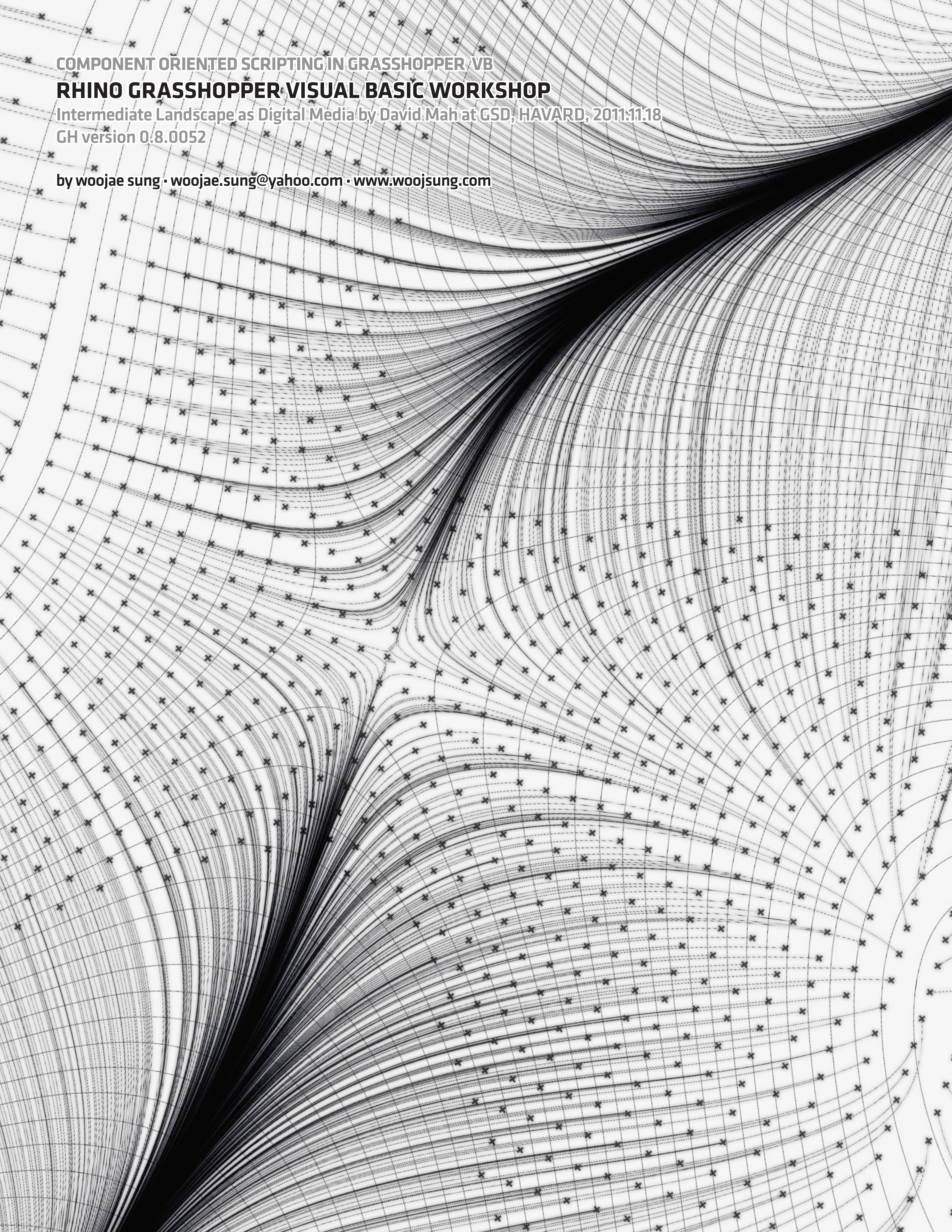
COMPONENT ORIENTED SCRIPTING IN GRASSHOPPER VB

# RHINO GRASSHOPPER VISUAL BASIC WORKSHOP

Intermediate Landscape as Digital Media by David Mah at GSD, HAVARD, 2011.11.18

GH version 0.8.0052

by woojae sung · woojae.sung@yahoo.com · www.woojsung.com



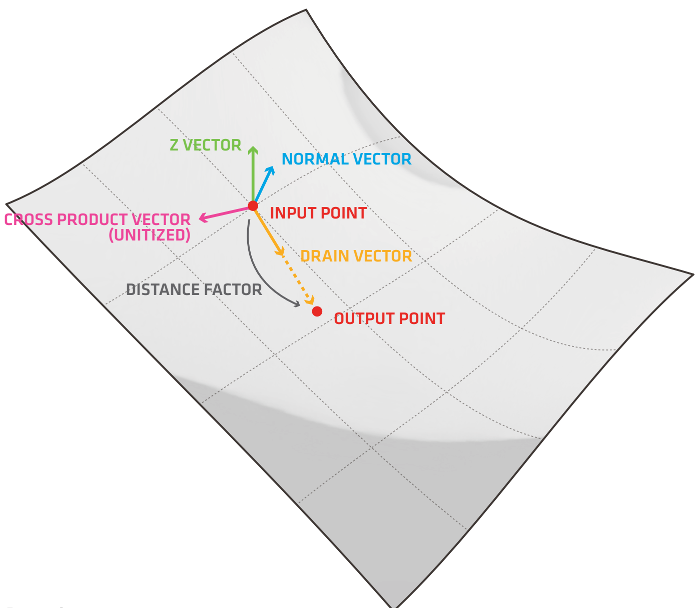
# IDEA

The goal of the definition is to simulate the way how water flows downwards on a hilly terrain. Let's start by dividing the process into three parts;

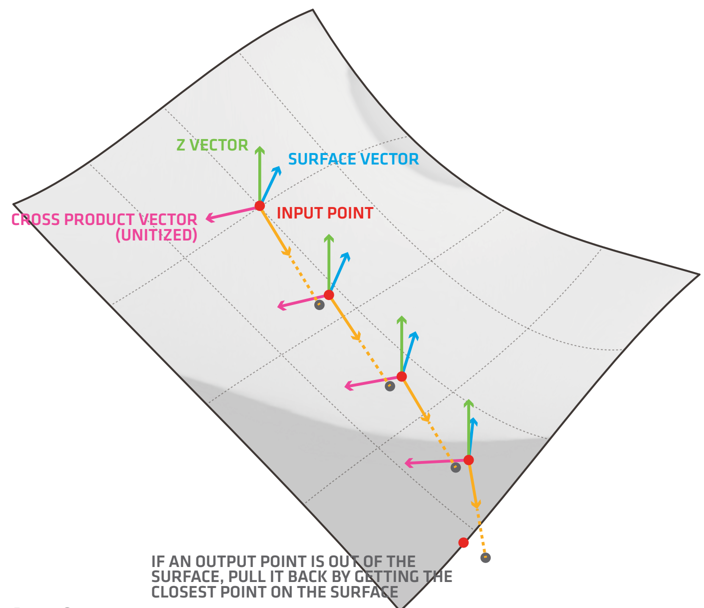
***Part 1*** *- Define a drain slope vector at an arbitrary point, called 'input point' in our definition, on a give surface, then decide possible position of an 'output point' on the surface. When we are given a vector at a specific time and position, by multiplying a factor, we can predict what next position would be. In our case, we call the factor 'distance_factor' and this number will decide how precise the process would be.*

***Part 2*** *- In Part 1, we got an 'output point' from an initial 'input point'. If we use the 'output point' as another 'input point', we can get another 'output point'. By repeating this until we cannot get a valid 'output point', we can get a water flow curve from the series of points.*

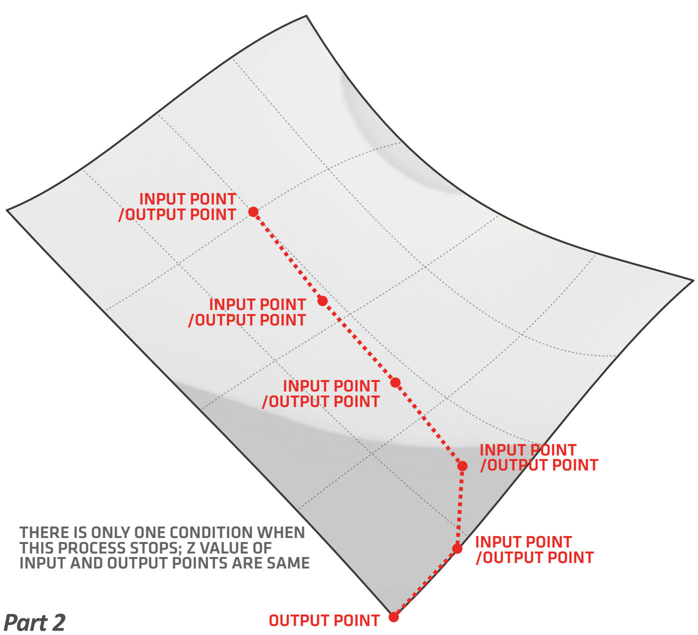
***Part 3*** *- By supplying multiple 'input points' to the process explained in the previous steps, we can simulate water flow from multiple source on a given surface.*



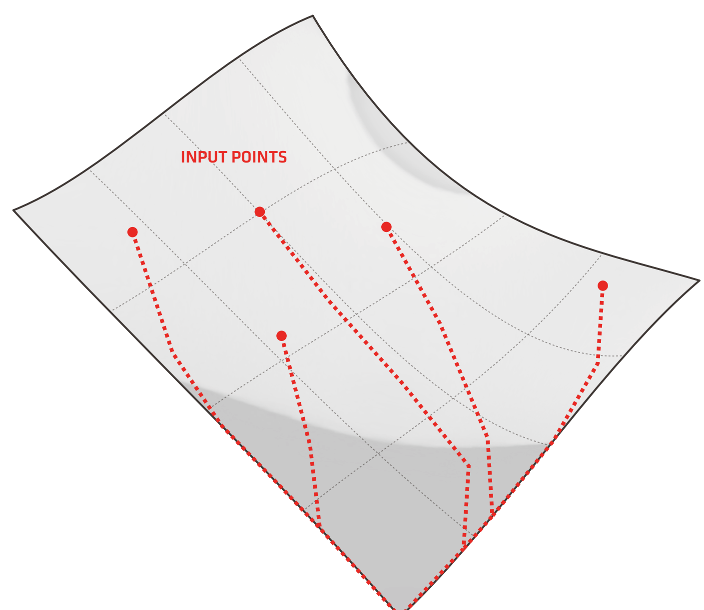

***Part 1***



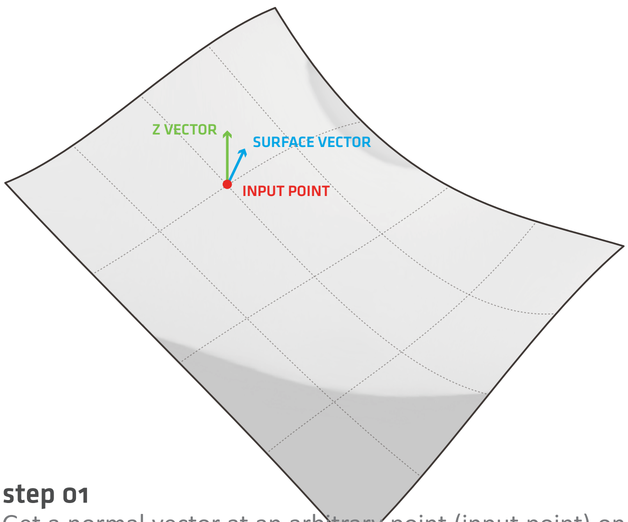

***Part 2***



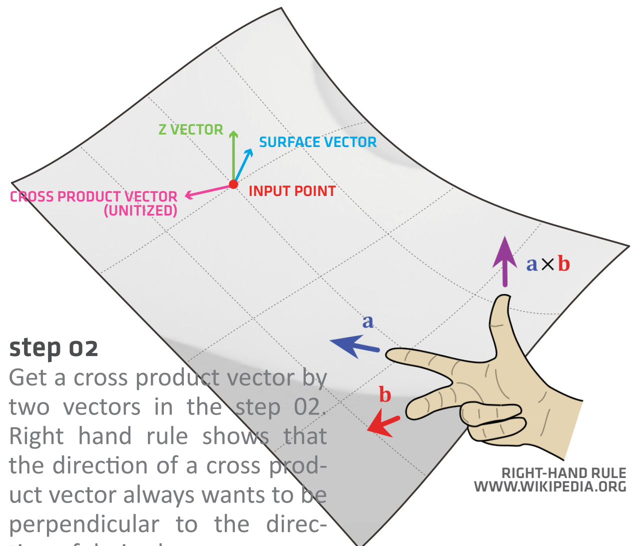

***Part 2***



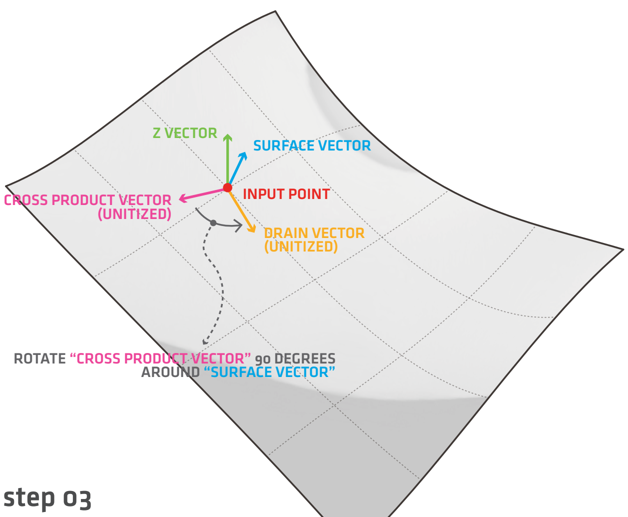

***Part 3***



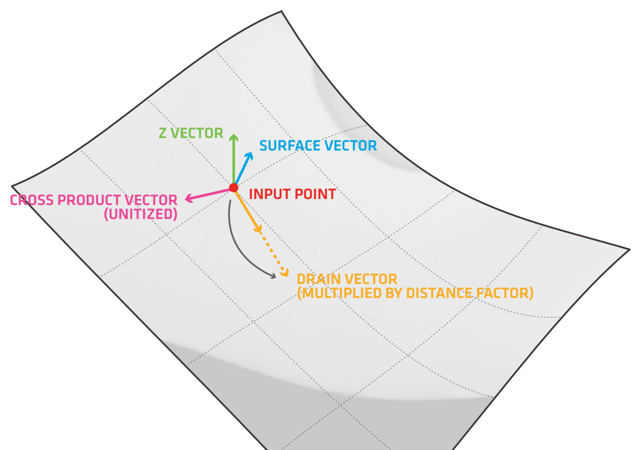

### step 01
Get a normal vector at an arbitrary point (input point) on a given surface. Get a Z vector at the point.



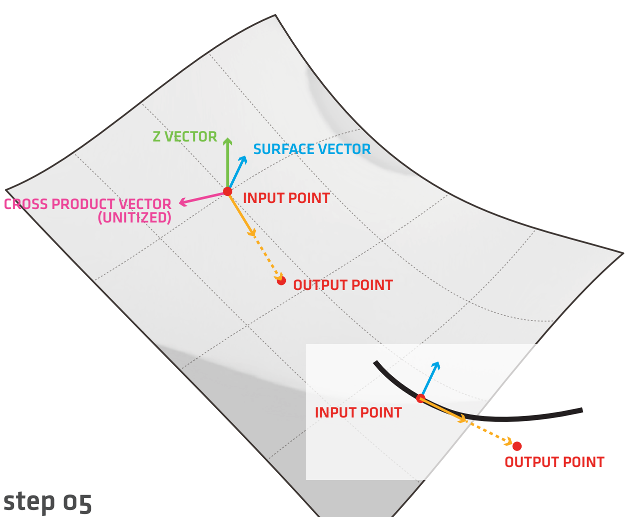

### step 02
Get a cross product vector by two vectors in the step 02. Right hand rule shows that the direction of a cross product vector always wants to be perpendicular to the direction of drain slope.



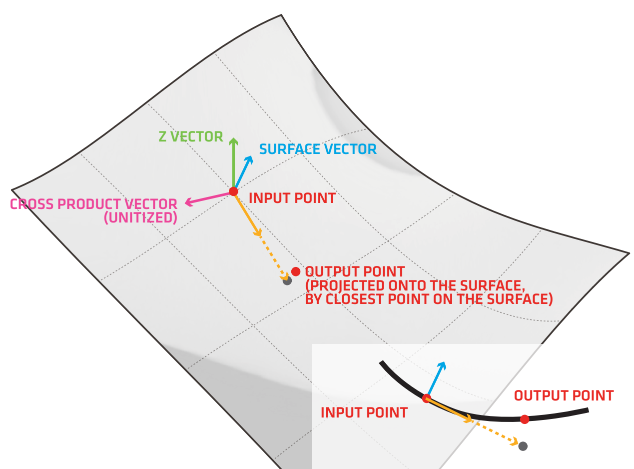

### step 03
Rotate 'cross product vector' 90 degrees count clock wise around 'surface normal vector to get 'drain slope vector'




### step 04
Multiply a certain number to the unitized 'drain slope vector' to get a output point. We call the number 'distance_factor' and it determines how accurate this process would be.

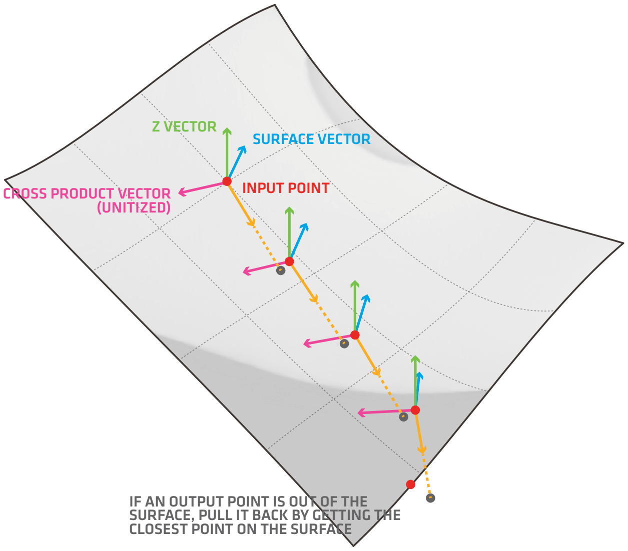



### step 05
Because of the surface curvature, an 'output point' most likely not on the surface.



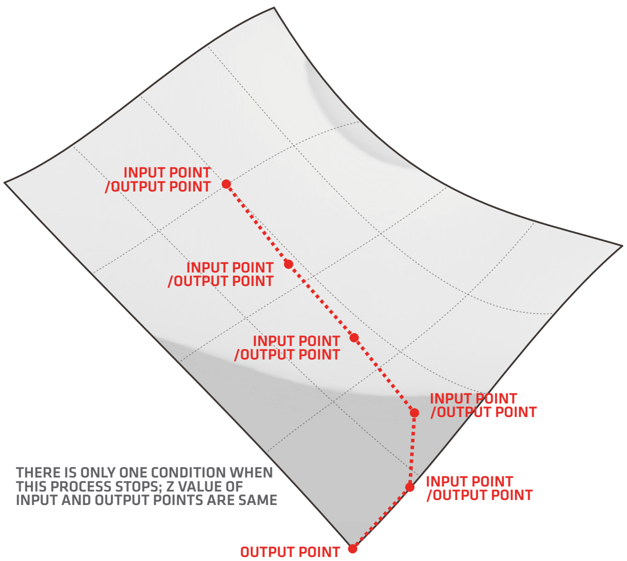

### step 06
So we want to pull the point back onto the surface by finding closest point from the point.

## PART 2 DEFINING A 'FLOW LINE'




**step 01**

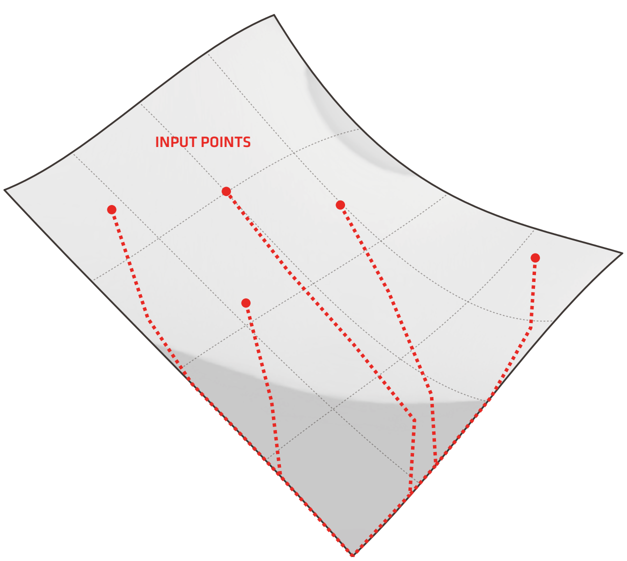
Repeat the process by continuously replacing an 'input point' in current iteration with an 'output point' in the previous one. If an 'output point' is out of the surface, we always can pull it back onto the surface by finding the closest point on it.




**step 02**

There is only one case when this chain reaction stops; when z value of both an 'input' and 'output' point are same. This means for whatever reason, an output point is not moving any further from an input point.

## PART 3 APPLYING TO MULTIPLE WATER SOURCES




**step 01**

We can get multiple flow lines by supplying series of input points depending on one's design intent.

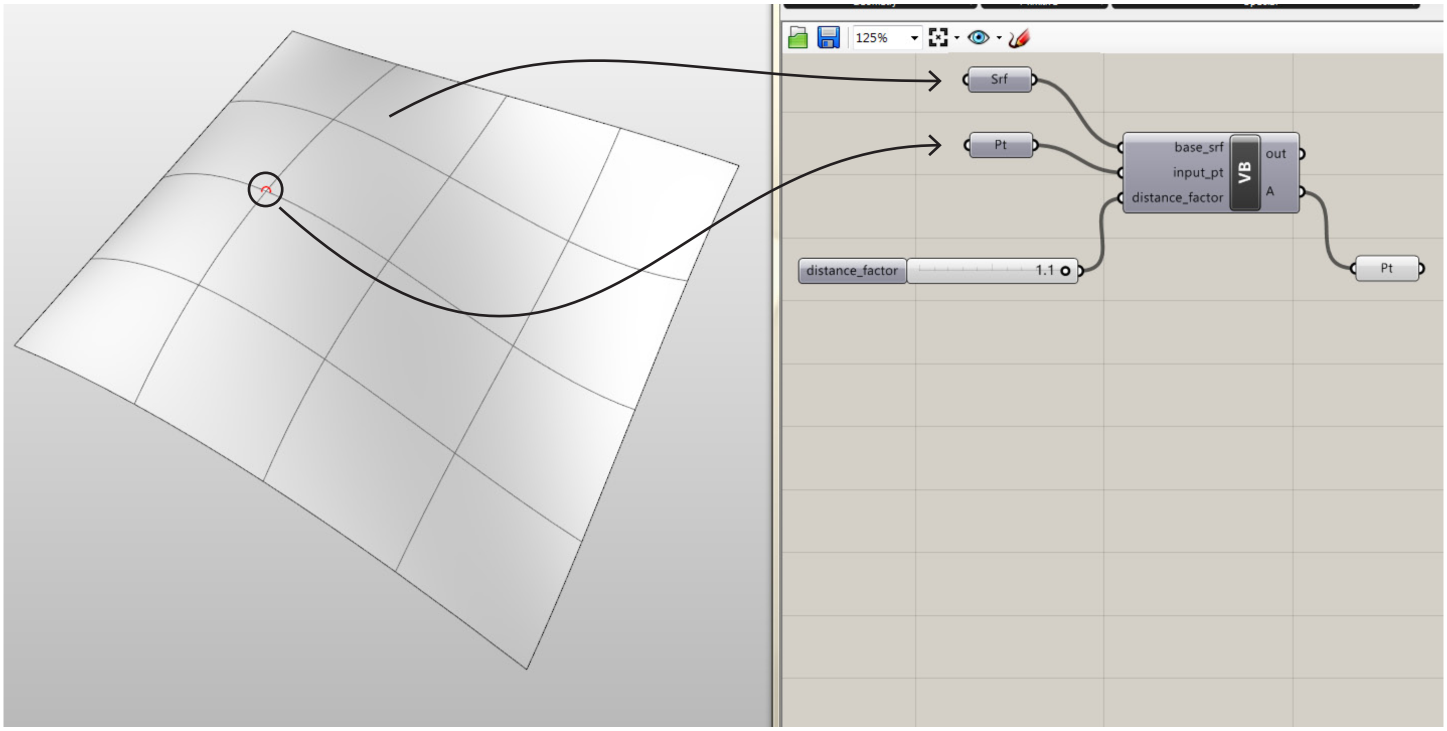
# CODE REVIEW

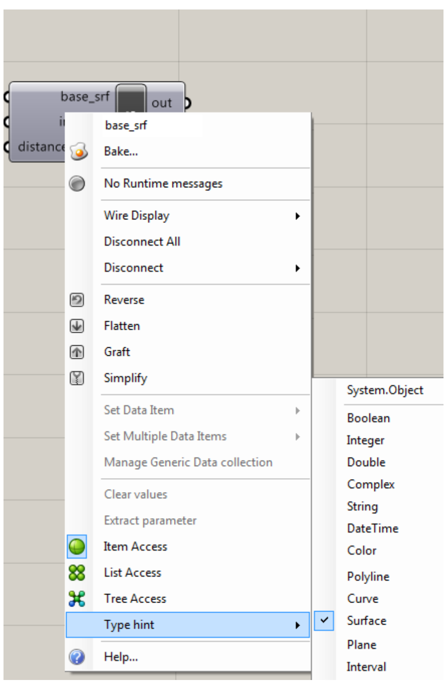
## PART 1 **FINDING AN 'OUTPUT POINT'**

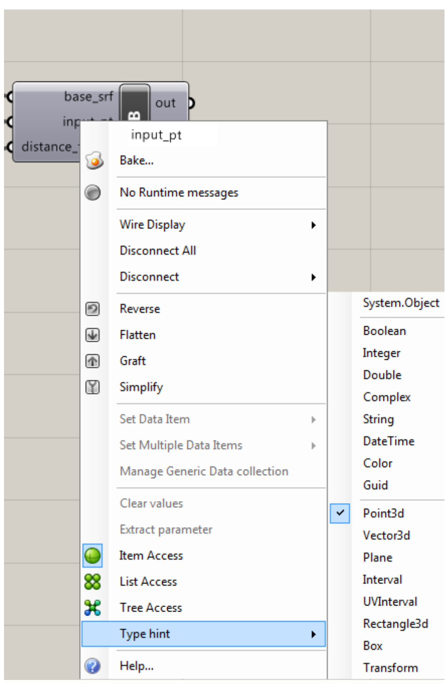
Refer to 'VB workshop part1.gh' and 'VB workshop.3dm' attached.

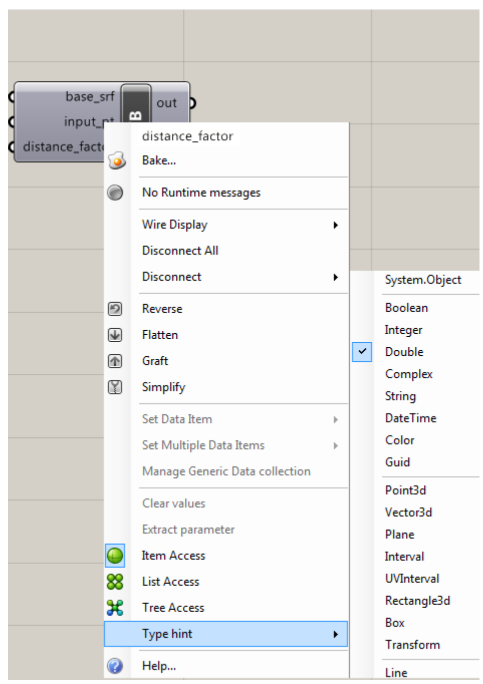
### scene setting

Set up the scene as illustrated below.



```
85   Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, ByRef A As Obj
86
87     Dim u, v As Double
88
89     base_srf.ClosestPoint(input_pt, u, v)
90
91     Dim normal_vector As Vector3d
92
93     normal_vector = base_srf.NormalAt(u, v)
94
95     Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
96
97     drain_vector.Unitize
98
99     drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
100
101    Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
102
103    base_srf.ClosestPoint(moved_pt, u, v)
104
105    Dim output_pt As Point3d = base_srf.PointAt(u, v)
106
107    A = output_pt
108
109  End Sub
```

**step 1** In this step we want to get a 'normal vector' and a 'z vector' on a 'base surface' at an 'input point'. We start by defining an empty 3d vector.

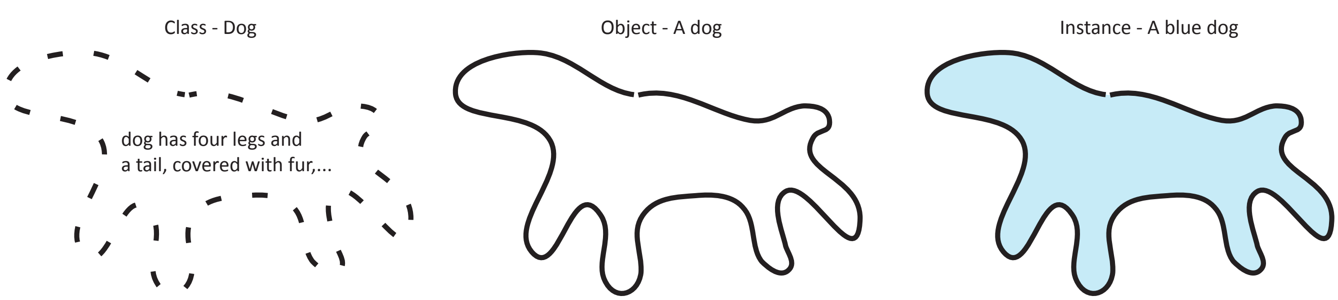
### 91 Dim normal_vector as Vector3d

When we declare something by '**Dim A as B**', **A** is a name of the variable and **B** is a type of the variable such as point3d, integer, vector3d, surface, etc.. We call these '**classes**' instead of '**types**'. So integer can be a class and surface can be another.

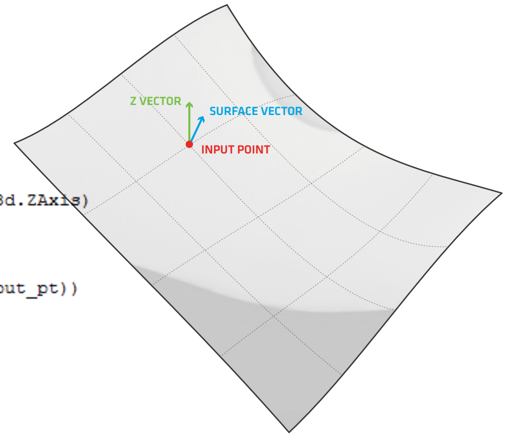
A **class** is, as you may feel, very abstract concept like dog, cat, or rhino(without indefinite article maybe?). For example, vector3d as a class doesn't even have a real name and we don't know anything about it; direction, magnitude, etc..

By giving it a name, '**normal_vector**', we get a vector3d out of vector3d class. Now '**normal_vector**' is an object. An **object** is like a dog, a cat, or a rhino(with indefinite article maybe?). However it is still bodiless and doesn't have any physical characteristics. '**normal_vector**' still can be any vector3d, as a dog can represent DiCaprio, my dog, or Pitt, your dog.

If you want to be more specific about your vecter3d, '**normal_vector**', you can do that by defining an instance. '**Dim A as New B(C)**' is a typical way of defining an **instance**. **A** is a name of the instance, **B** is a name of the class, and **C** is specific condition that gives physicality to **A**. For example, a '**normal_vector**' in line 91 is an object in vector3d class, but we don't know where it is and how big it is, etc.. But if we define a '**normal_vector**' by '**Dim normal_vector as New Vector3d(pointA, pointB)**', the vector comes alive in a space with physicalities such as a starting and ending point as well as length.



```
85  Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, ByRef A As Obj
86
87      Dim u, v As Double
88
89      base_srf.ClosestPoint(input_pt, u, v)
90
91      Dim normal_vector As Vector3d
92
93      normal_vector = base_srf.NormalAt(u, v)
94
95      Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
96
97      drain_vector.Unitize
98
99      drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
100
101     Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
102
103     base_srf.ClosestPoint(moved_pt, u, v)
104
105     Dim output_pt As Point3d = base_srf.PointAt(u, v)
106
107     A = output_pt
108
109 End Sub
```




## 93 normal_vector = base_srf.NormalAt(u, v)

Now we want to get a '**normal_vector**' on a '**base_surface**' at a '**input_point**'. It is clear that a normal vector cannot even exist without a surface. If so, there should be some kind of protocol that enable us to find a normal vector from a surface.

Of course there are. Every class, object, and instance has lower level services; '**constructor**, **methods**, and **properties**'.

'**Constructor**' is the way how we build new instances/objects out of classes. For example, when we draw a line by two points, we can code it in this way; '**Dim A as New Line(pointA, pointB)**'.

While 'constructor' is more about defining an instance/object itself, '**methods**' has to do with manipulating, evaluating or analyzing the instance/object to get something else other than its inherited properties. For example, if you want to get a surface normal vector at a certain point on a surface, you probably can find a 'method' that does this for you from a list of 'methods' in surface class.

'**Properties**' are inherited characteristics of an instance. Unlike 'methods', 'properties' can be retrieved free directly from an instance/object. For example, unlike a surface normal vector at a specific point, area of the surface doesn't change as long as the surface stays same, and we always can ask the surface like "how big are you?".

Good thing about these protocols are that you can always call them with a 'dot' connector. Whenever you want to ask an instance/object, simply type in a dot right next to its name then you will see promptly whatever would be available at that moment. So, in our case, since we have no idea how to find a 'method' that extracts a '**normal_vector**' from a surface, we can just type in '**base_srf**' and add '.' right next to it. Then you will see a list of possible 'methods'. In this case, we want to select '**NormalAt**' from the drop down list.

If you are not sure what do you need, or simply want to browse what is available, you can find useful reference/bible here at the Rhino Common SDK(Software Development Kit) in this page, **http://www.rhino3d.com/5/rhinocommon/index.html**. There you might want to browse in to Rhino.Geometry Namespace, where all of accessible rhino classes are listed up for you. There you can get this;

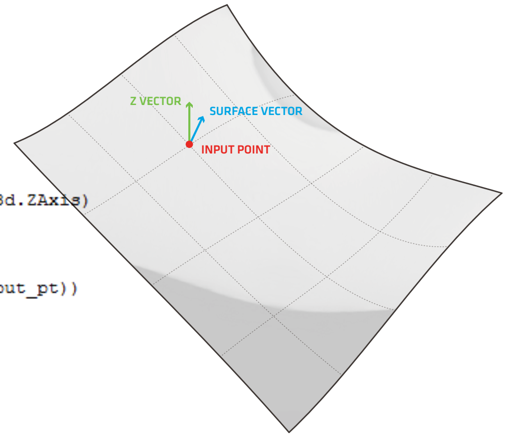
**Surface.NormalAt (u As Double, v As Double) As Vector3d**

'**NormalAt**' method, as you see it, needs two variables; u as double, and v as double. **u** and **v** or **(u,v)** is a local coordinate systme that represents a location of a point on a surface. Because we don't know yet how and where we can get both u and v, let's just put 'u' and 'v' as variables.

## 87 Dim u, v As Double

Although we don't know anything about them, one thing for sure is that they are double. And also they want to be declared before they are called in in order to avoid an error. Declare 'u' and 'v' as double in line 87.

```
85   Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, ByRef A As Obj
86
87       Dim u, v As Double
88
89       base_srf.ClosestPoint(input_pt, u, v)
90
91       Dim normal_vector As Vector3d
92
93       normal_vector = base_srf.NormalAt(u, v)
94
95       Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
96
97       drain_vector.Unitize
98
99       drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
100
101      Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
102
103      base_srf.ClosestPoint(moved_pt, u, v)
104
105      Dim output_pt As Point3d = base_srf.PointAt(u, v)
106
107      A = output_pt
108
109  End Sub
```

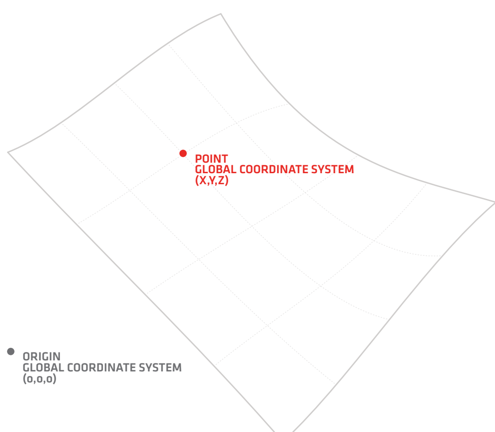


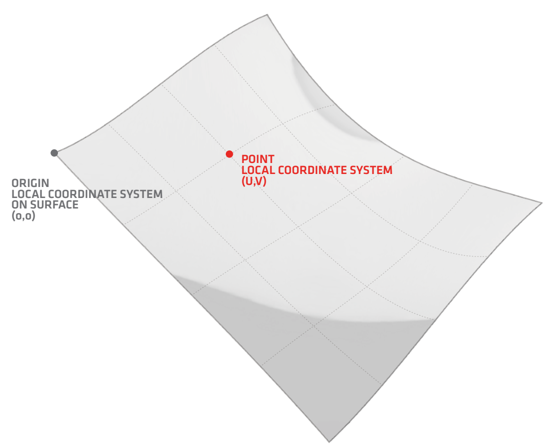

## 89 base_srf.ClosestPoint(input_pt, u, v)

Now we want to get **(u,v)** coordinate of the '**input_point**' on the '**base_surface**'. We all know from our experiences in Grasshopper that the easiest way to convert a global coordinate, **(x,y,z)**, into local one, **(u,v)** is to use 'finding the closest point on a surface' method. This is sort of pre-defined/built-in function in Grasshopper that returns you **(u,v)** coordinate when you supply a surface and a point. Since this method will clearly be part of surface class, browse in to surface class and there you will find the below in method tab.

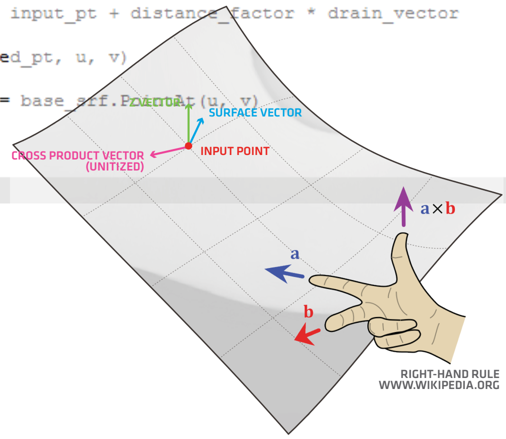
**Surface.ClosestPoint (testPoint As Point3d, ByRef u As Double, ByRef v As Double)**

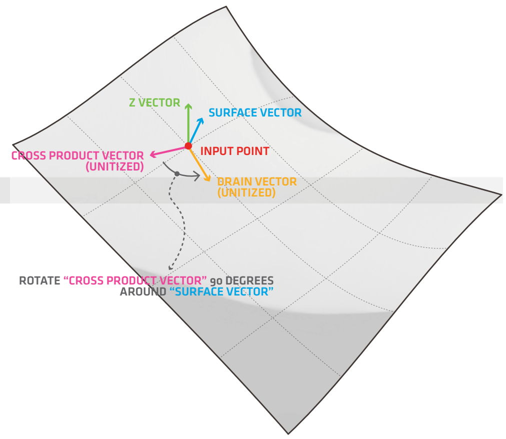
This method requires a testPoint from which it calculates the closest point, and then pass the local coordinate of the closest point by two output references, u and v. When you decipher a code in the SDK, ByRef usually means something you get not something you supply.



```
85  Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, ByRef A As Obj
86
87    Dim u, v As Double
88
89    base_srf.ClosestPoint(input_pt, u, v)
90
91    Dim normal_vector As Vector3d
92
93    normal_vector = base_srf.NormalAt(u, v)
94
95    Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
96
97    drain_vector.Unitize
98
99    drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
100
101   Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
102
103   base_srf.ClosestPoint(moved_pt, u, v)
104
105   Dim output_pt As Point3d = base_srf.PointAt(u, v)
106
107   A = output_pt
108
109 End Sub
```




**step 2** In this step, we want get a '**drain_vector**' by a '**normal_vector**' and a '**z_vector**' at the '**input_point**'.

### 95 Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)

We have two vectors springing from the '**input_point**'; a '**normal_vector**' from the previous step and a '**z_vector**'. From the illustration above, I believe that you can predict the direction of water flow very easily. Yes, the '**drain_vector**' is always perpendicular to the '**cross product vector**' of two input vectors. In other words, we can rotate the '**cross product vector**' 90 degrees CCW around '**normal_vector**' to get the '**drain_vector**'. We can compute the cross product of these vectors with vector3d.crossproduct method. Unlike '**NormalAt**' or '**ClosestPoint**' methods in the previous steps, this particular methods cannot be subordinate to any instance or object, because this method wants to calculate two inputs vectors in the same level. In this case, we can start with generic term, '**Vector3d**' instead of any instance/object name.

**Vector3d.CrossProduct (a As Vector3d, b As Vector3d) As Vector3d**

This method requires two vectors as inputs, and the method itself becomes another vector3d instance. Although the '**cross-product vector**' is not a '**drain_vector**', we can assign this vector to a '**drain_vector**' for now.

### 97 drain_vector.Unitize

In line 97, we unitize the vector, so we can have better control on its length. Otherwise, sometimes it will cause an unexpected error because of uncertainty in vector length.

### 99 drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))

In line 99, we want to rotate the vector 90 degrees counter clock wise around a '**normal_vector**' to get a '**drain_vector**'. We can use vector3d.transform method.

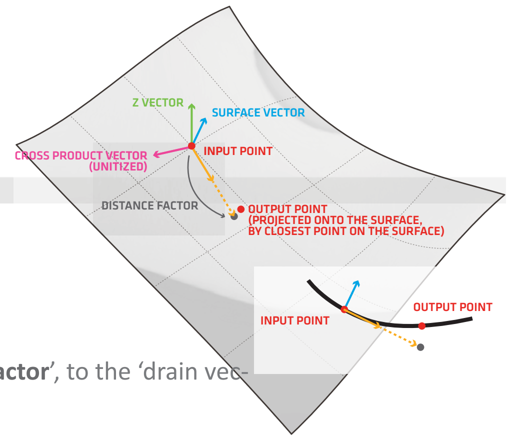
**Vector3d.Transform (transformation As Transform)**

This method requires 'transform' (transform is a class) as a variable.

**Transform.Rotation (angleRadians As Double, rotationAxis As Vector3d, rotationCenter As Point3d) As Transform**

Transform class has a rotation method, and it requires three variables. Now we get a '**drain_vector**'!

```
85   Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, ByRef A As Obj
86
87     Dim u, v As Double
88
89     base_srf.ClosestPoint(input_pt, u, v)
90
91     Dim normal_vector As Vector3d
92
93     normal_vector = base_srf.NormalAt(u, v)
94
95     Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
96
97     drain_vector.Unitize
98
99     drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
100
101    Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
102
103    base_srf.ClosestPoint(moved_pt, u, v)
104
105    Dim output_pt As Point3d = base_srf.PointAt(u, v)
106
107    A = output_pt
108
109  End Sub
```




**step 3** In this step, we will get an '**output_point**' by multiplying a number, '**distance_factor**', to the 'drain vector'.

### 101 Dim moved_pt As point3d = input_pt + distance_factor * drain_vector

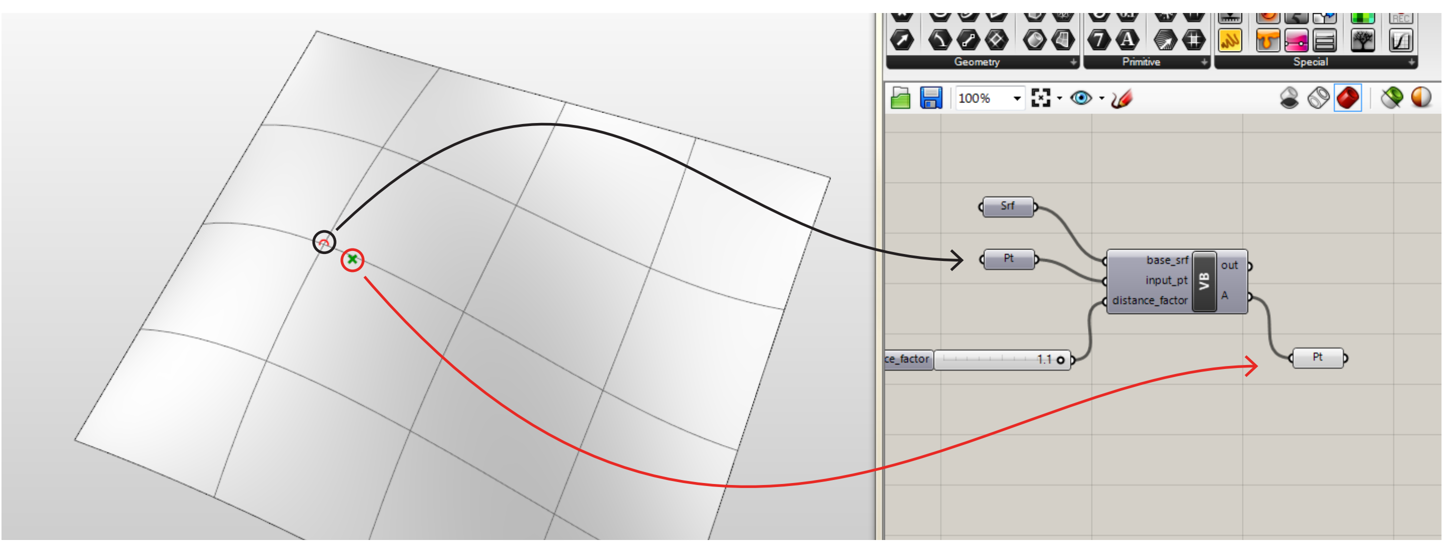
Line 101 is straight forward. This is to move an '**input_point**' by amplifying the '**drain_vector**' with '**distance_factor**'. We save this to a temporary space called '**moved_point**'.

### 103 base_srf.ClosestPoint(moved_pt, u, v)

Then we pull this temporary point back to the surface. Surface.closestpoint method gives **(u,v)** coordinates of the '**moved_point**'.

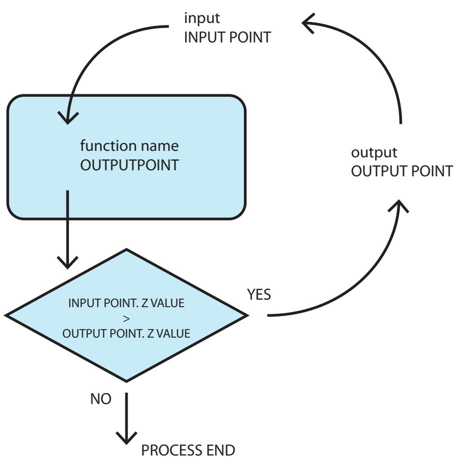
### 105 Dim output_pt As Point3d = base_srf.PointAt(u, v)
### 107 A = output_pt

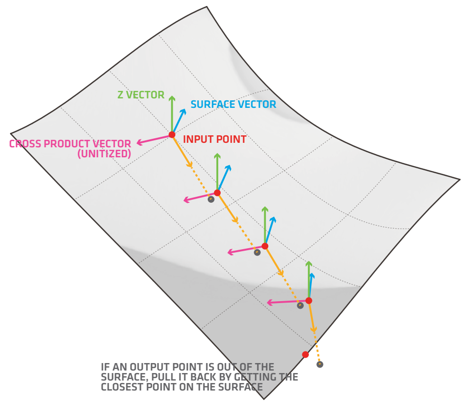
Then by supplying **(u,v)** values to surface.pointat method, we get an '**output_point**'. Then, export the point through an output tab, **A**.

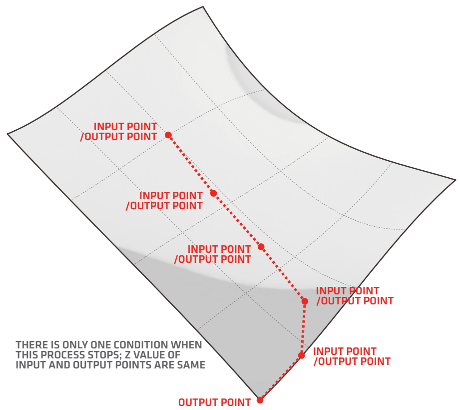


## PART 2 DEFINING A ‘FLOW LINE’

Refer to ‘VB workshop part2.gh’ and ‘VB workshop.3dm’ attached.




**step 1** In the part 1, we defined a simple system that returns an output by a certain logic. And now we want to convert this process into a modular system that keeps repeating its cycle until it satisfies a certain condition. For example, a recursive function; an output of current iteration becomes an input for the next iteration.

```
85  Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, ByRef A As Obj
86
87    Dim u, v As Double
88
89    base_srf.ClosestPoint(input_pt, u, v)
90
91    Dim normal_vector As Vector3d
92
93    normal_vector = base_srf.NormalAt(u, v)
94
95    Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
96
97    drain_vector.Unitize
98
99    drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
100
101   Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
102
103   base_srf.ClosestPoint(moved_pt, u, v)
104
105   Dim output_pt As Point3d = base_srf.PointAt(u, v)
106
107   A = output_pt
108
109 End Sub
```

Copy all the code from the step 1, from line 85 to 109. Then paste it to a place for custom script like below.

```
  End Sub

  '<Custom additional code>

  '</Custom additional code>

End Class
```

```
117  /**/
118
119  Private Function outputpoint(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As D
120                                                          ouble, ByRef A As Object) As Boolean
121    Dim u, v As Double
122
123    base_srf.ClosestPoint(input_pt, u, v)
124
125    Dim normal_vector As New Vector3d(base_srf.NormalAt(u, v))
126
127    Dim drain_vector As vector3d = vector3d.CrossProduct(normal_vector, vector3d.ZAxis)
128
129    drain_vector.Unitize
130
131    drain_vector.Transform(Transform.Rotation(Math.PI * 0.5, normal_vector, input_pt))
132
133    Dim moved_pt As point3d = input_pt + distance_factor * drain_vector
134
135    base_srf.ClosestPoint(moved_pt, u, v)
136
137    Dim output_pt As Point3d = base_srf.PointAt(u, v)
138
139    If output_pt.Z >= input_pt.Z Then
140
141      outputpoint = False
142
143    Else
144
145      outputpoint = True
146
147    End If
148
149    A = output_pt
150
151  End Function
152
153  /**/
```

After copy and paste, we might need to change some part of the code.

**Private Sub RunScript(..., ..., ..., ByRef A As Object)**

**119 Private Function outputpoint(..., ..., ..., ByRef output_pt As Object) As Boolean**

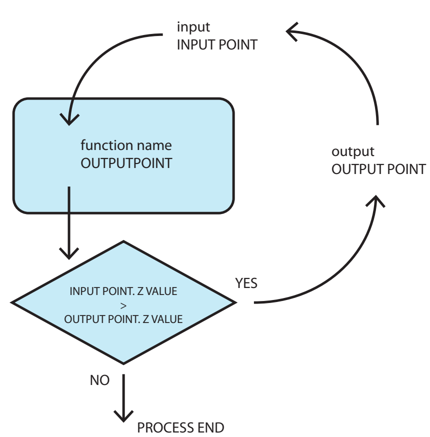
Above red is the first line of the code that you've just paste. And we will change the code like in blue.

**Private** means this portion of code doesn't share any name with the rest of code. You also can make it **public** if you want. However we don't want any naming conflictions, so we keep it **private**.

**Runscript** is a name of this portion of code. We change the name to something meaningful, '**outputpoint**'.

We change **Sub** to **Function**. Both of them refer to segments of code that are separate from main code. The difference between the two is '**sub**' doesn't return a value, while '**function**' does. And we want this function, '**outputpoint**', to store a boolean value for validity check. Confused? I found this link very useful for reference. http://www.homeandlearn.co.uk/NET/vbNet.html

**139 If output_pt.Z >= input_pt.Z Then**
**141 outputpoint = False**
**143 Else**
**145 outputpoint = True**
**147 End If**




This portion of code is to check if we want to repeat the process again or stop. As we can see from the diagram on the right hand side, we want to quit finding '**output_point**' process when **input** and **output** points are on the same elevation. So when this happens, we want to tell the main part of the code to stop this sub process. And we obviously need a messenger that delivers the message. That is why we wanted to make this portion of code to be '**function**' instead of '**sub**'. So depending on the validity, it assigns '**True**' or '**False**' to the function, so the main part of the code can decide if it wants to keep going or not.

If (conditional statement) Then
    (do this)
Else
    (do that)
End If

This is how ‘**if statements**’, one of conditional logic of VB works. For more information on VB, you can visit http://www.homeandlearn.co.uk/NET/vbNet.html

```
80  /**/
85  Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double,
86
87    Dim pt As point3d = input_pt
88
89    Dim output_pts As New List(Of Point3d)
90
91    output_pts.Add(pt)
92
93    Dim output_pt As point3d
94
95    Do
96
97      outputpoint(base_srf, pt, distance_factor, output_pt)
98
99      output_pts.add(output_pt)
100
101     pt = output_pt
102
103     If output_pts.Count > 100 Then
104
105       Exit Do
106
107     End If
108
109   Loop While outputpoint(base_srf, pt, distance_factor, output_pt) = True
110
111   Dim output_crv As New PolylineCurve(output_pts)
112
113   A = output_crv
114
115 End Sub
```

**step 2** In step 1, we defined a funtion ‘**outputpoint**’. In step 2, we call the function and run it until the function returns ‘**false**’.

**95 Do**
**109 Loop While outputpoint(base_srf, pt, distance_factor, output_pt) = True**

First off, we want to repeat the sub portion of the code, **function** or ‘**outputpoint**’ as long as its value is ‘True’.

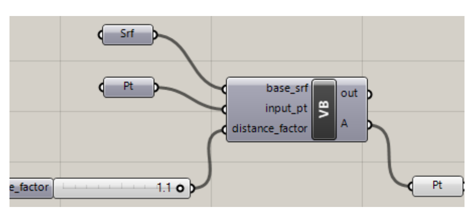
Do
(do this)
Loop While (conditional statement)

This is how ‘**Do ~ Loop**’, one of loop logics in VB, works. Note that ‘**while ()**’ part can be either after ‘Do’ or after ‘Loop’. And also note that the way how we call the function ‘**outputpoint**’. It is pretty much same as the way we use **methods** in the previous steps.

**97 outputpoint(base_srf, pt, distance_factor, output_pt)**

In line 97, finally we call the function. We have to supply this function with 4 parameters; first three as inputs and the last one as an output. It is exactly the same as the way we did in Grasshopper canvas.

We put ‘**base_surface**’ and ‘**distance_factor**’ as they are since they are pretty solid during the process unless we want to change them for some reasons. However, due to its nature as a recursive process, points tend to change frequently as it repeat process over the time. So we can start with two temporary spaces for both input and output points.



```
80  /**/
85  Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As Point3d, ByVal distance_factor As Double, I
87    Dim pt As point3d = input_pt
89    Dim output_pts As New List(Of Point3d)
91    output_pts.Add(pt)
93    Dim output_pt As point3d
95    Do
97      outputpoint(base_srf, pt, distance_factor, output_pt)
99      output_pts.add(output_pt)
101     pt = output_pt
103     If output_pts.Count > 100 Then
105       Exit Do
107     End If
109   Loop While outputpoint(base_srf, pt, distance_factor, output_pt) = True
111   Dim output_crv As New PolylineCurve(output_pts)
113   A = output_crv
115 End Sub
```

**87 Dim pt As point3d = input_pt**
**93 Dim output_pt As point3d**
**89 Dim output_pts As New List(Of Point3d)**
**91 output_pts.Add(pt)**

In line 87, we declare a temporary space '**pt**' for an input point, and then assign the '**input_point**' to '**pt**'. Also in line 93, we declare an empty point3d for output point, '**output_point**'. And in line 89, we declare a space to store a list of points, '**output_points**'. Then in line 91, we might want to add the initial input point '**pt**' to the output point list, so at the end of the process, polyline curve can start from the '**input_point**'.

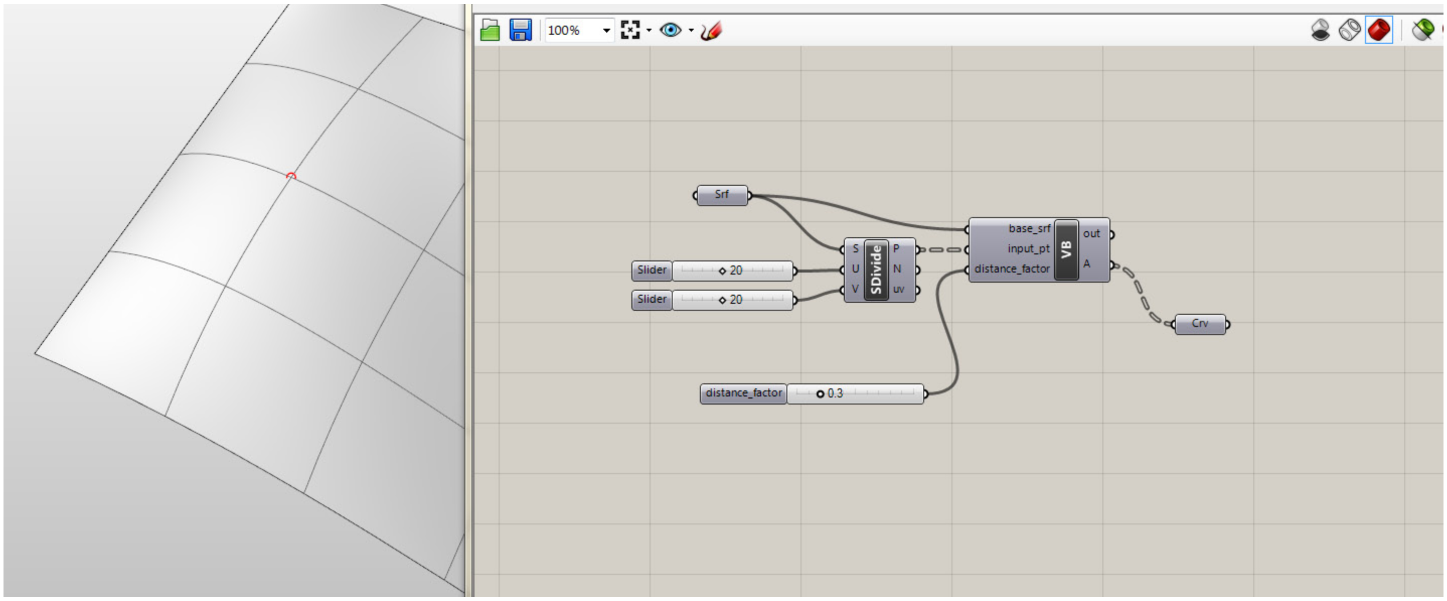
```
95   Do
97     outputpoint(base_srf, pt, distance_factor, output_pt)
99     output_pts.add(output_pt)
101    pt = output_pt
103    If output_pts.Count > 100 Then
105      Exit Do
107    End If
109  Loop While outputpoint(base_srf, pt, distance_factor, output_pt) = True
```

Back to the '**Do ~ Loop**' part, in line 99, we add the first output of function '**outputpoint**'. And in the next line, we switch output point of the current iteration with '**pt**', a temporary location for an input point.

From line 103 to 107, we check if total number of points in the output point list, '**output_pts**', is more than 100. Otherwise, without this, your code might crash because of unexpected heavy load.

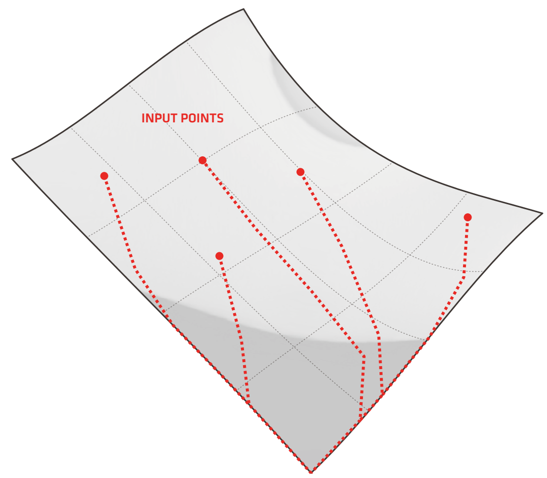
**111 Dim output_crv As New PolylineCurve(output_pts)**
**113 A = output_crv**

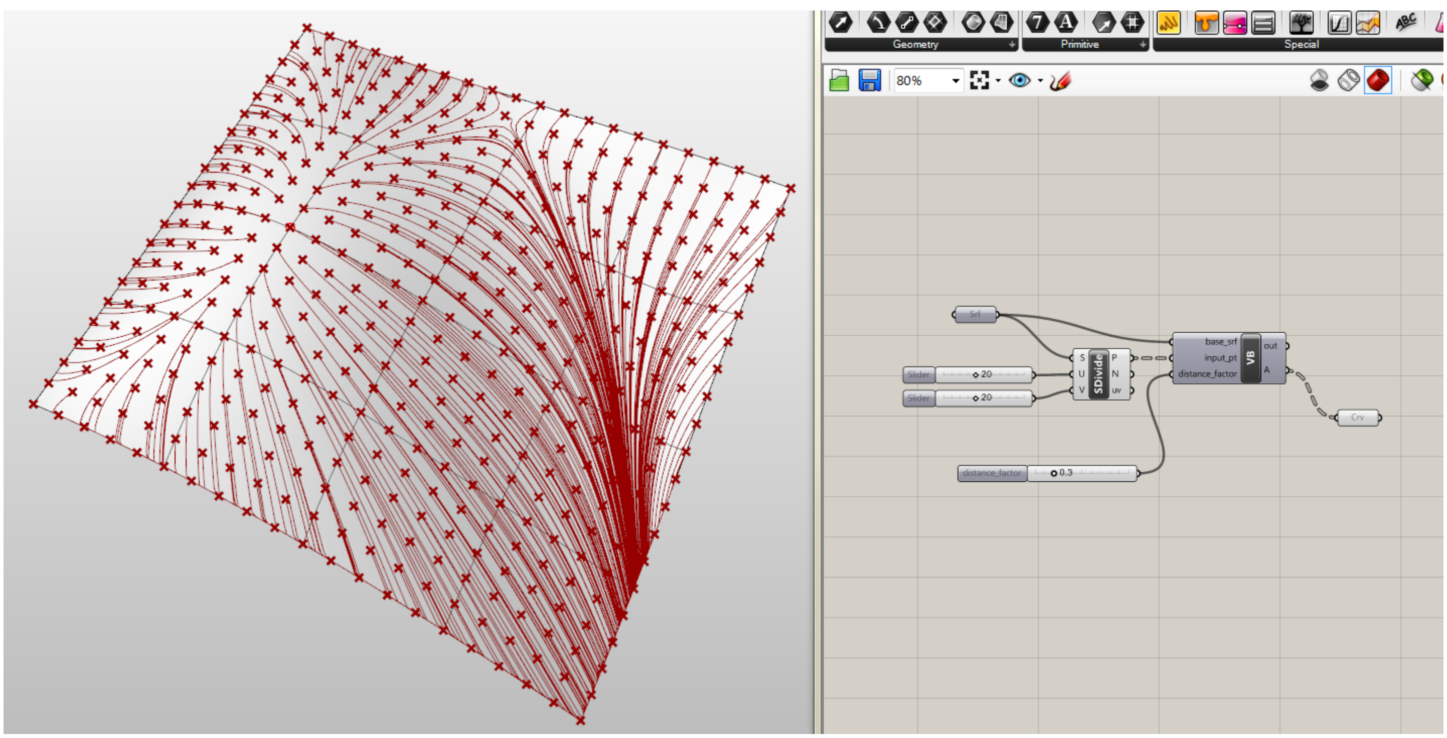
Get a polyline curve from the '**output_points**' list, and export the curve to A.




**PART 3 APPLYING TO MULTIPLE WATER SOURCES**
Refer to 'VB workshop part3.gh' and 'VB workshop.3dm' attached.

In the part 3, we apply the previously defined component to multiple points.



```
80  /**/
85  Private Sub RunScript(ByVal base_srf As Surface, ByVal input_pt As List(Of Point3d), ByVal distance_factor As
87    Dim output_crvs As New List(Of PolylineCurve)
89    For Each pt As point3d In input_pt
91      Dim output_pts As New List(Of Point3d)
93      output_pts.Add(pt)
95      Dim output_pt As point3d
97      Do
99        outputpoint(base_srf, pt, distance_factor, output_pt)
101       output_pts.add(output_pt)
103       pt = output_pt
105       If output_pts.Count > 100 Then
107         Exit Do
109       End If
111     Loop While outputpoint(base_srf, pt, distance_factor, output_pt) = True
113     Dim output_crv As New PolylineCurve(output_pts)
115     output_crvs.Add(output_crv)
117   Next
119   A = output_crvs
121 End Sub
```

**step 1** Now we want to repeat the process for every input point.

**87 Dim output_crvs As New List(Of PolylineCurve)**

Because, at the end of the process, we might want to get multiple polyline curves, we start off by declaring a place to store polyline curve list.

**89 For Each pt As point3d In input_pt**
**....**
**115 output_crvs.Add(output_crv)**
**117 Next**

This is another form of loop logic in VB, and unlike '**do ~ Loop**', '**for ~ next**' is unconditional. It simply do whatever it wants to do until there is no point left in the point list. In line 115, add '**output_curve**' to the output curve list.