surfaces. Curvature is defined as *the change in inclination of a tangent to a curve over unit length of arc*. For a circle or sphere, it is the reciprocal of the radius and it is constant across the full domain.

At any point on a curve in the plane, the line best approximating the curve that passes through this point is the tangent line. We can also find the best approximating circle that passes through this point and is tangent to the curve. The reciprocal of the radius of this circle is the curvature of the curve at this point.

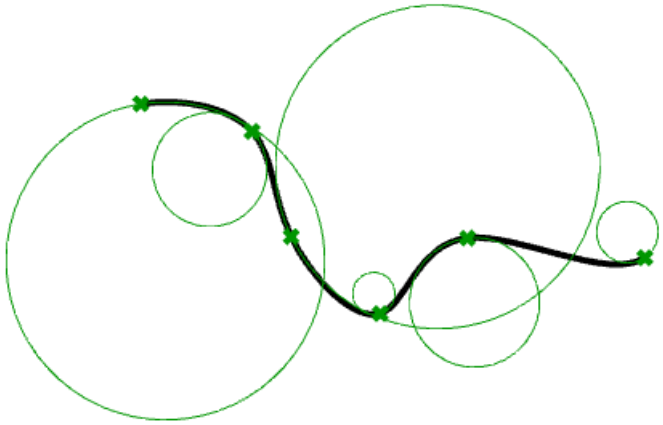


Figure (48): Examining curve curvature at different points.

The best approximating circle can lie either to the left or to the right of the curve. If we care about this, we establish a convention, such as giving the curvature positive sign if the circle lies to the left and negative sign if the circle lies to the right of the curve. This is known as signed curvature. Curvature values of joined curves indicate continuity between these curves.

Parametric surfaces

Surface parameters

A parametric surface is a function of two independent parameters (usually denoted *u, v)* over some two-dimensional domain. Take for example a plane. If we have a point P on the plane and two nonparallel vectors on the plane, **a** and **b**, then we can define a parametric equation of the plane in terms of the two parameters *u* and *v* as follows:

P = P′ + *u* * **a** + *v* * **b**

Where:

P′ is a known point on the plane

a is the first vector on the plane

b is the first vector on the plane

u is the first parameter

v is the first parameter

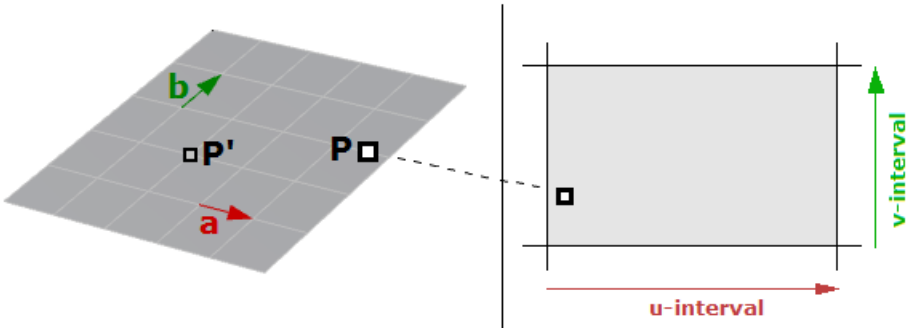


Figure (49): The parameter rectangle of a plane.

Another example is the sphere. The Cartesian equation of a sphere centered at the origin with radius R is

$x^2 + y^2 + z^2 = R^2$

That means for each point, there are three variables (x, y, z), which is not useful to use for a parametric representation that requires only two variables. However, in the spherical coordinate system, each point is found using the three values:

r: radial distance between the point and the origin

θ: the angle from the x-axis in the xy-plane

ø: the angle from the z-axis and the point

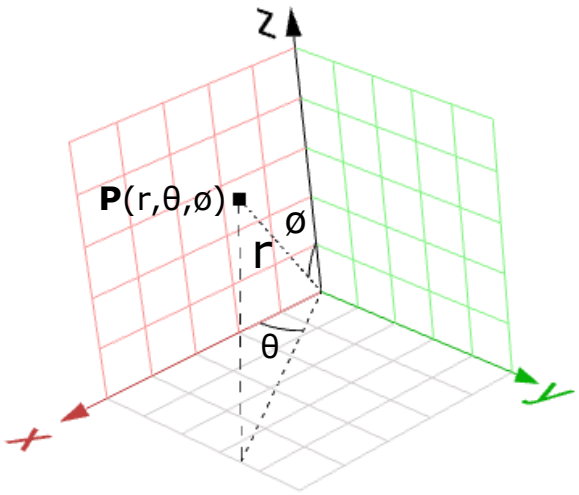


Figure (50): Spherical coordinate system.

A conversion of points from spherical to Cartesian coordinate can be obtained as follows:

$x = \boldsymbol{r} * \sin(ø) * \cos(θ)$

$y = \boldsymbol{r} * \sin(ø) * \sin(θ)$

$z = \boldsymbol{r} * \cos(ø)$

Where:

r is distance from origin ≥ 0

θ is running from 0 to 2π

ø is running from 0 to π

Since **r** is constant in a sphere surface, we are left with only two variables, and hence we can use the above to create a parametric representation of a sphere surface:

$u = θ$

$v = ø$

So that:

$x = \mathbf{r} * \sin(v) * \cos(u)$

$y = \mathbf{r} * \sin(v) * \sin(u)$

$z = \mathbf{r} * \cos(v)$

Where (u, v) is within the domain (2 π, π)

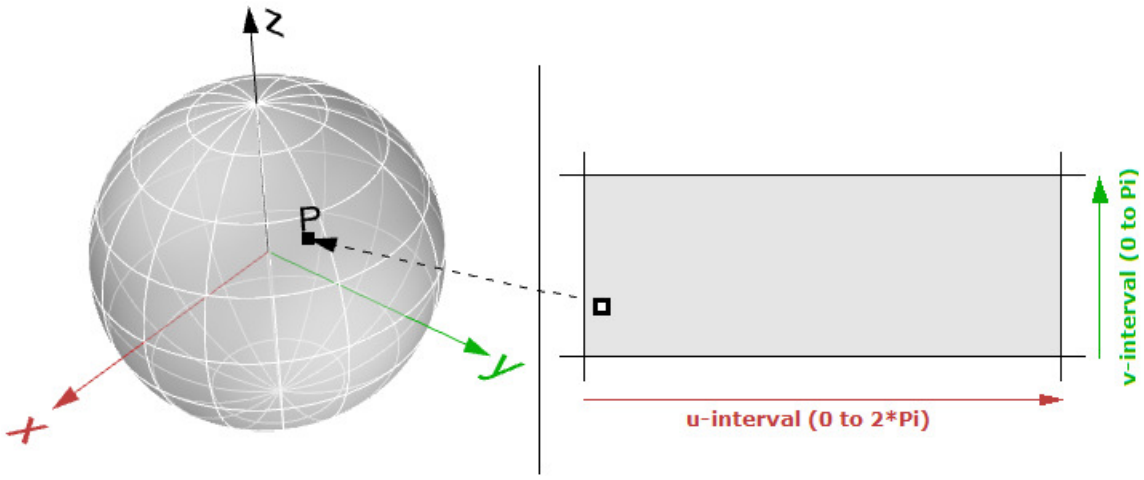


Figure (51): The parameter rectangle of a sphere.

The parametric surface follows the general form:

x = x(u,v)

y = y(u,v)

z = z(u,v)

Where:

u and v are the two parameters within the surface domain or region.

Surface domain

A surface domain is defined as the range of (u,v) parameters that evaluate into a 3-D point on that surface. The domain in each dimension (u or v) is usually described as two real numbers (u_min to u_max) and (v_min to v_max)

Changing a surface domain is referred to as *reparameterizing* the surface.
An increasing domain means that the minimum value of the domain points to the minimum point of the surface. Domains are typically increasing, but not always.

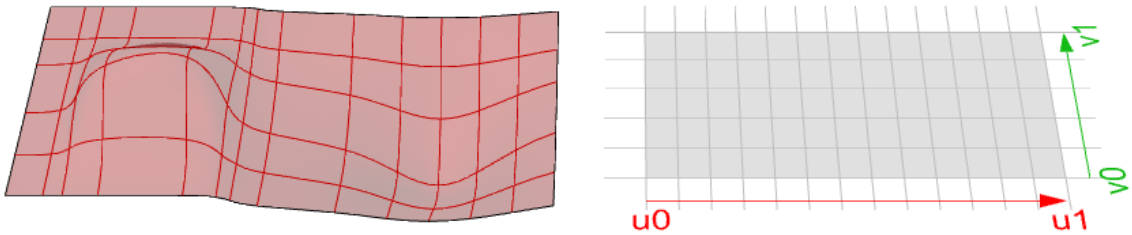


Figure (52): NURBS surface in 3-D modeling space (left). The surface parameter rectangle with domain spanning from u0 to u1 in the first direction and v0 to v1 in the second direction (right).

Surface evaluation

Evaluating a surface at a parameter that is within the surface domain results in a point that is on the surface. Keep in mind that the middle of the domain (mid-u, mid-v) might not necessarily evaluate to the middle point of the 3-D surface. Also, evaluating u- and v-values that are outside the surface domain will not give a useful result.

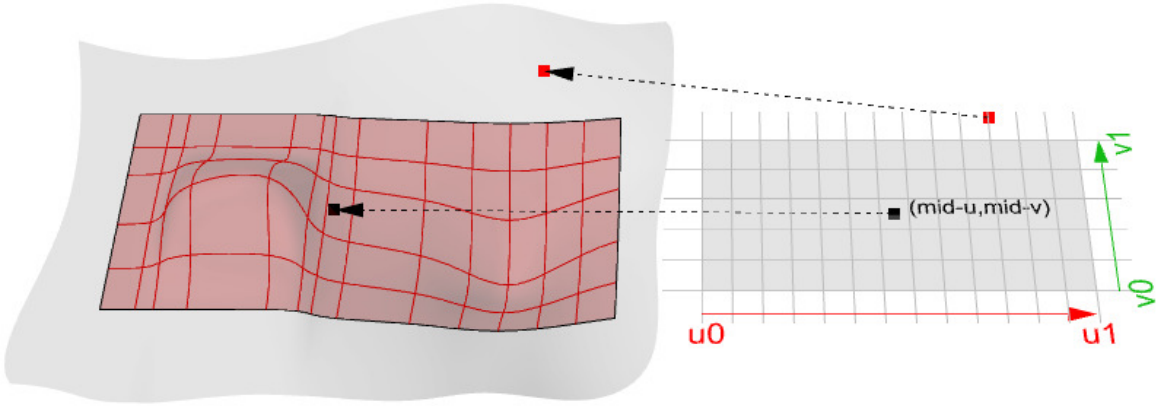


Figure (53): Surface evaluation.

Tangent plane of a surface

The tangent plane to a surface at a given point is the plane that touches the surface at that point. The z-direction of the tangent plane represents the normal direction of the surface at that point.

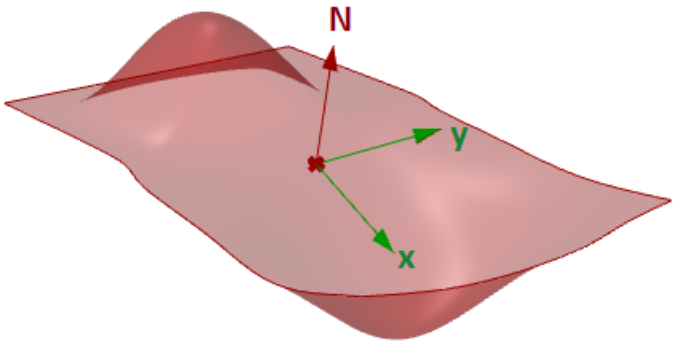


Figure (54): Tangent and normal vectors to a surface.

Surface geometric continuity

Many models cannot be constructed from one surface patch. Continuity between joined surface patches is important for visual smoothness, light reflection, and airflow.

The following table shows various continuities and their definitions:

G0 (Position continuous)	Two surfaces joined together.
G1 (Tangent continuous)	The corresponding tangents of the two surfaces along their joint edge are parallel in both u- and v-directions.
G2 (Curvature continuous)	Curvatures as well as tangents agree for both surfaces at the common edge.
GN	The surfaces agree to higher order.

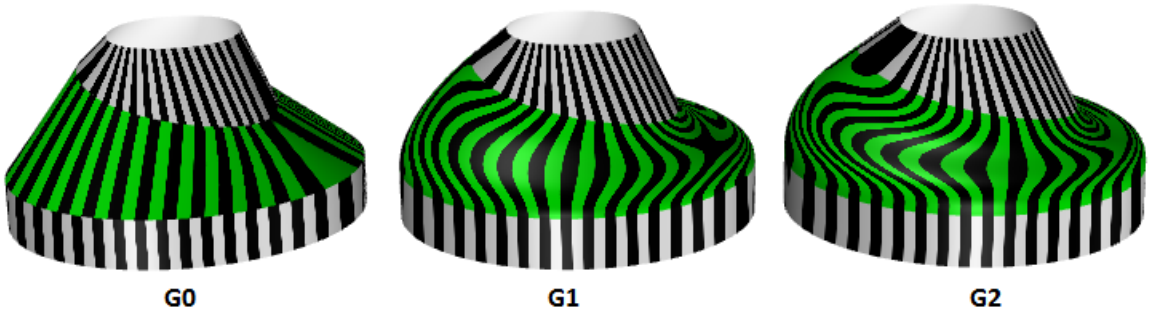


Figure (55): Examining surface continuity with zebra analysis.

Surface curvature

For surfaces, normal curvature is one generalization of curvature to surfaces. Given a point on the surface and a direction lying in the tangent plane of the surface at that point, the normal section curvature is computed by intersecting the surface with the plane spanned by the point, the normal to the surface at that point, and the direction. The normal section curvature is the signed curvature of this curve at the point of interest.

If we look at all directions in the tangent plane to the surface at our point, and we compute the normal curvature in all these directions, there will be a maximum value and a minimum value.

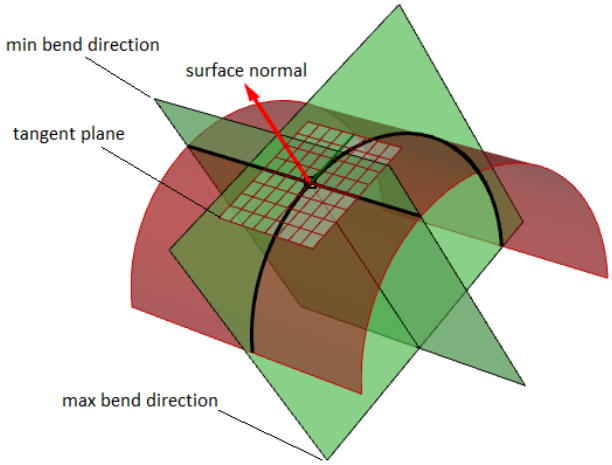


Figure (56): Normal curvatures.

Principal curvatures

The principal curvatures of a surface at a point are the minimum and maximum of the normal curvatures at that point. They measure the maximum and minimum bend amount of the surface at that point. The principal curvatures are used to compute the *Gaussian* and *mean* curvatures of the surface.

For example, in a cylindrical surface, there is no bend along the linear direction (curvature equals zero) while the maximum bend is when intersecting with a plane parallel to the end faces (curvature equals 1/radius). Those two extremes make the principle curvatures of that surface.

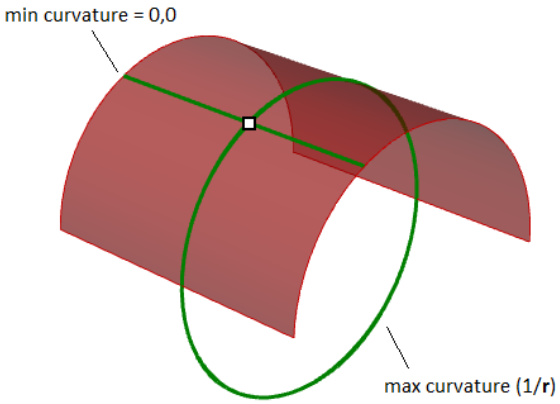
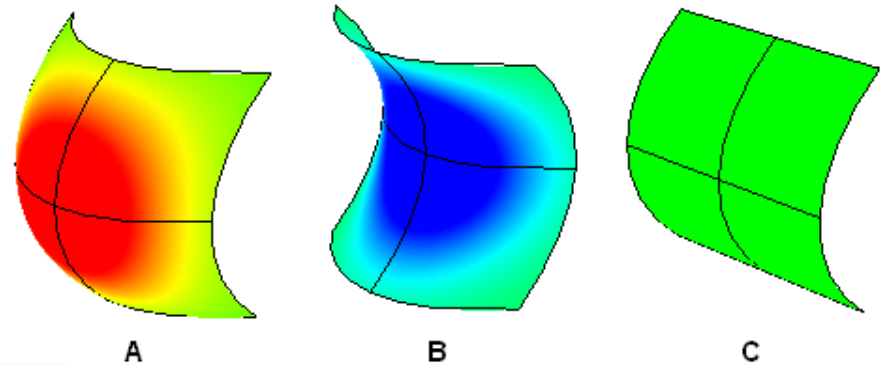


Figure (57): Principle curvatures at a surface point are the minimum and maximum curvatures at that point.

Gaussian curvature

The Gaussian curvature of a surface at a point is the product of the principal curvatures at that point. The tangent plane of any point with positive Gaussian curvature touches the surface locally at a single point, whereas the tangent plane of any point with negative Gaussian curvature cuts the surface.



A: Positive curvature when surface is bowl-like.

B: Negative curvature when surface is saddle-like.

C: Zero curvature when surface is flat in at least one direction (plane, cylinder).

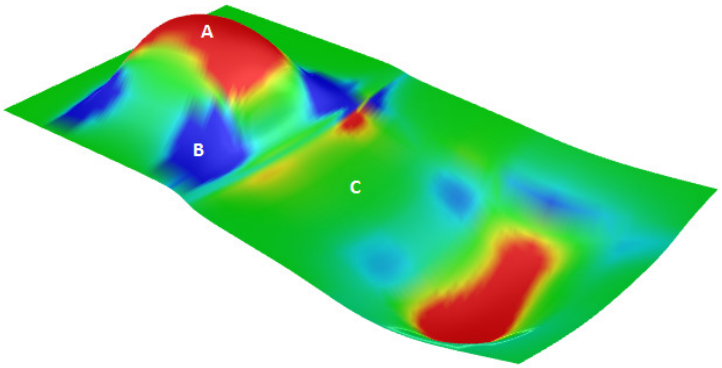


Figure (58): Analyzing the surface Gaussian curvature.

Mean curvature

The mean curvature of a surface at a point is one-half of the sums of the principal curvatures at that point. Any point with zero mean curvature has negative or zero Gaussian curvature.

Surfaces with zero mean curvature everywhere are minimal surfaces. Physical processes which can be modeled by minimal surfaces include the formation of soap films spanning fixed objects, such as wire loops. A soap film is not distorted by air pressure (which is equal on both sides) and is free to minimize its area. This contrasts with a soap bubble, which encloses a fixed quantity of air and has unequal pressures on its inside and outside. Mean curvature is useful for finding areas of abrupt change in the surface curvature.

Surfaces with constant mean curvature everywhere are often referred to as constant mean curvature (CMC) surfaces. CMC surfaces include the formation of soap bubbles, both free and attached to objects. A soap bubble, unlike a simple soap film, encloses a volume and exists in equilibrium where slightly greater pressure inside the bubble is balanced by the area-minimizing forces of the bubble itself.

NURBS surfaces

You can think of NURBS surfaces as a grid of NURBS curves that go in two directions. The shape of a NURBS surface is defined by a number of control points and the degree of that surface in each one of the two directions (u- and v-directions). NURBS surfaces are efficient for storing and representing free-form surfaces with a high degree of accuracy. The mathematical equations and details of NURBS surfaces are beyond the scope of this text. We will only focus on the characteristics that are most useful for designers.

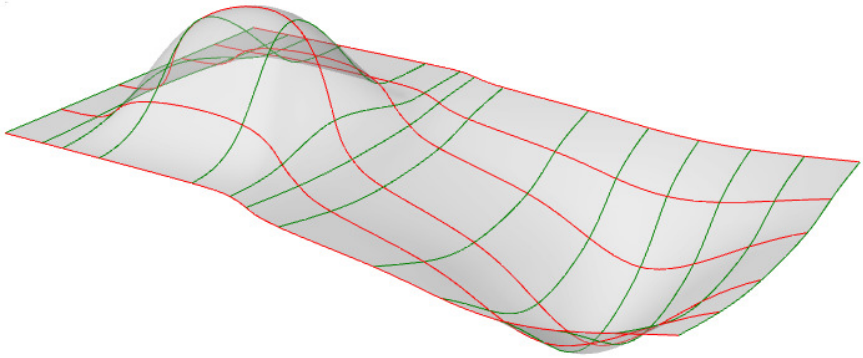


Figure (59): NURBS surface with red isocurves in the u-direction and green isocurves in the v-direction.

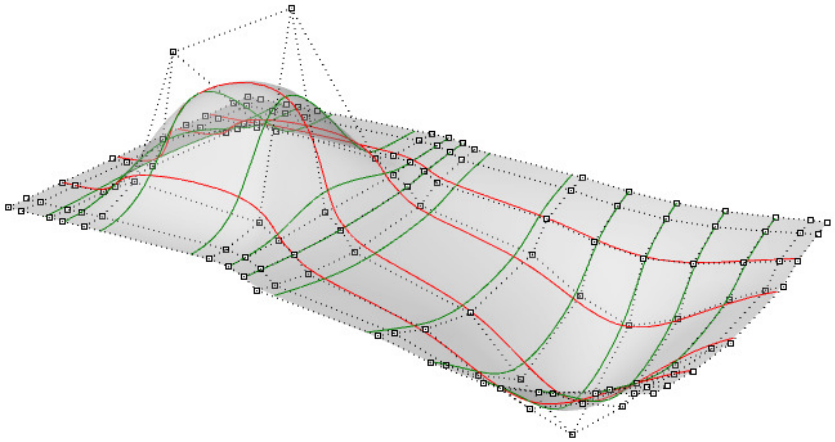


Figure (60): The control structure of a NURBS surface.

Figure (61): The parameter rectangle of a NURBS surface.

Evaluating parameters at equal intervals in the 2-D parameter rectangle does not translate into equal intervals in 3-D space in most cases.

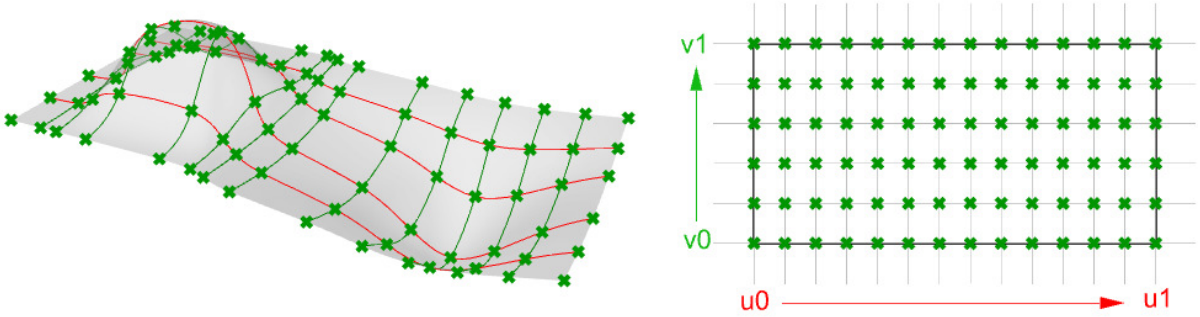


Figure (62): Evaluating surfaces.

Characteristics of NURBS surfaces

NURBS surface characteristics are very similar to NURBS curves except there is one additional parameter. NURBS surfaces hold the following information:

- Dimension, typically 3
- Degree in u-and v-directions: (sometimes use *order* which is degree + 1)
- Control points (points)
- Weights of control points (numbers)
- Knots (numbers)

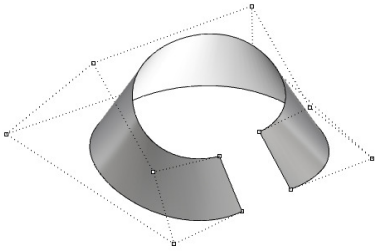
As with the NURBS curves, you will probably not need to know the details of how to create a NURBS surface, since 3-D modelers will typically provide good set of tools for this. You can always rebuild surfaces (and curves for that matter) to a new degree and number of control points. Surface can be open, closed, or periodic. Here are few examples of surfaces:

Degree-1 surface in both u- and v-directions.

All control points lie on the surface.

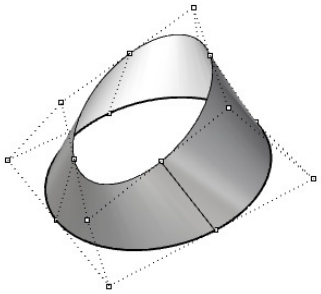
Degree-3 in the u-direction and degree-1 in the v-direction open surface.

The surface corners coincide with corner control points.

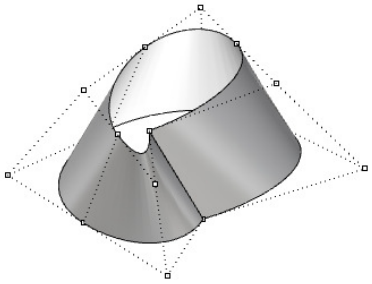


Degree-3 in the u-direction and degree 1 in the v-direction closed (non-periodic) surface.

Some control points coincide with the surface seam.

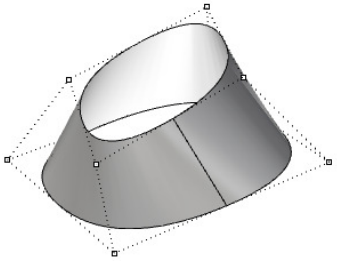


Moving control points of a closed (non-periodic) surface causes a kink and the surface does not look smooth.

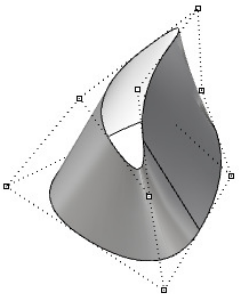


Degree 3 the u-direction and degree 1 in the v-direction periodic surface.

The surface control points do not coincide with the surface seam.



Moving the control points of a periodic surface does not affect surface smoothness or create kinks.



Singularity in NURBS surfaces

For example, if you have a linear edge of a simple plane, and you drag the two end control points of an edge so they overlap (collapse) at the middle, you will get a singular edge. You will notice that the surface isocurves converge at the singular point.

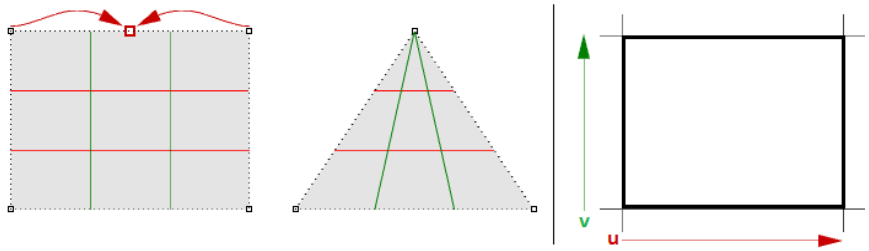


Figure (63): Collapse two corner points of a rectangular NURBS surface to create a triangular surface with singularity. The parameter rectangle remains rectangular.

The above triangular shape can be created without singularity. You can trim a surface with a triangle polyline. When you examine the underlying NURBS structure, you see that it remains a rectangular shape.

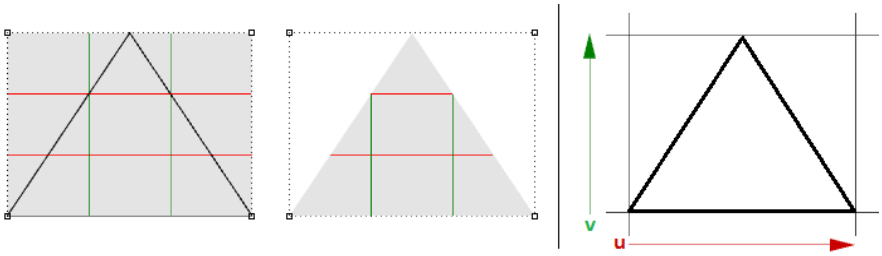


Figure (64): Trim a rectangular NURBS surface to create a trimmed triangular surface.

Other common examples of surfaces that are hard to generate without singularity are the cone and the sphere. The top of a cone and top and bottom edges of a sphere are collapsed into one point. Whether there is singularity or not, the parameter rectangle maintains a more or less rectangular region.

Trimmed NURBS surfaces

NURBS surfaces can be trimmed or untrimmed. Trimmed surfaces use an underlying NURBS surface and closed curves to trim out part of that surface. Each surface has one closed curve that defines the outer border (*outer loop*) and can have non-intersecting closed inner curves to define holes (*inner loops*). A surface with an outer loop that is the same as that of its underlying NURBS surface and that has no holes is what we refer to as an *untrimmed* surface.

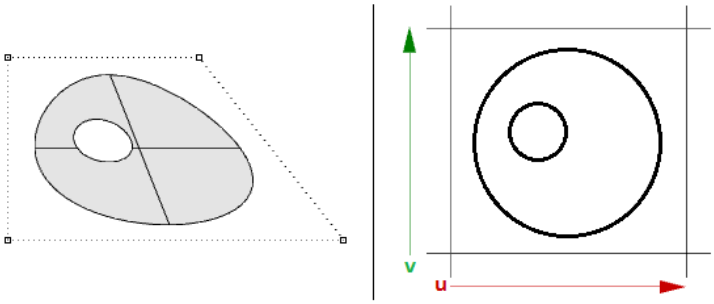


Figure (65): Trimmed surface in modeling space (left) and in parameter rectangle (right).

Polysurfaces

A polysurface consists of two or more (possibly trimmed) NURBS surfaces joined together. Each surface has its own structure, parameterization, and isocurve directions that do not have to match. Polysurfaces are represented using the boundary representation (*BRep*). The BRep structure describes surfaces, edges, and vertices with trimming data and connectivity among different parts. Trimmed surface are also represented using BRep data structure.

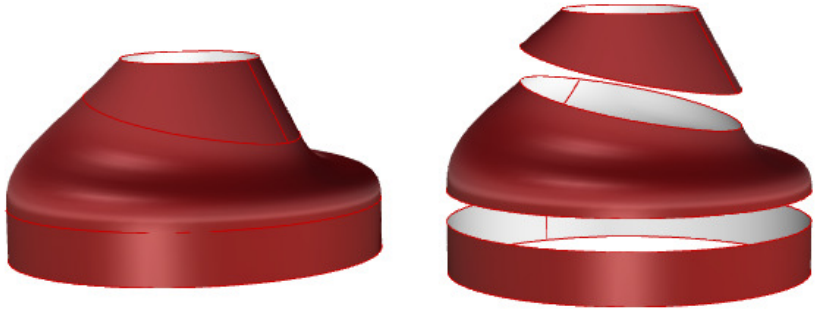


Figure (66): Polysurfaces are made out of joined surfaces with common edges aligning perfectly within tolerance.

The BRep is a data structure that describes each face in terms of its underlying surface, surrounding 3-D edges, vertices, parameter space 2-D trims, and relationship between neighboring faces. BRep objects are also called *solids* when they are closed (watertight).

An example polysurface is a simple box that is made out of six untrimmed surfaces joined together.

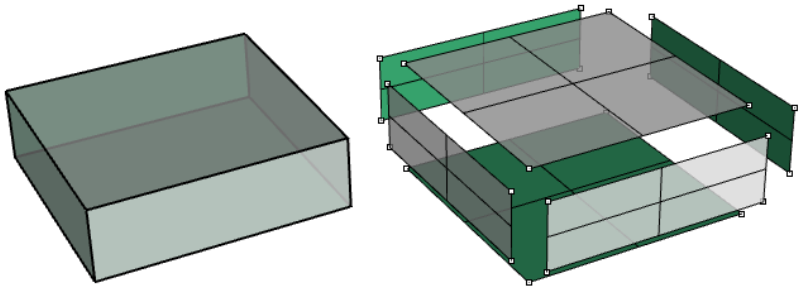


Figure (67): Box made out of six untrimmed surfaces joined in one polysurface.

The same box can be made using trimmed surfaces, such as the top one in the following example.

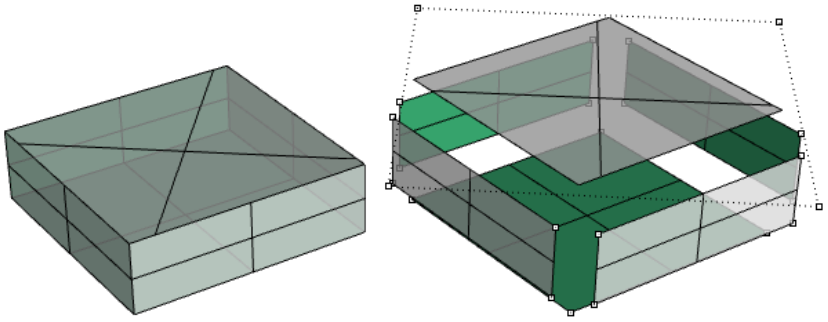


Figure (68): Box faces can be trimmed.

The top and bottom faces of the cylinder in the following example are trimmed from planar surfaces.

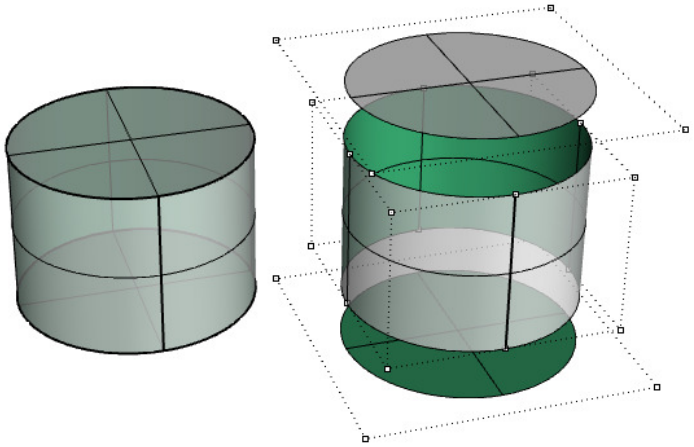


Figure (69) shows the control points of the underlying surfaces.

We saw that editing NURBS curves and untrimmed surfaces is intuitive and can be done interactively by moving control points. However, editing trimmed surfaces and polysurfaces can be challenging. The main challenge is to be able to maintain joined edges of different faces within the desired tolerance. Neighboring faces that share common edges can be trimmed and do not usually have matching NURBS structure, and therefore modifying the object in a way that deforms that common edge might result in a gap.

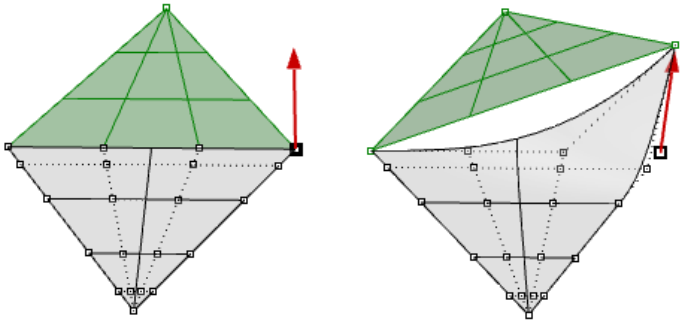


Figure (70): Two triangular faces joined in one polysurface but do not have matching joined edge. Moving one corner create a hole.

Another challenge is that there is typically less control over the outcome, especially when modifying trimmed geometry.

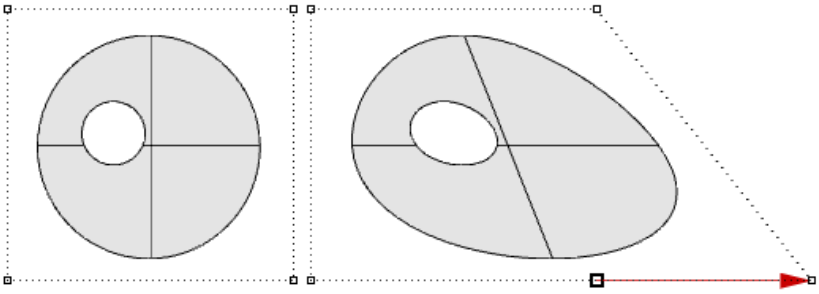


Figure (71): Once a trimmed surface is created, there is limited control to edit the result.

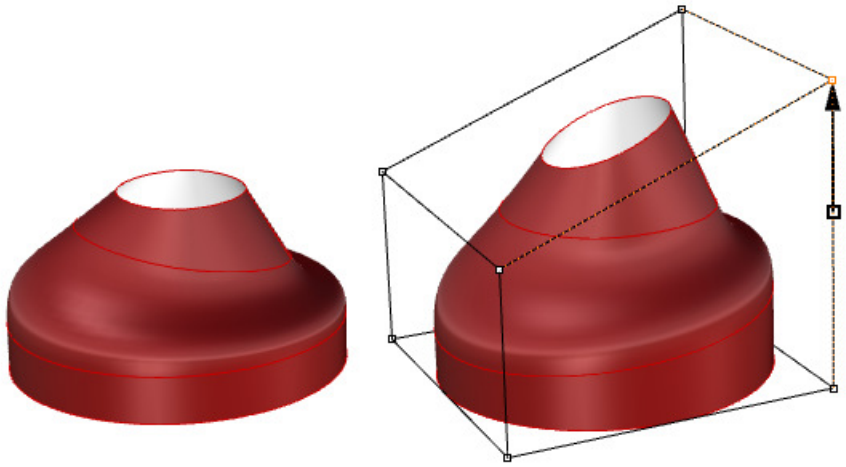


Figure (72): Use cage edit technique in Rhino to edit polysurfaces.

Trimmed surfaces are described in parameter space using the untrimmed underlying surface combined with the 2-D trim curves that evaluate to the 3-D edges within the 3-D surface.

Tutorials

The following tutorials use the concepts learned in this chapter. They use Rhinoceros 5 and Grasshopper 0.9.

Continuity between curves

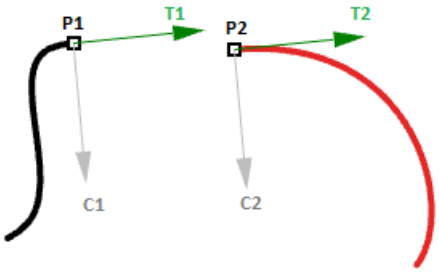
Examine the continuity between two input curves. Continuity assumes that the curves meet at the end of the first curve and the start of the second curve.

Input:

Two input curves.

Parameters:

Calculate the following to be able to decide the continuity between two curves:

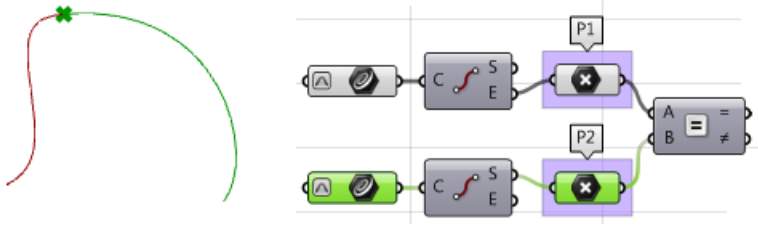


- The end point of the first curve (P1)
- The start point of the second curve (P2)

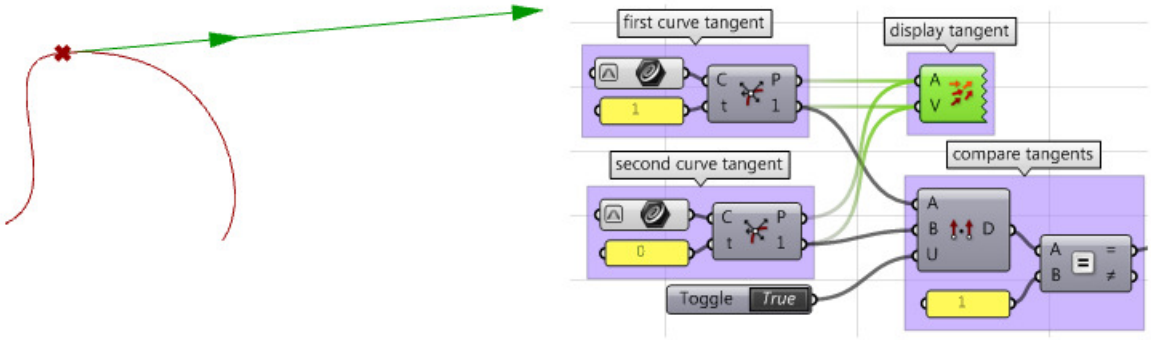
- The tangent at the end of the first curve and at the start of the second curve (T1 and T2).
- The curvature at the end of the first curve and at the start of the second curve (C1 and C2).

Solution:

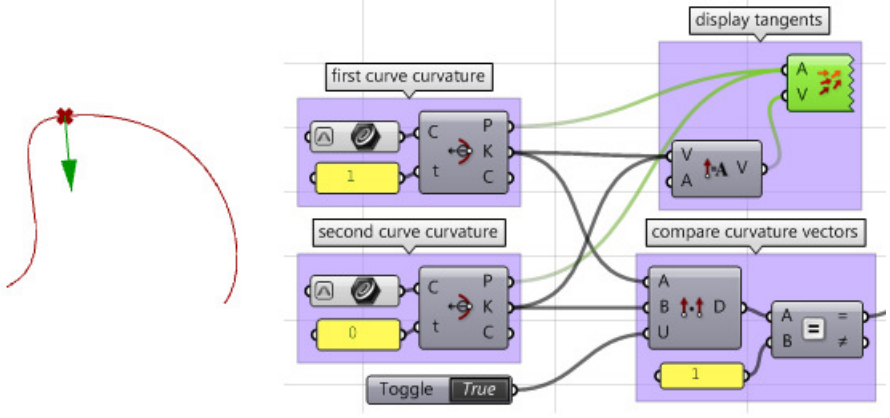
1. Reparameterize the input curves. We do that so that we know that the start of the curve evaluates at t=0 and the end at t=1.
2. Extract the end and start points of the two curves, and check whether they coincide. If they do, the two curves are at least G0 continuous.



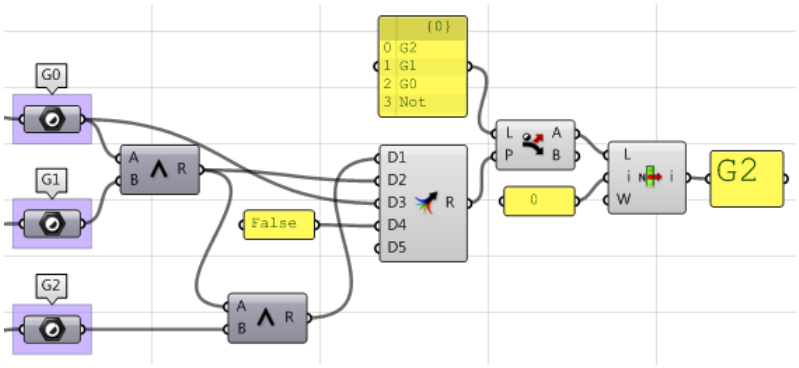
3. Calculate tangents.
4. Compare the tangents using the dot product. Make sure to unitize vectors. If the curves are parallel, then we have at least G1 continuity.



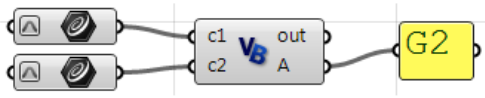
5. Calculate curvature vectors.
6. Compare curvature vectors, and if they agree, the two curves are G2 continuous.



7. Create logic that filters through the three results (G1, G2, and G3) and select the highest continuity.

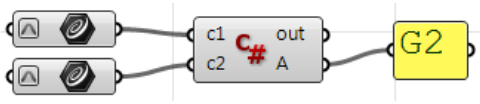

G0
G1
G2
A
B
R
False
{0}
0 G2
1 G1
2 G0
3 Not
D1
D2
D3
D4
D5
L
P
A
B
0
L
i
W
G2

Using the Grasshopper VBScript component:



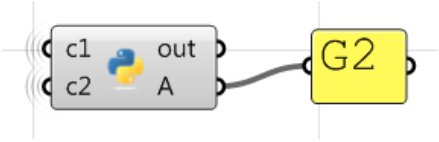

```
Private Sub RunScript(ByVal c1 As Curve, ByVal c2 As Curve, ByRef A As Object)

  'declare variables
  Dim continuity As New String("")
  Dim t1, t2 As Double
  Dim v_c1, v_c2, c_c1, c_c2 As Vector3d

  'extract start and end points
  Dim end_c1 = c1.PointAtEnd
  Dim start_c2 = c2.PointAtStart

  'check G0 continuity
  If end_c1.DistanceTo(start_c2) = 0 Then
    continuity = "G0"
  End If

  'check G1 continuity
  If continuity = "G0" Then
    'calculate tangents
    v_c1 = c1.TangentAtEnd
    v_c2 = c2.TangentAtStart
    'unitize tangent vectors
    v_c1.Unitize
    v_c2.Unitize
    'compare tangents
    If v_c1 * v_c2 = 1 Then
      continuity = "G1"
    End If
  End If

  'check G2 continuity
  If continuity = "G1" Then
    'extract the parameter at start and end of the curves domain
    t1 = c1.Domain.Max
    t2 = c2.Domain.Min
    'calculate curvature
    c_c1 = c1.CurvatureAt(t1)
    c_c2 = c2.CurvatureAt(t2)
    'unitize curvature vectors
    c_c1.Unitize
    c_c2.Unitize
    'compare vectors
    If c_c1 * c_c2 = 1 Then
      continuity = "G2"
    End If
  End If

  'Assign output
  A = continuity
End Sub
```

Using the Grasshopper C# component:




```
private void RunScript(Curve c1, Curve c2, ref object A)
{
  //declare variables
  string continuity = ("");
  double t1, t2;
  Vector3d v_c1, v_c2, c_c1, c_c2;

  //extract start and end points
  Point3d end_c1 = c1.PointAtEnd;
  Point3d start_c2 = c2.PointAtStart;

  //check G0 continuity
  if( end_c1.DistanceTo(start_c2) == 0){
    continuity = "G0";
  }

  //check G1 continuity
  if( continuity == "G0")
  {
    //calculate tangents
    v_c1 = c1.TangentAtEnd;
    v_c2 = c2.TangentAtStart;
    //unitize tangent vectors
    v_c1.Unitize();
    v_c2.Unitize();
    //compare tangents
    if( v_c1 * v_c2 == 1 ){
      continuity = "G1";
    }
  }

  //check G2 continuity
  if( continuity == "G1" )
  {
    //extract the parameter at start and end of the curves domain
    t1 = c1.Domain.Max;
    t2 = c2.Domain.Min;
    //calculate curvature
    c_c1 = c1.CurvatureAt(t1);
    c_c2 = c2.CurvatureAt(t2);
    //unitize curvature vectors
    c_c1.Unitize();
    c_c2.Unitize();
    //compare vectors
    if( c_c1 * c_c2 == 1 ){
      continuity = "G2";
    }
  }

  //assign output
  A = continuity;
}
```

Using the Grasshopper Python component:



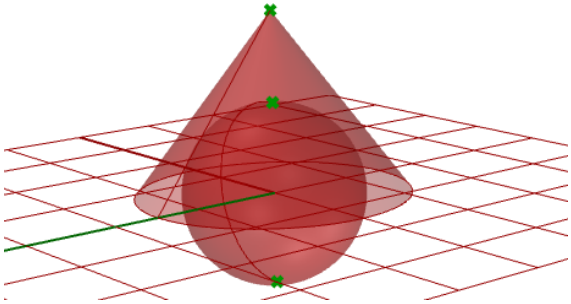

```
#decalre variables
continuity = ""

#extract start and end points
end_c1 = c1.PointAtEnd
start_c2 = c2.PointAtStart

#check G0 continuity
if end_c1.DistanceTo(start_c2) == 0:
    continuity = "G0"

#check G1 continuity
if continuity == "G0":
    #calculate tangents
    v_c1 = c1.TangentAtEnd
    v_c2 = c2.TangentAtStart
    #unitize tangent vectors
    v_c1.Unitize()
    v_c2.Unitize()
    #compare tangents
    dot = v_c1 * v_c2
    if dot == 1:
        continuity = "G1"
        
#check G2 continuity
if continuity == "G1":
    
    #extract the parameter at start and end of the curves domain
    t1 = c1.Domain.Max
    t2 = c2.Domain.Min
    #calculate curvature
    c_c1 = c1.CurvatureAt(t1)
    c_c2 = c2.CurvatureAt(t2)
    #unitize curvature vectors
    c_c1.Unitize()
    c_c2.Unitize()
    #compare vectors
    dot = c_c1 * c_c2
    if dot == 1:
        continuity = "G2"
        
#assign output
A = continuity
```

Surfaces with singularity

Extract singular points in a sphere and a cone.

Input:

One sphere and one cone.



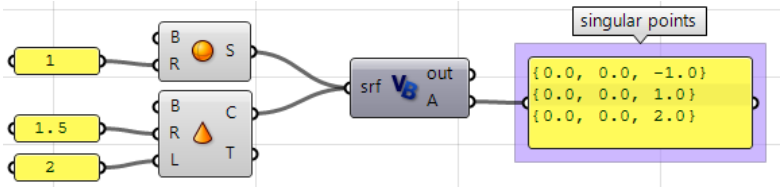
Parameters:

Singularity can be detected through analyzing the 2-D parameter space trims that have invalid or zero-length corresponding edges. Those trims ought to be singular.

Solution:

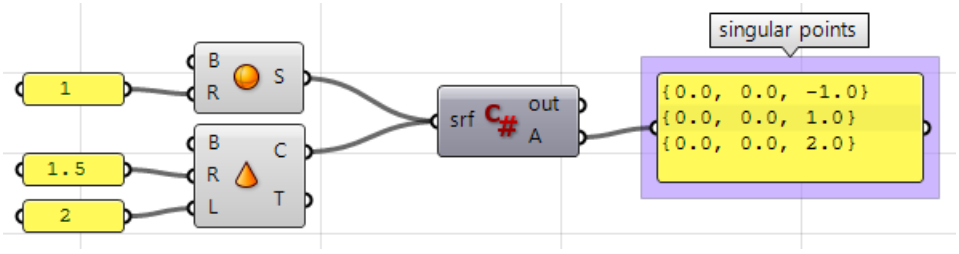
1. Traverse through all trims in the input.
2. Check if any trim has an invalid edge and flag it as a singular trim.
3. Extract point locations in 3-D space.

Using the Grasshopper VB component:



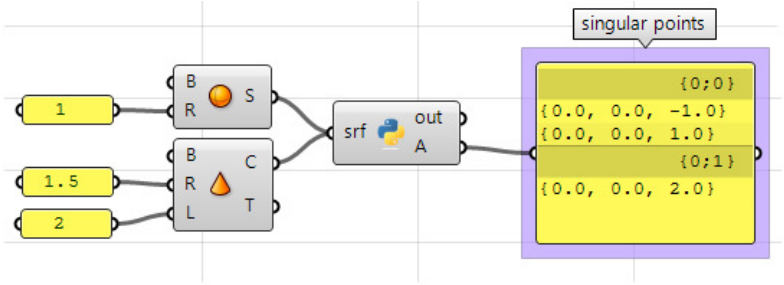

```
Private Sub RunScript(ByVal srf As Brep, ByRef A As Object)

  'Decalre a new list of points
  Dim singular_points As New List( Of Point3d)

  'Examine all trims in the input
  For Each trim As BrepTrim In srf.Trims

    'Null edge of a trim indicates a singularity
    If trim.Edge Is Nothing Then

      'Find the 2D parameter space point of the start or end of the trim
      Dim pt2d = New Point2d(trim.PointAtStart)

      'Evaluate trim end point on the object surface
      Dim pt3d = trim.Face.PointAt(pt2d.x, pt2d.y)

      'Add 3D point to the list of singular points
      singular_points.Add(pt3d)
    End If

  Next

  'Asign output
  A = singular_points

End Sub
```

Using the Grasshopper C# component:




```
private void RunScript(Brep srf, ref object A)
{
  //Decalre a new list of points
  List < Point3d > singular_points = new List<Point3d>();

  //Examine all trims in the input
  foreach( BrepTrim trim in srf.Trims)
  {
    //Null edge of a trim indicates a singularity
    if( trim.Edge == null)
    {
      //Find the 2D parameter space point of the start or end of the trim
      Point2d pt2d = new Point2d(trim.PointAtStart);

      //Evaluate trim end point on the object surface
      Point3d pt3d = trim.Face.PointAt(pt2d.X, pt2d.Y);

      //Add 3D point to the list of singular points
      singular_points.Add(pt3d);
    }
  }

  //Asign output
  A = singular_points;
}
```

Using the Grasshopper Python component:




```
#Decalre a new list of points
singular_points = []

#Examine all trims in the input brep
for trim in srf.Trims:

    #Null edge of a trim indicates a singularity
    if trim.Edge == None:
        #Find the 2D parameter space point at trim start or end
        pt2d = trim.PointAtStart
        
        #Evaluate trim end point on the object surface
        pt3d = trim.Face.PointAt(pt2d.X, pt2d.Y)
        
        #Add 3D point to the list of singular points
        singular_points.append(pt3d)
        
#Asign output
A = singular_points
```

References

Edward Angel, "Interactive Computer Graphics with OpenGL," Addison Wesley Longman, Inc., 2000.

James D Foley, Steven K Feiner, John F Hughes, "Introduction to Computer Graphics" Addison-Wesley Publishing Company, Inc., 1997.

James Stewart, "Calculus," Wadsworth, Inc., 1991.

Kenneth Hoffman, Ray Kunze, "Linear Algebra", Prentice-Hall, Inc., 1971

Rhinoceros® help document, Robert McNeel and Associates, 2009.

Notes

[i] Wikipedia: Projection (linear algebra)

[ii] Wikipedia: Cubic Hermite spline.

[iii] Wikipedia: Bézier curve.

[iv] Wikipedia: Non-uniform rational B-spline.

[v] Wikipedia; De Casteljau's algorithm.

[vi] Wikipedia: NURBS.

[vii] Wikipedia: De Boor's algorithm.

[viii] Michigan Tech, Department of Computer Science, De Boor's algorithm.