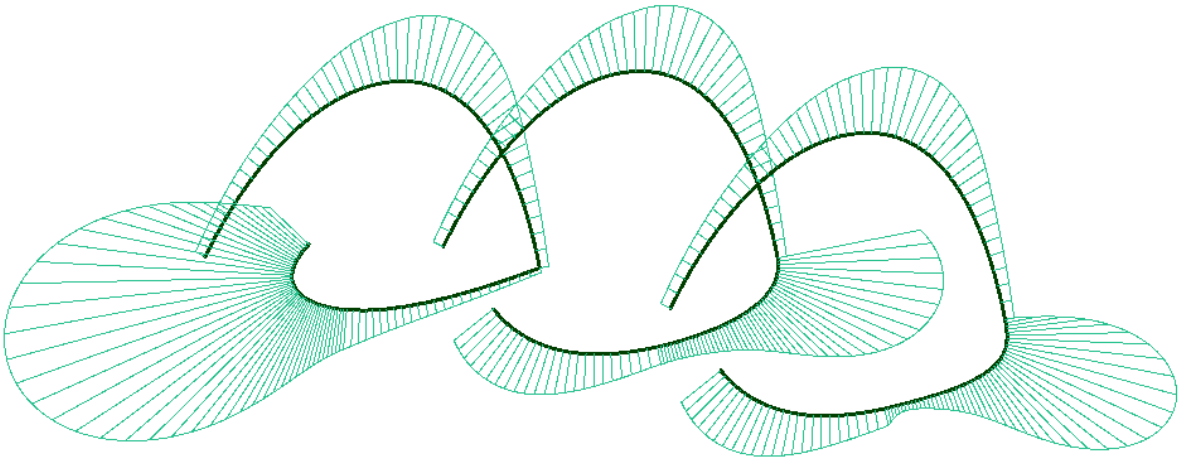


Essential Mathematics

for Computational Design - Fourth Edition

（日本語版）

Rajaa Issa
Robert McNeel & Associates

序文

Essential Mathematics for Computational Design では，3D モデリングやコンピュータグラフィックスの計算手法を効果的に開発するために必要となる数学概念の基礎を，デザインの専門家向けに紹介します．これは，網羅的にではなく，基本的で最もよく用いるような概念を概観する内容となっています．

この資料は，高校数学以降の知識がほとんど，または全くないデザイナーを対象としています．概念はすべて，Rhinoceros®（Rhino）のジェネレーティブモデリング環境である Grasshopper®（GH）を使用して，視覚的に説明されています．詳細については，www.rhino3d.com，および www.grasshopper3d.com をご覧ください．

内容は，3 つの章に分かれています． 第 1 章では，ベクトル表現，ベクトル演算，直線および平面の式を含むベクトルの数学について解説します．第 2 章では，行列の演算や変換について解説します．第 3 章では，NURBS 曲線やその連続性および曲率の概念に特に焦点を当て，パラメトリック曲線について詳細に解説します．NURBS 曲面とポリサーフェスについても触れています．

Robert McNeel&Associates の Dale Lear 博士による，優れた徹底的なテクニカルレビューに感謝します．彼の貴重なコメントは，このエディションの作成に役立ちました．テクニカルライティングのレビューとドキュメントのフォーマットについては，Robert McNeel&Associates の Margaret Becker 氏にも感謝します．

Rajaa Issa
Robert McNeel & Associates

目次

1 Vector Mathematics（ベクトル数学）

ベクトルは，速度や力のように大きさと向きを持った量を意味します．3 次元座標系のベクトルは，3 つの実数 を用いて次のように表されます．

v = <a1, a2, a3>

ベクトル表現

この資料では，太字の小文字はベクトルを表すとします．ベクトルの成分は，山括弧で囲みます．大文字は点を表すとします．点の座標は常に丸括弧で囲みます．

ベクトルは，ある座標系の任意の座標とアンカーポイント（始点）のセットを用いた線分表現として視覚化できます．矢印はベクトルの方向を示します．

例えば，ある 3 次元座標系において，x 軸方向に平行で大きさが 5 単位のベクトルがある場合，ベクトルは次のように書くことができます．

v = <5, 0, 0>

このベクトルを表現するには，座標系にアンカーポイントが必要となります．例えば，次の図の矢印はすべて異なる場所に位置しますが，同じベクトルを等しく表しています．

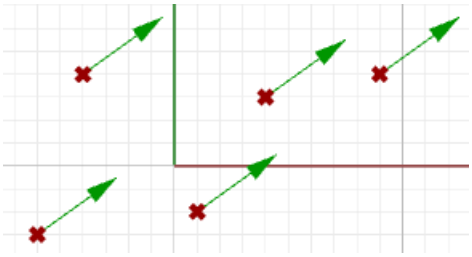


図 (1): 3 次元座標系におけるベクトル表現.

ある 3 次元ベクトル ***v*** = < a1, a2, a3 > の 成分 a1, a2, a3 が実数であるとすると，任意の点 A(x,y,z) から点 B(x+a1, y+a2, z+a3) への線分は，すべてベクトル ***v*** の同等の表現です．

では，与えられたベクトルを表現する線分の終点を定義するにはどうすれば良いでしょうか？

ここで，始点 A とベクトル ***v*** を次のように定義してみます．

A = (1, 2, 3)

v = <5, 6, 7>

ベクトルの先端(B) の位置は，始点 A とベクトル ***v*** のそれぞれの成分を足し合わせることで求められます．

B = A + **v**

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

B = (6, 8, 10)

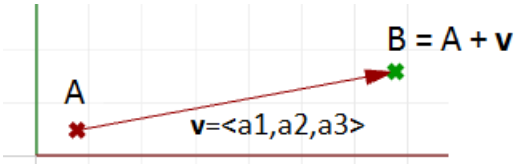


図 (2): ベクトルと始点とベクトルの先端に位置する点の関係.

位置ベクトル

ある特別なベクトル表現では，原点（0,0,0）をベクトルの始点として使用します（これを位置ベクトルと呼びます）．位置ベクトル **v** = <a1,a2,a3> は，原点と B の 2 点間の線分で次のように表せます.

原点 = (0,0,0)

B = (a1,a2,a3)

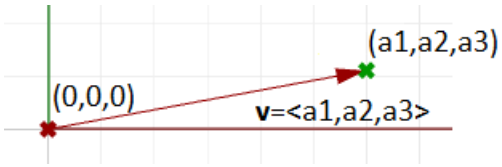


図 (3): 位置ベクトル. 終点の座標はそれぞれ対応するベクトル成分と等しい.

> ベクトル ***v***= < a1,a2,a3 >があるとき，位置ベクトルは原点 (0,0,0) から 点 (a1,a2,a3)への特別な線分表現です.

ベクトルと点

ベクトルと点を混同しないようにしましょう．これらの概念は全く異なります．前述のように，ベクトルは方向と長さを持つ量を表し，点は位置を示します．例えると，北方向はベクトルであり，北極は位置（点）を表すようなものです.

次のような同じ成分を持つベクトルと点がある場合,

v = <3,1,0>

P = (3,1,0)

ベクトルと点はそれぞれ次のように描画できます.

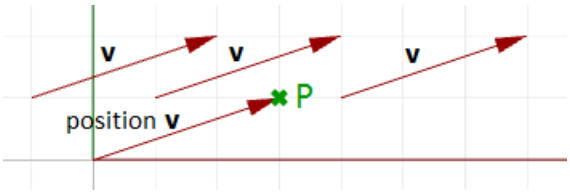


図 (4): ベクトルは方向と長さを定義し，点は位置を定義します.

ベクトルの長さ

前述のように，ベクトルには長さがあります．ベクトル **a** の長さは，**|a|** と表記します．以下がその例です.

a = <4, 3, 0>

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

$|\mathbf{a}| = 5$

一般に，ベクトル **a**<a1,a2,a3> の長さは 次のように計算します．

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

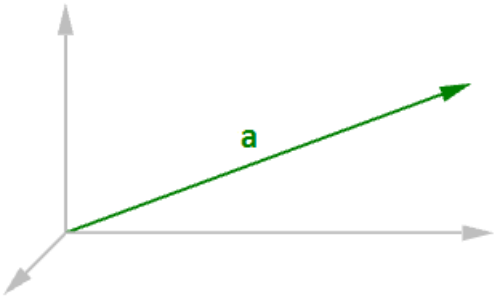


図 (5): ベクトル長さ.

単位ベクトル

単位ベクトルは，長さが 1 に等しいベクトルです．単位ベクトルは，ベクトルの方向を比較するためによく用いられます．

> 単位ベクトルは，長さが 1 に等しいベクトルです．

単位ベクトルを計算するには，まずベクトルの長さを求め，ベクトル成分をそれぞれ求めたベクトル長さで割り算します．以下がその例です．

a = <4, 3, 0>

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

$|\mathbf{a}|$ = 5 単位長さ

b を **a** の単位ベクトルとすると，

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 単位長さ

一般に以下が成り立ちます．

a = <a1, a2, a3>

a の単位ベクトル = <a1/|**a**|, a2/|**a**|, a3/|**a**|>

図 (6): 単位ベクトルは, あるベクトルの 1 単位長さのベクトルに等しくなります.

ベクトルの演算

ベクトルのスカラー演算

ベクトルのスカラー演算とは，ベクトルにスカラー量（方向を持たない大きさのみで表される量）を掛け算することです．例えば，次のようになります．

a = <4, 3, 0>
2***a*** = <2*4, 2*3, 2*0>
2***a*** = <8, 6, 0>

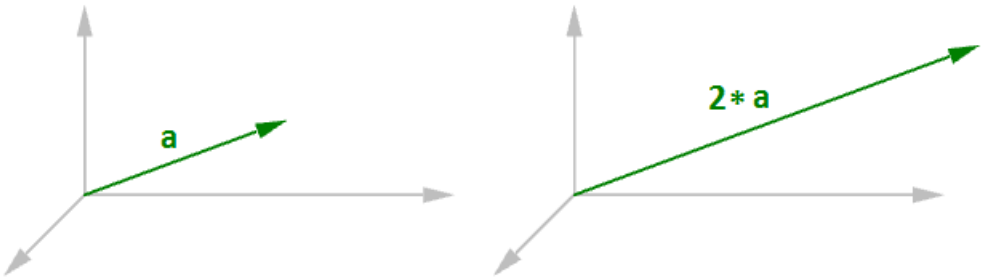


図 (7): ベクトルのスカラー演算.

一般に，ベクトル **a** = <a1, a2, a3> と 実数 *t* があるとき以下のようになります．
*t****a*** = < *t*a1, *t*a2, *t*a3 >

ベクトルの和

ベクトルの和では，2 つのベクトルから 3 つ目のベクトルが求まります．それぞれの成分を足し算すると，ベクトルの和となります．

> ベクトルの和は，それぞれ対応する成分を足し算することで求まる．

例えば，ある 2 つのベクトルがあった場合，次のようになります．

a<1, 2, 0>
b<4, 1, 3>
a+**b** = <1+4, 2+1, 0+3>
a+**b** = <5, 3, 3>

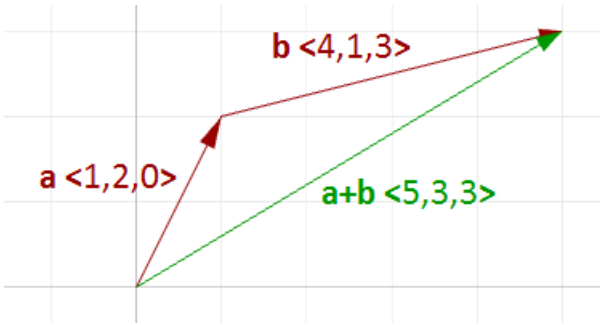


図 (8): ベクトルの和.

一般に，2 つのベクトル **a** と **b** の和は，次のように計算できます．

a = <a1, a2, a3>

b = <b1, b2, b3>

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

ベクトルの和は，2 つ以上のベクトルの平均の方向を求めるときに便利です．この場合は通常同じ長さのベクトルを用います．同じ長さのベクトルを用いた場合と異なる長さのベクトルを用いた場合のベクトルの和の結果の違いを以下の例に示します．

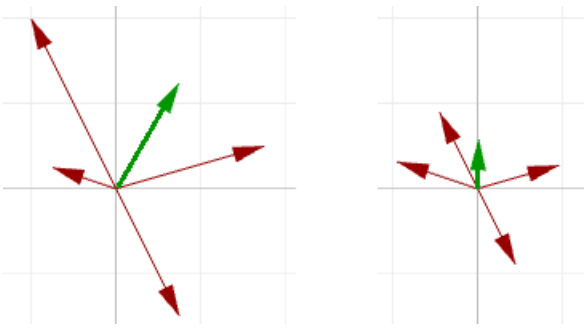


図 (9): 複数のベクトルの平均の方向を求めるときに，異なる長さのベクトルを足し合わせた場合 (左) と同じ長さのベクトルを足し合わせた場合 (右)．

入力するベクトルは，同じ長さとは限りません．平均の方向を求めるには，入力したベクトルの単位ベクトルを用いる必要があります．前述の通り，単位ベクトルは長さ 1 のベクトルです．

ベクトルの差

ベクトルの差でも，2 つのベクトルから 3 つ目のベクトルが求まります．対応する成分を引き算することで差を求めます．例えば，2 つのベクトル **a** と **b** があり，**a** から **b** を引き算する場合は次のようになります．

a<1, 2, 0>

b<4, 1, 4>

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

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

b から **a** を引き算する場合は，異なる結果となります．

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

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

ベクトル **b - a** の長さはベクトル **a – b** の長さと同じで，方向は反対になります．

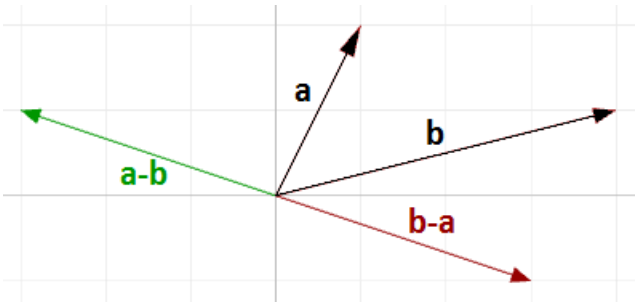


図 (10): ベクトルの差.

一般に，2 つのベクトル **a** と **b** があるとき，**a - b** は次のように計算できます．

a = <a1, a2, a3>

b = <b1, b2, b3>

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

ベクトルの差は，点同士を結ぶベクトルを求めるときによく用いられます．したがって，位置ベクトル **b** の先端から位置ベクトル **a** の先端へのベクトルを求めたい場合，図（11）のように，ベクトルの差 (**a**-**b**) を用います．

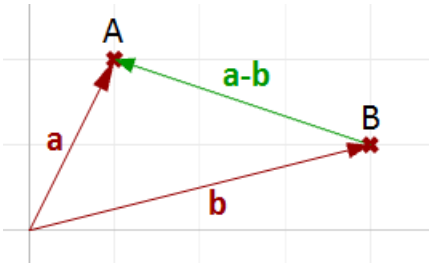


図 (11): 2 点間のベクトルを求めるためにベクトルの差を利用.

ベクトルの性質

ベクトルには 8 つの性質があります．**a**，**b**，**c** がベクトルを，*s* と *t* がスカラー量を表すとき，以下が成り立ちます．

a + **b** = **b** + **a**

a + 0 = **a**

s * (**a** + **b**) = *s* * **a** + *s* * **b**

s * *t* * (**a**) = *s* * (*t* * **a**)

a + (**b** + **c**) = (**a** + **b**) + **c**

a + (-**a**) = 0

(*s* + *t*) * **a** = *s* * **a** + *t* * **a**

1 * **a** = **a**

ベクトルの内積

内積（ドット積）では，2 つのベクトルから 1 つのスカラー量が求まります.

例えば，次のようなベクトル **a**，**b** がある場合を考えます.

a = <1, 2, 3>

b = <5, 6, 7>

このとき内積はそれぞれの成分を掛け算した和を表します.

a · **b** = 1 * 5 + 2 * 6 + 3 * 7

a · **b** = 38

一般に，2 つのベクトル **a**，**b** には以下が成り立ちます.

a = <a1, a2, a3>

b = <b1, b2, b3>

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

2 つのベクトルが互いに同じような方向を向いている場合，内積は正の値となります．2 つのベクトルが互いに反対方向を向いている場合，内積は負の値となります．

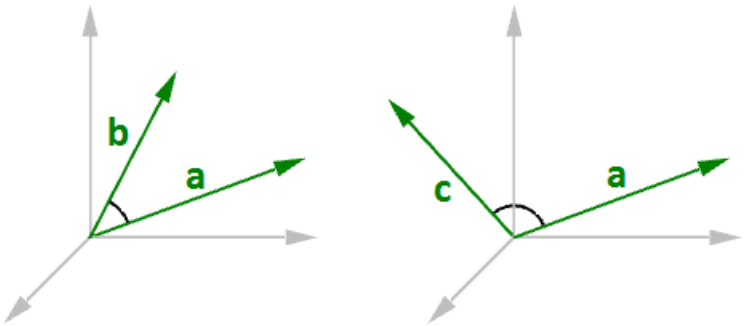


図 (12): 2 つのベクトルが同様の方向を向いている場合 (左), 内積の結果は正の値となります．2 つのベクトルが反対方向を向いている場合 (右), 内積の結果は負の値となります．

単位ベクトル間の内積の結果は常に，-1 から +1 の間の値となります．

a = <1, 0, 0>

b = <0.6, 0.8, 0>

a・**b** = (1 * 0.6, 0 * 0.8, 0 * 0) = 0.6

さらに，同じベクトルとの内積は，そのベクトルの長さを 2 乗した値に等しくなります．

a = <0, 3, 4>

a・**a** = 0 * 0 + 3 * 3 + 4 * 4

a・**a** = 25

ベクトル **a** の長さの 2 乗は次の通りです．

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

$|\mathbf{a}| = 5$

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

内積と長さ・角度の関係

2 つのベクトルの内積とそれらの間の角度の間には次の関係があります．

> 2 つの有効な（0 ではない）単位ベクトルの内積は，それらの間の角度の余弦に等しくなります．

一般に以下が成り立ちます．

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

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

ここで，θ はベクトル間の角度を表します．

もしベクトル **a**，**b** が単位ベクトルなら，よりシンプルな形で書けます．

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

また，角度が 90 度のとき cos の値が 0 となるので，次のことが言えます．

> ***a*・*b*** = 0 のとき，かつそのときに限り，ベクトル ***a***，***b*** は直行します．

例として, 2 つの直交ベクトル（World X 座標軸と World Y 座標軸）の内積を計算すると，結果は 0 になります.

x = <1, 0, 0>
y = <0, 1, 0>
x · **y** = (1 * 0) + (0 * 1) + (0 * 0)
x · **y** = 0

また次のように，内積は一方のベクトルをもう一方のベクトルに射影した長さに関係します.

a = <5, 2, 0>
b = <9, 0, 0>
unit(**b**) = <1, 0, 0>
a · unit(**b**) = (5 * 1) + (2 * 0) + (0 * 0)
a · unit(**b**) = 5 (ベクトル **a** を **b** へ射影した長さに等しくなります)

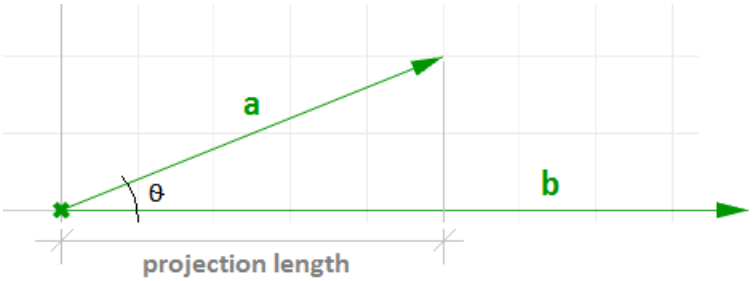


図 (13): 内積は，あるベクトルを 0 でない単位ベクトルへ射影した長さに等しくなります.

一般に，あるベクトル **a** と 0 でないベクトル **b** があるとき， **a** の **b** への射影 *pL* は内積を用いて次のように表せます.

pL = |**a**| * cos(*θ*)
pL = **a** · unit(**b**)

内積の性質

a，**b**，**c** がベクトルを，*s* が スカラー量を表すとき，以下が成り立ちます.

a · **a** = | **a** |^2
a · (**b** + **c**) = **a** · **b** + **a** · **c**
0 · **a** = 0
a · **b** = **b** · **a**
(*s* * **a**) · **b** = *s* * (**a** · **b**) = **a** · (*s* * **b**)

ベクトルの外積

外積（クロス積）では，2 つのベクトルから，両方に直交する 3 つ目のベクトルが求まります.

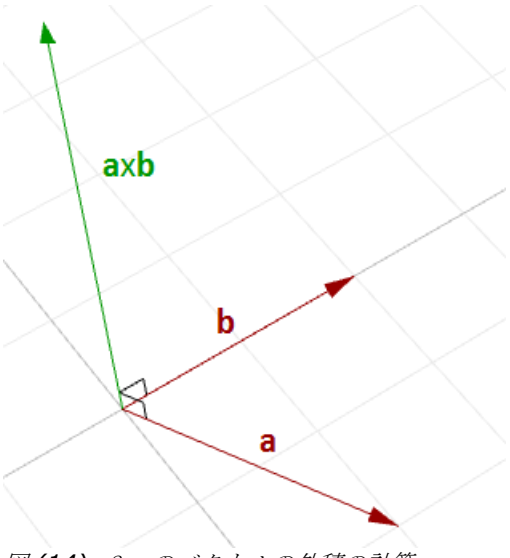


図 (14): 2 つのベクトルの外積の計算.

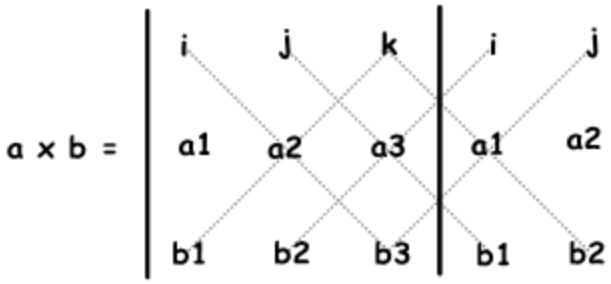
例えば，World XY 平面上に 2 つのベクトルがある場合，外積は World XY 平面に垂直なベクトルとなり，World Z 座標軸の正の方向，あるいは負の方向になります．以下はその例です.

a = <3, 1, 0>
b = <1, 2, 0>
a × **b** = < (1 * 0 – 0 * 2), (0 * 1 - 3 * 0), (3 * 2 - 1 * 1) >
a × **b** = <0, 0, 5>

> ベクトル **a** x **b** は， **a** と **b** の両方に直交します.

2 つのベクトルの外積を手計算する必要はおそらくありませんが，その方法に興味がある場合は，以下を参考にしてください．必要ない場合は，この節はスキップしても問題ありません．外積 a×b は，行列式を用いて定義されます．3 次元座標の基底ベクトル（**i**，**j**，**k**）を用いて行列式を計算する方法の簡単な説明を次に示します.

i = <1, 0, 0>
j = <0,1, 0>
k = <0, 0, 1>



2 つのベクトル **a**<a1, a2, a3> と **b**<b1, b2, b3> の外積は，上のダイヤグラムを用いて次のように計算できます.

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 >

外積と長さ・角度の関係

外積ベクトルの長さは，2 つのベクトル間の角度に関係します．角度が小さい（正弦が小さい）ほど，外積ベクトルの長さは短くなります. 外積ベクトルでは，演算の順序が重要になります. 例えば次の通りです.

a = <1, 0, 0>

b = <0, 1, 0>

a × **b** = <0, 0, 1>

b × **a** = <0, 0, -1>

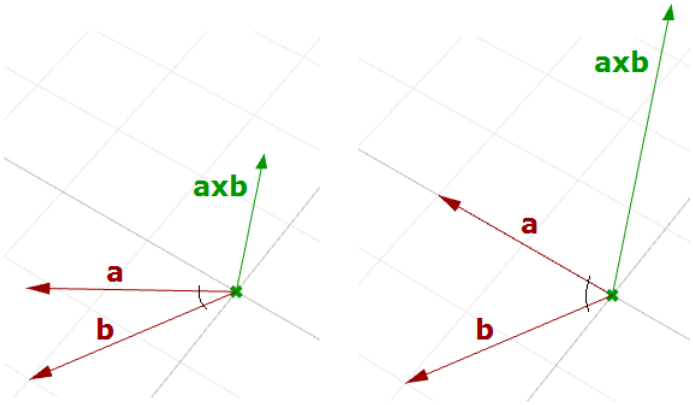
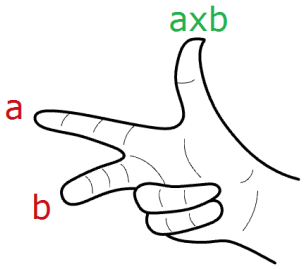


図 (15):2 つのベクトルの角度と外積ベクトルの長さの関係.

Rhino は右手座標系となっており，ベクトル **a** × **b** の方向は右手の法則（**a** = 人差し指, **b** = 中指，**a** × **b** = 親指)で与えられます.



一般に，2 つの 3 次元ベクトル **a**，**b** に対して以下が成り立ちます．

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

ここで，θ は位置ベクトル **a** と **b** の角度を表します．

a と **b** が単位ベクトルなら，外積の長さは 2 つのベクトル間の正弦に等しくなります．

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

外積は，2 つのベクトルが平行かどうかを判定するのに役立ちます．平行なら外積の結果はゼロベクトルとなるからです．

> ***a*** x ***b*** = 0 のとき，かつそのときに限り，ベクトル ***a***，***b*** は平行です．

外積の性質

a，**b**，**c** がベクトルを，s がスカラー量を表すとき，以下が成り立ちます．

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

直線のベクトル方程式

直線のベクトル方程式は，3D モデリングアプリケーションやコンピュータグラフィックスで用いられます．

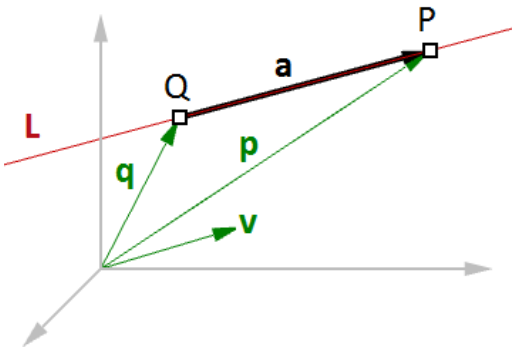


図 (16): 直線のベクトル方程式.

例えば，ある方向を持つ直線とその直線上の点がわかれば，次のようにベクトルを用いることでその直線上のあらゆる点を表現することができます．

L = 直線
v = <a, b, c>　（直線の方向を表す単位ベクトル）
Q = (x0, y0, z0)　（直線上の既知の点）
P = (x, y, z)　（直線上の求めたい点）

このとき，

$$\mathbf{a} = t * \mathbf{v} \quad \text{--- (2)}$$
$$\mathbf{p} = \mathbf{q} + \mathbf{a} \quad \text{--- (1)}$$

式(1) (2)より,

p = **q** + *t* * **v** --- (3)

式(3) は次のようにも書けます.

<x, y, z> = <x0, y0, z0> + <*t* * a, *t* * b, *t* * c>

<x, y, z> = <x0 + *t* * a, y0 + *t* * b, z0 + *t* * c>

すなわち,

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

これは次のように点を用いた場合と同じです.

P = Q + *t* * **v**

> 直線上の点 Q と方向 **v** が与えられると，その直線上の任意の点 P は，直線のベクトル方程式 P = Q + *t* * **v** を用いて計算できます．ここで，*t* はスカラー量です.

ほかの例としては，2 点間の中点を求める場合にも利用することができます．以下は，直線のベクトル方程式を用いて中点を見つける方法を示しています.

q は点 Q の位置ベクトル

p は点 P の位置ベクトル

a は点 Q から点 P へのベクトル

ベクトルの差演算から,

a = **p**-**q**

直線のベクトル方程式から,

M = Q + *t* * **a**

求めたいのは中点なので,

t = 0.5

したがって,

M = Q + 0.5 * **a**

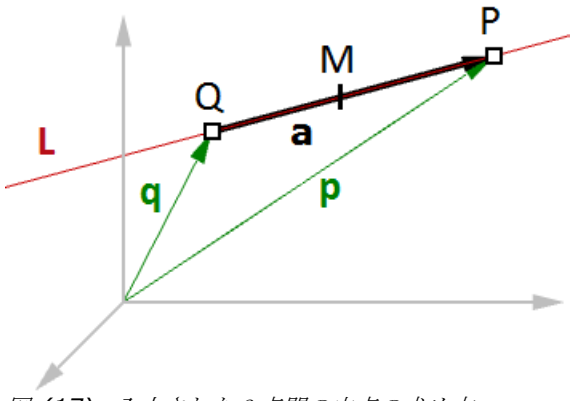


図 (17): 入力された 2 点間の中点の求め方.

より一般的に書くと，以下の一般式から *t* の値を 0～1 で変化させることで，点 Q と点 P 間にある点を求めることができます.

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

> 2 つの点 Q と点 P が与えられると，2 点間の任意の点 M は，方程式 M = Q + *t* * (P - Q)を使用して計算されます．*t* は 0～1 の数値です.

平面のベクトル方程式

平面を定義する 1 つの方法は，点と平面に垂直なベクトルを用いる方法です．そのベクトルは通常，平面の法線ベクトル（Normal vector）と呼ばれます．法線は，平面に対して上方向を指します.

法線ベクトルを計算する方法の 1 つは，平面上の任意の 3 点から求める方法です.

図(16)から,

A = 平面上の 1 つ目の点

B = 平面上の 2 つ目の点

C = 平面上の 3 つ目の点

また,

a = 点 A の位置ベクトル

b = 点 B の位置ベクトル

c = 点 C の位置ベクトル

とすると，法線ベクトル **n** は次のように求められます.

n = (**b** - **a**) × (**c** - **a**)

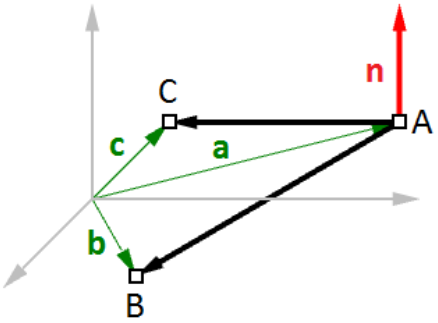


図 (18): ベクトルと平面.

また，ベクトルの内積から，平面のスカラー方程式を導出することができます.

n · (**b** - **a**) = 0

ここで,

n = <a, b, c>

b = <x, y, z>

a = <x0, y0, z0>

と置くと以下のように書き換えられます.

<a, b, c> · <x-x0, y-y0, z-z0 > = 0

上の内積の式を解くと，平面の一般的なスカラー方程式が得られます.

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

チュートリアル

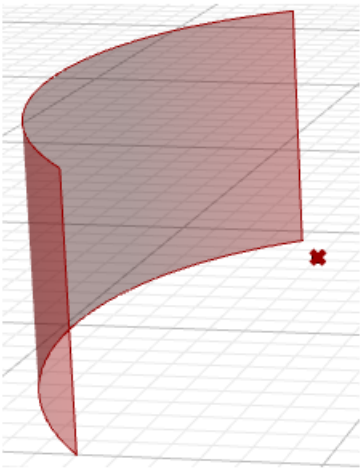
この章で紹介した概念はすべて，モデリングのときに直面する一般的なジオメトリの問題に直接適用できます．以下は，Rhino と GH 上でこの章で学習した概念を活用するステップバイステップのチュートリアルです.

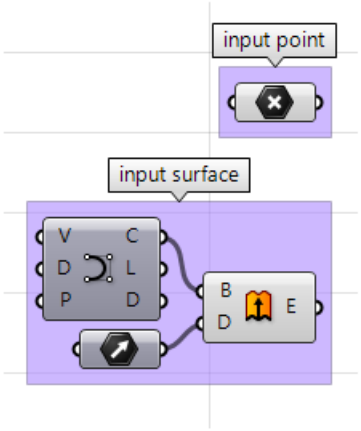
サーフェスの向き

ある点とサーフェスが与えられたとき，その点がサーフェスの表側と裏側どちらにあるかはどのように求めれば良いでしょう？

Input:

1. サーフェス
2. 点





Parameters:

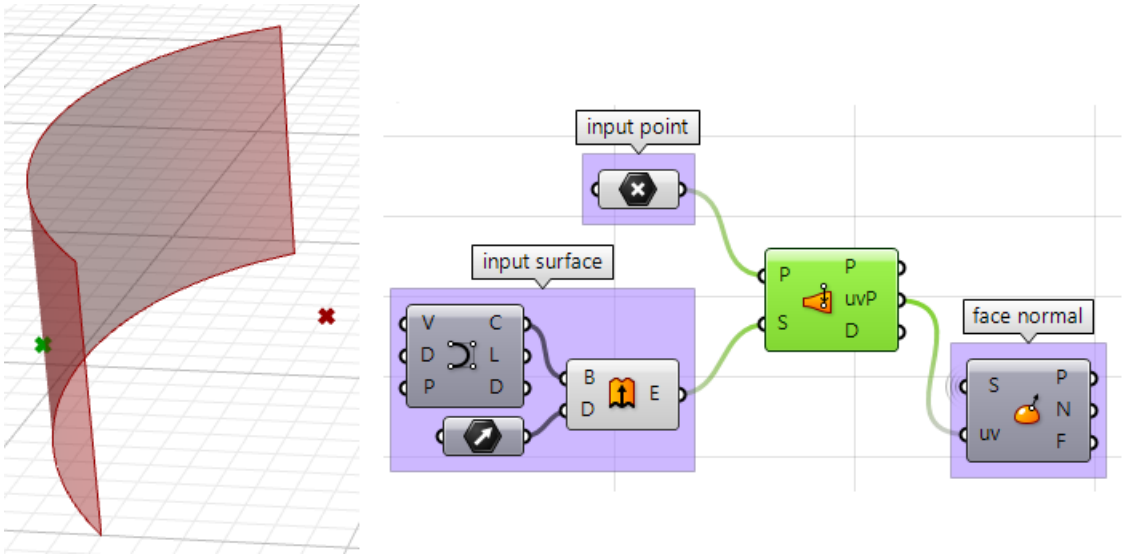
面の向きはサーフェスの法線方向で定義されます．ここで，以下の情報が必要になります.

- 入力した点に最も近いサーフェス位置での法線方向.
- 入力した点への最も近いサーフェス位置からのベクトル方向.

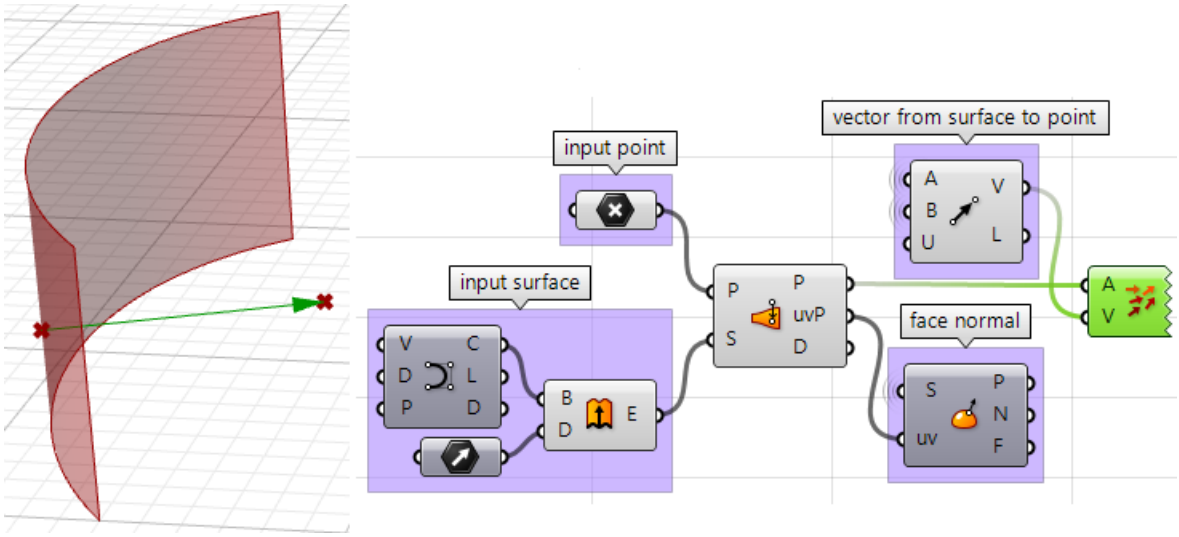
2 つの方向を比較したとき，同じ方向を向いていれば点は表側に，そうでなければ裏側にあることになります.

Solution:

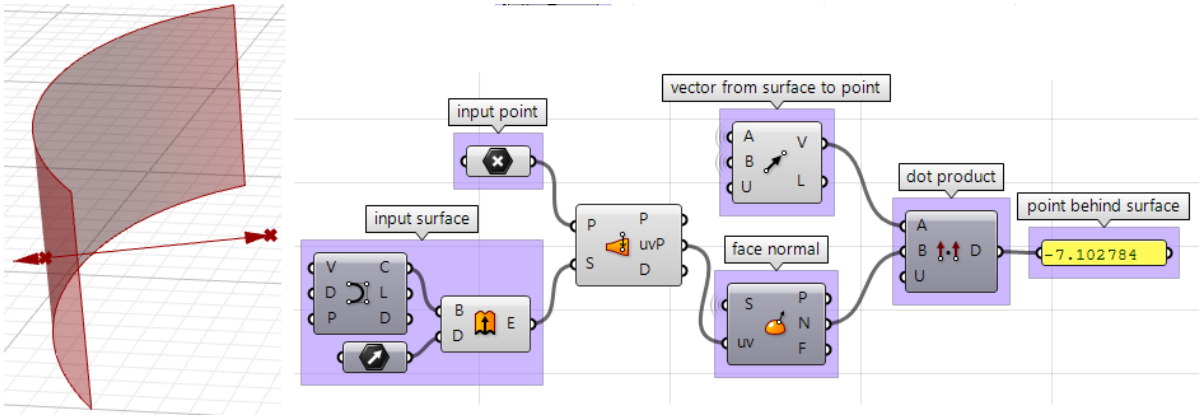
1. [**Pull**] コンポーネントを用いて，入力した点に最も近いサーフェス位置を求めます.これにより，最も近いサーフェス位置の UV 座標が得られるので，[**Evaluate Surface**] コンポーネントに繋ぐと法線方向が求まります.



2. 次のように，最も近い点から入力点へ向かうベクトルを描くこともできます.

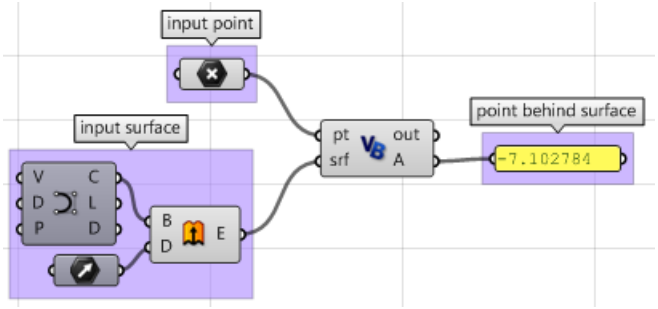


3. 内積を利用して 2 つのベクトルを比較します．結果が正であれば，点はサーフェスの表側にあります．結果が負であれば，点はサーフェスの裏側にあります．



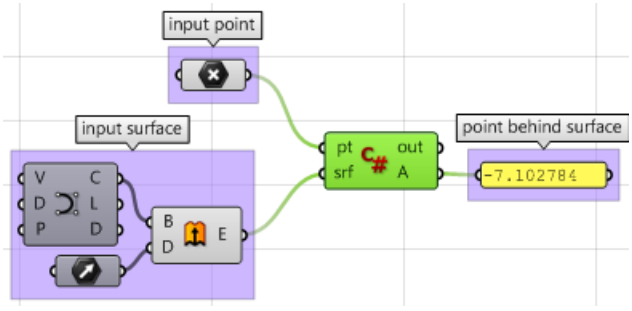
上記の手順は， 他のスクリプト言語を用いても解くことができます．

[VB Script] コンポーネントを用いた場合:



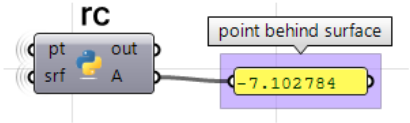

```
Private Sub RunScript(ByVal pt As Point3d, ByVal srf As Surface, ByRef A As Object)

  '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
```

[C# Script] コンポーネントを用いた場合**:**



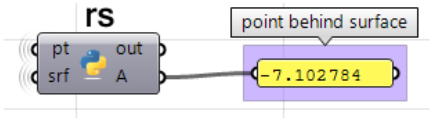

```
private void RunScript(Point3d pt, Surface srf, ref object A)
{
  //Decalre variables
  double u, v;
  Point3d closest_pt;

  //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;
}
```

[GhPython Script] コンポーネントと **RhinoCommon SDK** を用いた場合**:**




```
#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
```

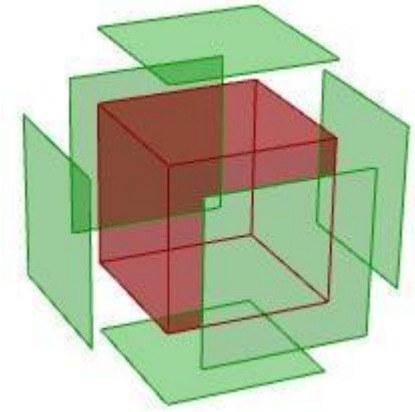
[GhPython Script] コンポーネントと RhinoScriptSyntax ライブラリを用いた場合:




```
#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
```

Box の分解図

このチュートリアルでは，ポリサーフェスを分解して表示する方法を紹介しています．以下の画像が最終的なイメージとなります．



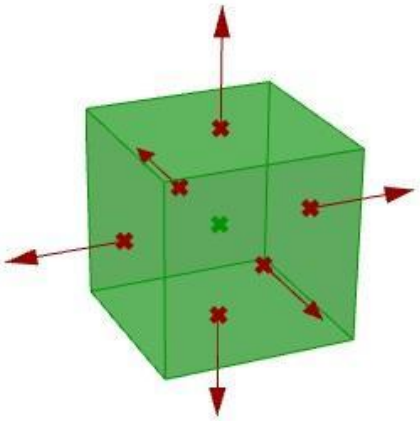
Input:

入力するものは Box です．GH では，[**Box**] パラメータを用います．

Parameters:

このチュートリアルを解くために必要なパラメータについて考えます.

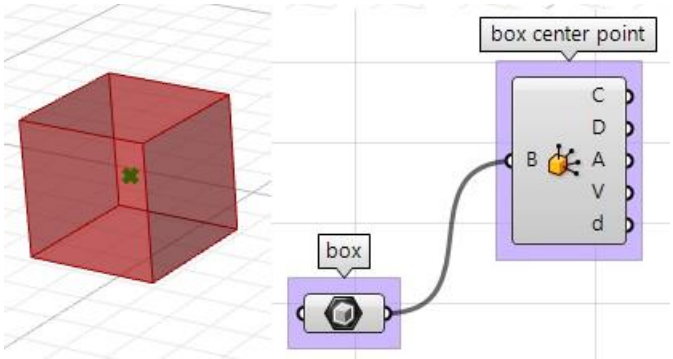
- 分解の基準となる点
- 分解する Box の面
- それぞれの面を移動させる方向



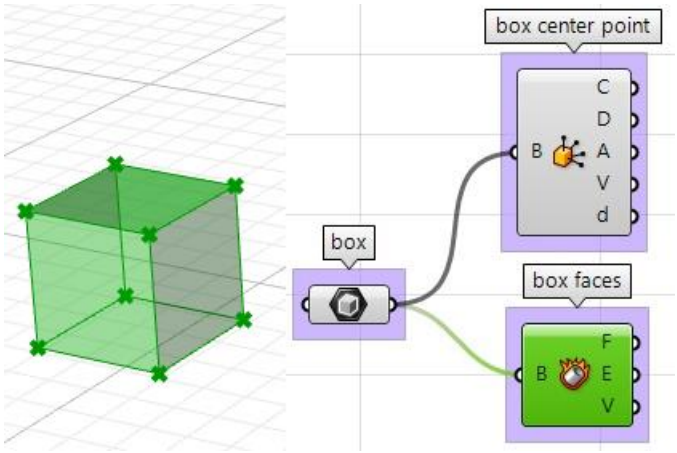
パラメータが特定できれば，答えに到達するために各ステップを論理的につなぎ合わせてまとめていくことが問題となります.

Solution:

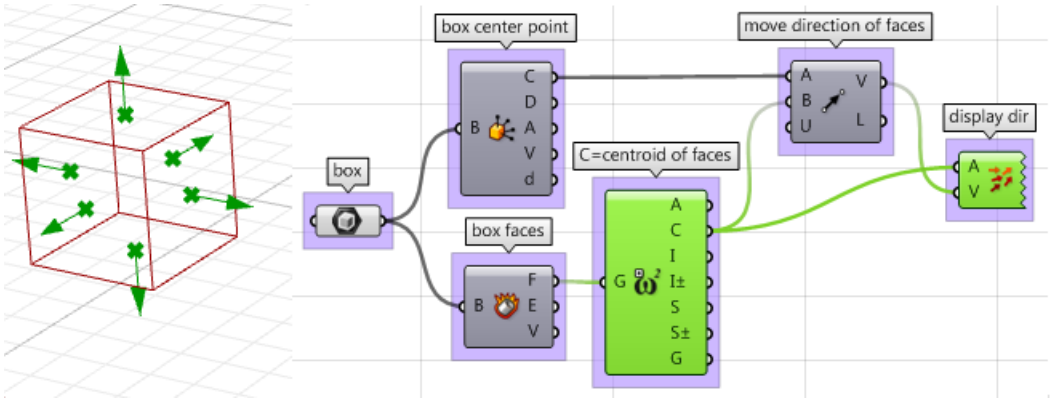
1. [**Box Properties**] コンポーネントを用いて Box の中心を求めます.



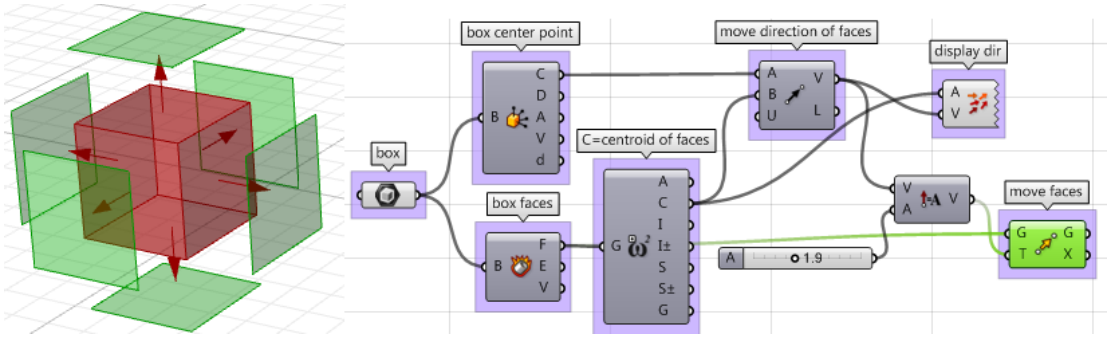
2. [**Deconstruct Brep**] コンポーネントで Box の各面を抽出します.



3. 面を移動する方向に関する部分は少々複雑です．以下のように，最初に各面の中心を求め，Box の中心からその点へ向かう方向を定義します．

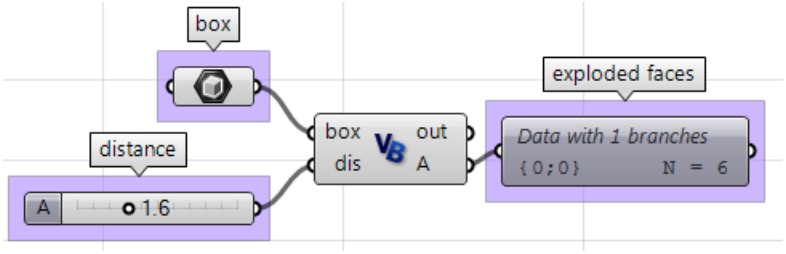


4. パラメータがすべて記述できたら，[**Move**] コンポーネントで各面を適切な方向で移動することができるようになります．移動ベクトルの大きさを希望の移動量に設定できるようにすれば完成です．



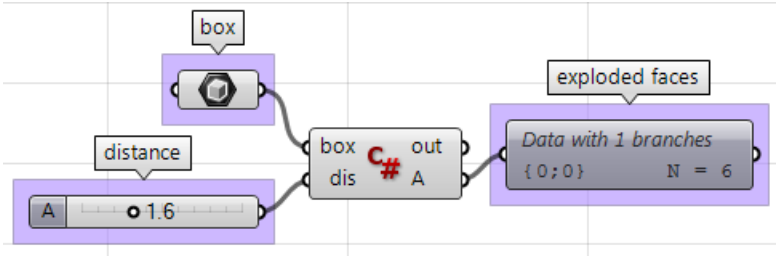
上記の手順は，VB script や C#，Python などを使っても解くことができます．以下はこれらのスクリプト言語を用いて解いた例です．

[VB Script] コンポーネントを用いた場合:



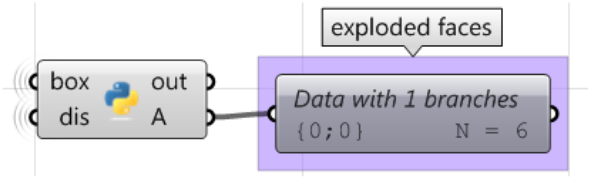
```
Private Sub RunScript(ByVal box As Brep, ByVal dis As Double, ByRef A As Object)

  '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
```

[C# Script] コンポーネントを用いた場合**:**




```
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;
}
```

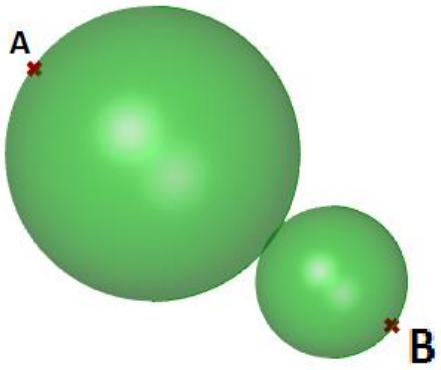
[GhPython Script] コンポーネントを用いた場合:




```
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
```

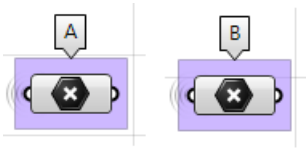
正接する 2 つの球

このチュートリアルでは，2 つの入力点から 2 つの正接した球を生成する方法を紹介します．
ここで求められる結果は次のようになります．



Input:

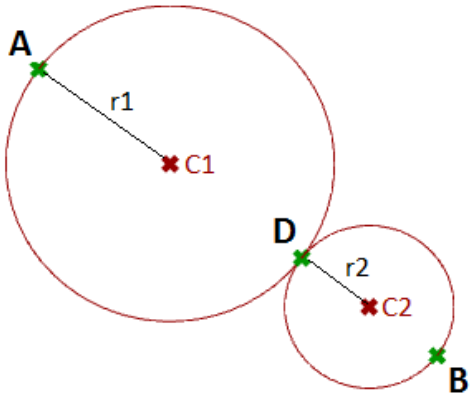
3 次元座標系内の 2 つの点 (点 A と点 B).



Parameters:

この問題を解く上で必要となるパラメータは次の通りです．

- 2 つの球が接する点 D（点 A–B 間でパラメータ *t*（0-1）を用いて設定する）．
- 1 つ目の球の中心点，すなわち点 A-D 間の中点 C1.
- 2 つ目の球の中心点，すなわち点 B-D 間の中点 C2.
- 1 つ目の球の半径 r1，すなわち点 A-C1 間の距離．
- 2 つ目の球の半径 r2，すなわち点 B-C2 間の距離．



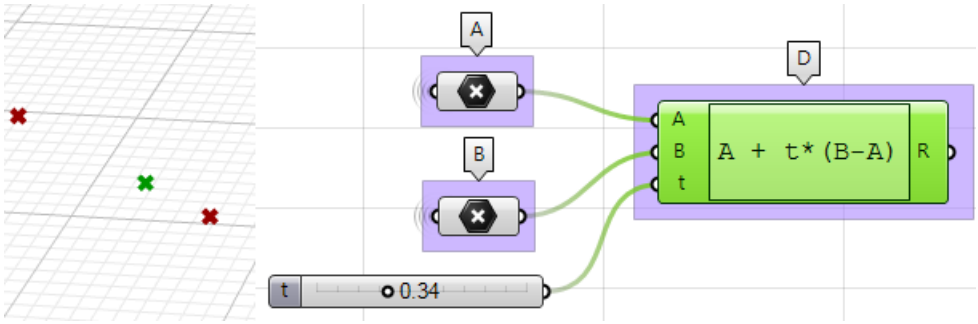
Solution:

1. [**Expression**] コンポーネントを用いて，パラメータ **t** で点 **A**-**B** 間の点 **D** を定義します． 数式は，直線のベクトル方程式: D = A + t*(B-A) に則ります.

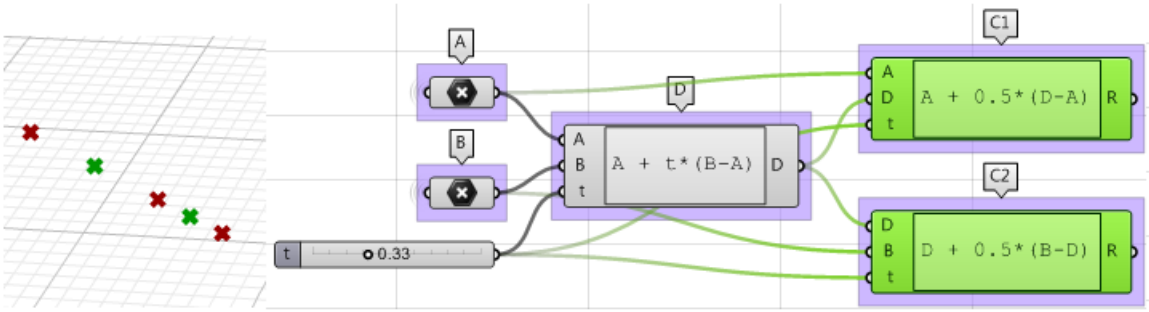
   B-A ：ベクトルの差演算から求まる点 B から点 A に向かうベクトルです.

   t*(B-A)：パラメータ *t* はベクトル上の位置を得るために 0 から 1 の値とします.

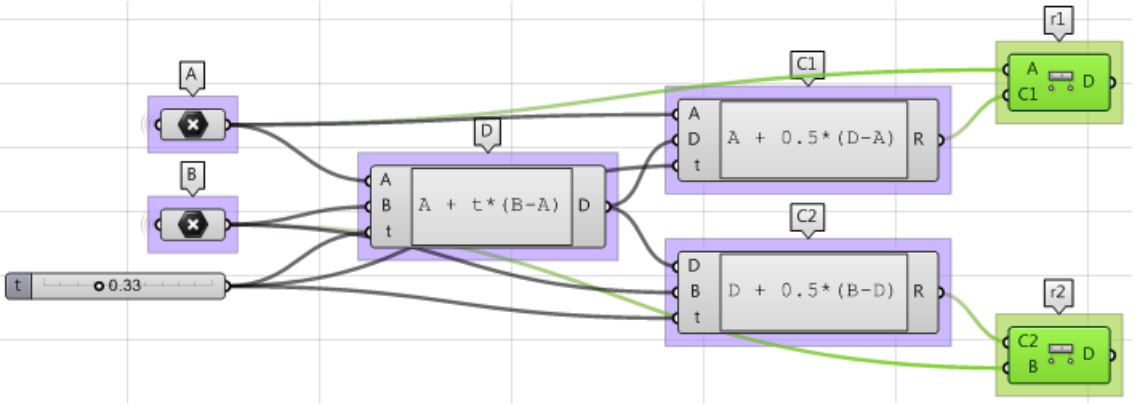
   A+t*(B-A): ベクトル AB 上の点を求めます.



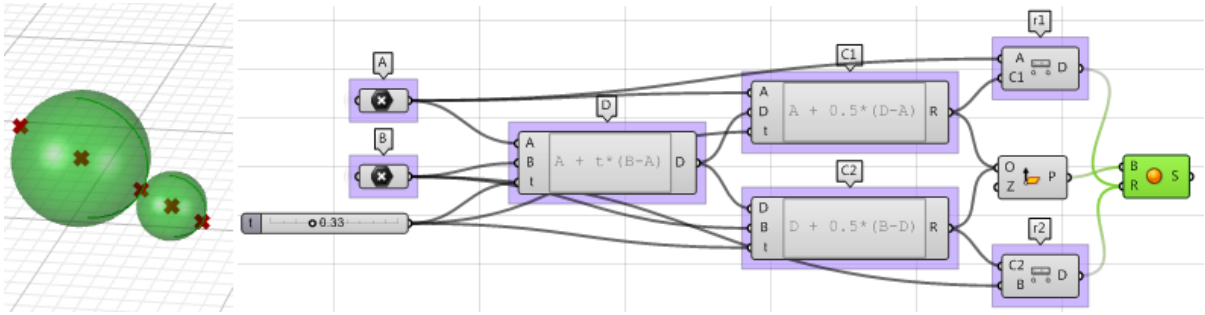
2. [**Expression**] コンポーネントを用いて，中点 **C1**，**C2** を求めます.



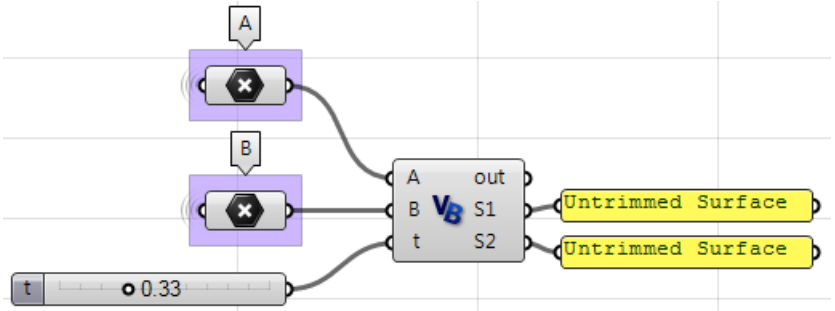
3. 1 つ目の球の半径（**r1**）と 2 つ目の球の半径 (**r2**)は [**Distance**] コンポーネントで計算できます.



4. 最後に，基準面（Base plane）と半径（Radius）から球を生成します．それぞれ，中心点が点 **C1**，**C2** に，半径が **r1** ，**r2** になっていることを確認します．

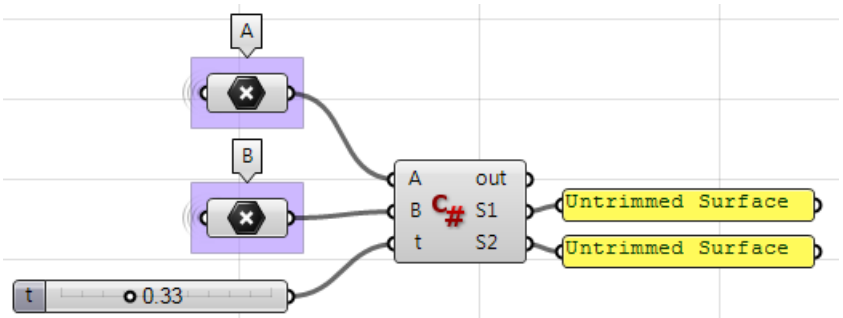


[VB Script] コンポーネントを用いた場合:



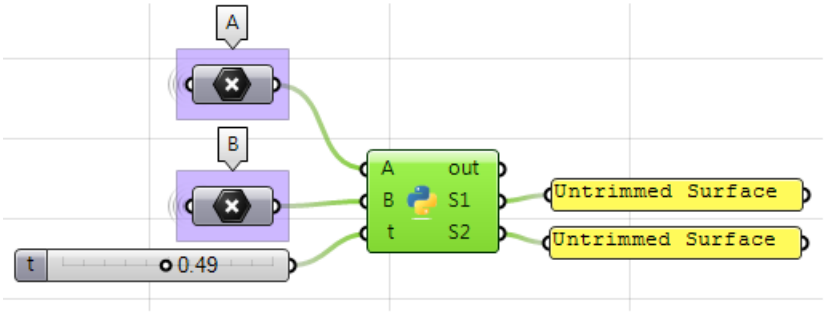

```
Private Sub RunScript(ByVal A As Point3d, ByVal B As Point3d, ByVal t As Double, ByRef S1 As Object, ByRef S2 As Object)

  '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
```

[C# Script] コンポーネントを用いた場合:



```
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);
}
```

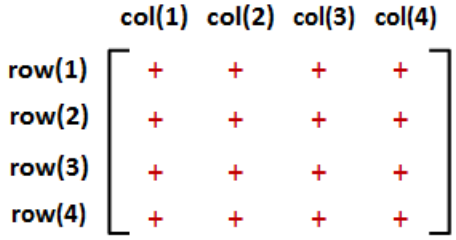
[GhPython Script] コンポーネントを用いた場合**:**




```
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（行列と変換）

「変換」とは，オブジェクトの移動（いわゆる平行移動），回転，スケーリングなどの操作を指します．それらは，数字を長方形に配列した行列を使用して 3 次元のプログラムとして保存されます．行列を用いると，複数の変換を非常に迅速に実行できます．4x4 行列の行列では，すべての変換を表現することができるので，すべての変換で統一された行列の次元を用いることで，計算時間を節約することができます．

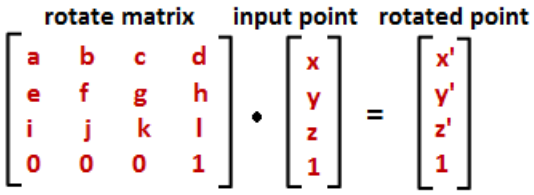


行列の演算

コンピュータグラフィックスで最も重要な操作は，「行列の乗法」です．詳細に解説します．

行列の乗法

行列の乗法は，ジオメトリに変換を適用するために用いられます．例えば，点が 1 つあり，それをある軸を中心に回転させたい場合，回転行列を用いてそれを点に掛けると，新しい回転位置が求まります．



ほとんどの場合，同じジオメトリに対し，複数回の変換を実行する必要があります．例えば，1000 個の点を移動および回転する必要がある場合，次のいずれかの方法を適用します．

Method 1

1. 1000 個の点に対し，平行移動行列を掛け，点を平行移動します．
2. 移動した 1000 個の点に対し，回転行列を掛け，点を回転します．

（操作の回数 ＝ **2000**）

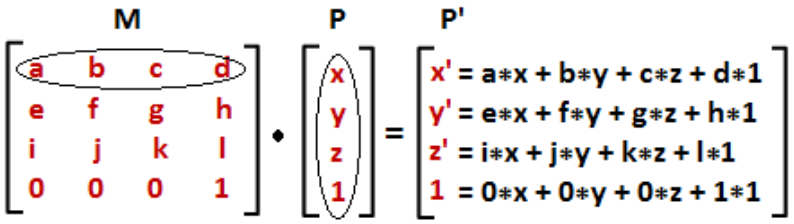
Method 2

1. 回転行列と平行移動行列を掛けて，組み合わせ行列を作成します．
2. 1000 個の点に対し，組み合わせ行列を掛け，同時に点を平行移動および回転します．

（操作の回数 = **1001**）

Method 1 では，同じ結果を得るための操作の数がほぼ 2 倍になります．Method 2 は非常に効率的ですが，平行移動行列と回転行列の両方が 4x4 行列の場合にのみ可能です．これが，コンピュータグラフィックスで 4x4 行列を使用してすべての変換を表し，4x1 行列を使用して点を表す理由です．

3 次元モデリングアプリケーションでは，変換を適用すると行列が乗算されます．行列を数学的に乗算する方法について興味がある人のために，以下にその簡単な例を紹介します． 2 つの行列を乗算するには，次元が一致している必要があります．言い換えると，最初の行列の列数は，2 番目の行列の行数と等しくなければならないということです．結果の行列のサイズは，最初の行列の行数と 2 番目の行列の列数に等しくなります．例えば，次元がそれぞれ 4x4 と 4x1 に等しい 2 つの行列 M と P がある場合，以下の図に示すように，乗算の結果の行列 M・P は 4x1 に等しい次元になります．

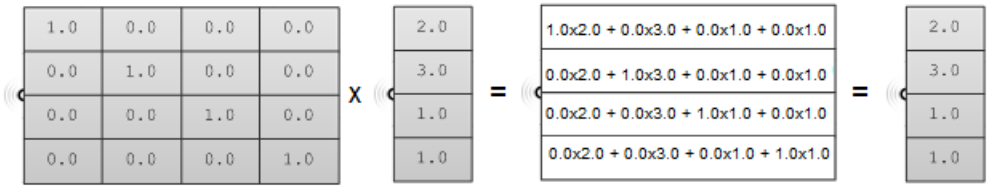


単位行列

「単位行列」は，すべての対角成分が 1 でその他の成分が 0 となる特殊な行列です．

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

単位行列の主な性質は，他の行列に乗算された場合，値が変わらないことです．



変換の演算

ほとんどの変換では，ジオメトリのパーツ間の平行関係は維持されます．例えば，同一直線上の点は，変換後も同一直線上にあります．また，1 つの平面上の点は，変換後も同一平面上にあります．このタイプの変換は，「アフィン変換」と呼ばれます．

平行移動変換

特定のベクトルを用いて，初期位置から点を移動する場合，次のように計算します．

P' = P + V

仮定:

P(x,y,z) は点,

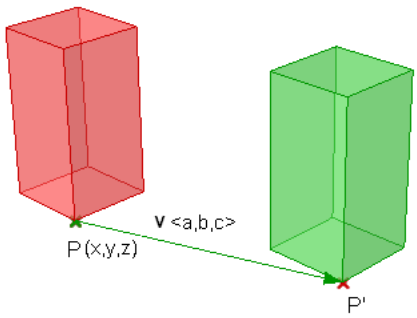
v<a,b,c> は変換ベクトルを表す

このとき:

P'(x) = x + a

P'(y) = y + b

P'(z) = z + c



点は，最後の行に 1 を挿入して 4x1 行列の形式で表されます．例えば，点 P(x,y,z) は右のように表せます．

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

3x3 行列の代わりに，変換に 4x4 行列（同次座標系と呼ぶ）を使用すると，平行移動を含むすべての変換を表すことができます．平行移動行列の一般的な形式は次の通りです．

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

例えば，点 P(2,3,1) をベクトル **v**<2,2,2>によって移動させる場合，新しい点の位置は，

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

行列形式を用いて，点に平行移動行列を掛けると，次のように新しい点の位置が求まります．

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

同様に，どんなジオメトリもその構成点に平行移動行列を掛けることで平行移動することができます．例えば，8 つのコーナーから定義されるボックスがあり，それを x 軸方向に 4 単位，y 軸方向に 5 単位，z 軸方向に 3 単位分移動させたい場合，次のような行列を掛ければ，新しいボックスの位置が得られる．

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

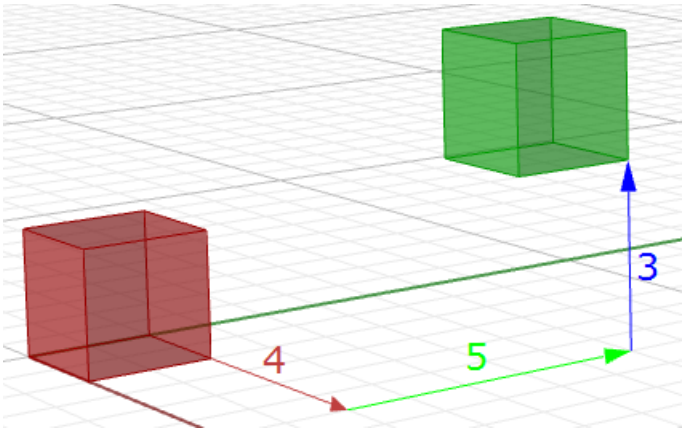


図 (19): ボックスのすべてのコーナーを平行移動.

回転変換

ここでは，三角法を用いて z 軸と原点回りの回転を計算し，回転に関する一般的な行列形式がどうなるかを見てみます．

xy 平面上の点 P(x,y) と，それを角度 b で回転することを考えます． 図より，以下のことが言えます．

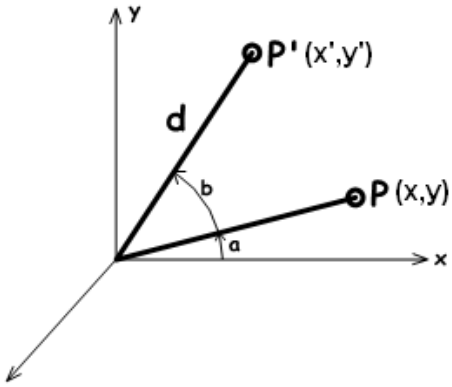
x = d cos(a) ---(1)
y = d sin(a) ---(2)
x' = d cos(b+a) ---(3)
y' = d sin(b+a) --- (4)

加法定理を用いて，x' と y' を展開すると，

x' = d cos(a)cos(b) - d sin(a)sin(b) ---(5)
y' = d cos(a)sin(b) + d sin(a)cos(b) ---(6)

式(1)(2)より，

x' = x cos(b) - y sin(b)
y' = x sin(b) + y cos(b)



z 軸回りの回転行列は以下のようになります.

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

同様に，**x 軸**回りの角度 **b** による回転行列は,

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

y 軸回りの角度 **b** による回転行列は,

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

例えば，あるボックスがあり，それを 30 度回転させたい場合，次のようにします.

1. まず，30 度の回転行列を作成します．一般的な行列形式と 30 度の cos および sin の値を使用すると，回転行列は次のようになります.

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

2. 回転行列に入力ジオメトリを乗算します．ボックスの場合は，各コーナーポイントを乗算してボックスの新しい位置を求めます.

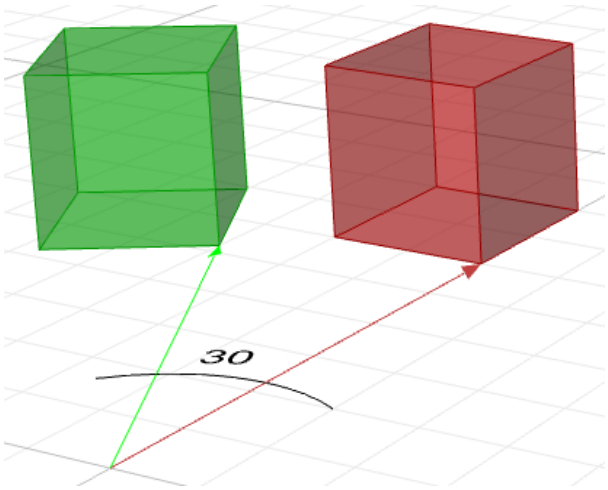


図 (20)：ジオメトリの回転.

スケール変換

ジオメトリをスケール変更するには，スケール係数とスケールの中心が必要となります．スケール係数は，x 方向，y 方向，z 方向すべて均一にもできますが，それぞれの方向で別々の値をとることもできます．点のスケーリングには次の式を用います．

P' = スケール係数 S * P

すなわち，

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

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

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

これは，スケールの中心が World 座標系の原点（0,0,0）であると仮定して，スケール変換するときの行列形式です．

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

例えば，World 座標系の原点を基準にして，ボックスを 0.25 倍スケーリングする場合，スケール行列は次のようになります．

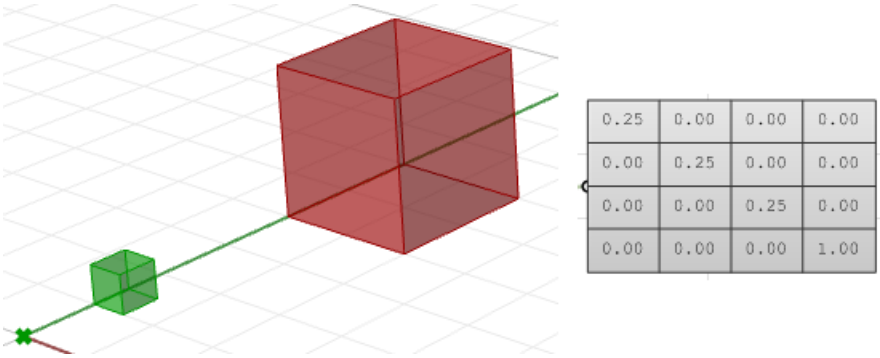


図 (21)：ジオメトリのスケール変換.

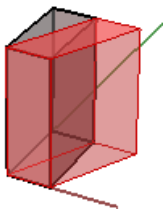
シアー変換

3 次元におけるシアー変形（せん断変形）は，ある軸に沿ってジオメトリを傾けたり歪めたりします．例えば，z 軸に沿うシアー変形は，その軸に沿うジオメトリは変更しませんが，x 軸および y 軸に沿ってジオメトリを変更します．シアー変形行列のいくつかの例を次に示します．

1. x 軸および z 軸方向のシアー変形．y 座標は固定．

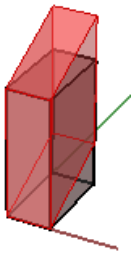
Shear x-axis

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



Shear z-axis

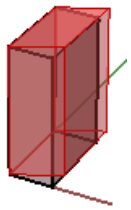
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.5 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



2. y 軸および z 軸方向のシアー変形．x 座標は固定．

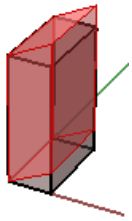
Shear y-axis

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



Shear z-axis

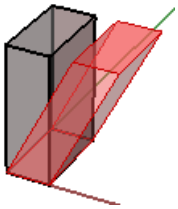
$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.5 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$



3. x 軸および y 軸方向のシアー変形．z 座標は固定．

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

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

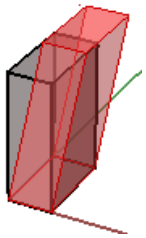
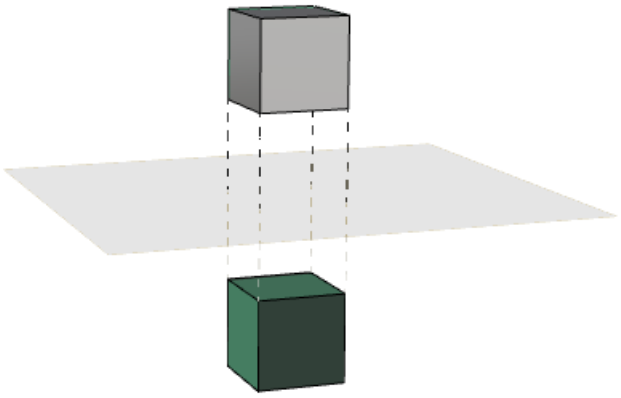


図 (22)：シアー変形行列.

ミラー変換

ミラー変換では，ある直線もしくは平面に対して鏡映を作成します．2 次元オブジェクトでは，直線に対してミラーリングされますが，3 次元オブジェクトでは，平面に対してミラーリングされます．ミラー変換により，ジオメトリ面の法線方向が反転することに注意が必要です．



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

図 (23): World xy 平面に対するミラー変換．面の方向が反転します.

平面投影変換

3 次元空間の点 P(x,y,z) の World xy 平面への投影は，z 座標を 0 にした点 $P_{xy}(x,y,0)$ である．同様に，点 P の xz 平面への投影は $P_{xz}(x,0,z)$となり，yz 平面への投影は P_{xz} = (0,y,z)となる．これらは正射影[1]と呼ばれます．

入力された曲線に平面投影変換を適用すると，その平面上に曲線が投影されます．以下は，行列形式を用いて曲線を xy 平面に投影した例です．

Note: NURBS 曲線(次章で解説します) では，曲線を定義するために制御点を用います．曲線を投影すると，その制御点が投影されます．

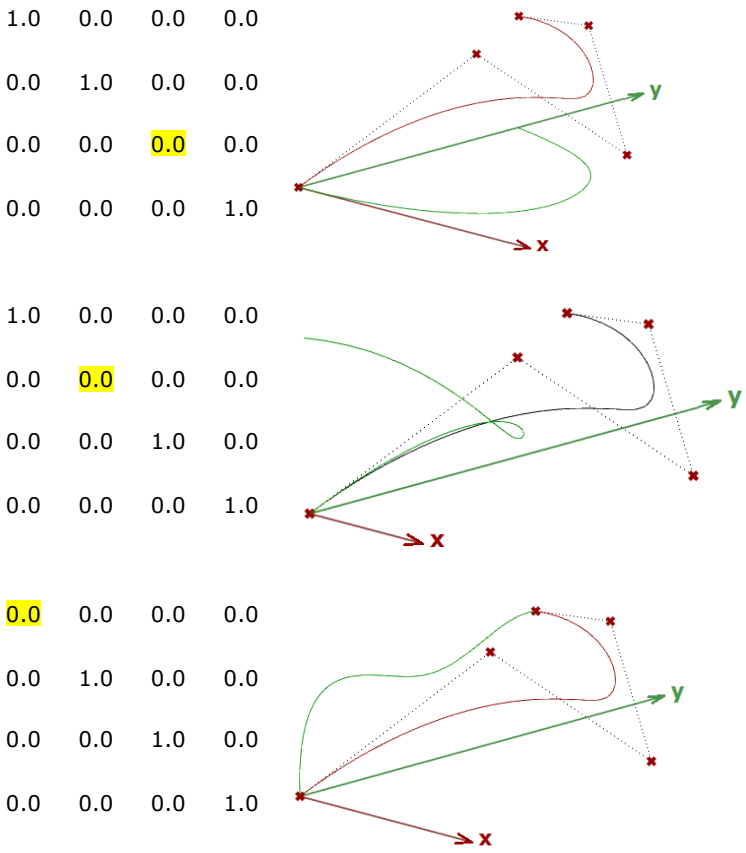


図 (24)：投影行列.

チュートリアル

複数の変換

1 つの行列を使用して，次のようにジオメトリを変換します．

[1] Projection (linear algebra). <https://en.wikipedia.org/wiki/Projection_(linear_algebra)>

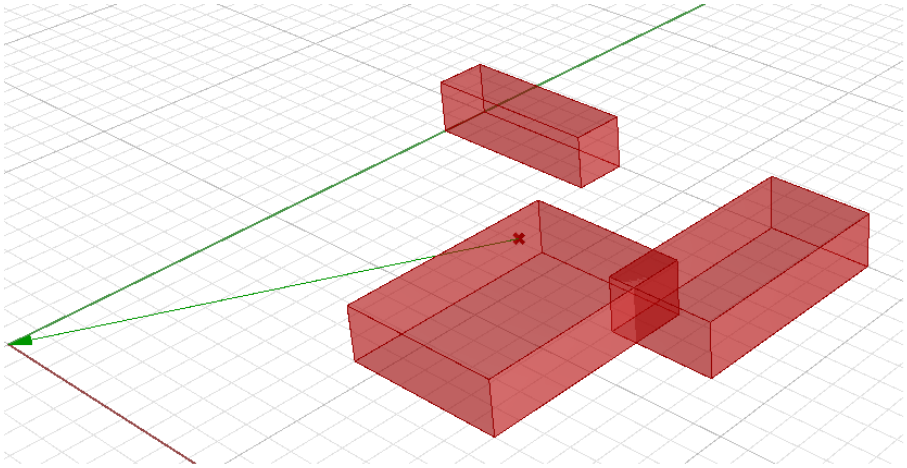
最初に，入力ジオメトリの中心が原点になるように移動し，次に z 軸を回転軸として 45 度回転し，0.2 倍で均一にスケーリングしてから元の位置に戻します.

Performance note:

変換する点またはオブジェクトの数が多い場合，それぞれの行列で複数回変換するのではなく，1 つのマスターとなる合成変換行列（最初にすべての行列を乗算）を作成して，その合成変換行列を使ってすべての入力を一括で変換する方がはるかに効率的です.

Input:

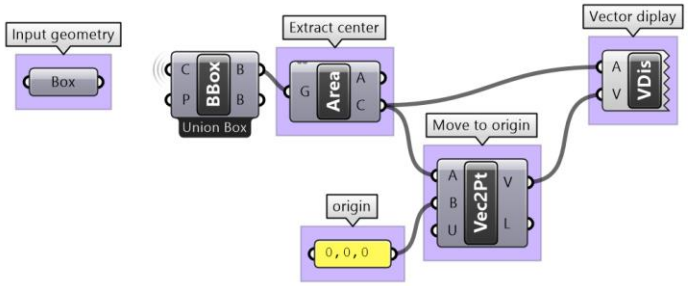
1. 変換するオブジェクト
2. 回転角度 45 度 と スケール係数 0.2



Additional input:

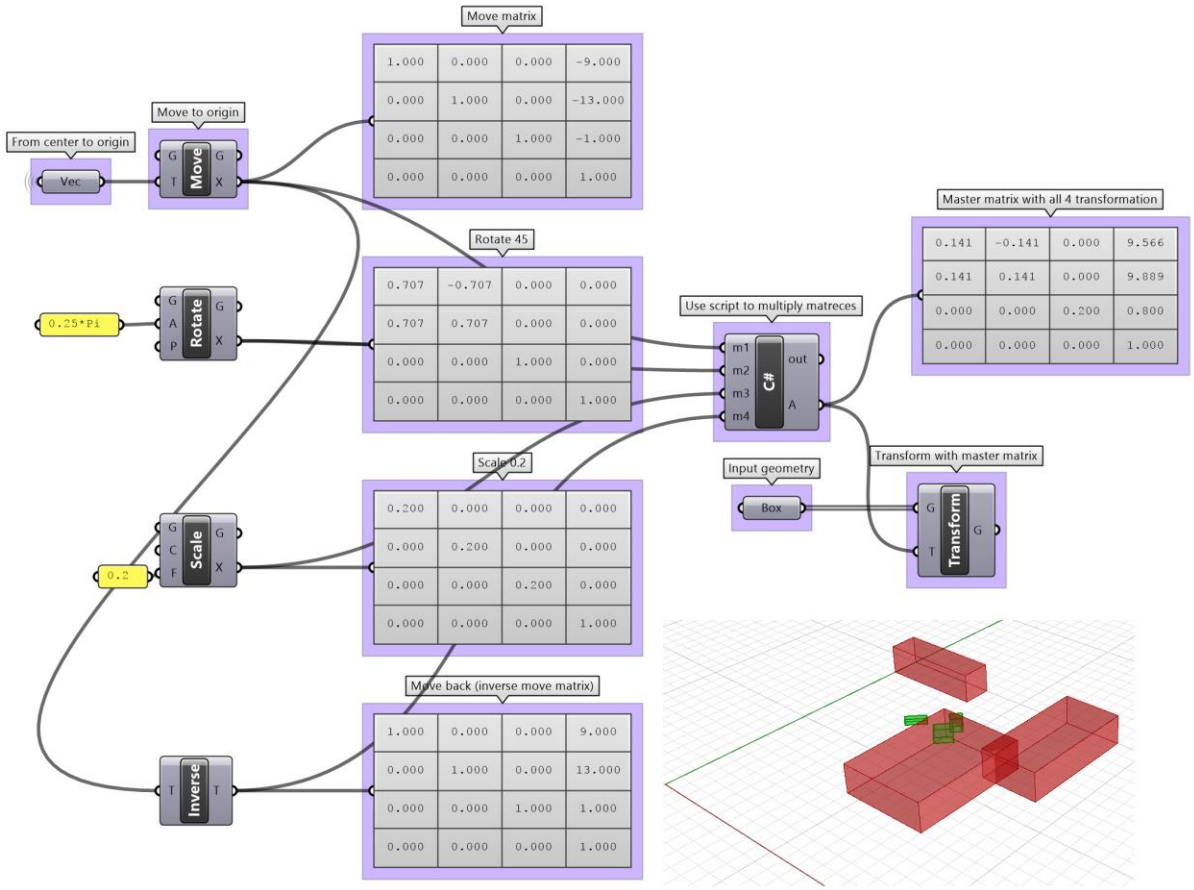
最初の移動には以下が必要です.

- バウンディングボックスの中心から原点へのベクトル



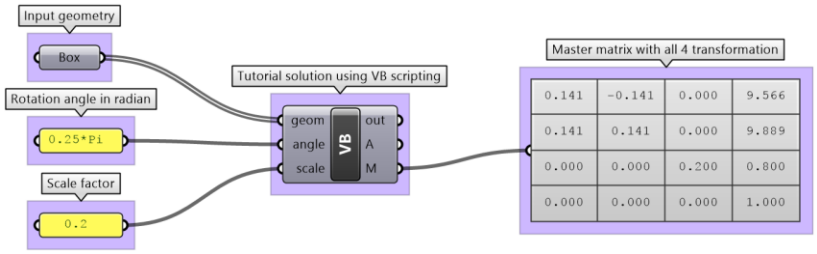
Solution:

1. すべての変換行列を作成します: 移動，回転，スケーリング，最初とは逆の移動.
2. 最後から最初へ行列を掛け合わせ，合成変換行列を生成します.
3. 合成変換行列を用いて入力ジオメトリを変換します.



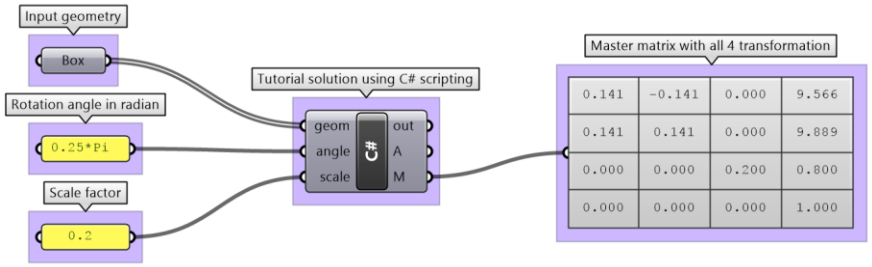
上記の手順は，スクリプトを使って解くこともできます.

[VB Script] コンポーネントを用いた場合:



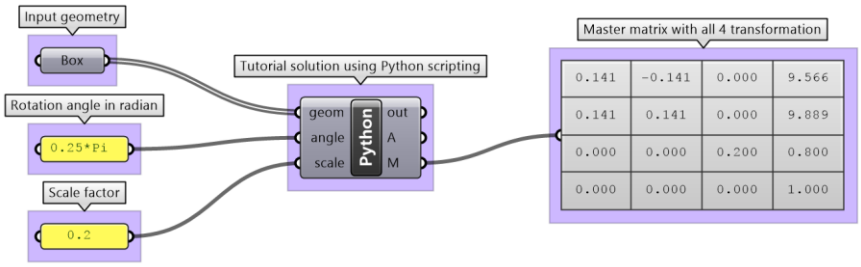
```
Private Sub RunScript(ByVal geom As List(Of GeometryBase), ByVal angle As Double, ByVal scale As Double, ByRef A As Object, ByRef M As Object)

  Dim center As Point3d = Point3d.Unset

  If geom.Count > 0 Then
    Dim bbox As BoundingBox = geom(0).GetBoundingBox(True)

    For i As Integer = 1 To geom.Count - 1
      bbox.Union(geom(i).GetBoundingBox(True))
    Next

    center = bbox.Center
  Else
    Return
  End If

  Dim m1 = Rhino.Geometry.Transform.Translation(Point3d.Origin - center)
  Dim m2 = Rhino.Geometry.Transform.Rotation(angle, Point3d.Origin)
  Dim m3 = Rhino.Geometry.Transform.Scale(Point3d.Origin, scale)
  Dim m4 = Rhino.Geometry.Transform.Identity
  m1.TryGetInverse(m4)
  Dim matrix = m4 * m3 * m2 * m1

  For i As Integer = 0 To geom.Count - 1
    geom(i).Transform(matrix)
  Next

  M = matrix
  A = geom

End Sub
```

[C# Script] コンポーネントを用いた場合:




```
private void RunScript(List<GeometryBase> geom, double angle, double scale, ref object A, ref object M)
{
  //Move to origin, find center
  Point3d center = Point3d.Unset;
  if(geom.Count > 0)
  {
    BoundingBox bbox = geom[0].GetBoundingBox(true);
    for(int i = 1; i < geom.Count ; i++)
    {
      bbox.Union(geom[i].GetBoundingBox(true));
    }
    center = bbox.Center;
  }
  else
    return;

  //Create move transformation matrix
  var m1 = Rhino.Geometry.Transform.Translation(Point3d.Origin - center);
  //Create rotation transformation matrix
  var m2 = Rhino.Geometry.Transform.Rotation(angle, Point3d.Origin);
  //Create scale transformation matrix
  var m3 = Rhino.Geometry.Transform.Scale(Point3d.Origin, scale);
  //Create move back transformation matrix
  var m4 = Rhino.Geometry.Transform.Identity;
  m1.TryGetInverse(out m4);

  //Multiply from last to first transformation
  var matrix = m4 * m3 * m2 * m1;

  //Apply transformation to all input geometry
  for(int i = 0; i < geom.Count ; i++)
  {
    geom[i].Transform(matrix);
  }

  //Assign output
  M = matrix;
  A = geom;
}
```

[GhPython Script] コンポーネントと RhinoCommon SDK を用いた場合:




```
import Rhino

#Move to origin, find center
center = Rhino.Geometry.Point3d.Unset
if geom.Count > 0:
    bbox = geom[0].GetBoundingBox(True)
    for i in range(1, geom.Count):
        bbox.Union(geom[i].GetBoundingBox(True))
    center = bbox.Center

origin = Rhino.Geometry.Point3d.Origin
#Create move transformation matrix
m1 = Rhino.Geometry.Transform.Translation(origin - center)
#Create rotation transformation matrix
m2 = Rhino.Geometry.Transform.Rotation(angle, origin)
#Create scale transformation matrix
m3 = Rhino.Geometry.Transform.Scale(origin, scale)
#Create move back transformation matrix
m4 = Rhino.Geometry.Transform.Identity
result, m4 = m1.TryGetInverse()

#Multiply from last to first transformation
matrix = m4 * m3 * m2 * m1

#Apply transformation to all input geometry
for g in geom:
    g.Transform(matrix)

#Assign output
M = matrix
A = geom
```

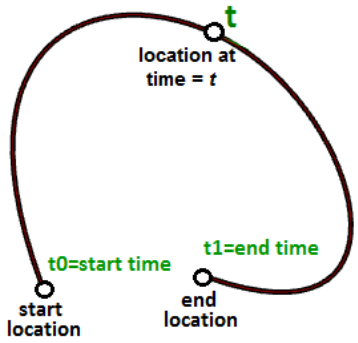
3 Parametric Curves and Surfaces（パラメトリック曲線と曲面）

例えば，平日，家から職場まで毎日移動するとします．午前 8 時に出発し，午前 9 時に到着します．8:00～9:00 の各時点では，途中のある場所にいます．移動中に 1 分毎に位置をプロットする場合，プロットした 60 点を接続することで，自宅と職場の間の経路を定義できます．毎日まったく同じ速度で移動すると仮定すると，8：00 には自宅（開始位置）に，9：00 には職場（終了位置）に，8：40 には経路上の 40 番目にプロットした点と同じ位置にいます．これで見事にパラメトリック曲線が定義できました．経路を定義するために，パラメータとして「時間」を使用したので，これも「パラメトリック曲線」であると言えます．開始から終了までにかかる時間の間隔は，「ドメイン（Domain）」や「区間（Interval）」と呼びます．

一般的に，パラメトリック曲線の x，y，z の位置は，次のようにパラメータ *t* を使って説明できます．

$x = x(t)$
$y = y(t)$
$z = z(t)$

ここで，*t* は実数を表します．



直線のベクトル方程式の節で，パラメータ *t* が次のように定義されることを確認しました．

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

ここで，

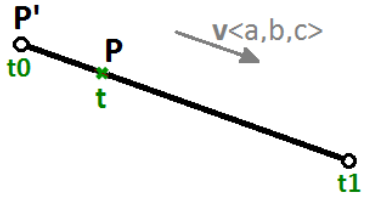
x，y，z は t の関数です．T は実数の範囲で表現されます．

x′，y′，z′は線分上にある点の座標を表します．

a，b，c は直線の傾きを定義し，ベクトル **v**<a, b, c>は直線に平行となります．

したがって，2 つの実数値 *t0*，*t1* の範囲にあるパラメータ *t* と，線分の方向を表す単位ベクトル v を用いて，線分のパラメトリック方程式は次のように記述できます．

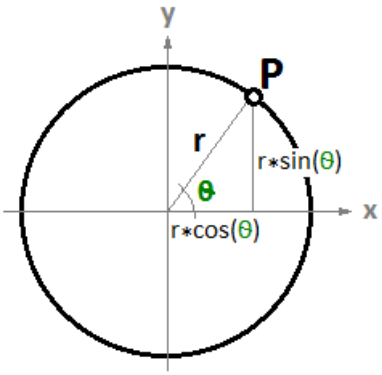
$P = P' + t * \mathbf{v}$



ほかの例としては円があります．中心が原点（0,0）で，角度パラメータ *t* が 0～2π ラジアンの範囲にある xy 平面上の円のパラメトリック方程式は次のとおりです．

$x = r \cos(t)$

$y = r \sin(t)$



円の一般式は次のように導くことができます．

$x/r = \cos(t)$

$y/r = \sin(t)$

またここで，

$\cos(t)^2 + \sin(t)^2 = 1$ (ピタゴラスの定理)

したがって，

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

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

パラメトリック曲線

曲線パラメータ

曲線上のパラメータは，その曲線上の点の位置を表します．前述のように，パラメトリック曲線は，一定の時間内に 2 つのポイント間を等速または変速で移動したときの経路と考えることができます．移動するのに時間 T が掛かる場合，パラメータ *t* は，T 以内の時間で表され，曲線上の位置（ポイント）として解釈できます．

2 点 AB 間が直線パス（線分）で結ばれていて，**v** が A から B へのベクトル（v = B-A）を表すと仮定すると，直線のパラメトリック方程式を用いて，AB 間の任意の点 M は次のように求められます．

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

このとき，*t* は 0～1 の値です．

パラメータ *t* の範囲（この場合は 0～1）は，ドメインや区間と呼びます． *t* がドメイン外の値（0 未満または 1 より大きい）の場合，点 M は線分 AB の外側の延長部分を表します．

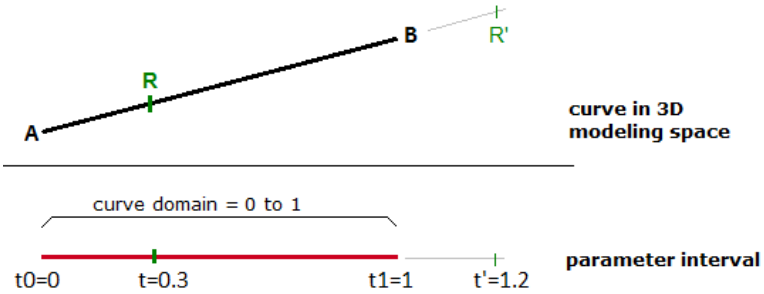


図 (25): 3 次元パラメトリック直線とパラメータの区間.

同じ原理がどのパラメトリック曲線にも適用されます．曲線上の任意の点は，曲線の端点を表す値から決まるドメインの範囲内のパラメータ t を用いて計算できます．一般にドメインの開始パラメータは t0，終了パラメータは t1 で表現します．

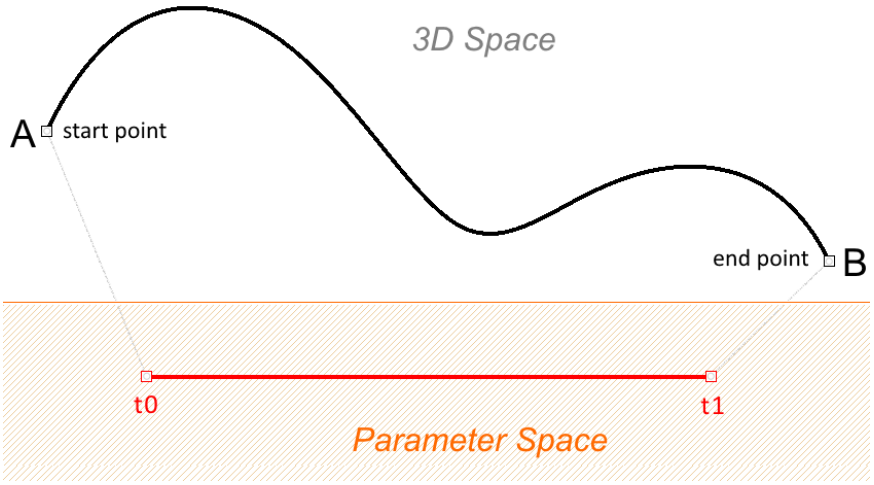


図 (26)：3 次元空間における曲線とパラメータ空間におけるドメインの関係.

曲線ドメイン

曲線のドメインは，その曲線上の点から評価されるパラメータ範囲として定義されます．ドメインは通常，「Min To Max」または「Min，Max」の形式で表されるドメインの最小値と最大値を定義する 2 つの実数で記述されます．ドメインの最大・最小値を表す 2 つの値は，曲線の実際の長さに関連する場合と関連しない場合があります．Min から Max で値が単調増加となるドメインの場合は，ドメインの Min が曲線の始点，ドメインの Max が曲線の終点を表します．

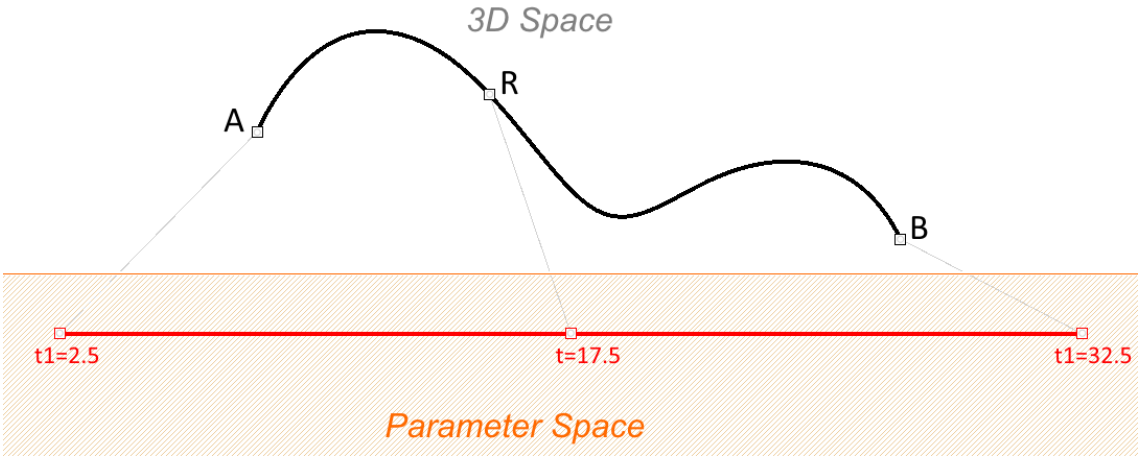


図 (27)：曲線のドメインは，通常，単調増加する 2 つの数値のセットです．可能な場合は，ドメイン長は曲線の実際の長さに近くなるよう設定されますが，曲線自体は変更せずに任意の長さに設定可能です.

曲線ドメインの変更プロセスは，「Reparameterize（パラメータの再定義）」と呼びます．例えば，ドメインを「0～1」の範囲に変更することは非常に一般的です（これを正規化と言います）．曲線のパラメータを変更しても，3 次元曲線の形状には影響しません．これは，徒歩の代わりに走ることによって経路の移動時間を変更するようなもので，経路の形状は変更されないことと似ています.

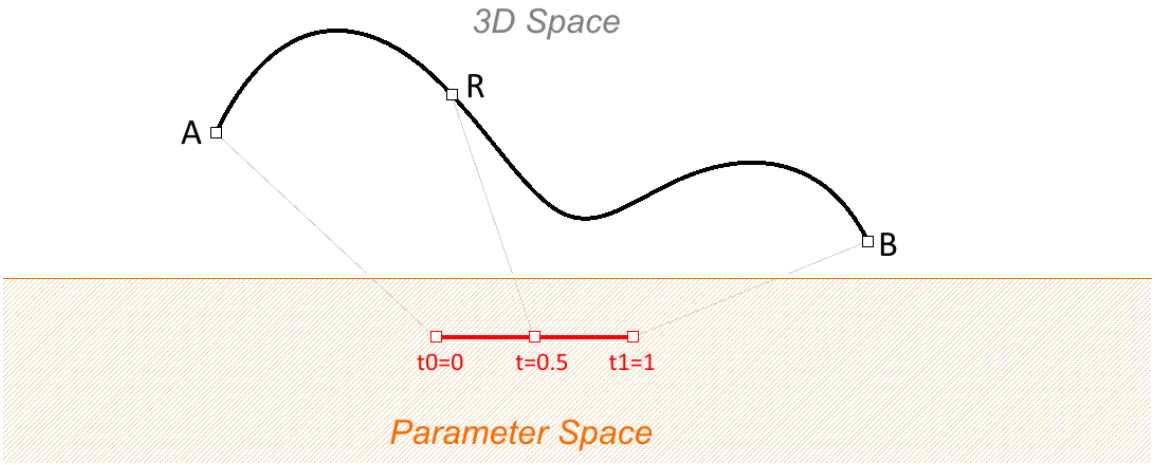


図 (28): 曲線ドメインは，正規化（0～1 に再定義）できます．曲線の実際の長さがドメイン長よりもはるかに大きい場合（10 倍以上），パラメータでの評価では 3 次元曲線上の厳密な位置が得られない可能性があることに注意が必要です.

ドメインが単調増加の場合は，ドメインの Min が曲線の始点を指すことを意味します．ドメインは通常，単調増加となりますが，必ずそうなる訳ではありません.

曲線の評価

曲線の区間は，3 次元曲線上の点で評価されるすべてのパラメータ値の範囲であることを学びました．ただし，例えばドメインの中間の値で評価したときに曲線の中点にある点が得られるという保証はありません.

曲線を均一なパラメータで表すことは，一定の速さで経路を移動するものと考えることができます． 2 点を結ぶ 1 次の直線は，図（29）に示すように，等間隔なパラメータを直線上の等間隔な点に変換できる例のひとつです．パラメトリック曲線では，等間隔なパラメータから 3 次元曲線を等間隔に評価できることは稀です.

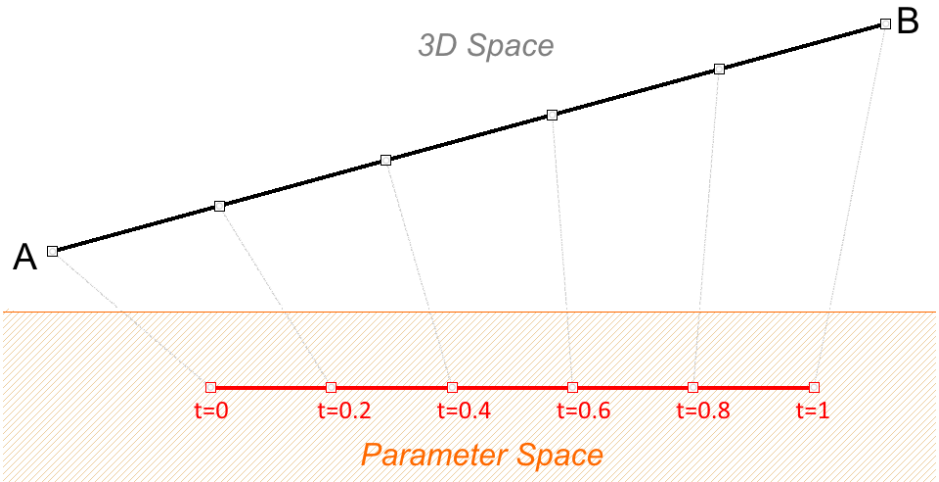


図 (29): 等間隔なパラメータから 3 次元曲線を等間隔に評価できる特殊なケースである次数 1 の直線.

移動する速さが，経路に沿って減少または増加することがよくあります．例えば，道路を移動するのに 30 分かかる場合に 15 分時点でちょうど真ん中の地点にいることはあまりないでしょう．図（30）は，等間隔のパラメータが，3D 曲線上に可変的に評価されるケースの典型です.

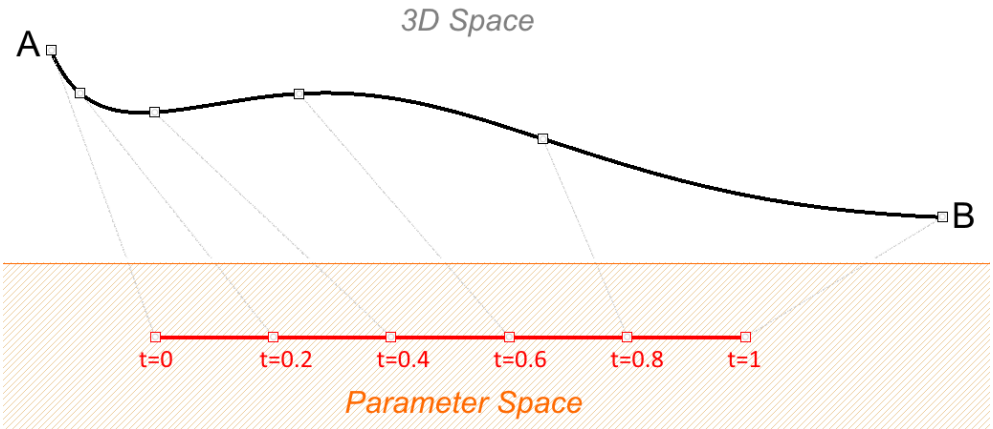


図 (30): NURBS 曲線のようなパラメトリック曲線では通常，等間隔なパラメータは，曲線上の等間隔の距離に変換されません.

実際の曲線長さの割合に応じて曲線上の点を評価したい場合もあるでしょう．例えば，曲線を等しい長さに分割する必要がある場合があるので，大抵の場合，3D モデラーには，弧の長さを基準に曲線を評価するツールがあります.

曲線の接線ベクトル

任意のパラメータ（または曲線上の点）における曲線の接線（Tangent）とは，その点で曲線に接触するが，交差はしないベクトルのことです．接線ベクトルの向きは，その点での曲線の勾配に等しくなります．以下の例は，2 つの異なるパラメータで曲線の接線を評価しています.

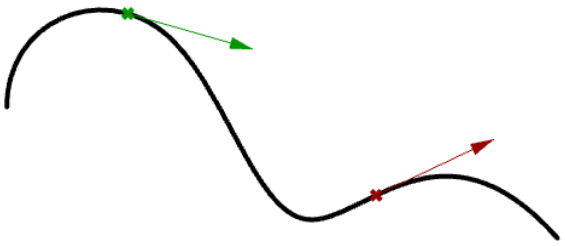


図 (31): 曲線の接線.

3 次多項式曲線

Hermite 曲線[2]と Bezier 曲線[3]は，4 つのパラメータによって決定される 3 次多項式曲線の例の 2 つです．Hermite 曲線は 2 つの端点とその点における 2 つの接線ベクトルによって決定され，Bezier 曲線は 4 つの点によって定義されます．それらは数学的には異なるものですが，互いによく似た特徴と制限を有します.

[2] Cubic Hermite spline. <https://en.wikipedia.org/wiki/Cubic_Hermite_spline>

[3] Bézier curve. <https://en.wikipedia.org/wiki/B%C3%A9zier_curve>

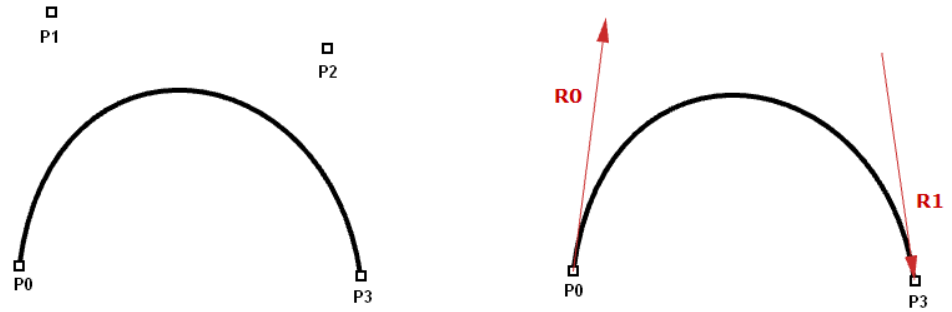


図 (32): 3 次多項式曲線. Bezier 曲線（左）は，4 つの点で定義されます. Hermite 曲線（右）は 2 つの点と 2 つの接線ベクトルで定義されます.

ほとんどの場合，曲線は複数のセグメントから成り立ちます．このためには，区分多項式曲線と呼ばれる曲線を作成する必要があります．以下は，7 つの格納点を用いて 2 セグメントの 3 次曲線を作成する区分的な Bezier 曲線の図です．最終的にできた曲線は結合されていますが，滑らかにも連続にも見えないことがわかるでしょう．

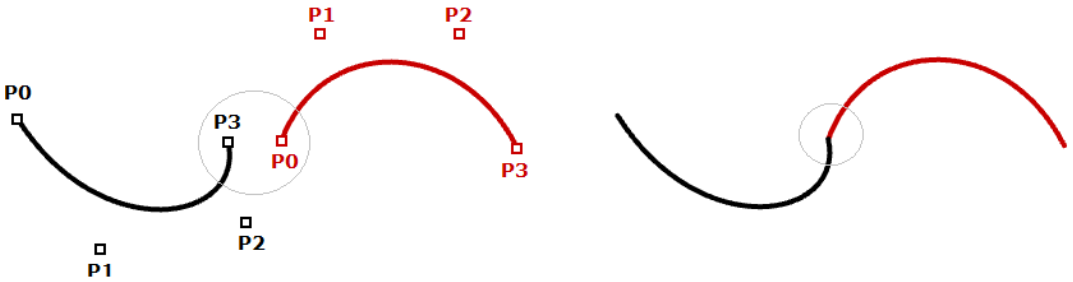


図 (33): 1 つの点を共有した 2 本の Bezier 曲線.

Hermite 曲線では，Bezier 曲線と同じ数のパラメータ（1 つの曲線を定義するための 4 つのパラメータ）を用いますが，次の曲線との接続部分で共有できる接線の情報を用いるので，以下のようにより少ない情報量でより滑らかな曲線が作成できます．

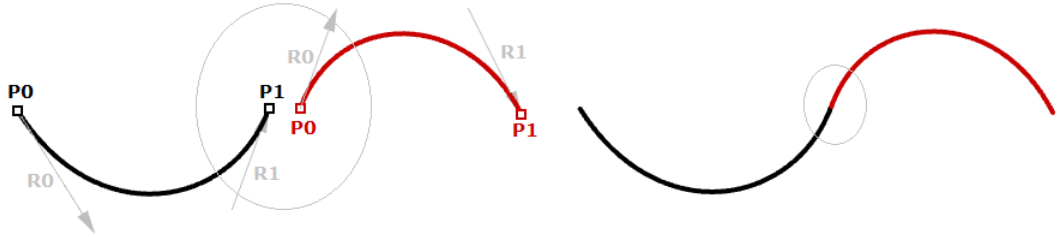


図 (34): 1 つの点と 1 つの接線ベクトルを共有した 2 本の Hermite 曲線.

非一様有理 B スプライン（NURBS）[4]は，滑らかさや連続性をより維持することができる強力な曲線表現です．セグメントはより多くの制御点を共有し，少ない情報量で滑らかな曲線を実現します．

[4] Non-uniform rational B-spline. <https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline>

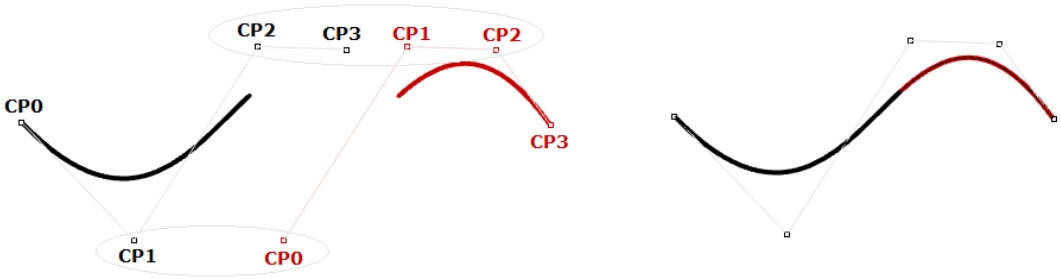


図 (35): 3 つの制御点を共有した 2 本の 3 次 NURBS 曲線.

NURBS 曲線と NURBS 曲面は，ジオメトリを表現するために Rhino で使用される代表的な数学的表現です．NURBS 曲線の特徴と成り立ちについては，この章の後半で詳しく説明します．

3 次 Bezier 曲線の評価

発明者の Paul de Casteljau にちなんで名付けられたド・カステリョのアルゴリズム[5]は，再帰的手法により Bezier 曲線を評価します．アルゴリズムの手順は次のように要約できます．

Input:

4 つの点 A, B, C, D で定義された曲線

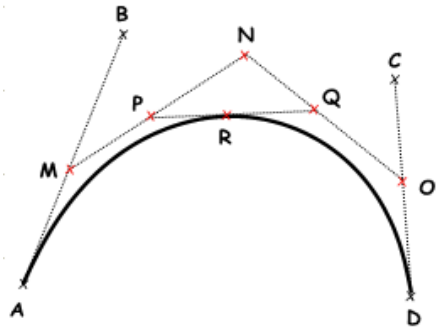
曲線のドメイン内のパラメータ *t*

Output:

パラメータ *t* における曲線上の点 R

Solution:

1. 線分 AB でのパラメータ *t* の点 M を求める
2. 線分 BC でのパラメータ *t* の点 N を求める
3. 線分 CD でのパラメータ *t* の点 O を求める
4. 線分 MN でのパラメータ *t* の点 P を求める
5. 線分 NO でのパラメータ *t* の点 Q を求める
6. 線分 PQ でのパラメータ *t* の点 R を求める



NURBS 曲線

NURBS は，曲線を非常に直感的にかつ正確に編集できる数学的表現です．NURBS によって自由曲線を簡単に表現でき，制御点構造により，編集が簡単でわかりやすくなります．

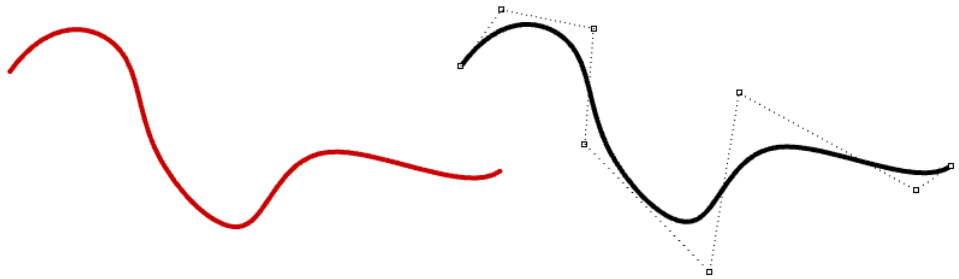


図 (36): 非一様有理 B スプラインとその制御点構造.

[5] De Casteljau's algorithm. <https://en.wikipedia.org/wiki/De_Casteljau%27s_algorithm>

NURBS についてより深く知りたい人向けに多くの書籍や参考文献がありますが，NURBS モデラーをより効果的に使用するには，まずは基本の理解が重要です．NURBS 曲線を定義する要素は主に 4 つあります．「次数（Degree）」「制御点（Control points）」「ノット（knots）」「評価公式（Evaluation rules）」です．

次数

曲線の次数は，正の整数となります．Rhino では，1 から始まる任意の次数の曲線を用いることができます．次数 1，2，3，5 が最もよく用いられ，4 や 6 以上は実際あまり使われません．以下は，曲線とその次数のいくつかの例です．

直線 や **ポリライン** は，
次数 1 の NURBS 曲線です．

円 や **楕円** は，
次数 2 の NURBS 曲線です．

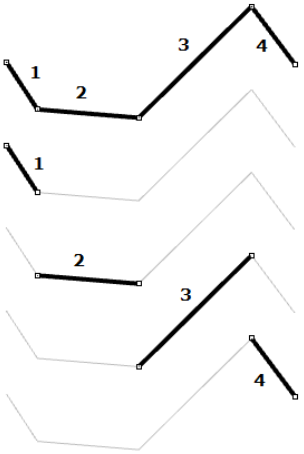
自由曲線 は，通常，次数 3 か 5 の NURBS 曲線で表されます．

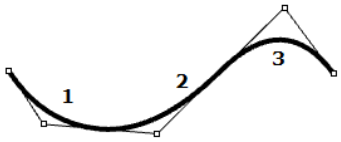
制御点

NURBS 曲線の制御点は，少なくとも（次数+1）個の点のリストです．NURBS 曲線の形状を変更する最も直感的な方法は，制御点を移動することです．

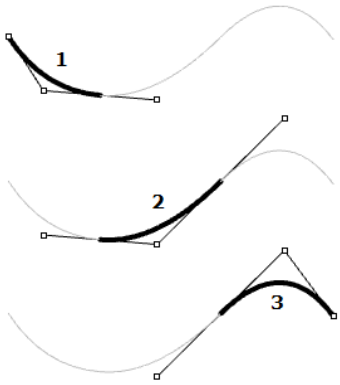
NURBS 曲線の各区間に影響する制御点の数は，曲線の次数で定義されます．例えば，次数 1 の曲線のそれぞれの区間は，それら区間の端点における制御点 2 つにのみ影響を受けます．次数 2 の曲線では，3 つの制御点がそれぞれの区間に影響します．

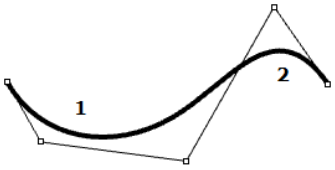
次数 1 の曲線は，すべての制御点を通過します．次数 1 の NURBS 曲線では，2 つ（次数+1）の制御点が各区間を定義します．5 つの制御点を使用すると，曲線は 4 つの区間から成り立ちます．



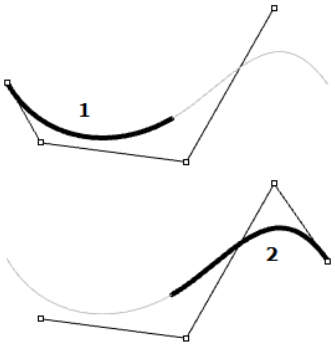


円と楕円は，2 次曲線の例です．次数 2 の NURBS 曲線では，3 つ（次数 +1）の制御点が各区間を定義します．5 つの制御点を使用すると，曲線は 3 つの区間から成り立ちます．





次数 3 の曲線の制御点は，開いた曲線の端点を除き，通常は曲線に接触しません．次数 3 の NURBS 曲線では，4 つ（次数+ 1）の制御点が各区間を定義します． 5 つの制御点を使用すると，曲線には 2 つの区間があります．



ウェイト

それぞれの制御点は，「ウェイト（重み）」と呼ばれる値を持ちます．例外もありますが，基本的にウェイトは正の数値となります．すべての制御点が同じウェイト（通常 1）を持つ場合，その曲線は，非有理（non-rational）である，と言います．直感的には，ウェイトはそれぞれの制御点が持つ重力の大きさであると考えることができます．制御点のウェイトが相対的に大きいほど，曲線はその制御点に対して引き寄せられます．

ただし，曲線のウェイトを変更することは避けた方が無難です．ウェイトを変更しても，交差などの演算において多くの計算上の課題が発生する一方で，望ましい結果が得られることはほとんどありません．有理曲線（rational curves）を扱う唯一のメリットは，円や楕円など，他の方法では描画できない曲線を表すことです．

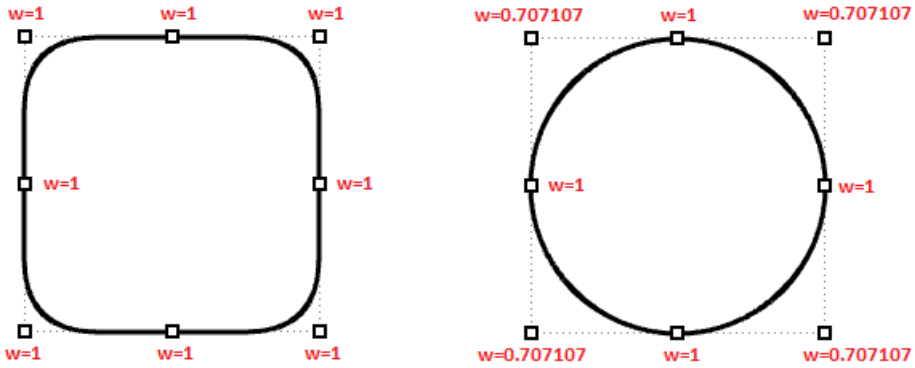


図 (37)：曲線の制御点のウェイトを変えた場合の影響．
左の曲線は，ウェイトが均一な非有理な曲線．
右側の円は，コーナーの制御点のウェイトが 1 より小さい有理曲線です．

ノット

それぞれの NURBS 曲線は，「ノット（ノットベクトルと呼ぶこともある）」と呼ばれる，それに関連付けられた数値のリストを持ちます．ノットについての理解と設定は少し難しくなりますが，3D モデラーを使用している間は，作成する曲線毎にノットを手動で設定する必要はありません．ノットについて学ぶのに役立つ事柄をいくつか見てみましょう．

ノットはパラメータ

ノットは，曲線ドメイン範囲内の単調増加のパラメータのリストです．Rhino では，ノット数は制御点の個数より（次数-1）個多くなり，式で表すと以下のようになります．

|ノット数| = |制御点数| + 次数 - 1

一般に，非周期的な曲線の場合，最初の方のノットの多くはドメインの最小値に等しくなり，最後の方のノットの多くはドメインの最大値に等しくなります．

例えば，7 つの制御点から成る，ドメインが 0 から 4 の開いた 3 次 NURBS 曲線のノットは，<0, 0, 0, 1, 2, 3, 4, 4, 4>のようになります．

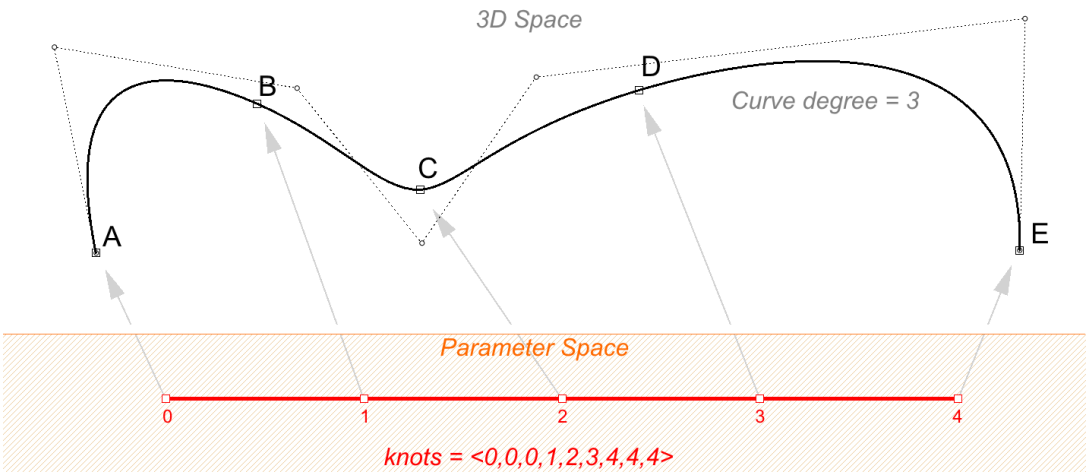


図 (38): ノット数は制御点の個数より（次数-1）個多くなります．制御点数= 7，次数= 3 の場合，ノット数は 9 です．ノット値は，3D 曲線上の各点で評価されるパラメータです．

ノットのリストをスケール変更しても，3D 曲線には影響しません．前述の例の曲線ドメイン「0 から 4」を「0 から 1」に変更すると，ノットのリストもスケール変更されますが，3D 曲線は変わりません．

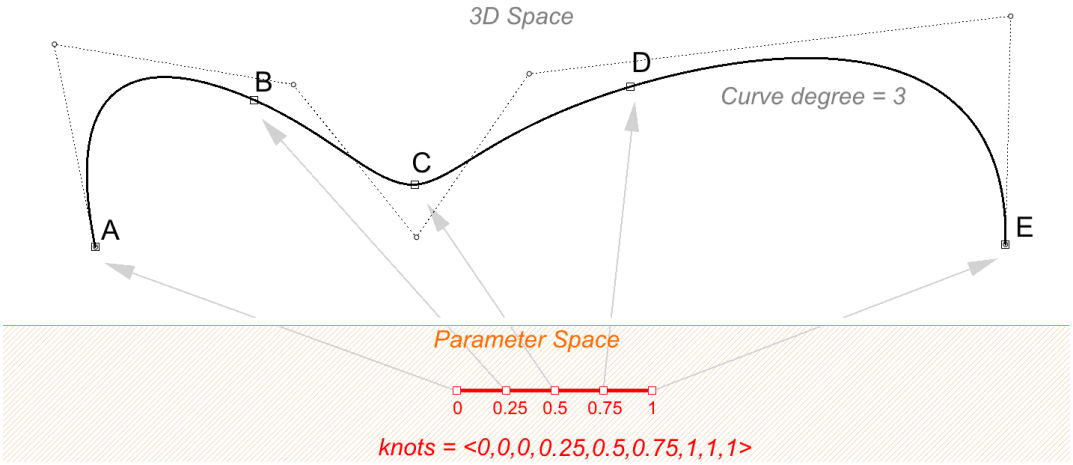


図 (39): ノットのリストをスケール変更しても 3D 曲線は変わりません．

1 回のみ現れるノットを単純ノットと呼びます．端点以外のノットは通常，前述の 2 つの例のように単純ノットです．

ノットの多重度

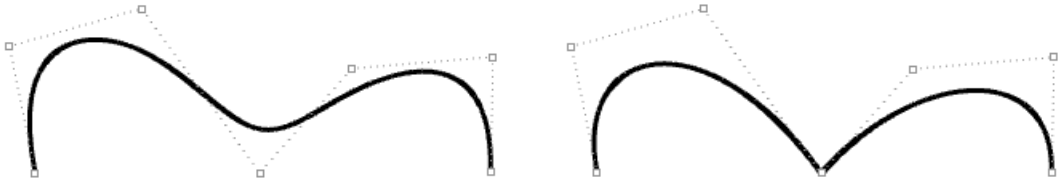
ノットの多重度とは，ノットのリストの中にそのノットと同じ値が何度現れるかを表した数値です．ノットの多重度は，曲線の次数より大きくはなりません．ノットの多重度は，対応する曲線上の点における連続性を制御するために用いられます．

完全多重ノット

完全多重ノットとは，多重度が曲線の次数と等しいノットです．完全多重ノットには，対応する制御点があり，曲線がその制御点を通ります．

開いた曲線は，端点に完全多重ノットを持ちます．つまり，曲線の端点の制御点は曲線の終点と一致します．曲線内部に完全多重ノットがある場合，対応する点で曲線に折れ曲がり（キンク）ができます．

例えば，次の 2 つの曲線は両方とも次数 3 で制御点の数と位置が同じですが，ノットが異なるため形状が異なります．完全多重ノットにより，曲線が強制的に関連する制御点を通るようになります．



(A): **knots = <0,0,0,1,2,3,4,4,4>**　　(B): **knots = <0,0,0,2,2,2,4,4,4>**

図 (40) (A): 開いた曲線は，曲線の次数（この場合は 3）に等しい完全多重ノットを開始点と終了点に持ちます．残りのノットは単純ノットです．(B): 真ん中の完全多重ノットはキンクを作り，曲線は関連する制御点を強制的に通過します．

一様ノット

開いた曲線において，ノットが一様な場合，次のような状態を満たしています．

ノットは完全多重ノットで始まり，単純ノットが続き，完全多重ノットで終わります．ノットの値は等間隔に増加します．これは，一般的な開いた曲線の場合です．後で説明するように，周期的な閉曲線の場合は異なります．

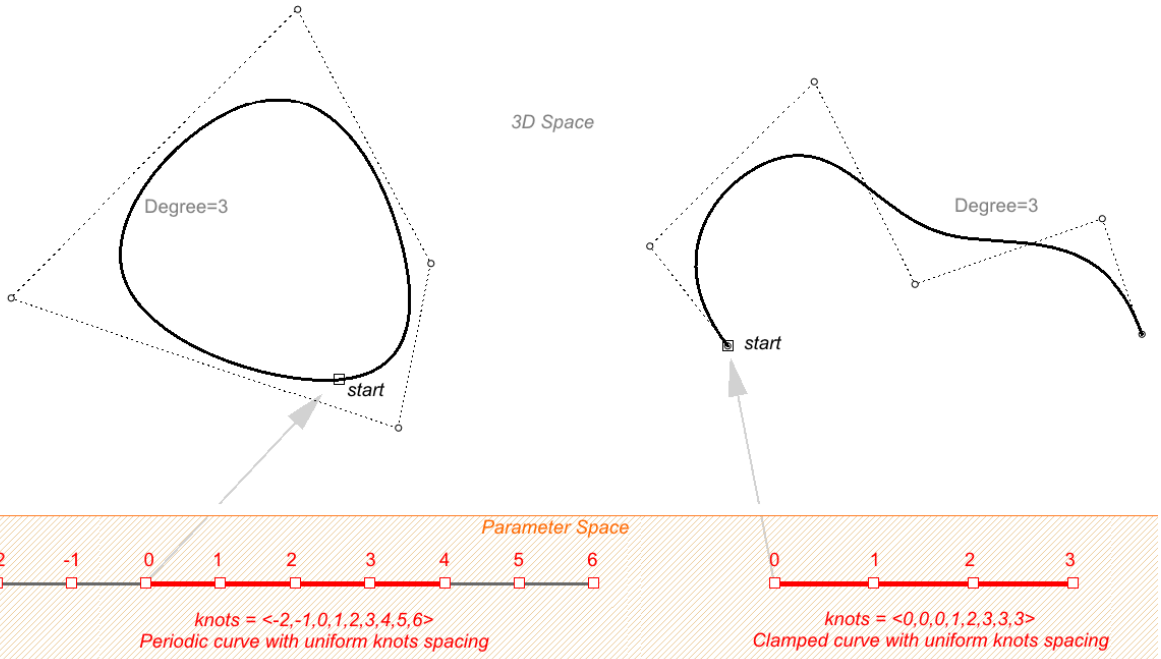


図 (41) 一様なノットリストとは，ノットの間隔が一定であることを意味します（ただし，開いた曲線の場合では，始点と終点が完全多重ノットとなるので，この部分は除外して考えます）．左側は周期曲線（Periodic curve，キンクなしで閉じています），右側は開いた曲線です．

非一様ノット

NURBS 曲線は，ノット間の間隔を非一様にすることができます．これにより，曲線の曲率がコントロールでき，より滑らかな曲線を作成できます．次の例は，左が非一様なノットリストを用いて点を補間した曲線，右が一様なノットリストを用いた曲線です．一般に，NURBS 曲線のノットの間隔が，制御点間の間隔に比例する場合，曲線はより滑らかになります．

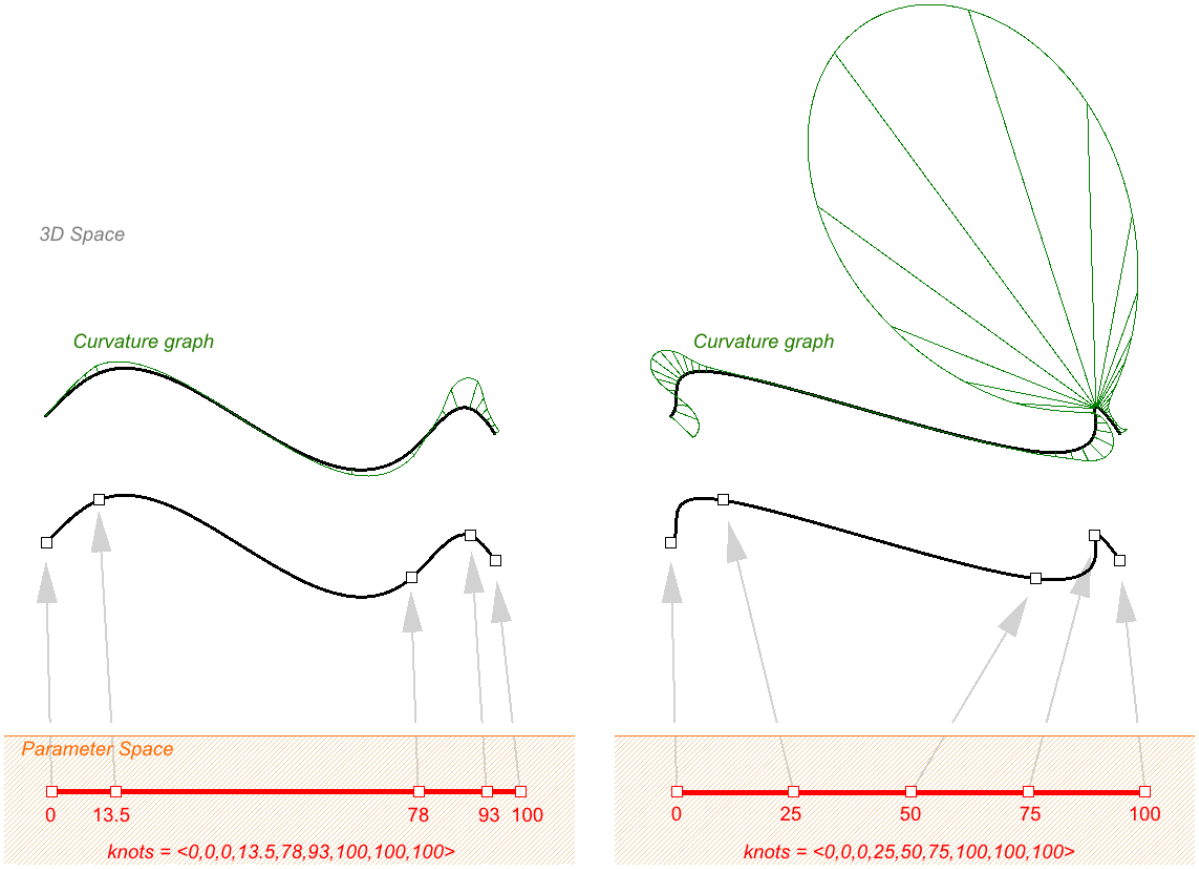


図 (42) 非一様なノットリストは，より滑らかな曲線を作成するのに役立ちます．左の曲線は，非一様なノットで点を補間しており，滑らかな曲率の曲線を生成しています．右の曲線は，同じ点を補間していますが，ノットの間隔を一様にしており，結果の曲線は滑らかではありません．

非一様と有理の両方の条件を満たす曲線の例は，NURBS 円です．以下は，9 つの制御点と 10 個のノットを持つ次数 2 の曲線です．ドメインは 0～4 で，ノット間隔は 0 と 1 で交互になります．

ノットベクトル = <0,0,1,1,2,2,3,3,4,4> --- (内部のノットが完全多重度を持つ)

ノットの間隔 = [0,1,0,1,0,1,0,1,0] --- (非一様)

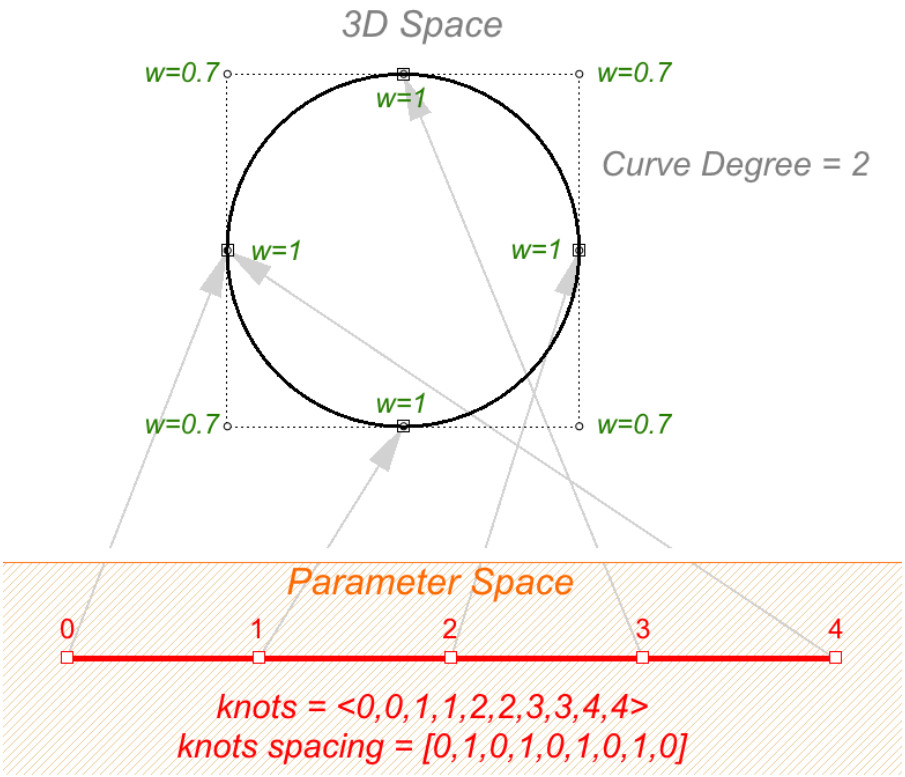


図 (43) NURBS による近似円は，有理で非一様な NURBS 曲線.

評価公式

評価公式として，曲線ドメインの数値を取得し，点を割り当てる数式を使用します．数式では，次数，制御点，ノットが考慮されます．

この式を用いると，特殊な曲線関数から，曲線パラメータを取得し，その曲線上に対応する点を生成できます．パラメータは，曲線ドメイン内の数値です．通常，ドメインは単調に増加し，2 つの数値で構成されます．ドメインの最小パラメータ（通常 t0 で表す）は，曲線の始点として評価され，最大パラメータ（t1）は曲線の終点として評価されます．

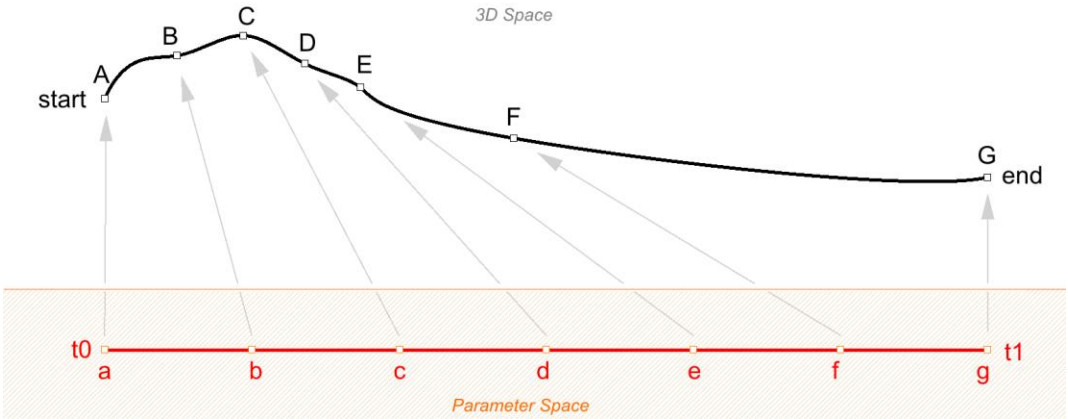


図 (44): パラメータ（a，b，c，…）による 3 次元曲線上の点（A，B，C，…）の評価．最小および最大パラメータ（t0 および t1）は，3 次元曲線の始点と終点を評価します.

NURBS 曲線の特徴

NURBS 曲線の作成のためには，次の情報が必要になります.

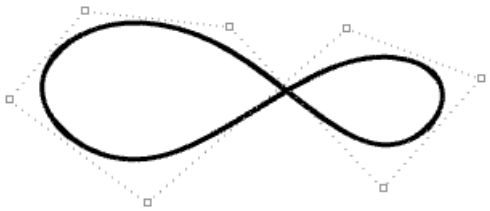
- 次元（通常は 3）
- 次数
- 制御点（点座標のリスト）
- 制御点の重み（数値のリスト）
- ノット（数値のリスト）

NURBS 曲線を作成するときは，少なくとも次数と制御点の位置を定義する必要があります．曲線作成に必要な残りの情報は自動的に生成されます．通常，終点を始点と一致するように選択すると，周期的な滑らかな閉じた曲線が作成されます．以下の表に，開いた曲線と閉じた曲線の例を示します.

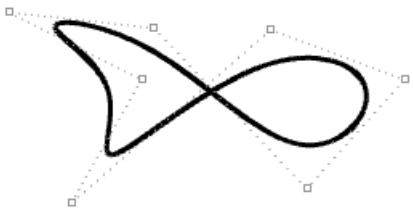
次数 1 の開いた曲線.
曲線はすべての制御点を通過します.

次数 3 の開いた曲線.
曲線の両端のみ，端部の制御点と一致します.

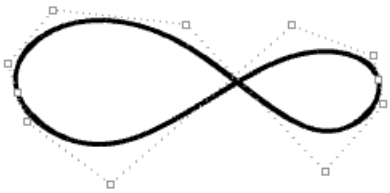
次数 3 の閉じた曲線.
曲線は制御点を通過しません.



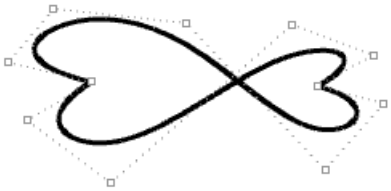
周期曲線（Periodic curve）は，制御点を移動させても，曲線の滑らかさを維持します.



曲線が制御点を通過する場合，キンク（折れ曲がり）が生成されます.



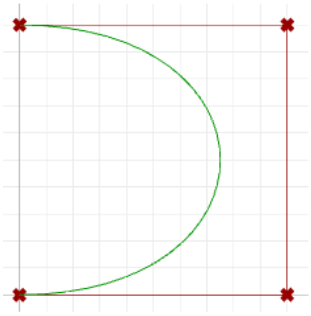
非周期曲線では，制御点を移動すると滑らかな連続性は保証されませんが，結果をより詳細に制御できます.



開いた NURBS 曲線と周期 NURBS 曲線の比較

開いた曲線の端点は，両端の制御点と一致します．周期曲線は，滑らかに閉じた曲線です．両者の違いを理解する最良の方法は，制御点とノットを比較することです.

以下は，開いたコの字の非有理（ウェイトが一様）な NURBS 曲線の例です．この曲線は，4 つの制御点から成り，両端のノットが完全多重ノットで，制御点のウェイトがすべて 1 です.



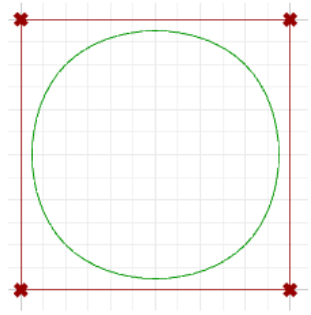
control points
{0.0, 0.0, 0.0}
{10.0, 0.0, 0.0}
{10.0, 10.0, 0.0}
{0.0, 10.0, 0.0}

knot vector
0.0
0.0
0.0
1.0
1.0
1.0

weights
1.0
1.0
1.0
1.0

図 (45)：次数 3 の開いた非有理 NURBS 曲線の分析.

以下の円形の曲線は，3 次の閉じた周期曲線の例です．ウェイトがすべて同じ値なのでこの曲線も非有理です．周期曲線では，より多くの制御点が必要で，少し重複していることに注意してください．ノットはすべて単純ノットです.



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}
{48.0, 12.0, 0.0}

knot vector
-2.0
-1.0
0.0
1.0
2.0
3.0
4.0
5.0
6.0

weights
1.0
1.0
1.0
1.0
1.0
1.0
1.0

図 (46)：次数 3 の閉じた周期 NURBS 曲線の分析.

周期曲線では，4 つの入力点が 7 つの制御点（次数+4）に変更されるのに対し，コの字の開いた曲線では，4 つの制御点のみが用いられています．周期曲線のノットは，単純なノットのみを使用するのに対し，コの字の開いた曲線の開始ノットと終了ノットは，完全な多重度を持ち，曲線は開始制御点と終了制御点を通過します.

前の例の次数を 3 ではなく 2 に設定すると，ノットはより小さい値となり，周期曲線の制御点の数が変化します.

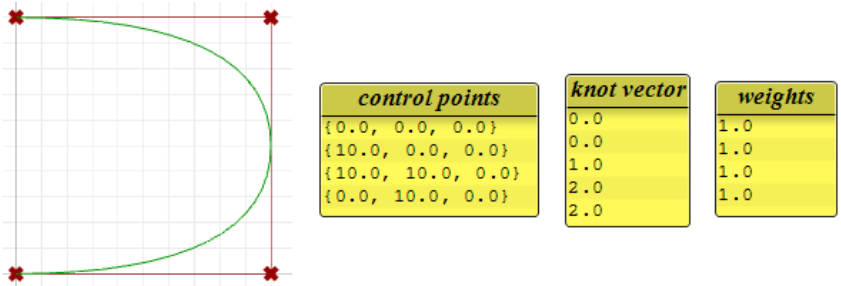


図 (47): 次数 2 の開いた NURBS 曲線の分析.

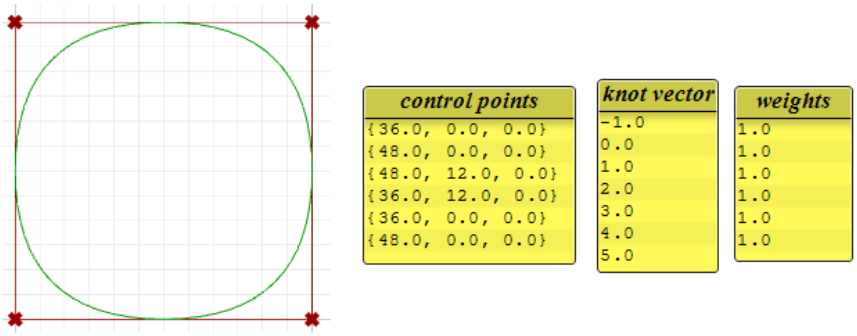


図 (48): 次数 2 の閉じた周期 NURBS 曲線の分析.

ウェイトの影響

均一な NURBS 曲線における制御点のウェイトは 1 に設定されますが，この値は変えることもできます（有理 NURBS 曲線）．次の例では，制御点のウェイトを変えた場合に曲線にどう影響するかを示しています.

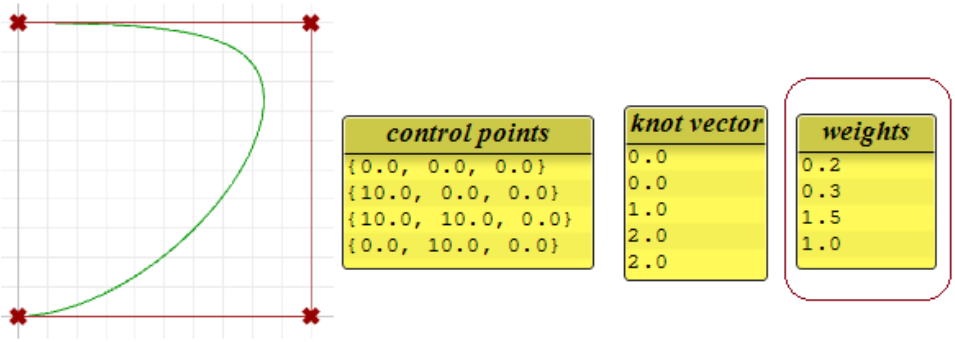


図 (49): 開いた NURBS 曲線におけるウェイトの分析.

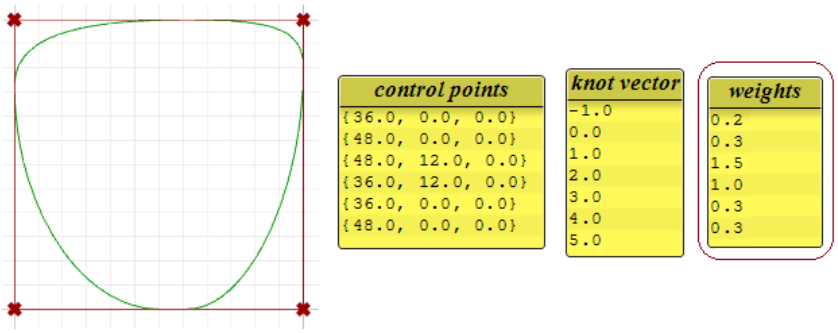


図 (50)：閉じた NURBS 曲線におけるウェイトの分析.

NURBS 曲線の評価

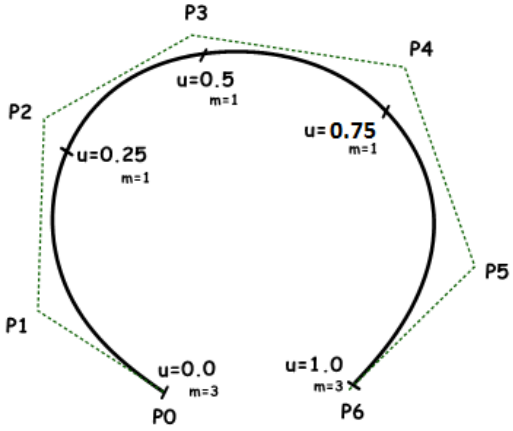
その発明者である Carl de Boor にちなんで名付けられたド・ボアのアルゴリズム[6]は，Bezier 曲線用のド・カステリョのアルゴリズムを一般化したものです．数値的に安定しており，3D アプリケーションで NURBS 曲線上の点を評価するために広く使用されています．以下は，ド・ボアのアルゴリズム[7]を使用して，次数 3 の NURBS 曲線上の点を評価する例です.

Input:

P_0 から P_6 の 7 点の制御点.

ノット:

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:

u=0.4 における曲線上の点.

[6] De Boor's algorithm. <https://en.wikipedia.org/wiki/De_Boor's_algorithm>

[7] De Boor's Algorithm. <https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/spline/de-Boor.html>

Solution:

1. 最初の反復計算の係数を計算します.

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

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

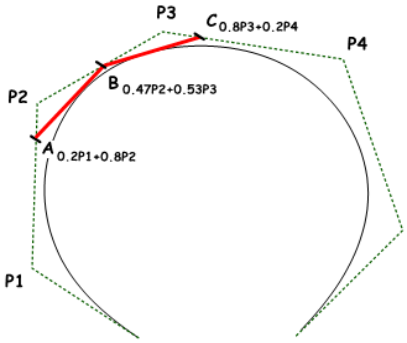
   $Cc = (u - u_3) / (u_{3+3} - u_3) = 0.2$

2. 求めた係数を使って点を算出します.

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

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

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



3. 2 回目の反復計算の係数を求めます.

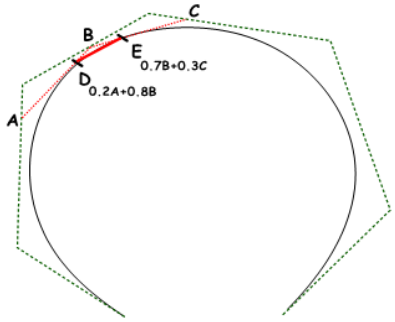
   $Dc = (u - u_2) / (u_{2+3-1} - u_2) = 0.8$

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

4. 求めた係数を使って点を算出します.

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

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

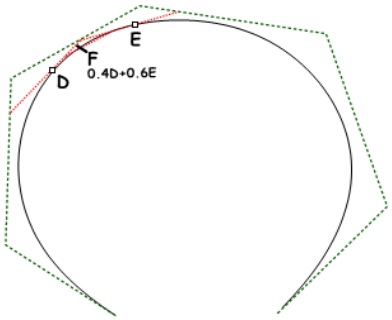


5. 最後の係数を計算します.

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

   パラメータ u=0.4 における曲線上の点を算出します.

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



曲線の幾何学的連続性

連続性（Continuity）は，3D モデリングにおける重要な概念です．連続性は，視覚的な滑らかさを実現し，滑らかな光と空気の流れを得るために重要です．

以下の表に，さまざまな連続性とその定義を示します．

G0 (位置連続)	2 つの曲線セグメントが互いに結合．
G1 (接線連続)	結合部における曲線セグメントの接線方向が一致．
G2 (曲率連続)	結合部における曲線セグメントの接線方向と曲率が一致．
GN	より高次で一致．

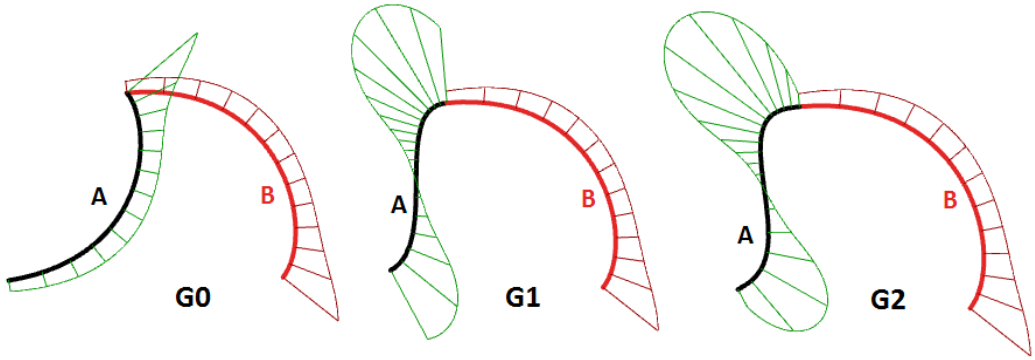


図 (51)：曲率グラフを用いた曲線の連続性の分析．

曲線の曲率

曲率（Curvature）は，3 次元の曲線や曲面のモデリングで広く利用される概念です．曲率は「円弧の単位長さにおける曲線の接線の傾きの変化率」として定義されます．円または球の場合は，半径の逆数となり，ドメイン全体で一定となります．

平面内の曲線上の任意の点において，この点を通過する曲線を最適に近似する線は接線です．また，この点を通り，曲線に接する最適な近似円を求めることもできます．この円の半径の逆数がこの点における曲線の曲率です．

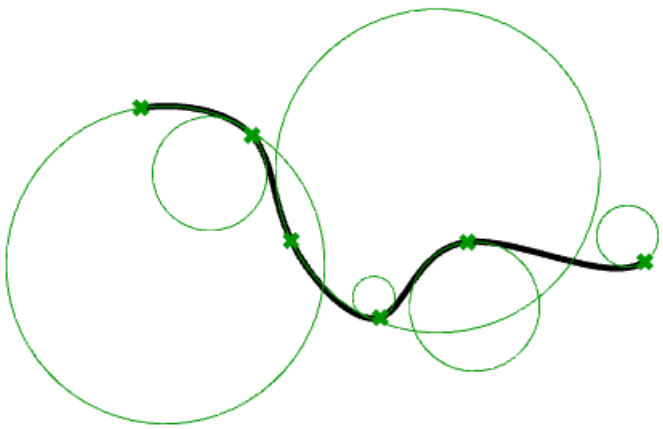


図 (52)：異なる点における曲線の曲率の分析．

最適な近似円は，曲線の左側または右側のいずれかにあります．これらを別物として扱う場合は，曲線が曲線の左側にある場合は正の符号を，曲線が曲線の右側にある場合は負の符号を与えるなどの規則を確立します．これは符号付き曲率として知られています．結合された曲線の曲率は，結合部における 2 つの曲線間の連続性を示します．

パラメトリック曲面

曲面パラメータ

パラメトリック曲面（サーフェス）は，2 次元領域における 2 つの独立したパラメータ（通常は u，v）の関数です．平面を例にとってみましょう．平面上に点 P と 2 つの非平行ベクトル a，b があるとき，2 つのパラメータ u，v を用いて，次のように平面のパラメトリック方程式を定義できます．

$$P = P' + u * \mathbf{a} + v * \mathbf{b}$$

ここで，

P′ は平面上の既知の点．

a は平面上の 1 つ目のベクトル．

b は平面上の 2 つ目のベクトル．

u は 1 つ目のパラメータ．

v は 2 つ目のパラメータ．

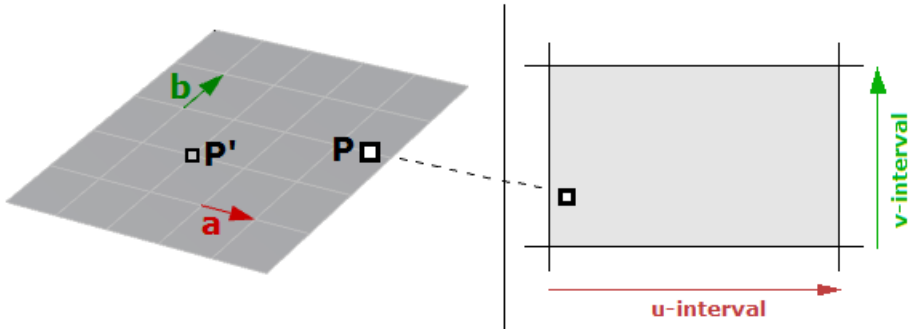


図 (53)：平面の 2 次元パラメータ領域.

もう 1 つの例は球です．直交座標系における原点が中心の半径 R の球面の方程式は以下です．

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

この式では，各点には 3 つの変数（x，y，z）があり，2 つの変数のみを必要とするパラメトリック表現に用いるのは有用ではないことがわかります．これに対し，3 次元極座標系で表した場合，各点は 3 つの値を用いて求められます．

r: 点から中心点への半径距離．

θ: xy 平面における x 軸からの角度．

ø: z 軸から点への角度．

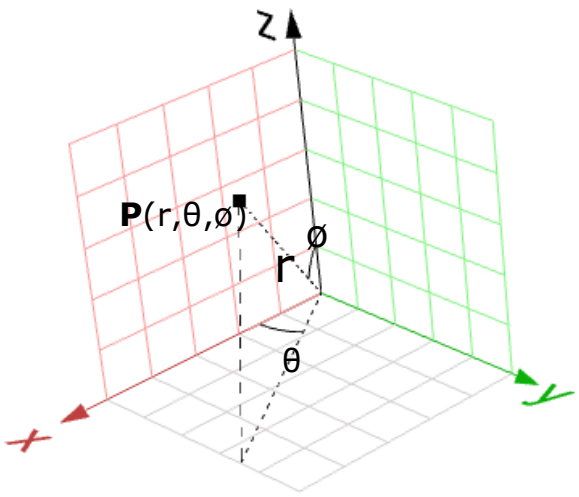


図 (54): 3 次元極座標系.

極座標から直交座標への変換は次の式で行うことができます.

x = ***r*** * sin(ø) * cos(θ)

y = ***r*** * sin(ø) * sin(θ)

z = ***r*** * cos (ø)

ここで,

r は中心からの距離で r ≥ 0.

θ は 0 ～ 2π.

ø は 0 ～ π.

r は球面で一定であることから，変数は 2 つしか残っていないため，これらを用いて球面のパラメトリック表現を作成できます.

u = θ

v = ø

すなわち,

x = **r** * sin(v) * cos(u)

y = **r** * sin(v) * sin(u)

z = **r** * cos(v)

ここで (u, v) は，ドメイン (0 to 2 π, 0 to π)です.

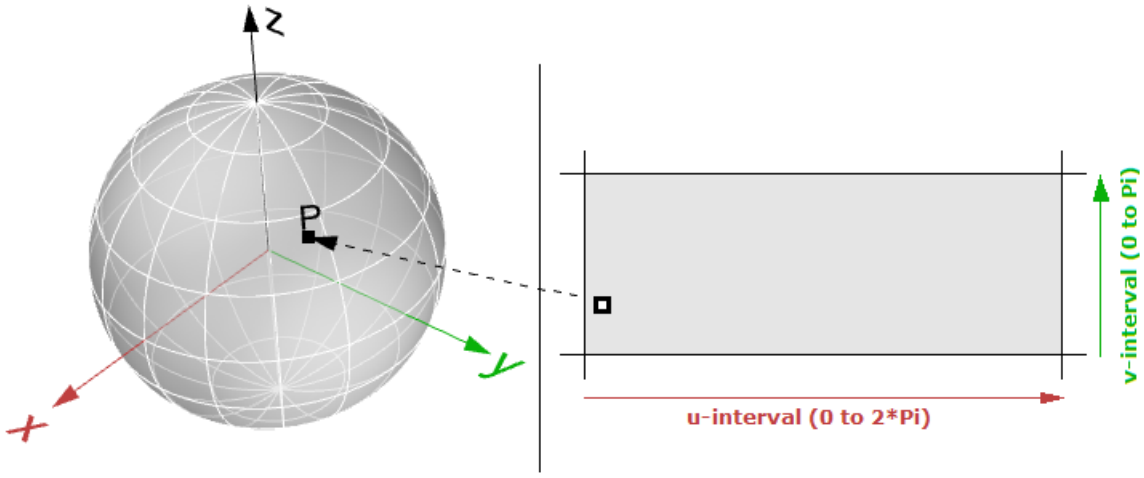


図 (55): 球の 2 次元パラメータ領域.

一般的な形式で表すと，パラメトリック曲面は次のように表現できます.

x = x(u,v)
y = y(u,v)
z = z(u,v)

ここで，u，v は曲面ドメイン内の 2 つのパラメータです.

曲面ドメイン

曲面ドメインは，その曲面上のそれぞれの点の位置で評価される（u，v）パラメータの範囲として定義されます．それぞれの方向（u または v）のドメインは通常，2 つの実数（u_min から u_max および v_min から v_max）で記述されます.

曲面ドメインの変更は，曲面の「Reparameterize（パラメータの再定義）」と呼びます．ドメインが単調増加することは，ドメインの最小値が表面の最小点を指すことを意味します．ドメインは通常単調に増加しますが，常にではありません.

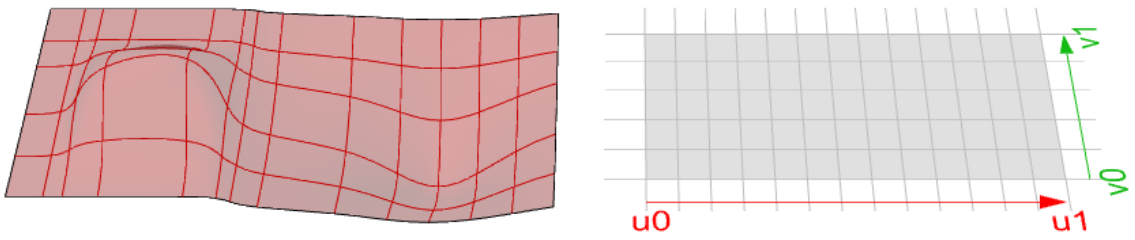


図 (56): 3D モデリング空間における NURBS 曲面 (左). 曲面パラメータの一方向のドメインが u0 から u1，もう一方向のドメインが v0 から v1 の曲面の 2 次元パラメータ領域 (右).

曲面の評価

曲面ドメイン内のパラメータを用いて曲面を評価すると，曲面上にある点になります．ただし，ドメインの中央の値（mid-u，mid-v）は必ずしも 3 次元曲面の中心点として評価されるとは限らないことに注意が必要です．また，曲面ドメイン外にある u 値と v 値を評価しても，有用な結果は得られません．

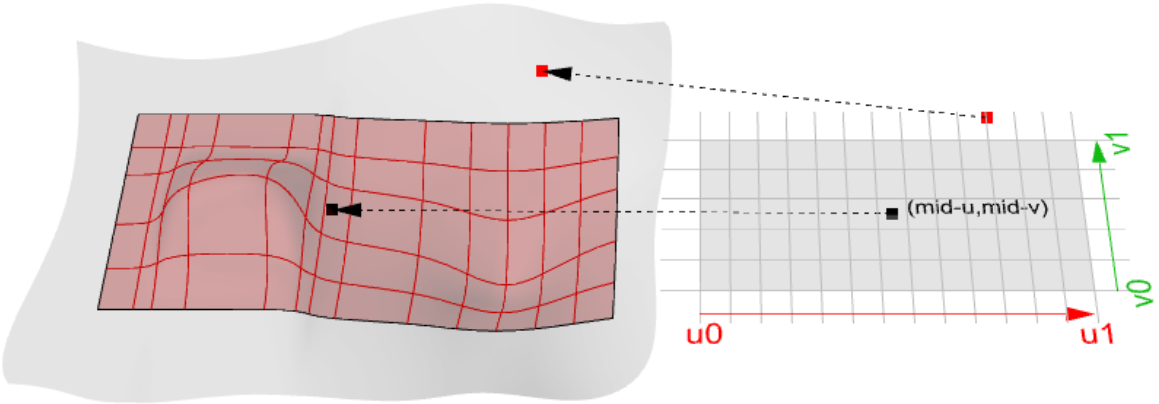


図 (57)：曲面（サーフェス）の評価.

曲面の接平面

任意の点における曲面の接平面（Tangent plane）とは，その点で曲面に接する平面です．接平面の z 方向は，その点における曲面の法線方向を表します．

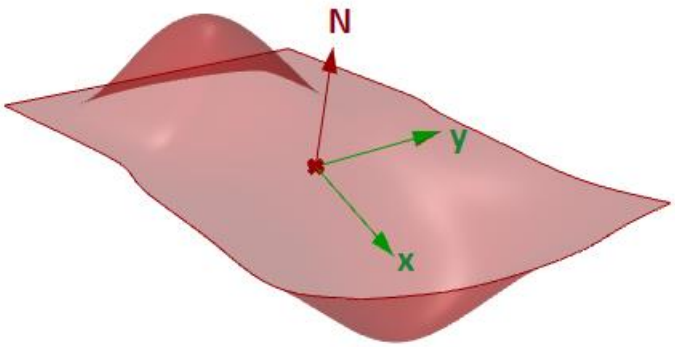


図 (58)：曲面における接線ベクトルと法線ベクトル.

曲面の幾何学的連続性

1 枚の曲面だけで構築できないモデルはたくさんあります．結合された曲面間の連続性は，視覚的な滑らかさや光の反射，エアフローなどにとって重要となります．

次の表では，さまざまな曲面連続とその定義を示しています．

G0 (位置連続)	2 枚の曲面が互いに結合.
G1 (接線連続)	2 枚の曲面の結合エッジにおける接線方向が u 方向，v 方向ともに一致.
G2 (曲率連続)	2 枚の曲面の結合エッジにおける接線方向と曲率が u 方向，v 方向ともに一致.
GN	より高次で一致.

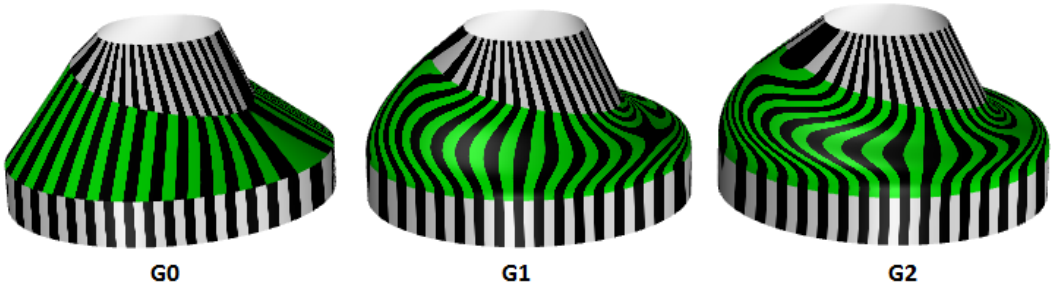


図 (59)：ゼブラ解析による曲面の連続性の分析.

曲面の曲率

曲面では，法曲率（normal curvature）が曲面の曲率の一般化の 1 つです．曲面上の点とその点における接平面上のベクトルが与えられると，それらとその点の法線を通る平面が得られます．その平面と曲面との交線から求まる符号付き曲率が，法曲率です．

接平面上の方向には無数の方向が考えられますが，これらすべての方向について法曲率を計算すると，最大値と最小値があることがわかります．

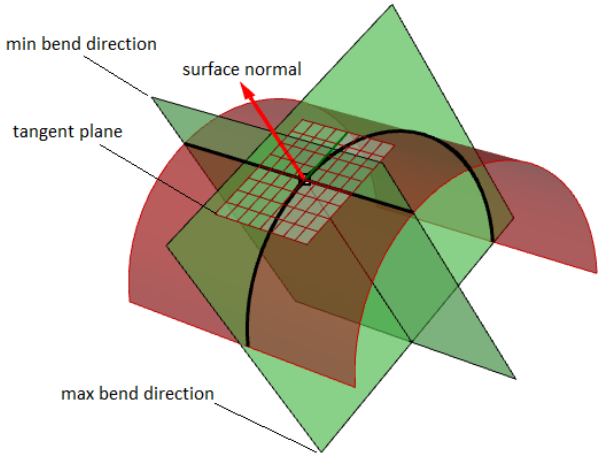


図 (60)：法曲率.

主曲率

ある点における曲面の主曲率（Principal curvature）とは，その点における法曲率の最小値と最大値です．それらは，その点での曲面の最大および最小の曲げ量を与えます．主曲率は，ガウス曲率（Gaussian curvature）と平均曲率（Mean curvature）を計算するために用いられます．

例えば，円柱曲面では，高さ方向に沿う曲げはありません（曲率はゼロに等しい）が，最大の曲げは，端面に平行な平面との交線（曲率は 1 /半径に等しい）です．これら 2 つが，その曲面の主曲率となります．

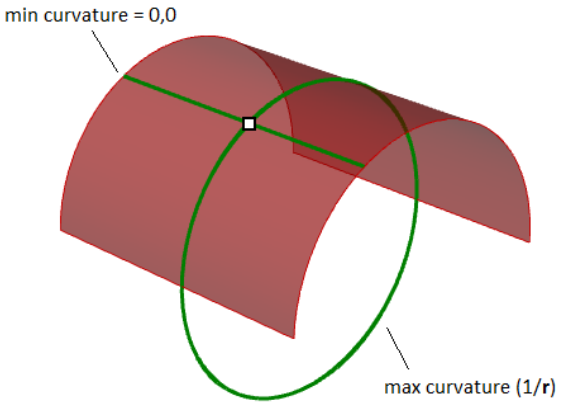
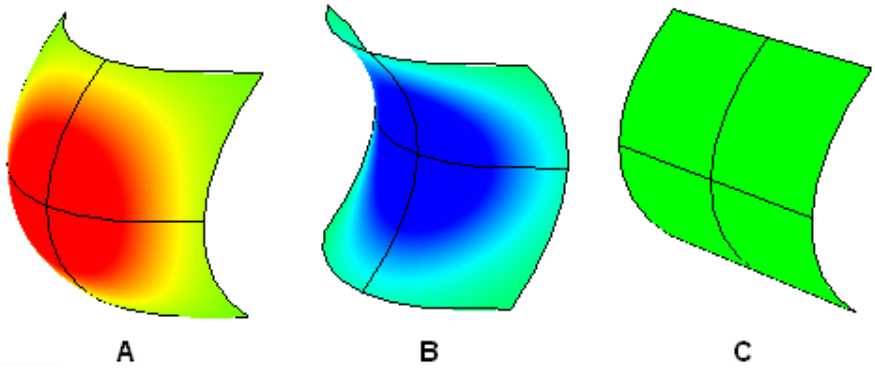


図 (61): 曲面上の点における主曲率には，最小と最大の曲率があります.

ガウス曲率

ある点における曲面のガウス曲率（Gaussian curvature）とは，その点での主曲率の積です．正のガウス曲率を持つ任意の点における接平面は，単一の点で局所的に曲面に接します．それに対し，負のガウス曲率を持つ任意の点における接平面は，曲面を分断します．



A: 曲面がボウルのような形の場合，ガウス曲率は正となります．

B: 曲面がサドルのような形の場合，ガウス曲率は負となります．

C: 少なくとも 1 方向がフラット（平面状や円柱状）な場合，ガウス曲率はゼロとなります．

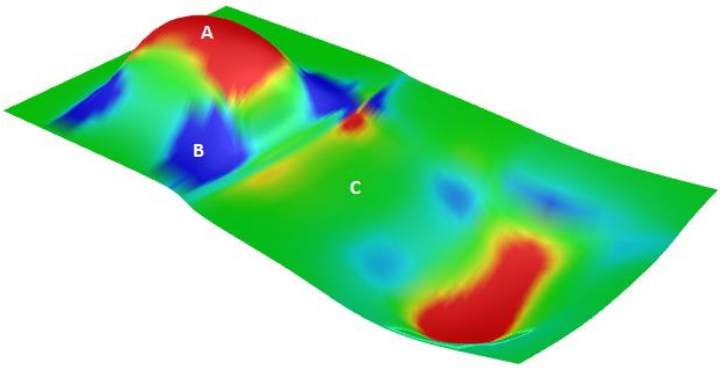


図 (62): ガウス曲率の解析.

平均曲率

ある点での曲面の平均曲率（Mean curvature）とは，その点での主曲率を合計し，半分に割った値です．平均曲率がゼロの場合，ガウス曲率は負またはゼロとなります．

平均曲率がどの位置でもゼロになる曲面を，極小曲面（minimal surface）と呼びます．極小曲面でモデル化できる物理プロセスとしては，ワイヤーの輪など固定された物体にできる石鹸膜の形成があります．石鹸膜は，空気圧（両側で等しい）によって歪むことなく，その表面積を最小限に抑えることができます．これは，一定量の空気を封入し，内部と外部で等しくない圧力を持つシャボン玉とは対照的です．平均曲率は，曲面の曲率が急激に変化する領域を見つけるのに役立ちます．

どの位置でも平均曲率が一定の曲面は，平均曲率一定（constant mean curvature：CMC）曲面と呼ばれます．CMC 曲面の例としては，シャボン玉の形成があり，浮遊している場合と物体に付着した場合の両方に当てはまります．シャボン玉は，単純な石鹸膜とは異なり，体積を持ち，気泡内のわずかに大きな圧力が気泡自体の面積を最小化しようとする力と均衡することによって平衡状態で存在します．

NURBS 曲面

NURBS 曲面は，2 方向に進む NURBS 曲線の格子と考えることができます．NURBS 曲面の形状は，2 つの方向（u 方向と v 方向）それぞれにおける制御点の数とその曲面の次数によって定義されます．NURBS 曲面は，自由曲面を高い精度で保存および表現するのに効率的です．NURBS 曲面に関する数式や詳細については，このテキストの範囲を超えていますので，デザイナーにとって特に重要な特徴にのみフォーカスします．

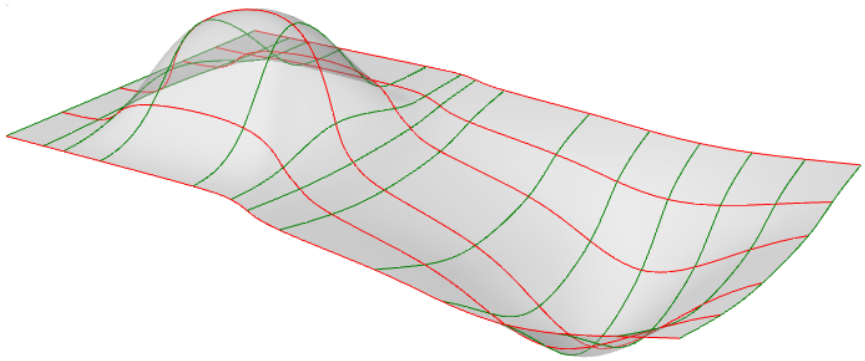


図 (63): U 方向に赤いアイソカーブ，V 方向に緑のアイソカーブを持つ NURBS 曲面.

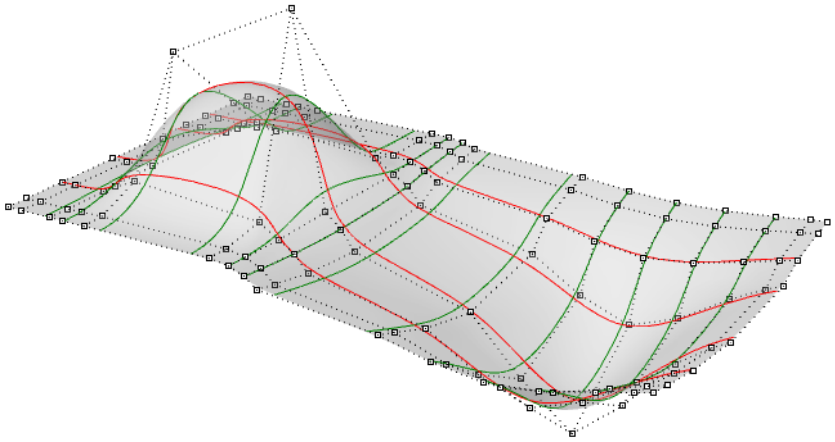


図 (64): NURBS 曲面の制御点.

図 (65): NURBS 曲面の 2 次元パラメータ領域.

2 次元パラメータの長方形で等間隔にパラメータを評価しても，ほとんどの場合，3 次元空間の等間隔には変換されません.

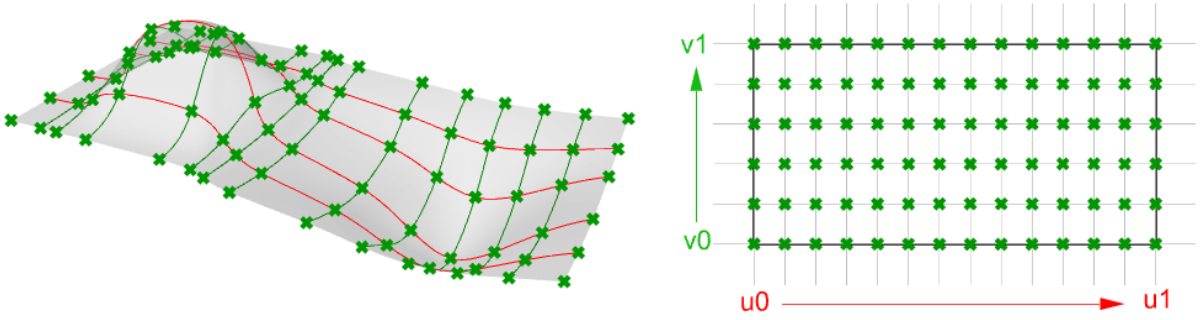


図 (66): 曲面の評価.

NURBS 曲面の特徴

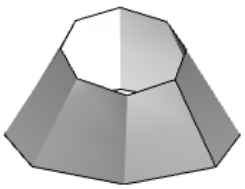
NURBS 曲面の特徴は，追加パラメータが 1 つあることを除いて，NURBS 曲線によく似ています．NURBS 曲面でも以下の情報を持ちます.

- 次元 (通常は 3)
- 次数 (u 方向，v 方向)

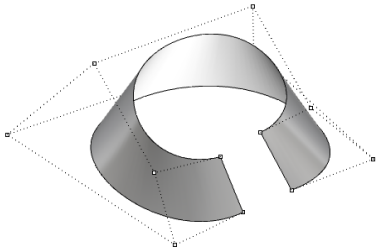
- 制御点 (点座標のリスト)
- 制御点の重み (数値のリスト)
- ノット (数値のリスト)

NURBS 曲線の場合と同様，3D モデラーでは曲面生成のための優れたツールが備わっているため，NURBS 曲面がどのように生成されるかの詳細を知る必要はないでしょう．曲面（曲線も同様）の次数と制御点は常に変更（リビルド）できます．曲面には，開いているもの，閉じているもの，または周期的なものがあります．曲面のいくつかの例を以下に示します．

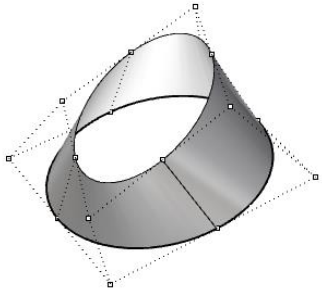
u 方向，v 方向の両方の次数が 1 のサーフェス．
制御点はすべてサーフェス上にあります．



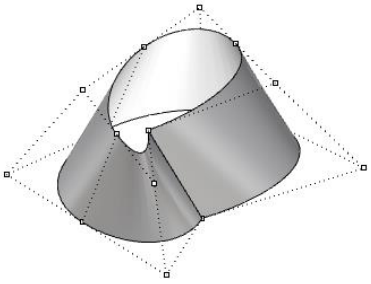
次数が，u 方向：3，v 方向：1 の開いたサーフェス．サーフェスのコーナーは，コーナーの制御点と一致します．



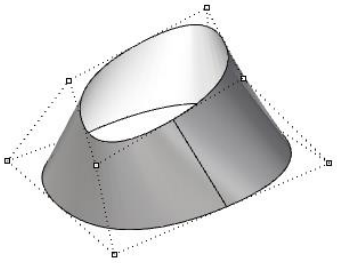
次数が，u 方向：3，v 方向：1 の閉じた（非周期的な）サーフェス．いくつかの制御点が，サーフェスのシームで重なります．



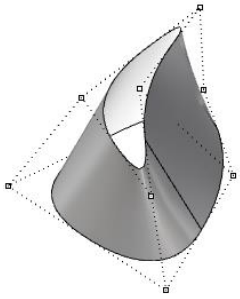
閉じた（非周期的な）サーフェスの制御点を移動するとキンクが発生し，サーフェスは滑らかに見えません．



次数が，u 方向：3，v 方向：1 の周期サーフェス（periodic surface）です．
サーフェスの制御点は，サーフェスシームと一致しません．



周期サーフェスの制御点を移動しても，表面の滑らかさに影響したり，キンクが発生することはありません．



NURBS 曲面の特異点

例えば，シンプルな平面の直線エッジがあった場合に，エッジの端の 2 つの制御点をドラッグして中央で重なる（折りたたむ）ようにすると，特異なエッジが得られます．曲面のアイソカーブは特異点（Singularity）で収束することがわかります．

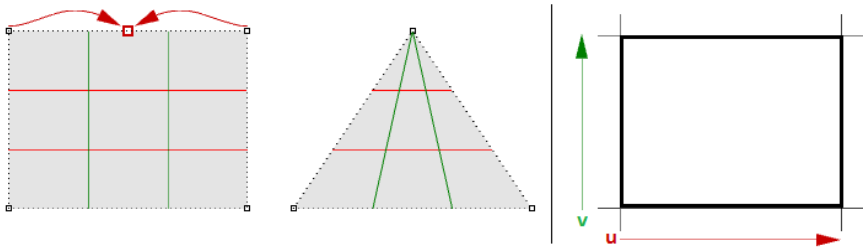


図 (67): 長方形の NURBS 曲面の 2 つのコーナーを重ね，特異点を持つ三角形の曲面を作成．2 次元パラメータ領域の長方形は長方形のままです．

上記のような三角形は，三角形のポリラインで曲面をトリムすれば特異点なしで作成できます．ただし，制御点を表示し，基礎となる NURBS 構造を確認すると，長方形の形状のまま（トリムサーフェス）であることがわかります．

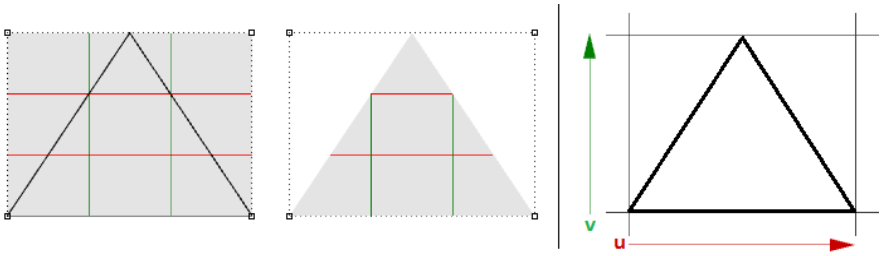


図 (68): 長方形の NURBS 曲面をトリムし，三角形のトリムサーフェスを作成．

特異点なしで生成するのが難しい曲面の例は，円錐と球です．円錐の上端と球の上下のエッジは，1 点に集約されます．特異性があるかどうかに関わらず，パラメータ領域は，ほぼ長方形の領域を維持します．

トリムされた NURBS 曲面

NURBS 曲面は，トリムしたり，トリム解除したりできます．トリムサーフェスは，ベースとなる NURBS 曲面と境界となる曲線を用いて，その曲面の一部をトリムします．それぞれの曲面には，外側の境界を定義する 1 つの閉じた曲線（Outer loop）があり，交差しない閉じた内側の曲線（Inner loop）を用いて，穴を定義します．Outer loop でできる曲面が基礎となる NURBS 曲面で，穴のない曲面を非トリムサーフェス（Untrimmed surface）と呼びます．

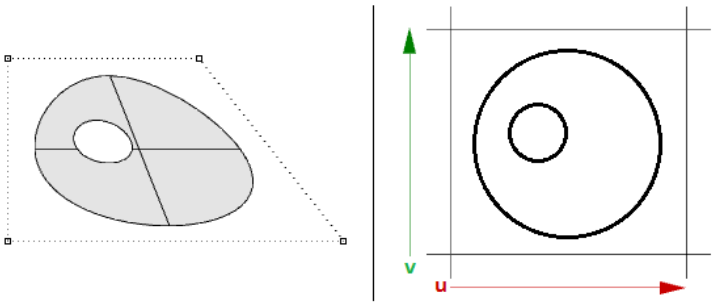


図 (69): モデル空間のトリムサーフェス (左) とそのパラメータ領域 (右).

ポリサーフェス

ポリサーフェスとは，2 つ以上の（トリムされていても良い）NURBS 曲面が結合されて構成された複合曲面のことを表します．それぞれの曲面は，独自の構造やパラメータ，アイソカーブの方向を持ち，それらは同じである必要はありません．ポリサーフェスは，境界表現（Boundary Representation: BRep）を用いて表現されます． BRep では，サーフェス，エッジ，頂点を，トリム情報と異なるパーツ間の接続性とともに記述します．また，トリムサーフェスも BRep のデータ構造を用いて表されます．

図 (70): ポリサーフェスは，共通のエッジが許容範囲内で完全に一致した結合されたサーフェスから作成られます.

BRep は，曲面とそれらを囲むエッジや頂点，トリム情報，隣接する面との関係性を用いて，それぞれの面を記述するデータ構造です．また，BRep オブジェクトは，閉じている（水密状態の）場合，「ソリッド」とも呼ばれます．

ポリサーフェスの例の 1 つは，6 つの非トリムサーフェスが結合したシンプルなボックスです.

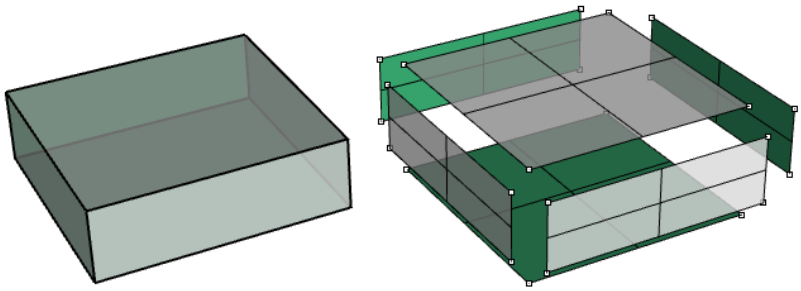


図 (71)：1 つのポリサーフェスとして，結合された 6 つの非トリムサーフェスで構成されるボックス.

以下の例のトップのようにトリムサーフェスを使っても同じボックスを作ることができます.

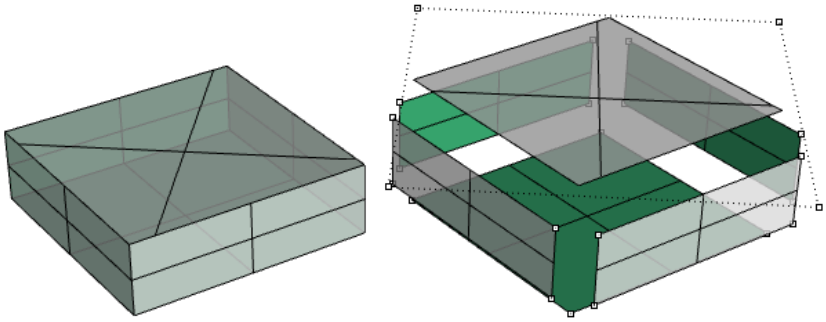


図 (72)：ポリサーフェスの面はトリムされていても良い.

以下の例の円柱のトップとボトムは，平面サーフェスからトリムされています.

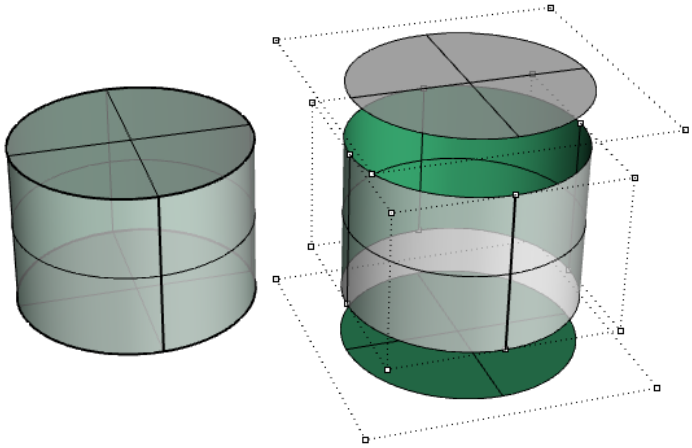


図 (73) それぞれの面の制御点の表示.

NURBS 曲線と非トリムサーフェスの編集は，直感的に行うことができ，制御点を移動することでインタラクティブに実行できます．しかし，トリムサーフェスとポリサーフェスの編集は困難な場合があります．主な課題は，さまざまな面が結合されたエッジを望ましい許容範囲内に維持できるようにすることです．共通のエッジを共有する隣接する面はトリムでき，通常は一致する NURBS 構造を持たないため，その共通のエッジを変形するようにオブジェクトを変更すると，ギャップが生じる可能性があります．

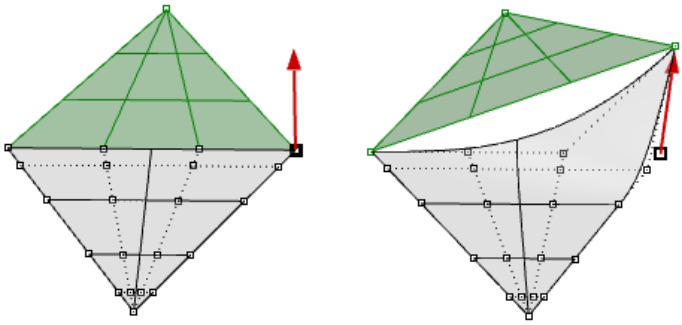


図 (74): 2 つの三角形の面が 1 つのポリサーフェスとして結合されていますが，一致する結合エッジがありません． 1 つのコーナー点を移動させると隙間ができます.

もう 1 つの課題は，特にトリムされたジオメトリを変更する場合は，結果に影響する制御点が一般的に少ないことです．

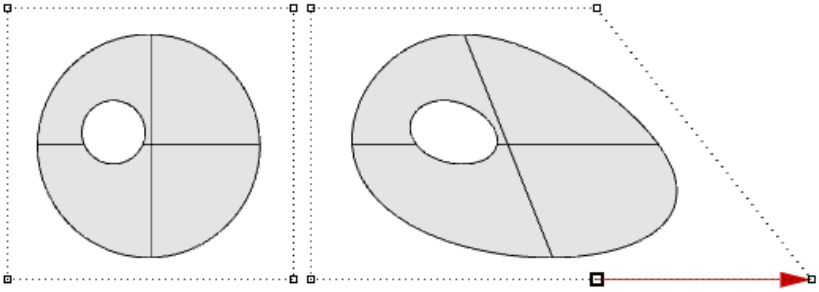


図 (75): トリムサーフェスを生成したとき，結果を編集する制御点が限られています.

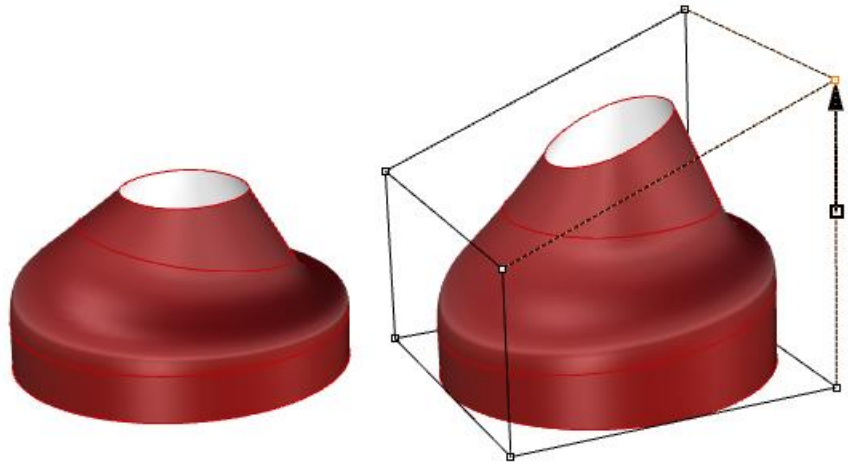


図 (76): Rhino の [Cage Edit] 機能を用いて，ポリサーフェスを編集します.

トリムサーフェスは，3D 曲面内のエッジを評価する 2D トリム曲線と非トリムサーフェスを用いて，パラメータ空間で記述されます.

チュートリアル

このチュートリアルでは，この章で学習した概念を用います（Rhinoceros 5 と Grasshopper 0.9 を使用しています）.

曲線間の連続性

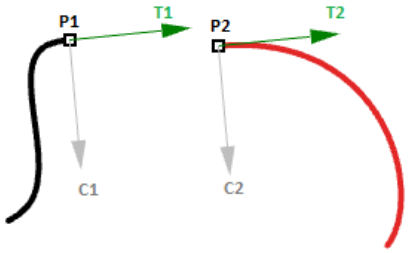
入力した 2 つの曲線間の連続性を調べます．連続性は，曲線が最初の曲線の終点と 2 番目の曲線の始点で一致することを前提としています.

Input:

2 つの入力曲線.

Parameters:

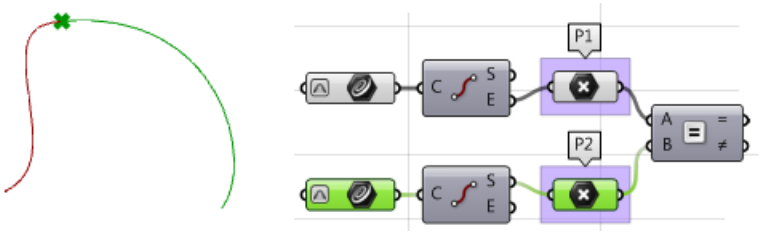
2 つの曲線間の連続性を決定できるように，以下を計算します.



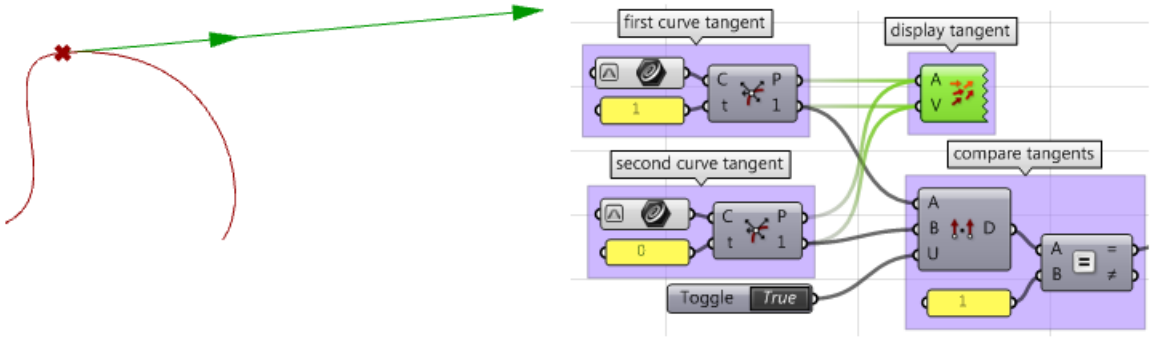
- 最初の曲線の終点 (P1)
- 2 番目の曲線の始点 (P2)
- 最初の曲線の終点と 2 番目の曲線の始点における接線 (T1，T2).
- 最初の曲線の終点と 2 番目の曲線の始点における曲率 (C1，C2).

Solution:

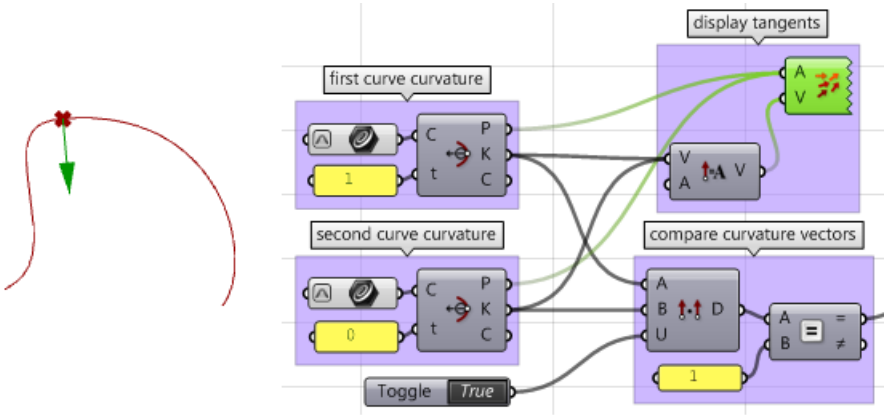
1. 入力曲線を Reparameterize します．そうすると，曲線の始点が t = 0，終了点が t = 1 で評価されるようになります.
2. 2 つの曲線の終点と始点を抽出し，それらが一致するかどうかを確認します．一致する場合，2 つの曲線は少なくとも G0 連続です.



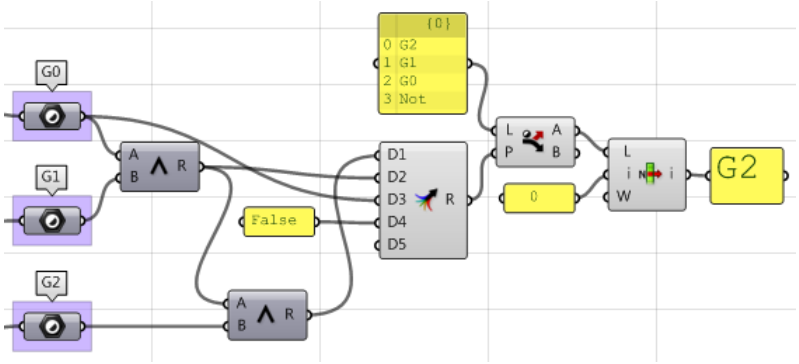
3. 接線を計算します.
4. 内積を利用して接線を比較します．必ずベクトルを単位ベクトル化（Unitize）してください．曲線が平行（内積が 1）であれば，少なくとも G1 連続です.



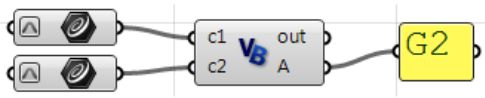
5. 曲率ベクトルを計算します.
6. 曲率ベクトルを比較し，それらが一致する場合，2 つの曲線は G2 連続です.



7. 3 つの結果（G1，G2，および G3）をフィルタリングするロジックを作成し，最も高い連続性を抽出します.

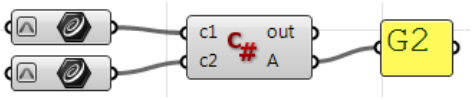


[VB Script] コンポーネントを用いた場合**:**



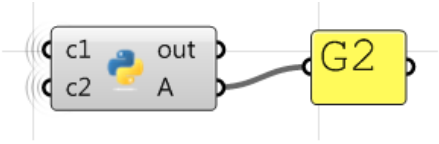

```
Private Sub RunScript(ByVal c1 As Curve, ByVal c2 As Curve, ByRef A As Object)

  '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
```

[C# Script] コンポーネントを用いた場合**:**




```
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;
}
```

[GhPython Script] コンポーネントを用いた場合:



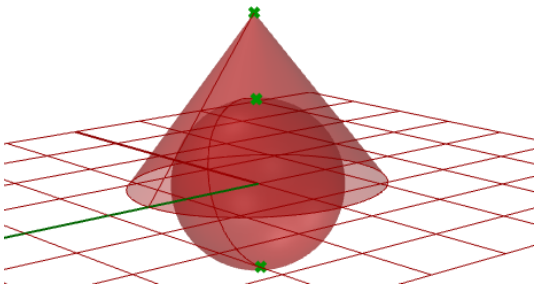

```
#decalre variables
continuity = ""

#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
```

サーフェスの特異点

球と円錐の特異点を抽出します.

Input:

1 つの球と 1 つの円錐.



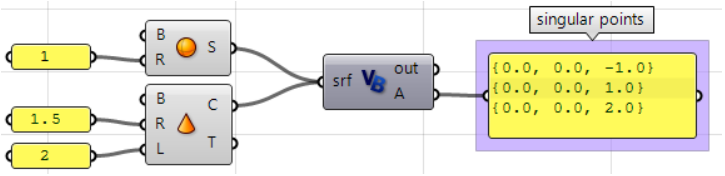
Parameters:

特異点は，無効または長さゼロに対応するエッジを持つ 2 次元パラメータ領域のトリム情報を分析することで検出できます．それらのトリム情報には特異なものがあるはずです.

Solution:

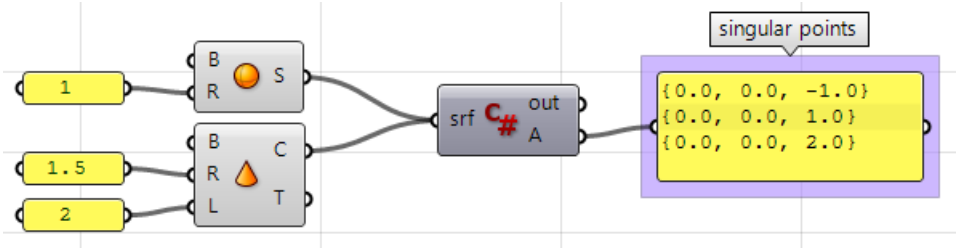
1. すべてのトリム情報を入力します.
2. トリムをチェックし，無効なエッジがある場合は，特異なトリムとしてフラグを立てます.
3. 3 次元空間での点の位置を抽出します.

[VB Script] コンポーネントを用いた場合:



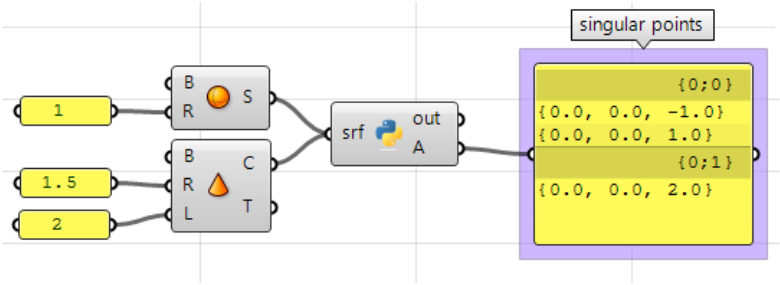

```
Private Sub RunScript(ByVal srf As Brep, ByRef A As Object)

  '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
```

[C# Script] コンポーネントを用いた場合**:**




```
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;
}
```

[GhPython Script] コンポーネントを用いた場合**:**




```
#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
```

参考文献

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