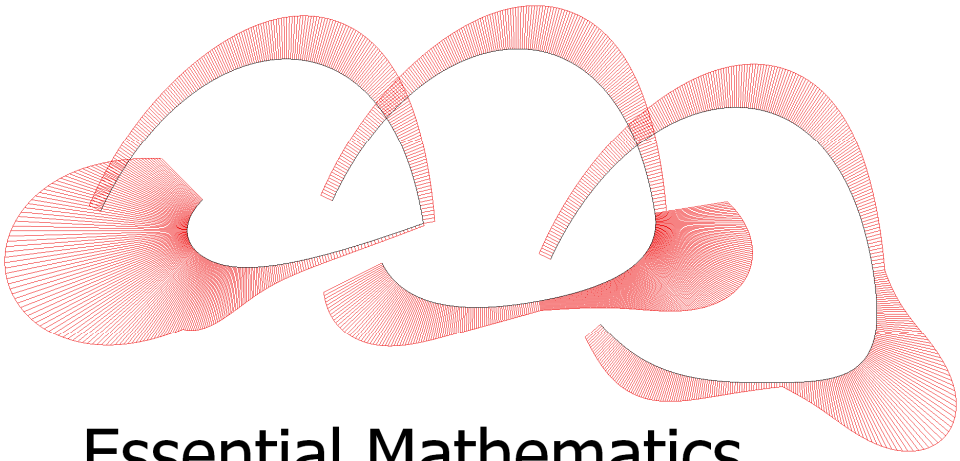


Essential Mathematics
for computational design

Rajaa Issa

Rhinoceros development
Robert McNeel & Associates

Preface

Essential Mathematics for Computational Design introduces design professionals to foundation mathematical concepts that are necessary for effective development of computational methods for 3D 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.

This book is directed towards designers who have little or no background in the subject. It assumes that readers have good knowledge of Rhinoceros® (Rhino), NURBS modeling for Windows, and Grasshopper® (GH), the visual scripting environment for Rhino. Both are used as tools to explain various concepts. For more information, go to www.rhino3d.com and www.grasshopper3d.com.

This book has three parts. The first discusses vector math including vector representation, vector operation, and line and plane equations. The second part reviews matrix operations and transformations. The third part includes a general review of parametric curves with special focus on NURBS curves and the concepts of continuity and curvature. The material in this manual is based partly on a workshop I held at the University of Texas at Arlington for the Tex-Fab event February, 2010.

Rajaa Issa
Rhino Development
Robert McNeel & Associates

Table of Contents

1 Vector Mathematics

Vector representation

Vectors indicate a quantity that has "direction" and "magnitude" such as velocity or force. Vectors in 2D coordinate systems are represented with two real numbers in the form:

V = <a1, a2>

Similarly, in 3-D coordinate system, vectors are represented by three real numbers and would look like:

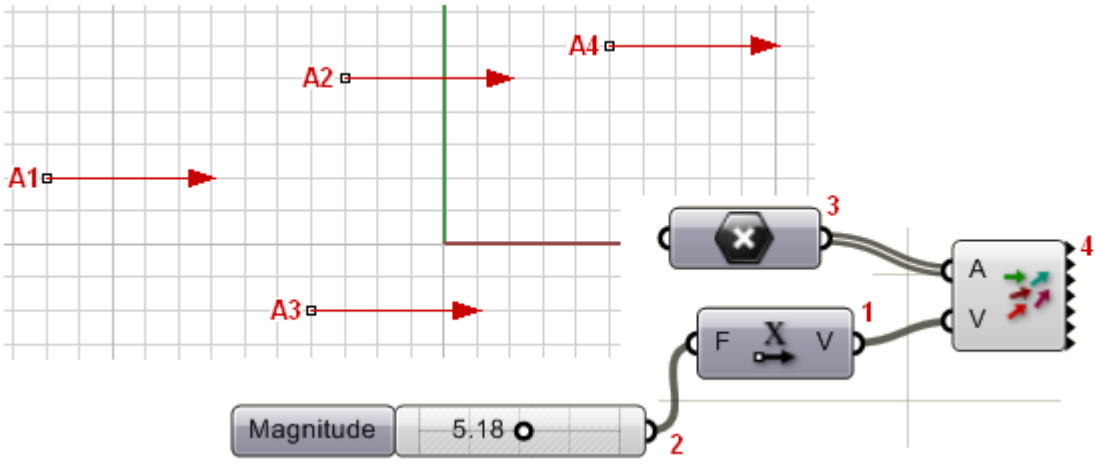
V = <a1, a2, a3>

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. We usually put an arrowhead to show the direction of vectors.

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 magnitude equal to 5.18 units, then we can write the vector as follows:

V = <5.18, 0, 0> (Angle brackets differentiate a vector from point coordinates.)

To represent that vector, we need an anchor point in the coordinate system. For example, all of the red line segments in the following figure are equal representations of the same vector.



1. Grasshopper unit x-axis component
2. Grasshopper number slider component
3. Grasshopper point components that is set to reference multiple points in Rhino (in this case referencing A1, A2, A3 and A4)
4. Grasshopper vector display component

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 representation of vector <a>.

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

Let us define an anchor point using Grasshopper x,y,z point component:

A = (1,2,3)

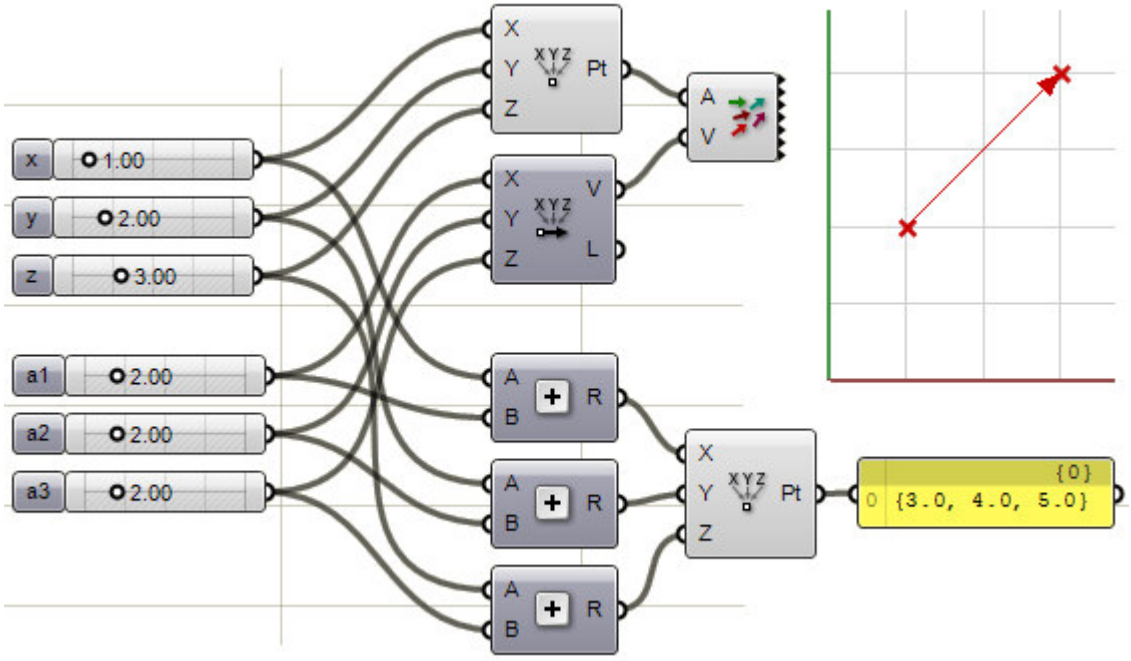
And a vector using Grasshopper xyz vector component that takes as an input three real numbers:

V = <2,2,2>

The tip point of the vector is calculated by adding the corresponding components from point A and vector V:

B = (3,4,5)

The following definition displays this vector using the Grasshopper vector display component, and marks the end of the displayed vector that coincides with point B:

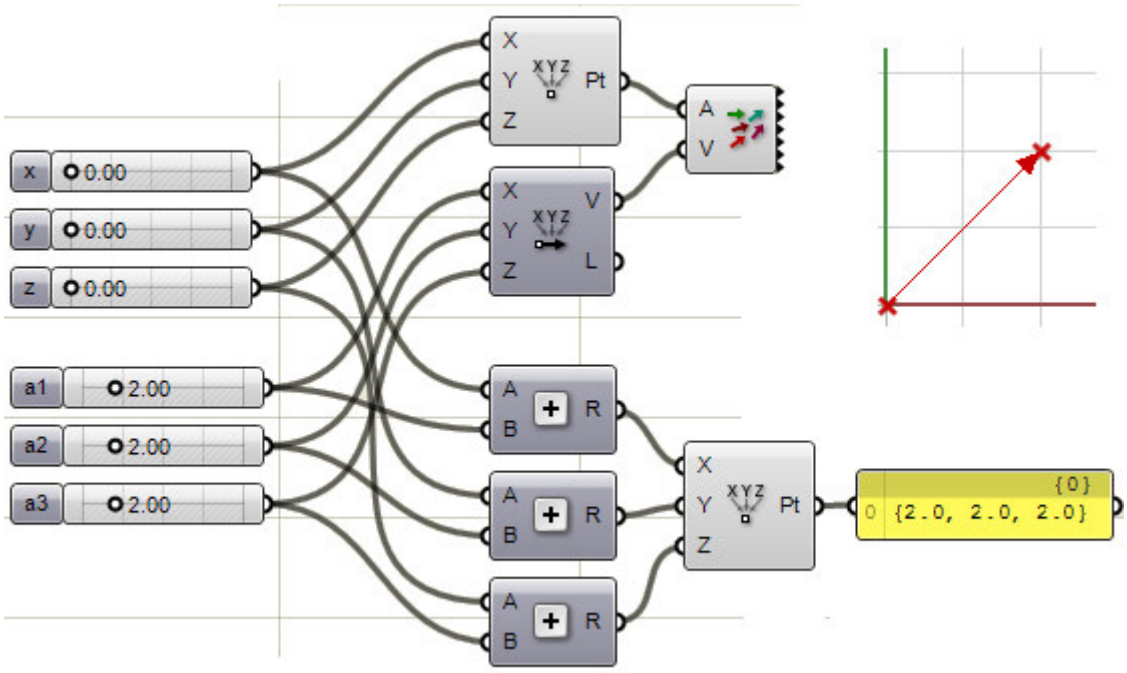


Position vector

There is one special vector representation that 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 A and B so that:

A = (0,0,0)
B = (a1,a2,a3)

Note that in the following Grasshopper definition, point B coordinates are equal to vector components.



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

Vector operations

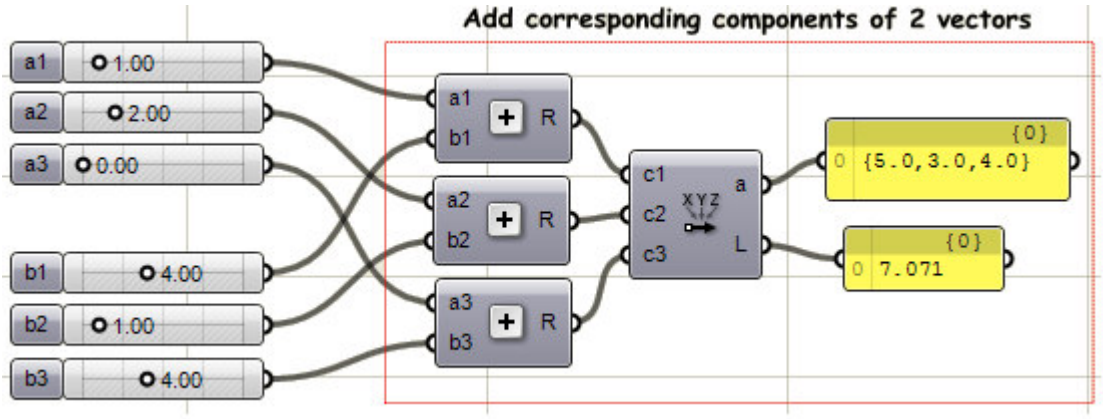
Vector addition

We add vectors by adding corresponding components. That is, if we have two vectors, **a** and **b**, the addition vector **a+b** is calculated as follows:

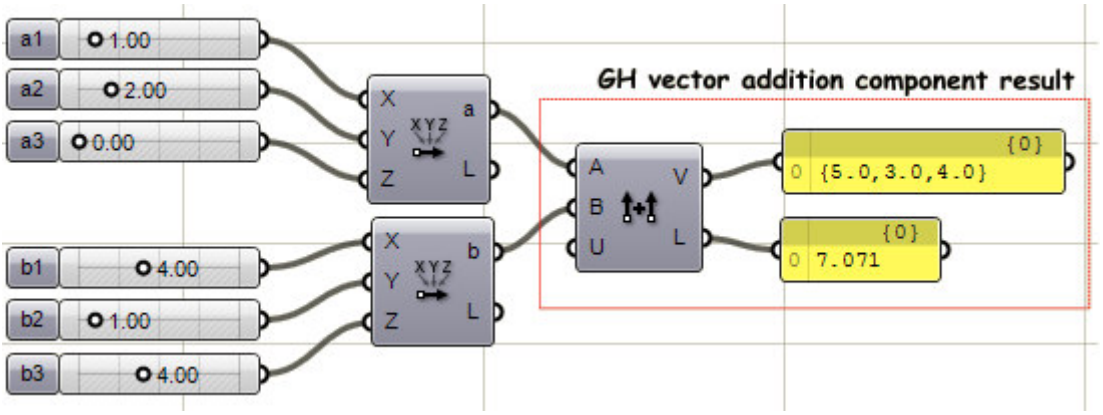
a = <a1, a2, a3>
b = <b1, b2, b3>
a+b = <a1+b1, a2+b2, a3+b3>

For example, if we have **a<1, 2, 0>** and **b<4, 1, 4>** the **a+b=<5, 3, 4>** addition is shown in the following figure:

The following Grasshopper definition shows how to create the **a+b** vector by adding corresponding components of the two input vectors **a** and **b**.

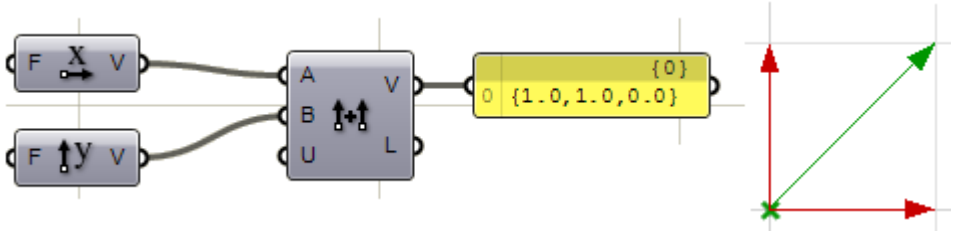


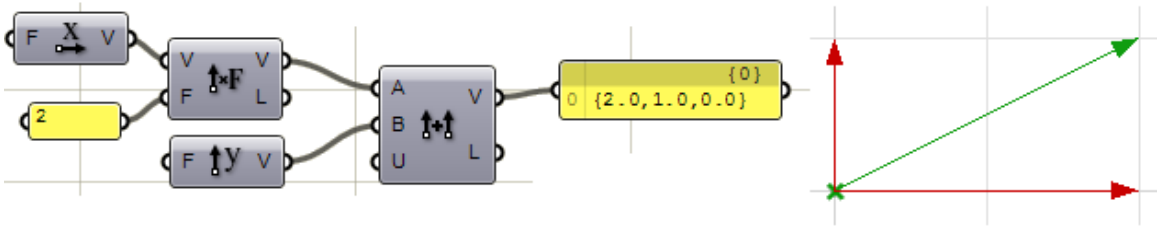
The resulting vector is the same of that resulting from using Grasshopper's built-in addition component:



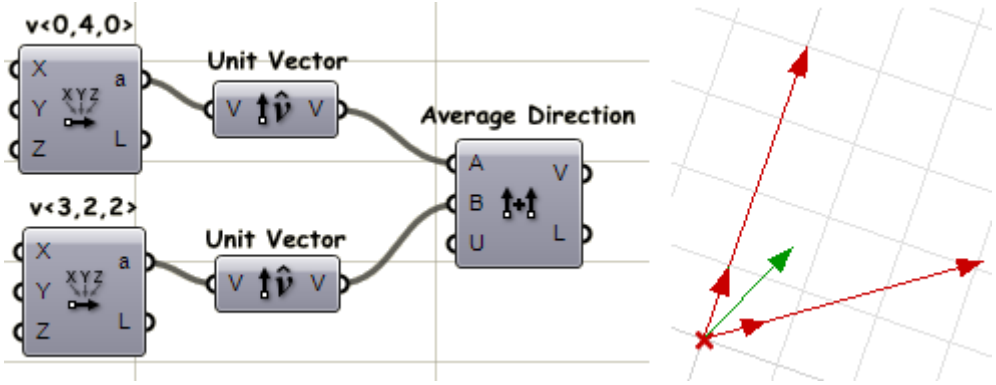
Add two vectors by adding their corresponding components.

Vector addition is also useful for finding the average direction of multiple 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:





The general case of how to solve adding vectors of different lengths to find average directions.



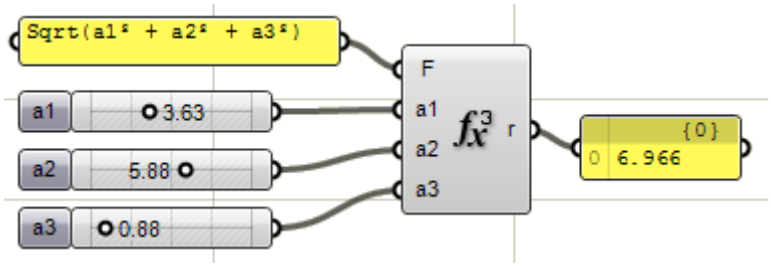
Vector scalar operation

A vector magnitude or length is calculated using the following:

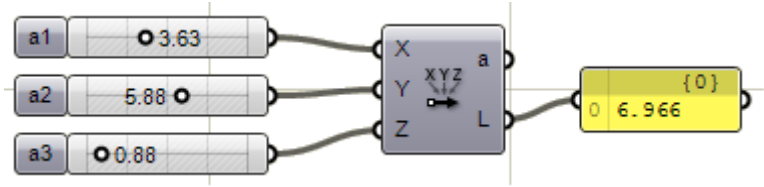
For vector **a = <a1, a2, a3>**

The **magnitude** of vector **a** = **sqrt(a1^2 + a2^2 +a3^2)**

Here is an example of calculating vector magnitude using Grasshopper function or expression:



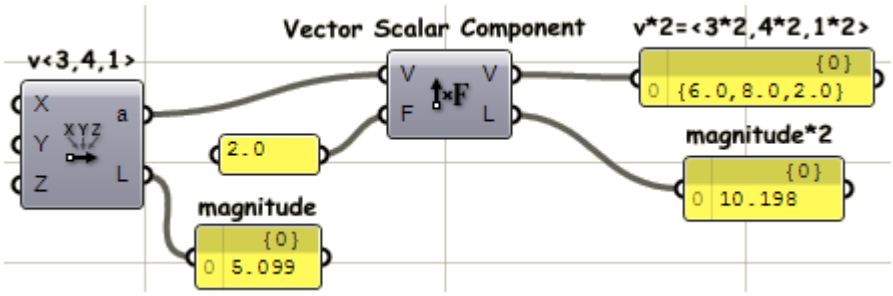
Note that Grasshopper vector component has an output **L** that is the vector magnitude:



Given vector **a = <a1, a2, a3>**, and factor **t** = some real number,

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

Here is the equation implemented in Grasshopper:



Unit vector

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

A vector is called a unit vector when its length or magnitude equal to one unit.

Vector properties

There are eight properties of vectors. If **A**, **B**, and **C** are vectors and **i** and **j** are scalar, then:

1. A + B = B + A
2. A + 0 = A
3. i(A+B) = iA + iB
4. ij(A) = i(jA)
5. A+(B + C) = (A+B) + C
6. A + (-A) = 0
7. (i + j)A = iA + jA
8. 1 * A = A

Vector dot or scalar product

The dot product of two vectors is defined as follows:

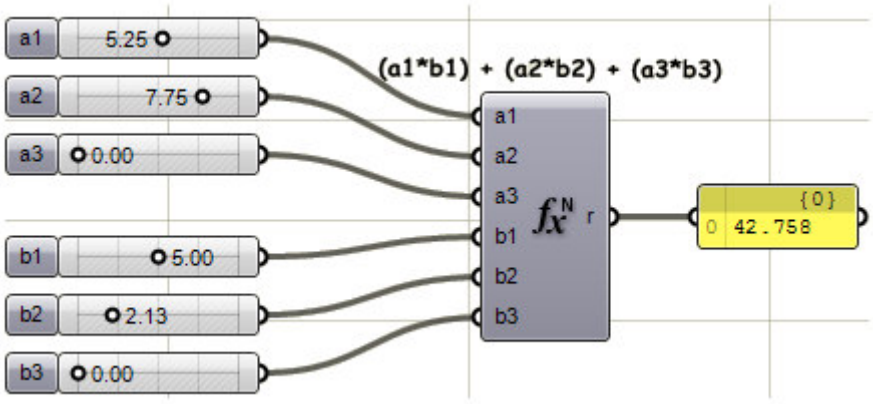
Given:

vector a = <a1, a2, a3>

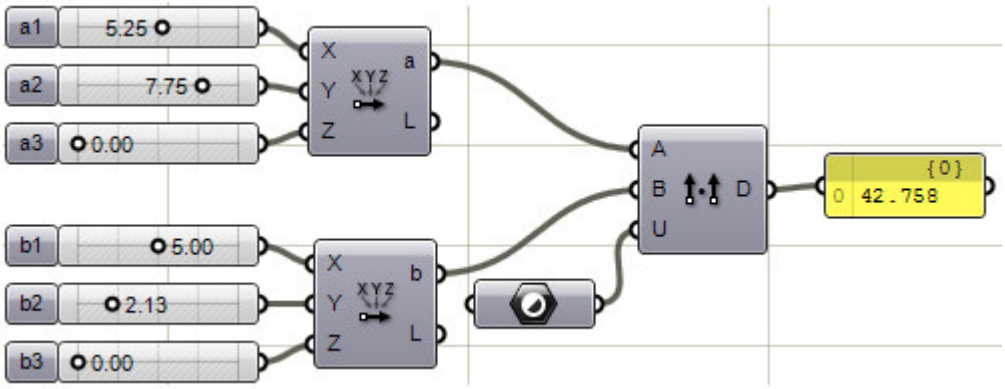
vector b = <b1, b2, b3>

a.b = a1*b1 + a2*b2 + a3*b3

In the following illustration, we will show that Grasshopper vector dot product component yields the same result as this **a.b** equation:



Grasshopper has built-in vector dot product component as shown in the following illustration:



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

The dot product of a vector with itself is that vector length to the power of two:

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

Proof:

If vector **a = <a1, a2, a3>** then from the definition of dot product of two vectors:

$$\mathbf{a.a} = a1*a1 + a2*a2 + a3*a3$$

or

$$\mathbf{a.a} = a1^2 + a2^2 + a3^2$$

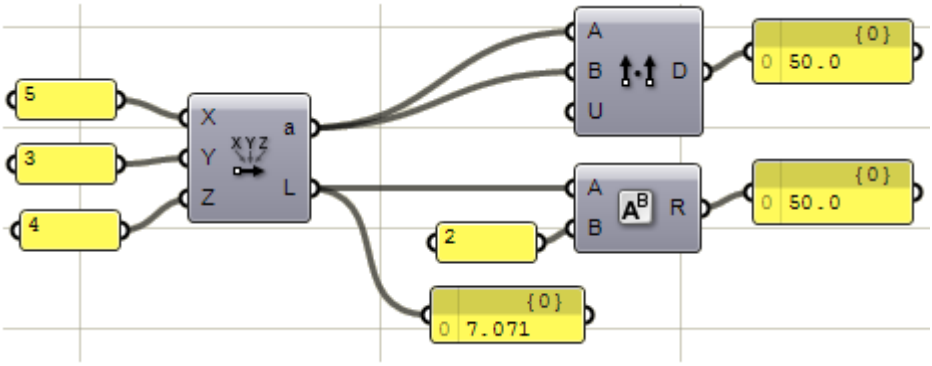
Since we know that:

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

Therefore,

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

Here is a Grasshopper sample to prove this property:



Dot product and angle between vectors

One important theorem for vector dot product is:

$a.b = |a||b|\cos(\theta)$, or

$\cos(\theta) = a.b / (|a||b|)$

And if vectors a and b are unit vectors, we can simply say:

$\cos(\theta) = a.b$

The dot product of two unit vectors equals the cosine of the angle between them

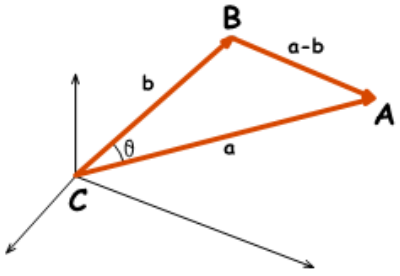
Proof:

From the law of cosines on triangle ABC

$|AB|^2 = |CA|^2 + |CB|^2 - 2|CA||CB|\cos(\theta)$

or:

$|a-b|^2 = |a|^2 + |b|^2 - 2|a||a|\cos(\theta)$ --- (1)



$|AB|^2$ is the same as $|a-b|^2$, so we can say:

$|a-b|^2 = \{a-b\} . (a-b)$

$= a.a - a.b - b.a + b.b$

$= |a|^2 - 2a.b + |b|^2$ --- (2)

from (1) & (2)

$|a|^2 - 2a.b + |b|^2 = |a|^2 + |b|^2 - 2|a||a|\cos(\theta)$

then:

$2a.b = 2|a||a|\cos(\theta)$

or:

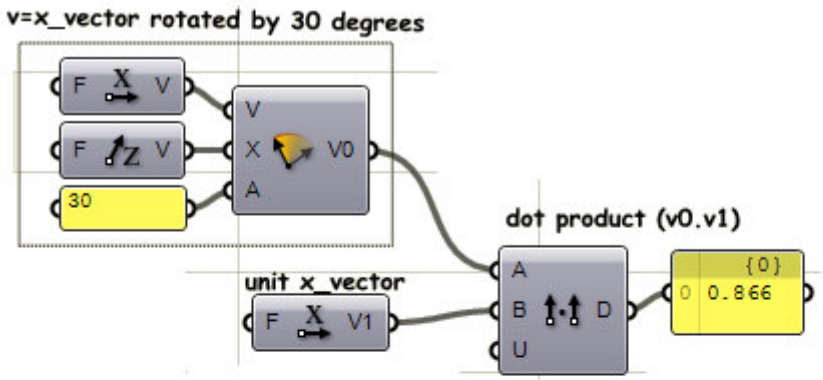
$\cos(\theta) = a.b / (|a||b|)$

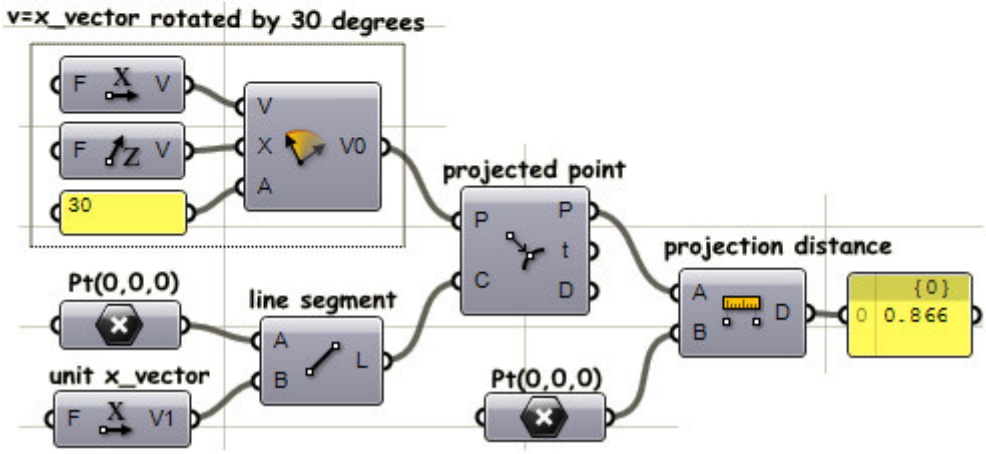
Vectors a and b are orthogonal if, and only if, a.b = 0.

What is the dot product of two unit vectors if they are parallel?

In the most practical way, you can think of the dot product of two vectors to be the projection length of one vector on the other.

Here is a proof using Grasshopper:





Dot product properties

If **A**, **B**, and **C** are vectors and **i** is scalar then:

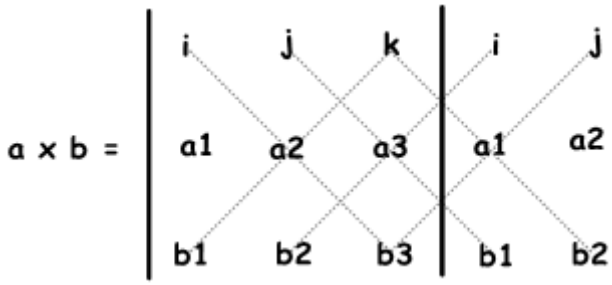
1. $A \cdot A = |A|^2$
2. $A \cdot (B + C) = A \cdot B + A \cdot C$
3. $0 \cdot A = 0$
4. $A \cdot B = B \cdot A$
5. $(iA) \cdot B = i(A \cdot B) = A \cdot (iB)$

Vector cross product

The cross product of two vectors produces a third vector that is orthogonal to both input vectors. Given:

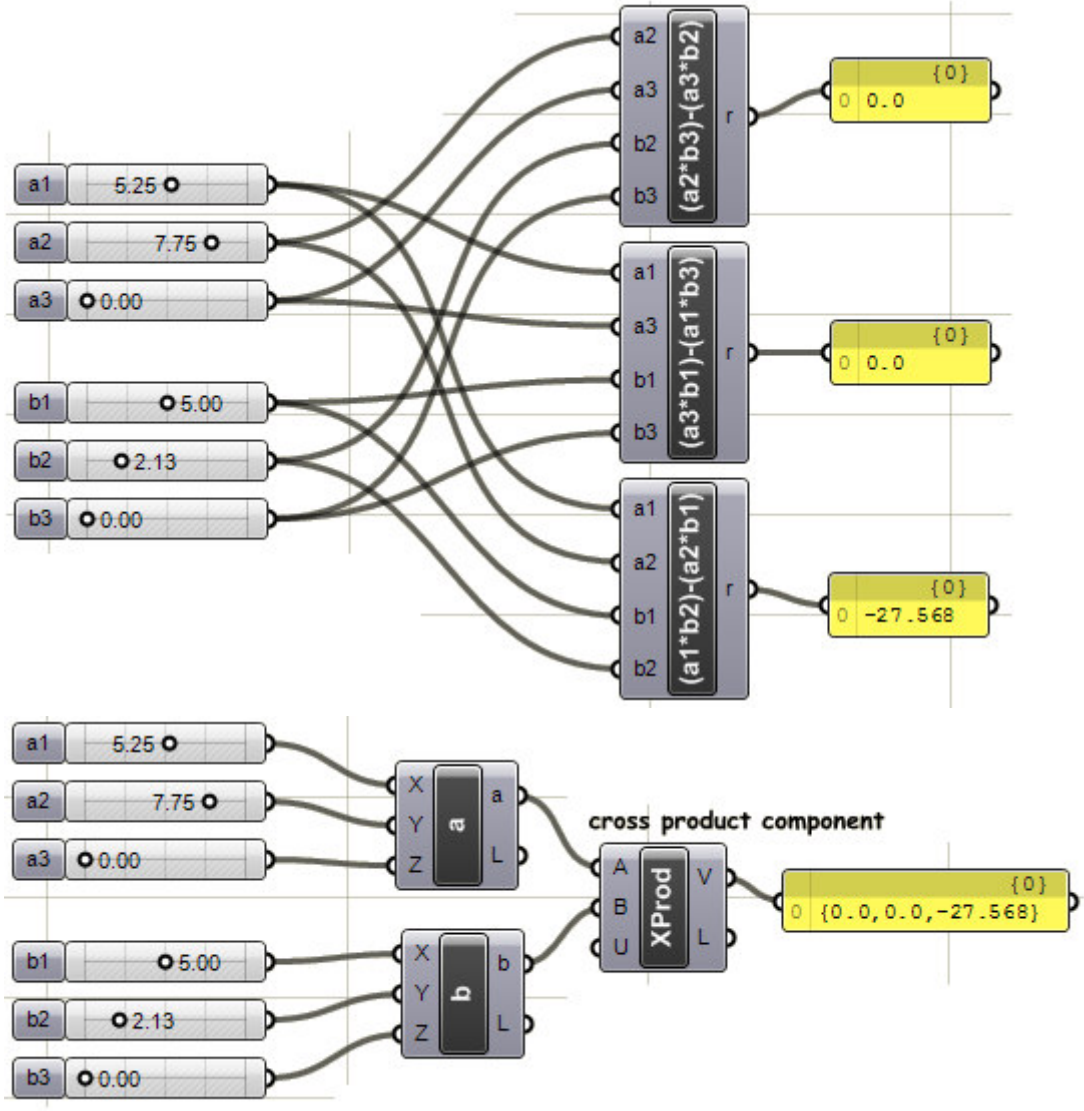
Vector a = <a1, a2, a3>
Vector b = <b1, b2, b3>

The cross product **a X b** is solved used determinants. Here is a quick illustration of how to calculate a determinate mathematically:



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

This is the Grasshopper definition for solving the cross product using expressions and comparing it with the vector cross product built-in component.



The vector a x b is orthogonal to both a & b

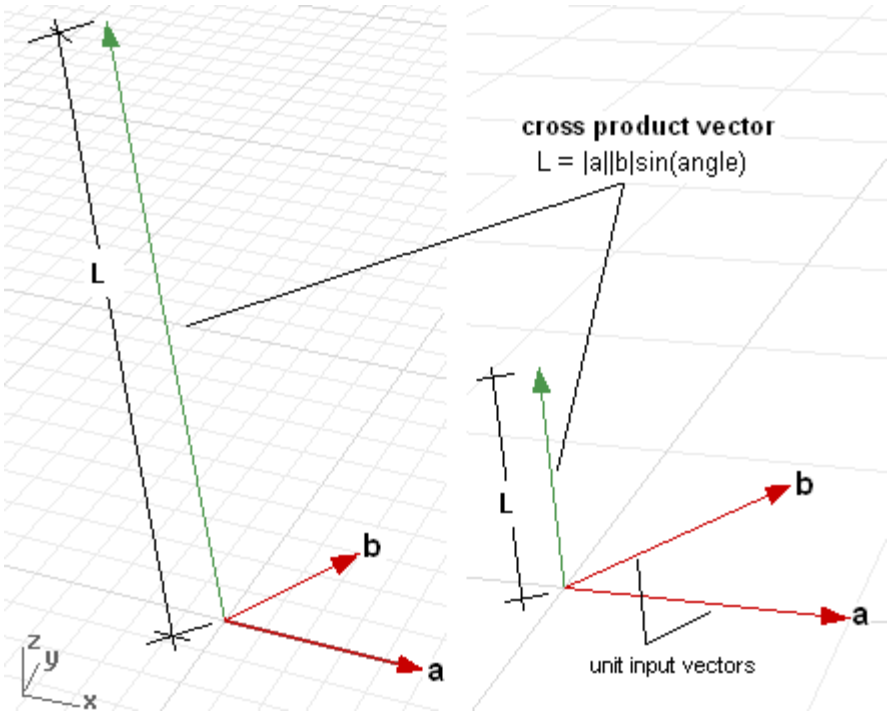
Theorem

If the angle between vectors **a** and **b** is between 0 and 180, then:

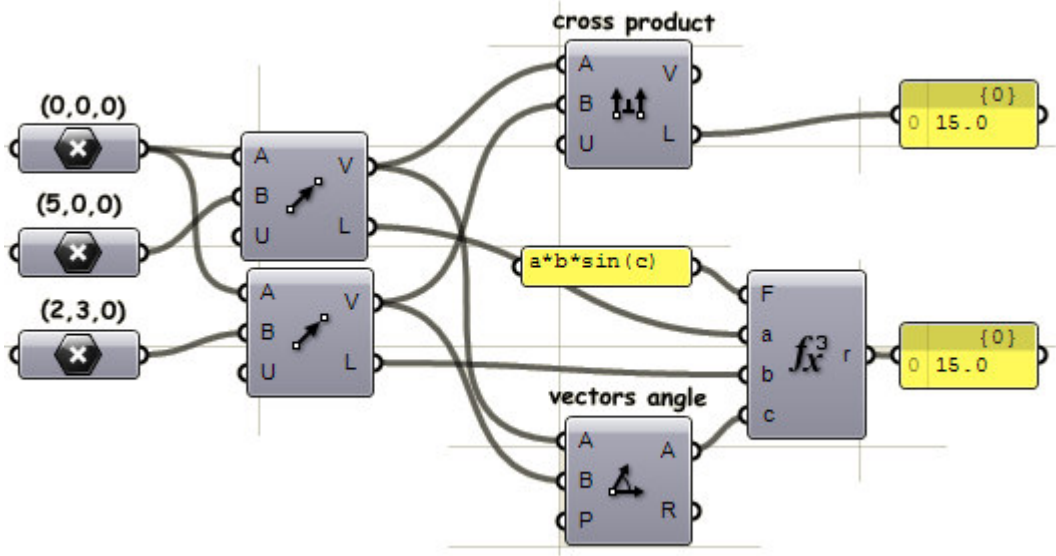
|a x b| = |a||b|sin(ɵ)

Or if **a** and **b** are unit vectors, then:

|a x b| = sin(ɵ)



This is the Grasshopper example to calculate the length of the cross product.



In determining the cross product, the order of operation is important. For example:

a = (1, 0, 0)
b = (0, 1, 0)
a x b = (0, 0, 1)
b x a = (0, 0, -1)

In Rhino's right-handed system, **a X b** is defined as a vector **c** that is perpendicular to both **a** and **b**, with a direction given by the right-hand rule (where a = index finger, b = middle finger, and result = thumb).

Vectors a and b are parallel if, and only if, a x b = 0

Cross product properties

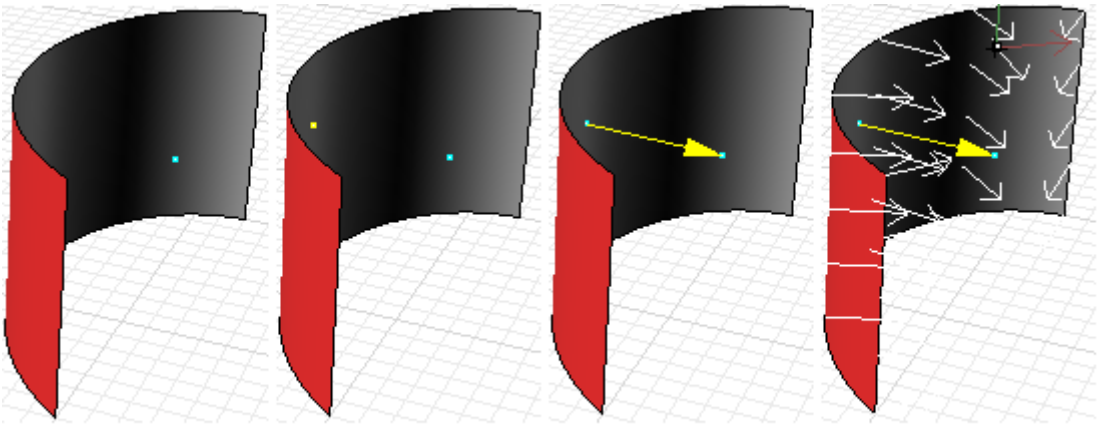
If **A**, **B**, and **C** are vectors and **i** is scalar then:

1. A X B = -B X A
2. (iA) X B = i(A X B) = A X (iB)
3. A X (B + C) = A X B + A X C
4. (A + B) X C = A X C + B X C
5. A . (B X C) = (A X B) . C
6. A X (B X C) = (A . C)B – (A . B)C

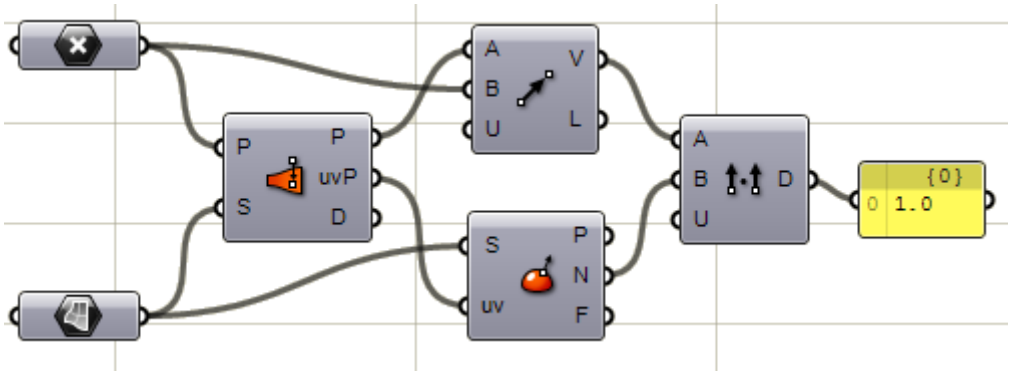
Geometry example

Given a point and a surface, determine whether the point is on the front or back of that surface.

Here are the steps to solve the problem:



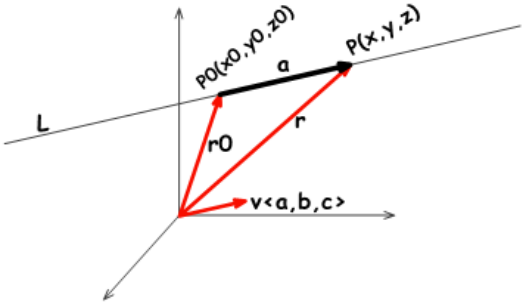
Here is the Grasshopper solution following the same steps. Note that in this case the **dot product > 0** which means the point is facing the front side of the surface. If the dot product were < 0 then the point would be on the back.



Vector equation of line

The vector line equation is used in 3-D modeling applications and computer graphics. Here is a description of that equation and how it might be used.

In the figure:



L = line
v = line direction
P0 = line position
r = r0 + a --- (1)
a = t * v --- (2)

Therefore from 1 and 2:

r = r0 + tv --- (3)

However, we can write (3) as follows:

<x,y,z> = <x0,y0,z0> + <ta, tb, tc>

<x,y,z> = <x0+ta, y0+tb, z0+tc>

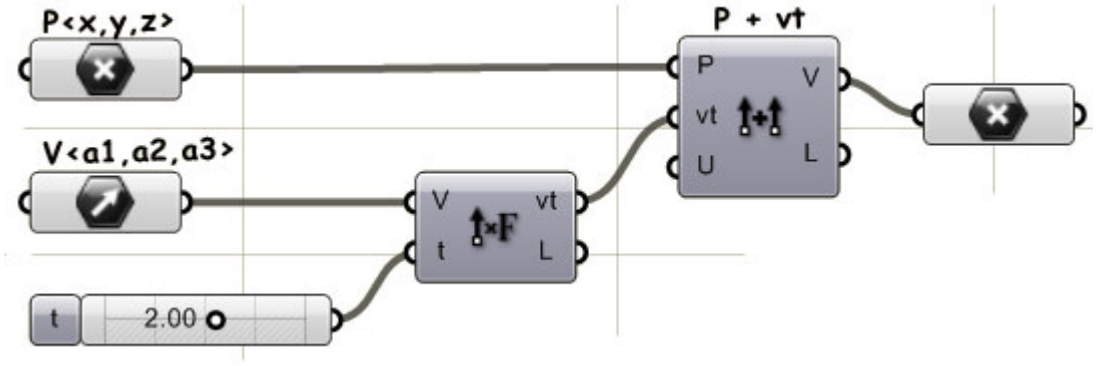
Therefore:

x = x0 + ta
y = y0 + tb
z = z0 + tc

Which is the same as:

P = P0 + tv

This is a Grasshopper definition to get any point on a line:

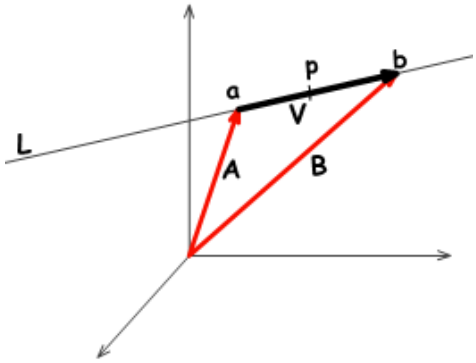


To illustrate when line equation is used, consider the following problem:

Given two points **A** and **B**, find the mid point using the line equation (P = P0 + tv).

We will show how to solve this problem and write a Grasshopper definition to solve the mid point problem. In the following figure, given points **a** and **b**, find point **p**. Notice that:

A is the position vector for point **a**
B is the position vector for point **b**
V is the vector going from **a** to **b**



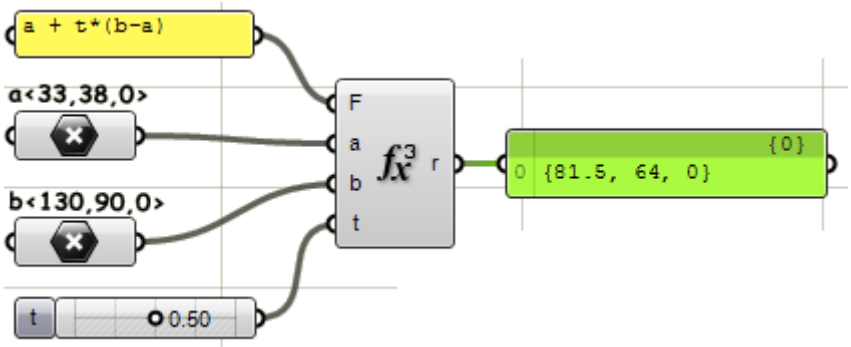
From vector addition property:

A + V = B, or
V = B - A

However, the line equation is: P = A + t*V, and since t=0.5 and V=B-A (from the above), then we can say:

P = A + 0.5(B-A)

Use the above equation to create a Grasshopper definition:



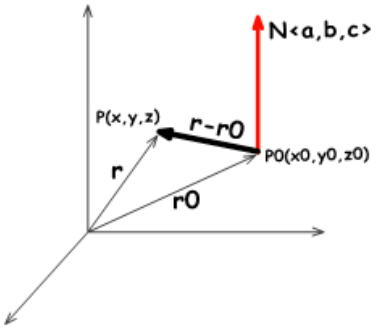
To find any point between **A** and **B**, we can change the **t** value.

Vector equation of a plane

In the figure above:

P0(x0,y0,z0) = a given point on the plane
r0<x0,y0,z0> = P0 position vector
N<a,b,c> = normal vector of the

P(x,y,z) = arbitrary point on the plane
r<x,y,z> = P position vector



Because dot product of two orthogonal vectors = 0, then:

N . (r - r0) = 0

Or we can say:

<a,b,c> . <x-x0 , y-y0 , z-z0 > = 0

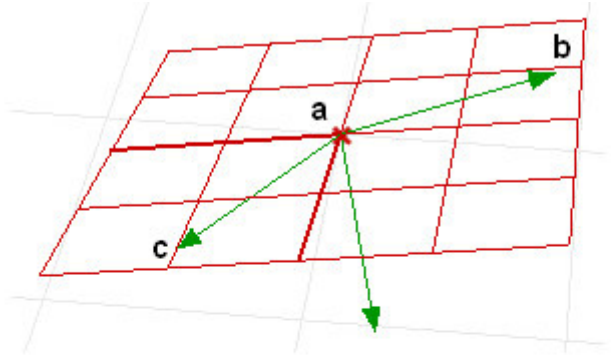
Solving the dot product gives the scalar equation of the plane:

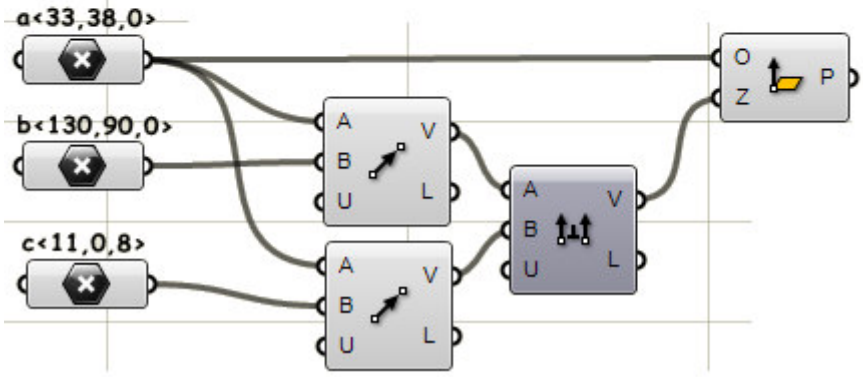
a(x-x0) + b(y-y0) + c(z-z0) = 0

How can we find the plane that goes through three points using the origin point and plane normal?

In order to find a plane, we need an origin and a plane normal. We have an origin point, which can be any of our three points, so how do we find the normal?

We know that the cross product of two vectors is a third vector normal to both of them. This would be the plane normal. Hence, this is how we may solve the question using Grasshopper:





2 Matrices and Transformations

Introduction

Although designers might not need to use matrix mathematics directly in computational design, knowledge of the basics is very useful for appreciating what is happening behind the scene. Transformation matrices are responsible for moving, rotating, projecting, and scaling objects. Matrices are also used for transformations between coordinate systems, for example from the 3-D world coordinate to the 2-D screen coordinate system.

We can define transformation as a function that takes a point (or a vector) and maps that point into another point (or vector). What is a matrix, and why do we need it for transformations?

A matrix is a rectangular array of numbers. A matrix dimension is m-by-n where:

m: number of rows
n: number of columns

Matrices have proven to be very a useful representation for transformations. Multiple transformations can be performed very quickly using this representation. The key is to find one format that can represent ALL transformations such as translation (move), rotation, and scale.

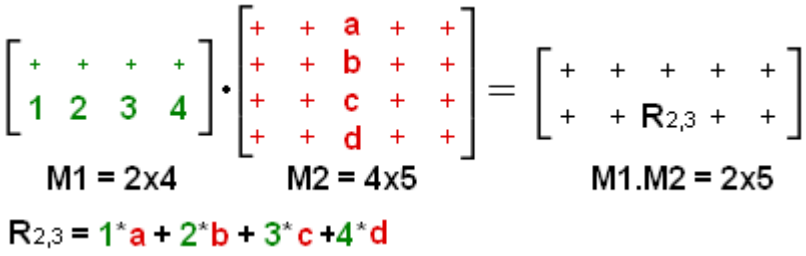
Matrix multiplication

Matrix multiplication is used to apply transformation to geometry. A series of transformation matrices is first multiplied to get a final transformation matrix that is in turn used for transforming geometry. Matrix multiplication is one of the frequently used matrix operations, so it is useful to elaborate on.

In order to multiply two matrices, they have to match. In other words, the number of columns of the first matrix must equal the number of rows of the second matrix. The resulting matrix has size equal to the number of rows from the first matrix and the number of columns from the second matrix. Here are few examples:

M_1=[2x4], M_2=[4x3] then M_1xM_2=[2x3]

We cannot multiply M2xM1 because they do not match.



Here are the general steps to multiply two matrices:

1. Make sure they match.
   That is, given two matrices of size M_1 =[axb], M_2=[cxd], **b** must be equal to **c**.

2. Find the sum of multiplying corresponding items from the first row of the left matrix with the first column of the right matrix to get the item at index(1,1) of the resulting matrix.
3. Repeat step 2 to get all items of the resulting matrix.
   For example the sum of multiplying third row of left matrix with second column of right matrix yields item at index (3,2) in the resulting matrix.

One special matrix is the identity matrix. The main property of this matrix is that if it is multiplied by any other matrix, it does not change its values as in the following:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*2+0*3+0*1+0*1 \\ 0*2+1*3+0*1+0*1 \\ 0*2+0*3+1*1+0*1 \\ 0*2+0*3+0*1+1*1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix}$$

Affine transformations

In this section, we will cover a special, but very common, type or transformation called *affine transformation*. When applied to geometry affine transformations have the property of preserving parallel line relationships, but not length or angles. Translation (move), rotation, scale, and shear are affine transformations.

Translation (move) transformation

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

P' = P + V

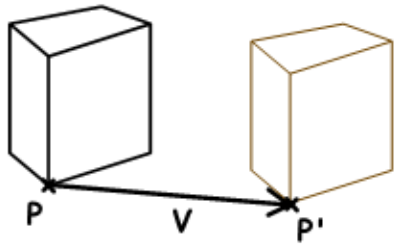
Suppose:

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

V<a,b,c> was a translation vector

then:

P'(x) = x + a
P'(y) = y + b
P'(z) = z + c



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, then we get the following:

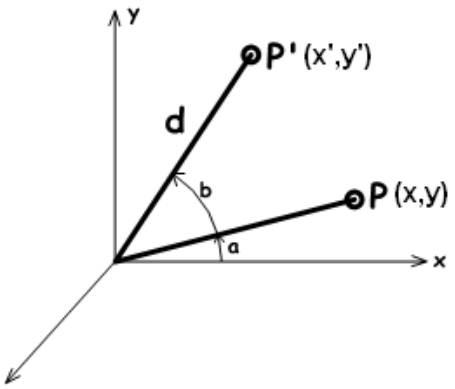
$$\begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 3 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1*2+0*3+0*1+2*1 \\ 0*2+1*3+0*1+2*1 \\ 0*2+0*3+1*1+2*1 \\ 0*2+0*3+0*1+1*1 \end{bmatrix} = \begin{bmatrix} 4 \\ 5 \\ 3 \\ 1 \end{bmatrix}$$

Rotation transformation

This example shows how to calculate rotation around z-axis and origin point using trigonometry, and then 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 3 and 4 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)

Using the homogenous coordinate system, the rotation matrix around **z-axis** looks like:

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

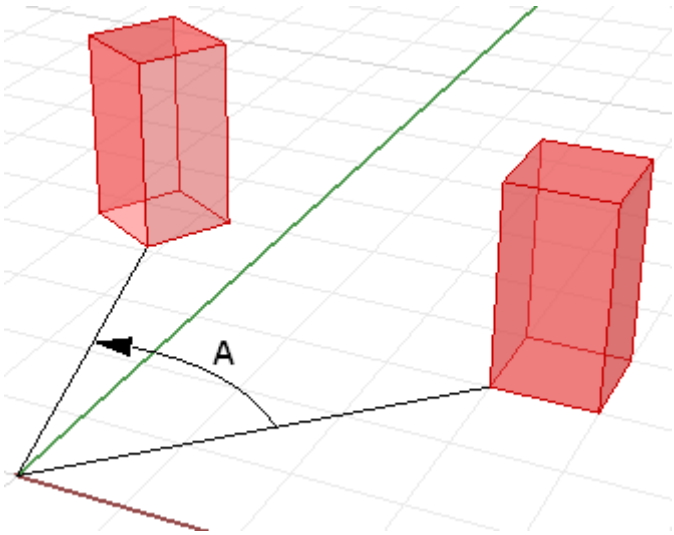
The rotation matrix around **x-axis** by **A**:

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

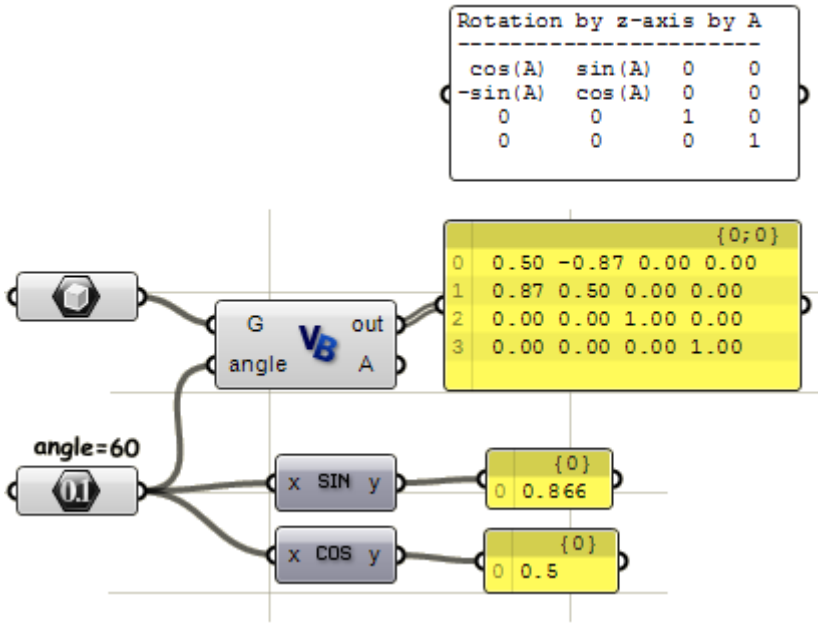
The rotation matrix around **y-axis** by **A**:

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

OpenNURBS™, the Rhino geometry library (http://www.openNURBS.org), contains a class called OnXform that handles transformations. It stores transformation matrix and performs matrix operations. The following is an example of how to rotate an object and examines OnXform matrix values to compare to the general format of the rotation matrix. You can use the same principle to examine other transformations.



Here is the Grasshopper definition for rotating geometry and output matrix values to compare with the general matrix format:



Here is the script in the VB component that generates a rotation matrix and prints data to the output window:

```
Private Sub RunScript(ByVal G As OnBrep,
                      ByVal angle As Double,
                      ByRef A As Object)

  'declare transformation
  Dim rotate As New OnXform

  'Set transformation
  Dim rotation_axis As New On3dVector(OnUtil.On_zaxis)
  Dim rotation_center As New On3dPoint(OnUtil.On_origin)
  rotate.Rotation(angle, rotation_axis, rotation_center)

  'Transform/move the input geometry
  G.Transform(rotate)

  'Print transformation matrix
  Dim xform_string As String
  Dim i As Int32
  Dim j As Int32
  Dim num_str As String
  Dim num As Double
  For i = 0 To 3
    xform_string = ""
    For j = 0 To 3
      num = rotate.m_xform(i, j)
      num_str = String.Format("{0:n2}", num)

      xform_string = xform_string + " " + num_str
    Next
    Print(xform_string)
  Next

  'Assign new move geometry to output
  A = G

End Sub
```

Scale transformation

We know that:

P' = ScaleFactor(S) * P

or:

P'.x = S * P.x

P'.y = S * P.y

P'.z = S * P.z

This is the matrix format for scale transformation.

$$\begin{bmatrix} xS & 0 & 0 & 0 \\ 0 & yS & 0 & 0 \\ 0 & 0 & zS & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear transformation

Shear in 3-D is measured along a pair of axes relative to the third axis. For example, a shear along a z-axis will not change geometry along that axis, but will alter x and y values. The general form of shear matrix can be represented by the following:

$$\begin{bmatrix} 1 & Sxy & Sxz & 0 \\ Syx & 1 & Syz & 0 \\ Szx & Szy & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformations in openNURBS

OnXform is a class in openNURBS for storing and manipulating transformation matrix. This includes, but not limited to, defining matrices to move, rotate, scale, or shear objects.

OnXform is a 4x4 matrix of double-precision numbers. The class also has functions that support matrix operations such as inverse and transpose. Here are few of the member functions related to creating different transformations.

```
Dim xform As New OnXform
xform.
```

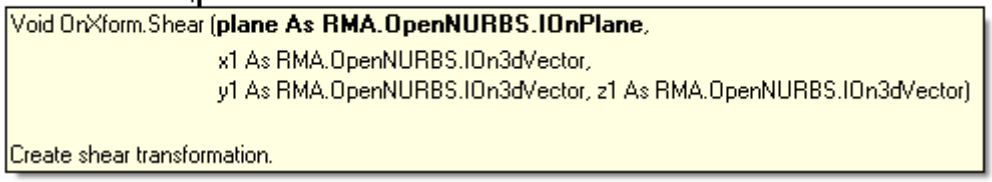
One nice auto-complete feature (available to all functions) is that once a function is selected, the auto-complete shows all overloaded functions. For example, translation (move) accepts either three numbers or a vector as shown in the picture.

```
Dim xform As New OnXform
xform.Translation(
```

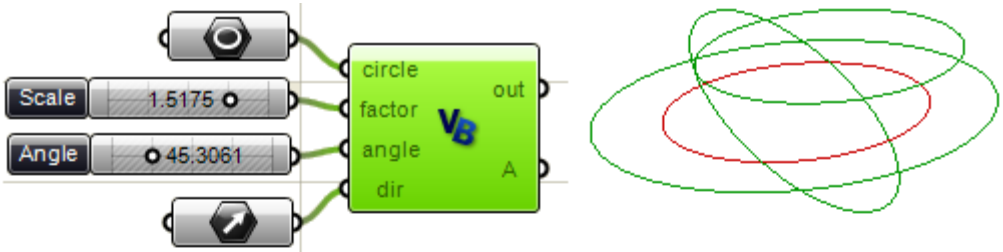
Here are few more OnXform functions:

```
Dim xform As New OnXform
xform.PlanarProjection(
```


```
Dim xform As New OnXform
xform.Shear(
```



The following example takes an input circle and outputs three circles. The first is scaled copy of the original circle, the second is rotated circle, and the third is translated one.




```
Sub RunScript(ByVal circle As OnCircle,
              ByVal factor As Double,
              ByVal angle As Double, ByVal dir As On3dVector)

  Dim circles As New List( Of OnCircle)

  'Scaled circle
  Dim scale As New OnXform
  scale.Scale(OnUtil.On_origin, factor)
  Dim s_circle As New OnCircle(circle)
  s_circle.Transform(scale)
  circles.Add(s_circle)

  'Rotated circle
  Dim rotate As New OnXform
  rotate.Rotation(angle, OnUtil.On_yaxis, OnUtil.On_origin)
  Dim r_circle As New OnCircle(circle)
  r_circle.Transform(rotate)
  circles.Add(r_circle)

  'Moved circle
  Dim move As New OnXform
  move.Translation(dir)
  Dim m_circle As New OnCircle(circle)
  m_circle.Transform(move)
  circles.Add(m_circle)

  'Assign output
  A = circles

End Sub
```

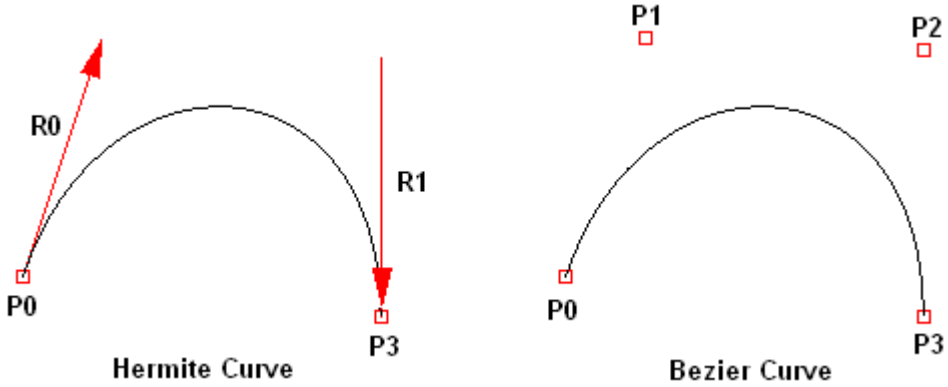
3 Parametric Curves

Introduction

Parametric curves are very compact and intuitive way to represent smooth curves. They also are very easy to modify compared to other representation formats. For example, polylines use first-degree piecewise approximation and therefore use a large number of points to store a curve that is somewhat smooth. In addition, curve manipulation is very tedious, especially if the smoothness of the curve needs to be maintained. The higher the accuracy of the curve, the heaver the curve storage grows and the more difficult to it is edit.

Cubic polynomial curves

Hermite[1] and Bézier[2] 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 directions at these points, while a Bézier curve is defined by four points. While they differ mathematically, they share similar characteristics and limitations.



A smoother and more continuous representation of curves is achieved with Uniform Non-Rational B-Splines[3] (UNRBS). UNRBS curves inherently have one more degree of continuity (a concept we will explain in the next section) than Hermite or Bézier curves and hence produce smoother curves.

There is yet another more powerful form of B-Spline called Non-Uniform Rational B-Spline[4] (NURBS). As the name implies, NURBS curves do not have to have uniformly spaced knot vectors[5]. Because of that, computation time to create and edit those curves is longer. The main advantage of NURBS over UNRBS is that NURBS curves are easier to interpolate through control points without introducing a straight-line segments (which is undesirable in most cases). In addition, it is possible to insert control points into NURBS curves, which you cannot do with UNRBS curves.

---

[1] For more details see: http://en.wikipedia.org/wiki/Cubic_Hermite_spline

[2] For more details, see: http://en.wikipedia.org/wiki/B%C3%A9zier_curve

[3] For more details, see: http://en.wikipedia.org/wiki/B-spline

[4] For more details, see: http://en.wikipedia.org/wiki/Non-uniform_rational_B-spline

[5] For more details, see: http://en.wikipedia.org/wiki/Spline_(mathematics)

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

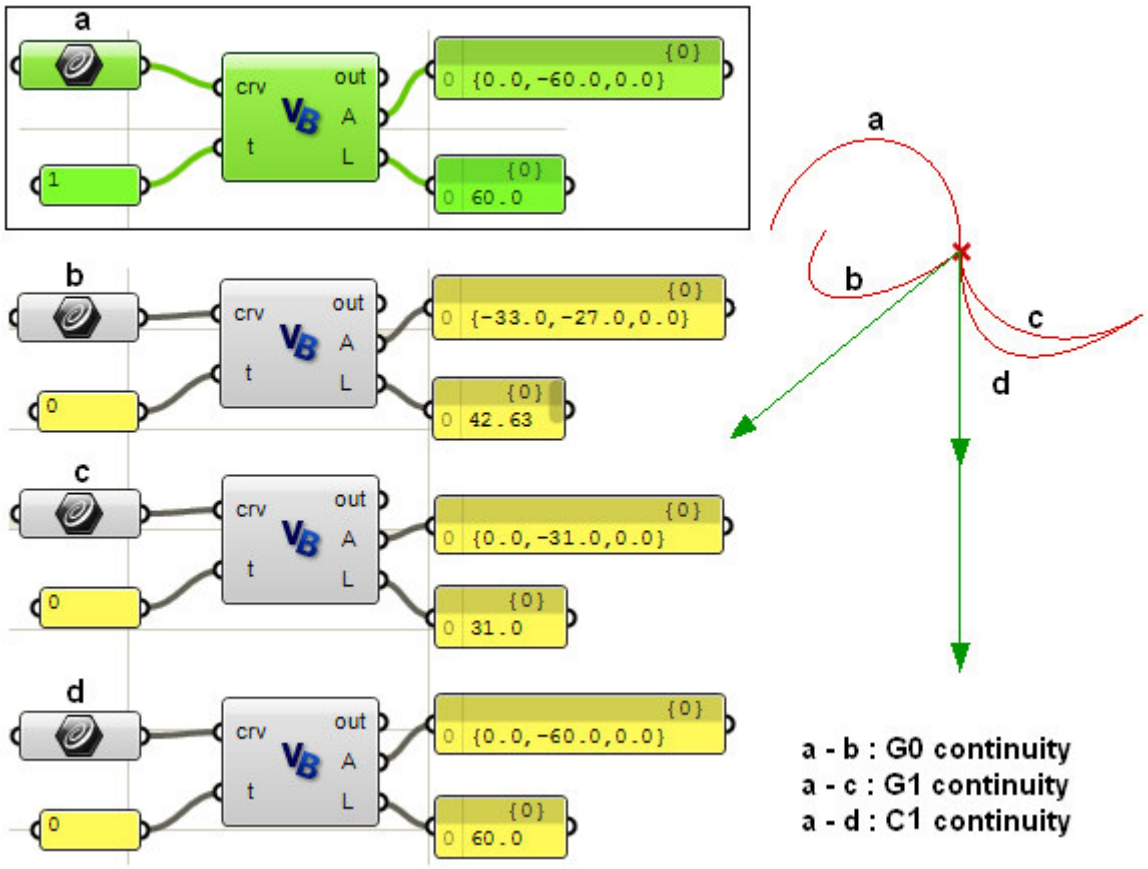
Geometric continuity

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 continuity)	Two curve segments joined together
G1 (Tangent continuity)	Direction of tangent at joint point is the same for both curve segments
G2 (or C1 Curvature continuity)	Direction & magnitude of tangent at joint point is equal for both curve segments
GN (or CN)	The N derivative of the two curves at joint point is equal

The following definition compares curves continuities between curve **a** and curves **b**, **c**, and **d**:



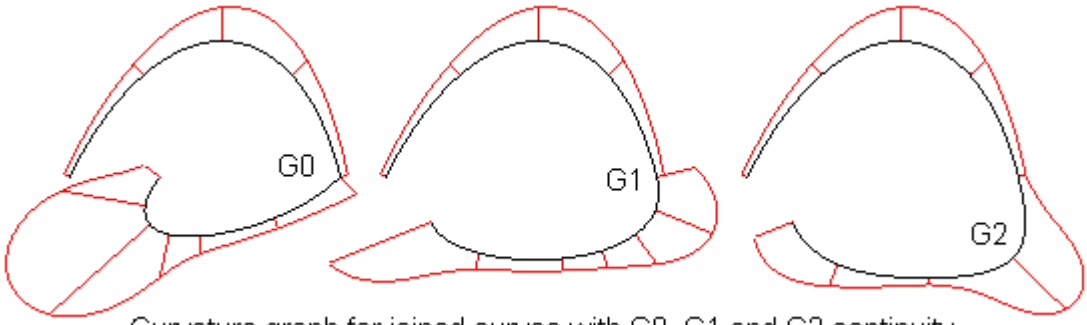
Curvature

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.*

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.

The best approximating circle can lie either to the left or to the right of the curve. If we care about this, then 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 value of blended curves also indicates continuity among these curves as in the following illustration.



Curvature graph for joined curves with G0, G1 and G2 continuity

For surfaces, normal section 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.

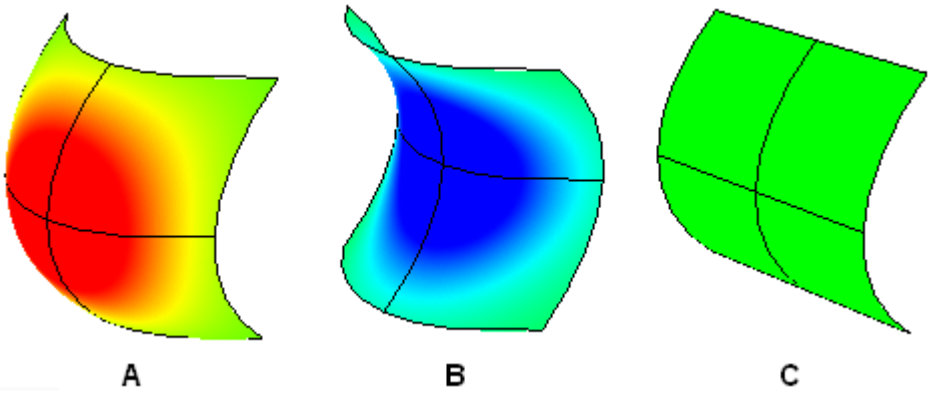
If we look at all directions in the tangent plane to the surface at our point, and we compute the normal section curvature in all these directions, then there will be a maximum value and a minimum value.

Principal curvatures

The principal curvatures of a surface at a point are the minimum and maximum of the normal curvatures at that point. (Normal curvatures are the curvatures of curves on the surface lying in planes including the tangent vector at the given point.) The principal curvatures are used to compute the *Gaussian* and *mean* curvatures of the surface.

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 at a single point, whereas the tangent plane of any point with negative Gaussian curvature cuts the surface. Any point with zero mean curvature has negative or zero Gaussian curvature.



A: Positive curvature when surface is bowl-like

B: Negative curvature when surface s saddle-like

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

Mean curvature

The mean curvature of a surface at a point is one half the sum 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. Surfaces with constant mean curvature everywhere are often referred to as constant mean curvature (CMC) surfaces.

CMC surfaces have the same mean curvature everywhere on the surface.

Physical processes which can be modeled by 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.

Minimal surfaces are the subset of CMC surfaces where the curvature is zero everywhere.

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.

Algorithms for evaluating parametric curves

De Casteljau[6] algorithm for evaluating cubic Bézier curves

Named after its inventor, Paul De Casteljau, this algorithm evaluates Bézier curves using a recursive method.

[6] De Casteljua's algorithm details are found in http://en.wikipedia.org/wiki/De_Casteljau%27s_algorithm

We will show the algorithm to find any point on the curve at parameter **t** using De Casteljau algorithm using Grasshopper. We will need the following input:

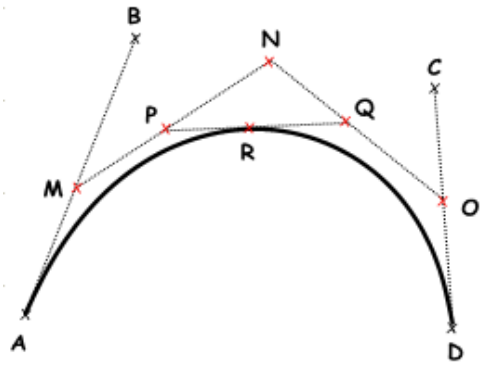
4 points A, B, C, D

t, which is a parameter on the curve within curve domain (0-1)

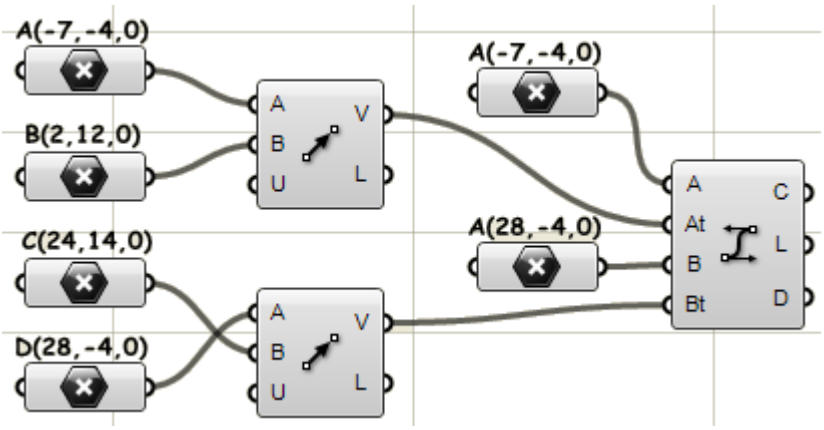
Output:

Point on curve that is at t parameter

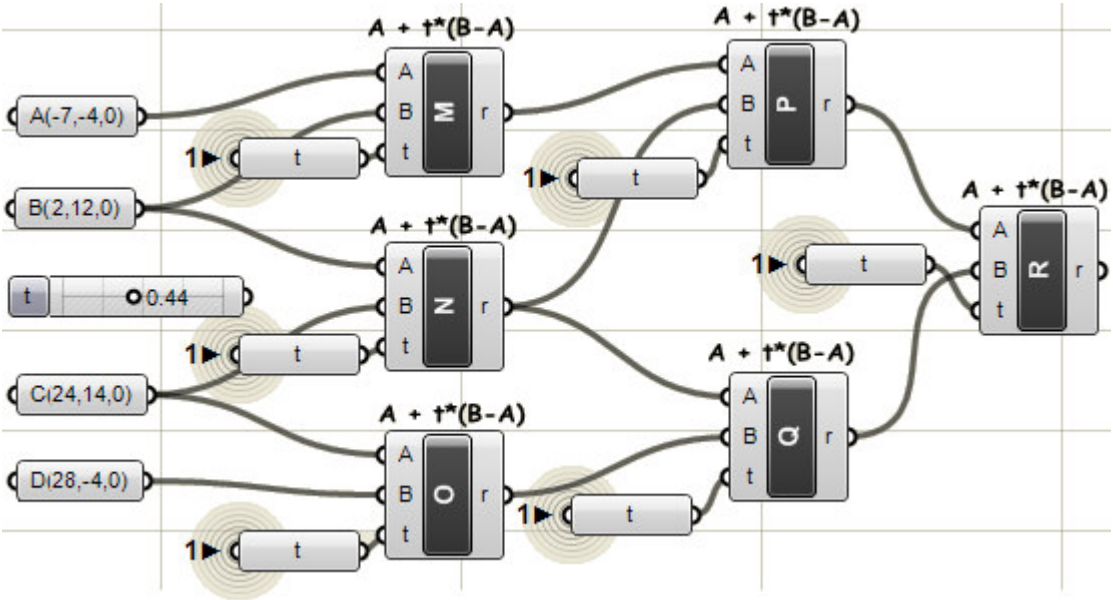
Solution steps:



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



This is the Grasshopper definition to evaluate a parameter on a Bézier curve using De Casteljau algorithm. Note that you can change t value between 0 and 1 to find points between the start and end of the Bézier curve.



De Boor[7] algorithm for evaluating NURBS curves

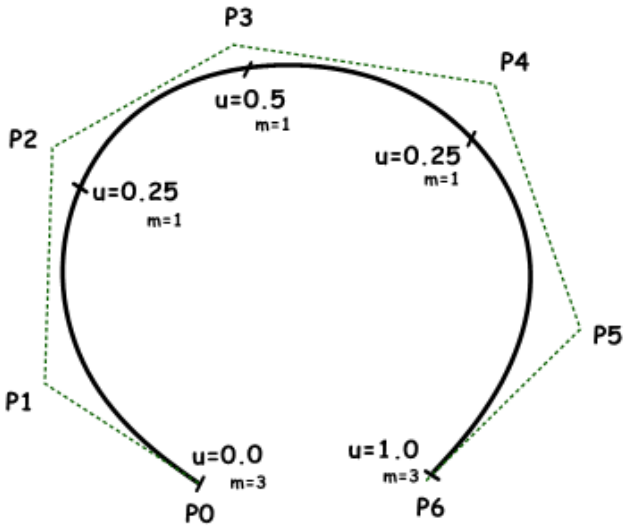
DeBoor's algorithm is a generalization of De Casteljau algorithm for Bezier curves. It is numerically stable and is widely used to evaluate a point on NURBS curves in 3D applications. The following is an example to evaluate a point on a degree 3 NURBS curve using DeBoor's algorithm[8].

Input

7 control points P_0 to P_6

Knot vector:

u0 = 0.0
u1 = 0.0
u2 = 0.0
u3 = 0.0
u4 = 0.25
u5 = 0.5
u6 = 0.75
u7 = 1.0
u8 = 1.0
u9 = 1.0
u10 = 1.0



Output:

Point on curve that is at u=0.4

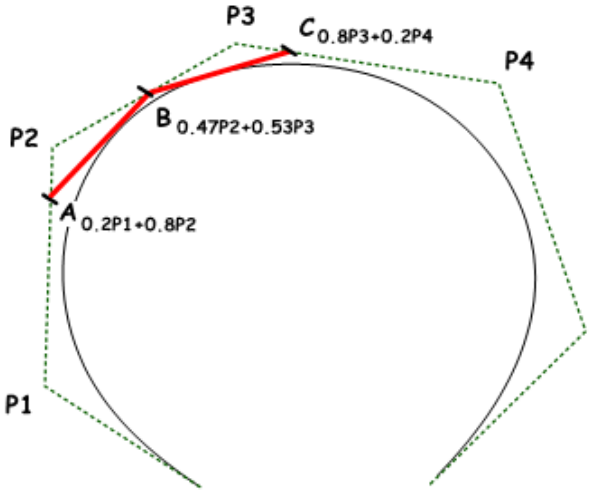
Solution steps:

1. Calculate coefficients for the first iteration:

$Ac = (u - u_2) / (u_{2+3} - u_2) = 0.8$
$Bc = (u - u_3) / (u_{3+3} - u_3) = 0.53$
$Cc = (u - u_4) / (u_{4+3} - u_4) = 0.2$

2. Calculate points using coefficient data:

$\mathbf{A} = 0.2\mathbf{P_1} + 0.8\mathbf{P_2}$
$\mathbf{B} = 0.47\,\mathbf{P_2} + 0.53\,\mathbf{P_3}$
$\mathbf{C} = 0.8\,\mathbf{P_3} + 0.2\,\mathbf{P_4}$



[7] De Boor's algorithm details are found in http://en.wikipedia.org/wiki/De_Boor's_algorithm

[8] The general description of the algorithm and the example detailsare found in: http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html

3. Calculate coefficients for the second iteration:

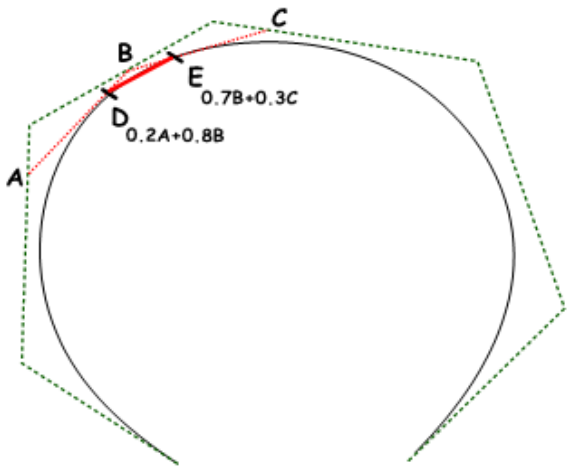
$Dc = (u - u_3) / (u_{3+3-1} - u_3) = 0.8$

$Ec = (u - u_4) / (u_{4+3-1} - u_4) = 0.3$

4. Calculate points using coefficient data:

D = 0.2**A**+ 0.8**B**

E = 0.7**B** + 0.3**C**

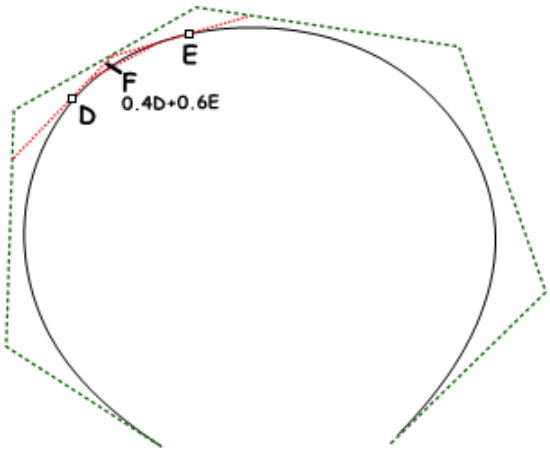


5. Calculate the last coefficient

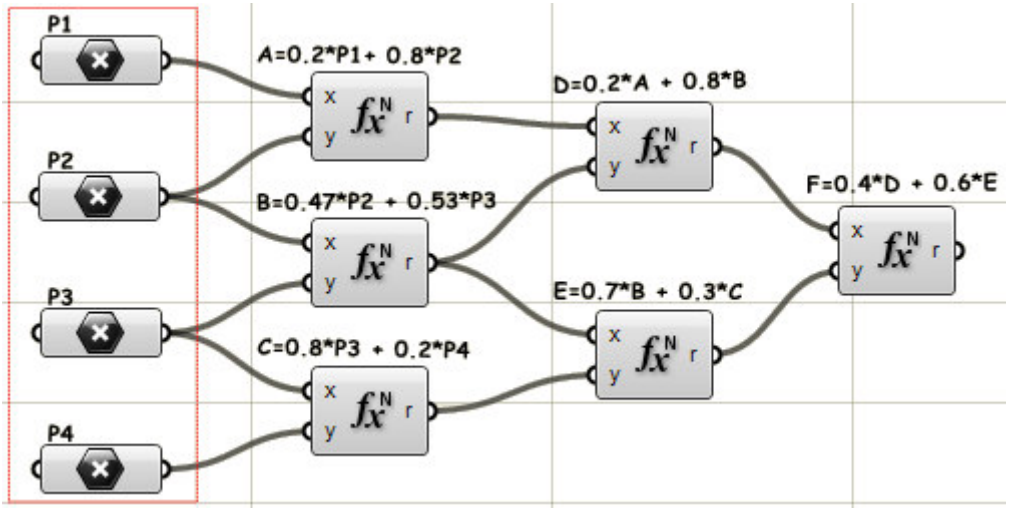
$Fc = (u - u_4)/ (u_{4+3-2} - u_4) = 0.6$

6. Find the point on curve at u=0.4 parameter

F= 0.4**D** + 0.6**E**



This is the Grasshopper definition to evaluate the u=0.4 parameter on a NURBS curve using DeBoor's algorithm.



NURBS curves

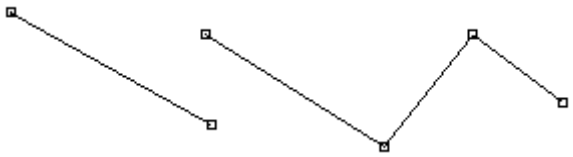
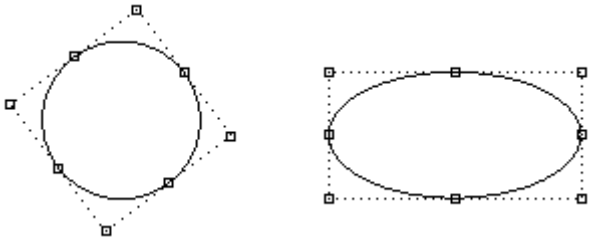
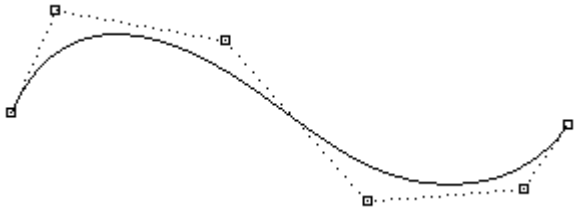
NURBS is an accurate mathematical representation of curves and surfaces that is highly intuitive to edit.

There are many books and references for those of you interested in an in-depth reading about NURBS (http://en.wikipedia.org/wiki/NURBS). A basic understanding of NURBS is necessary to help you use the NURBS modeler more effectively.

Four attributes define a NURBS curve: degree, control points, knots, and evaluation rules:

Degree

Degree is a whole positive number. Rhino allows working with any degree starting with 1. Degree 5 is also common, but the degrees above 5 are not very useful in the real world. Following are few examples of curves and their degree:

Lines and **polylines** are degree 1 NURBS curves. Order = 2 (order = degree + 1)	
Circles and **ellipses** are examples of degree 2 NURBS curves. They are also rational or non-uniform curves. Order = 3.	
Free-form **curves** are usually represented as degree 3 NURBS curves. Order = 4	

Control points

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

Control points have an associated number called a **weight**. With a few exceptions, weights are positive numbers. When a curve's control points all have the same weight (usually 1), the curve is called non-rational. We will have an example showing how to change the weights of control points interactively in Grasshopper.

Knots or knot vector

Each NURBS curve has a list of numbers associated with it that is called a knot vector. Knots are a little harder to understand and set, but luckily there are SDK functions that do the job for you. Nevertheless, there are few things that will be useful to learn about knot vectors.

Knot multiplicity

The number of times a knot value is duplicated. Any knot value cannot be duplicated more times than the curve degree.

Full-multiplicity knot or fully multiple knot

A knot duplicated a number of times equal to curve degree. Clamped curves have knots with full multiplicity at the two ends of the curve. This is why end control points coincide with curve end points. If there were full-multiplicity knot in the middle of the knot vector, then the curve will go through the control point and there would be a kink.

Simple knot

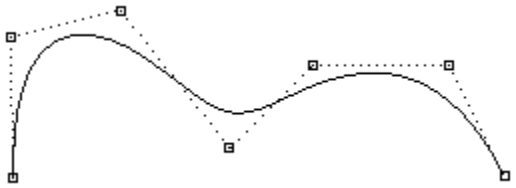
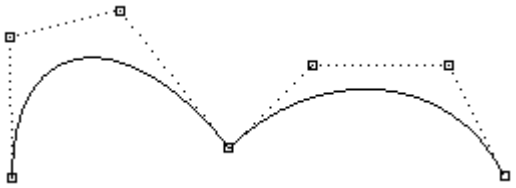
A knot with value appearing only once.

Uniform knot vector

A uniform knot vector satisfies two conditions:

1. The number of knots equals number of control points + degree – 1.
2. Knots start with a full-multiplicity knot, are followed by simple knots, and terminate with a full-multiplicity knot. The values are increasing and equally spaced. This is typical of clamped curves. Periodic curves work differently as we will see later.

Here are two curves with identical control points but different knot vectors:

Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,2,3,5,5,5)	
Degree = 3 Number of control points = 7 knot vector = (0,0,0,1,1,1,4,4,4) **Note**: Full knot multiplicity in the middle creates a kink and the curve is forced to go through the associated control point.	

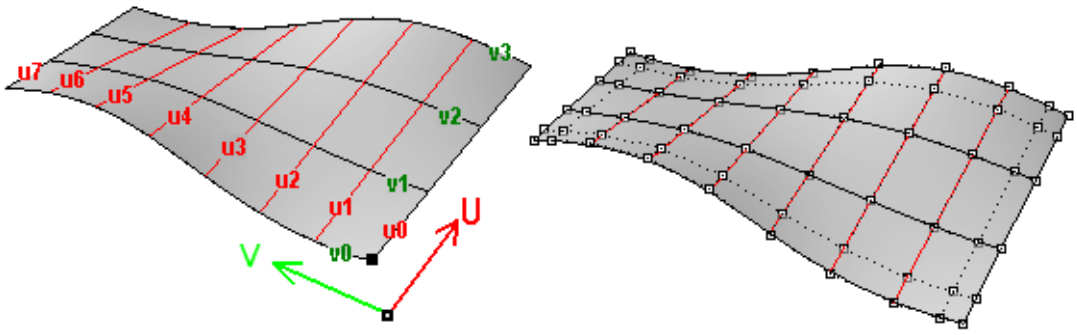
Evaluation rule

The evaluation rule uses a mathematical formula that takes a number and assigns a point. The formula involves the degree, control points, and knots.

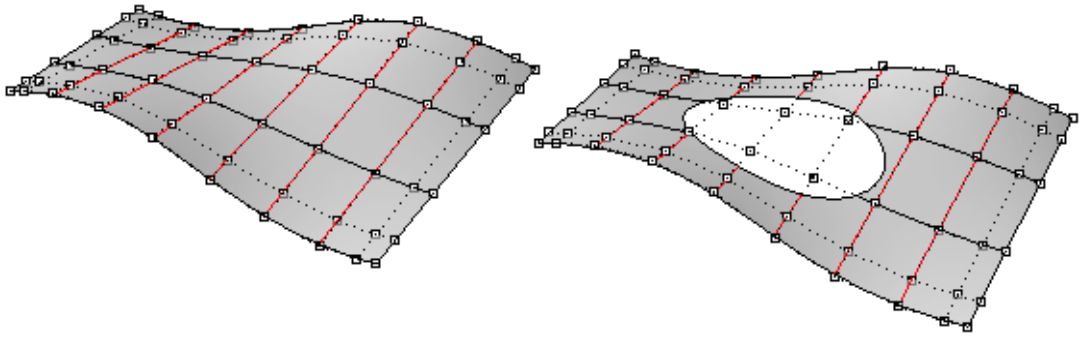
Using this formula, SDK functions can take a curve parameter and produce the corresponding point on that curve. A parameter is a number that lies within the curve domain. Domains are usually increasing and they consist of two numbers: minimum domain parameter (m_t(0)) that is usually the start of the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

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 can be trimmed or untrimmed. Trimmed surfaces use an underlying NURBS surface and closed curves to cut a specific shape of that surface. Each surface has one closed curve that defines the outer border (*outer loop*) and 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.

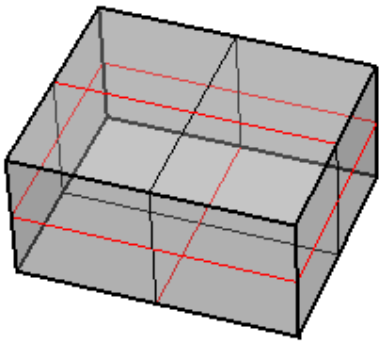


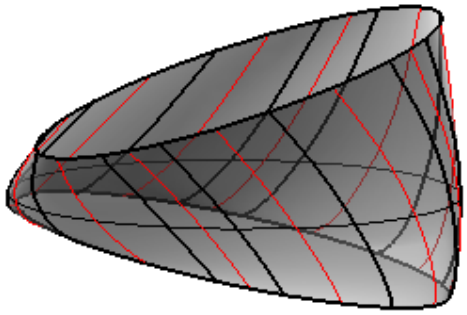
The surface on the left is untrimmed. The surface on the right is the same surface trimmed with an elliptical hole. Notice that the NURBS structure of the surface does not change when trimming.

Polysurfaces

A polysurface consists of two or more (possibly trimmed) surfaces joined together. Each surface has its own parameterization and u-,v-directions that do not have to match. Polysurfaces and trimmed surfaces are represented using what is called boundary representation (brep for short). It basically describes surface, edge, and vertex geometry with trimming data and relationships among different parts. For example, the brep describes each face, its surrounding edges and trims, normal direction relative to the surface, relationship with neighboring faces and so on. Breps can also be called *solids*.

OnBrep is probably the most complex data structure in OpenNURBS, and it might not be easily digested, but fortunately, there are many tools and global functions that come with the Rhino SDK to help create and manipulate breps.



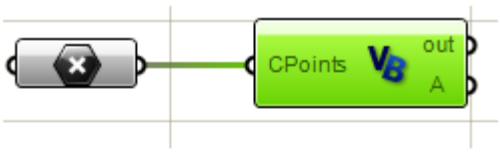


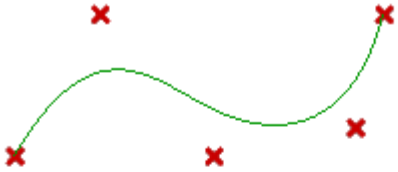
NURBS curves in openNURBS

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

- Dimension, which is typically 3.
- Order: Curve degree + 1.
- Control points (array of points).
- Knot vector (array of numbers).
- Curve type (clamped or periodic).

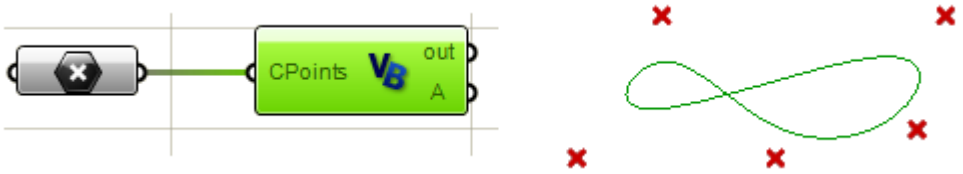
Functions help create the knot vector, as we will see shortly, so basically you need to decide on the degree and have a list of control points and you are good to go. The following example creates a clamped curve.





```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim nc As New OnNurbsCurve

  'Create open (Clamped) Nurbs Curve
  nc.CreateClampedUniformNurbs(dimension, order, CPoints.ToArray())

  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

For smooth, closed curves, you should create periodic curves. Using same input control points and curve degree, the following example shows how to create a periodic curve.

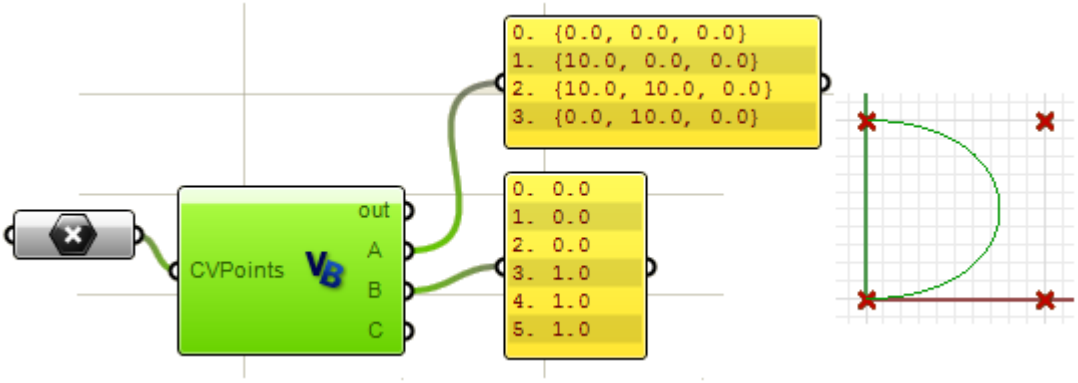



```
Sub RunScript(ByVal CPoints As List(Of On3dPoint))

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim nc As New OnNurbsCurve

  'Create closed (Periodic) Nurbs Curve
  nc.CreatePeriodicUniformNurbs(dimension, order, CPoints.ToArray())

  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

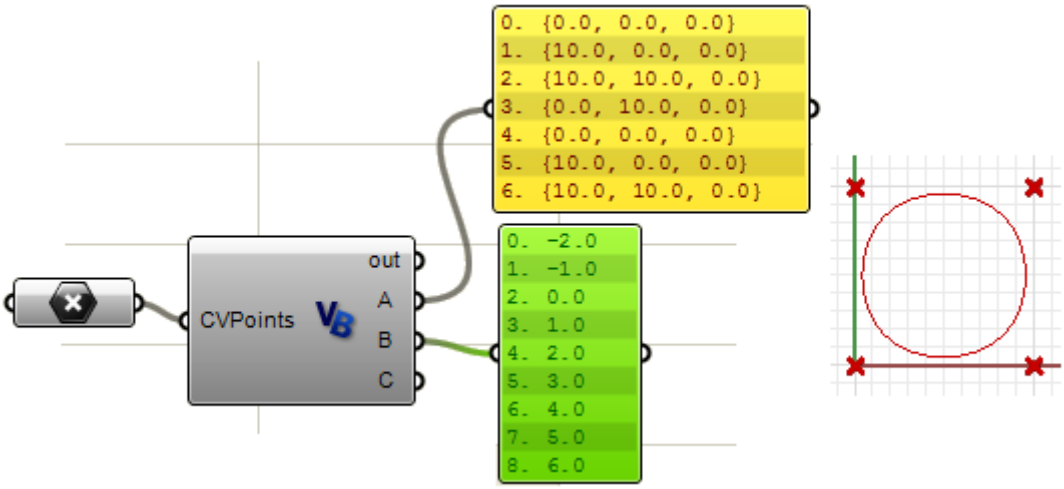
Clamped vs. periodic NURBS curves

Clamped curves are curves (usually open) where curve ends 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.

The following component creates clamped NURBS curve and outputs:

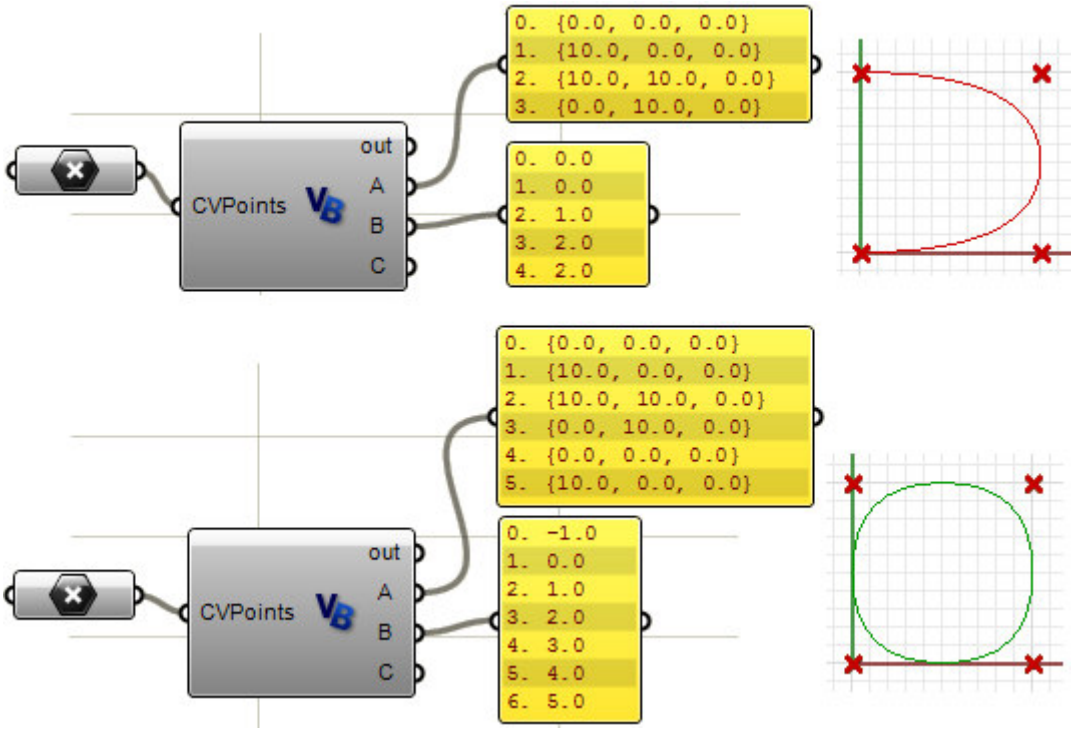


Here is the periodic curve using same input (control points and curve degree):



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

Here are same examples with degree 2 curves. As you may have guessed, number of control points and knots of periodic curves change when degree changes.

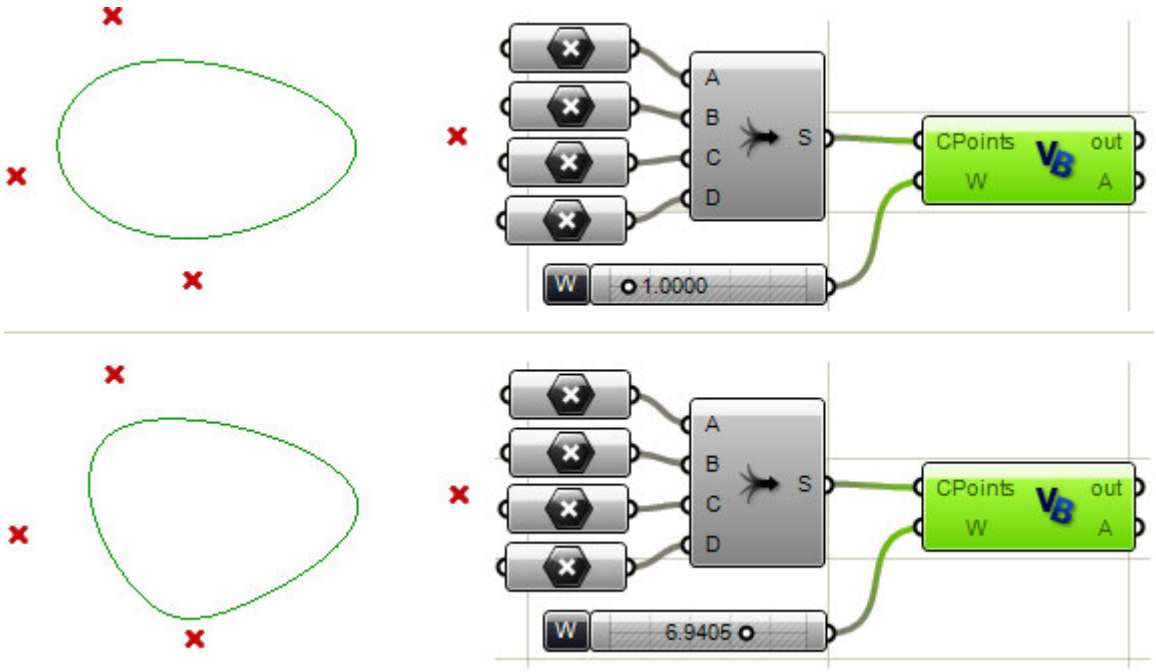


This is the code used to navigate through control points and knots in the previous examples:

```
'Output control points
Dim count As Double = nc.CVCount()
Dim i As Integer
Dim cvs As New List( Of On3dPoint )
For i = 0 To count - 1
  Dim cv As New On3dPoint(0, 0, 0)
  nc.GetCV(i, cv)
  cvs.Add(cv)
Next

'Output knots
Dim knots As New List( Of Double )
count = nc.KnotCount()
For i = 0 To count - 1
  knots.Add(nc.Knot(i))
Next
```

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 how to modify weights of control points interactively in Grasshopper.



```
Sub RunScript(ByVal CPoints As List(Of On3dPoint), ByVal W As Double)

  Dim i As Integer

  'Create nurbs curve
  Dim dimension As Integer = 3
  Dim order As Integer = 4
  Dim cv_count As Integer = CPoints.Count
  Dim nc As New OnNurbsCurve
  nc.CreatePeriodicUniformNurbs(dimension, order, CPoints.ToArray())
  nc.MakeRational()

  'Assign weights
  Dim cv As New On3dPoint
  For i = 0 To cv_count - 1
    nc.GetCV(i, cv)
    cv = cv * W
    nc.SetCV(i, cv)
    nc.SetWeight(i, W)
  Next


  'Assign curve to the output value A
  If( nc.IsValid() ) Then
    A = nc
  End If
End Sub
```

References

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.

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