Chapter Two: C# Programming Basics

2_1: Introduction

This chapter covers basic C# programming concepts. It serves as an introduction and quick reference to the language syntax. It is not meant to be complete by any measure, so please refer to the C# resources available online and in print[3]. All examples in this chapter are implemented using the Grasshopper C# component.

[3] The Microsoft documentation is a good resource: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/

C# - Comments

2_2: Comments

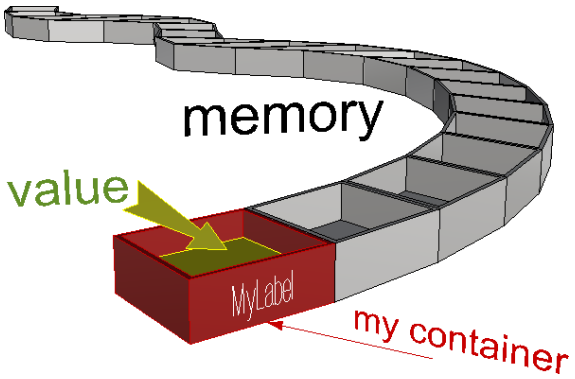
Comments are very useful to describe your code in plain language. They are useful as a reminder for you, and also for others to help them understand your code. To add a comment, you can use two forward slashes // to signal that the rest of the line is a comment and the compiler should ignore it. In the Grasshopper C# code editor, comments are displayed in green. You should enclose a multi-line comment between /* and */ as in the following example.

```
// The compiler ignores this line
/*
  The compiler ignores all lines enclosed
  within this area
*/
```

C# - Variables

2_3: Variables

You can think of variables as labeled containers in your computer's memory where you can store and retrieve data. Your script can define any number of variables and label them with the names of your choice, as long as you do not use spaces, special characters, or reserved words by the programming language. Try to always variable names that are descriptive of the data you intend to store. This will make it easier for you to remember what kind of information is stored in each variable.



Figure(16): How variables are stored in the computer memory

Your script uses variable names to access the value stored in them. In general, you can assign new values to any variable at any point in your program, but each new value wipes out the old one. When you come to retrieve the value of your variable, you will only get the last one stored. For example, let's define a variable and name it ***x***. Suppose we want ***x*** to be of type integer. Also,suppose we would like to assign it an initial value of 10. This is how you write a statement in C# to declare and assign an integer:

```
int x = 10;
```

Let us dissect all the different parts of the above statement:

int	The type of your data. ***int*** is a special keyword in ***C#*** that means the type of the variable is a signed integer (can be a positive or negative whole number)
x	The name of the variable.
=	Used for assignment, and it means that the value that follows will be stored in the variable x.
10	The initial value stored in the ***x*** variable.
;	The Semicolon is used to end a single statement[4].

In general, when you declare a variable, you need to explicitly specify the ***data type***.

C# - Operators

2_4: Operators

Operators are used to perform arithmetic, logical and other operations. Operators make the code more readable because you can write expressions in a format similar to that in mathematics. So instead of having to use functions and write ***C=Add(A, B)***, we can write ***C=A+B***, which is easier to read. The following is a table of the common operators provided by the C# programming language for quick reference:

Type	Operator	Description
Arithmetic Operators	^	Raises a number to the power of another number.
	*	Multiplies two numbers.
	/	Divides two numbers and returns a floating-point result.
	\	Divides two numbers and returns an integer result.
	%	Remainder: divides two numbers and returns only the remainder.
	+	Adds two numbers or returns the positive value of a numeric expression.
	-	Returns the difference between two numeric expressions or the negative value of a numeric expression
Assignment Operators	=	Assigns a value to a variable
	*=	Multiplies the value of a variable by the value of an expression and assigns the result to the variable.
	+=	Adds the value of a numeric expression to the value of a numeric variable and assigns the result to the variable. Can also be used to concatenate a String expression to a String variable and assign the result to the variable.
	-=	Subtracts the value of an expression from the value of a variable and assigns the result to the variable.
Comparison Operators	<	Less than
	<=	Less or equal
	>	Greater than
	>=	Greater or equal
	==	Equal
	!=	Not equal
Concatenation Operators	&	Generates a string concatenation of two expressions.
	+	Concatenate two string expressions.
Logical Operators	&&	Performs a logical conjunction on two Boolean expressions
	!	Performs logical negation on a Boolean expression
	||	Performs a logical disjunction on two Boolean expressions

C# - Namespaces

2_5: Namespaces

Namespaces are very useful to group and organize classes especially for large projects. The ***.NET*** uses namespaces to organize its classes. For example ***System*** namespace has many classes under it including the ***Math*** class, so if you like to calculate the square root of a number, your code will look like the following:

```
double num = 16;
double sqrNum = System.Math.Sqrt(num);
```

Notice how you use the namespace followed by "." to access the classes within that namespace. If you do not want to type the namespace each time you need to access one of the classes within that namespace, then you can use the ***using*** keyword as in the following.

```
using System;
```

```
double num = 16;
double sqrNum = Math.Sqrt(num);
```

C# Data Types : Primitives

2_6: Data

Data types refer to the kind of data stored in the variable. There are two main data types. The first is supplied by the programming language and involves things like numbers, logical true or false, and characters. Those are generally referred to as primitive or built-in types. Variables of any data type can be composed into groups or collections.

2_6_1: Primitive data types

Primitive types refer to the basic and built-in types provided by the programming language. Following examples declare variables of primitive data types:

Declare primitive data types	Notes
double pi = 3.1415;	***double***: big number with decimal point.
bool pass = true;	***bool***: set to either true or false. Used mainly to represent the truth value of a variable or a logical statement.
char initial = 'R';	***char***: stores exactly one character.
string myName = "Mary";	***string***: a sequence of characters.
object someData = "Mary";	***object*** type can be used to store data of any type. The use of ***object*** type is inefficient and should be avoided if at all possible.

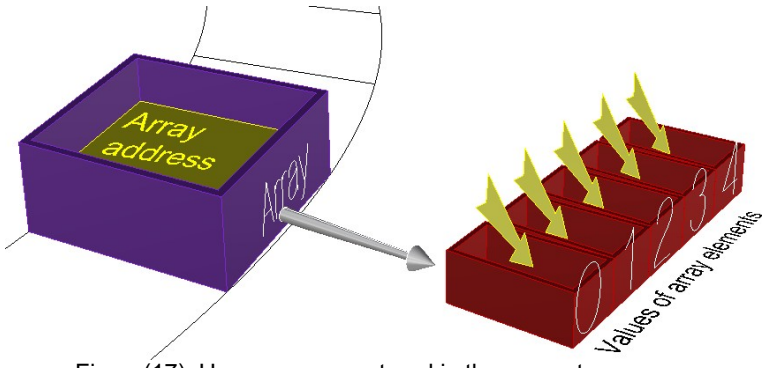
C# Data Types : Arrays

2_6_2: Collections

In many cases, you will need to create and manage a group of objects. There are generally two ways to group objects: either by organizing them in arrays or using a collection. Collections are more versatile and flexible, and they allow you to expand or shrink the group dynamically. There are many ways to create collections, but we will focus on ordered ones that contain elements of the same data type. For more information, you can reference the literature.

Arrays

Arrays are a common way to assemble an ordered group of data. Arrays are best suited if you already know the values you are storing, and do not need to expand or shrink the group dynamically.



Figure(17): How arrays are stored in the computer memory

For example, you can organize the days of the week using an array. The following declares and initializes the weekdays array in one step.

```
//Declare and initialize the days of the week array
string[ ] weekdays = {"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"};
```

You can also declare the array first with a specific dimension representing the number of elements to allocate in memory for the array, then assign the values of the elements in separate statements. In C# language, each statement ends with a semicolon.

```
//Declare the days of the week in one statement, then assign the values later
string[ ] weekdays = new string[7];
weekdays[0] = "Sunday";
weekdays[1] = "Monday";
weekdays[2] = "Tuesday";
weekdays[3] = "Wednesday";
weekdays[4] = "Thursday";
weekdays[5] = "Friday";
weekdays[6] = "Saturday";
```

Let's take the statement that sets the first value in the weekdays array in the above.

```
weekdays[0] = “Sunday”;
```

Sunday is called an ***array element***. The number between brackets is called the ***index*** of the element in the array. Notice that in ***C# language*** the array always starts with index=0 that points to the first element. The last index of an array of seven elements therefore equals 6. If you try to retrieve data from an invalid index, for example 7 in the weekday example above, then you will get what is called an ***out of bound error*** because you are basically trying to access a part of the memory that you did not allocate for your array and it can lead to a crash.

C# Data Types : Lists

>>

Lists

If you need to create an ordered collection of data dynamically, then use a ***List***. You will need to use the ***new*** keyword and specify the data type of the list. The following example declares a new list of integers and appends elements incrementally in subsequent statements. Note that the indices of a list are also zero based just like arrays.

```
//Declare a dynamic ordered collection using “List”
List<string> myWorkDays =  new List<string>();

//Add any number of new elements to the list
myWorkDays.Add("Tuesday");
myWorkDays.Add("Wednesday");
myWorkDays.Add("Thursday");
myWorkDays.Add("Friday");
```

You need to use the keyword ***new*** to declare an instance of a ***List***.. We will explain what it means to be a reference type with some more details later.

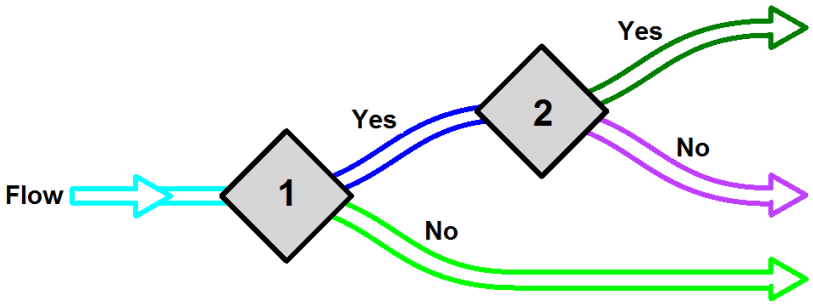
C# Flow Control : If Statements

2_7: Flow control

The flow of your scripts indicates the order of code execution. Sometimes you might need to branch or loop through specific statements several times. For example, you might want to branch your script to implement a different sequence based on some condition. Other times, you might need to repeat a certain sequence multiple times. We will discuss three important mechanisms to control the flow of a program; conditional statements, loops and methods.

2_7_1: Conditional statements

Conditional statements allow you to decide on which part of your code to use based on some condition that can change during run time. Unlike regular programming instructions, conditional statements can be described as decision-making logic. Conditional statements help control and regulate the flow of the program.



Figure(17): Conditional statements 1 and 2, and how they affect the flow of the program.

The following script examines a number variable and prints “zero” if it equals zero:

```
if (myNumber == 0)
{
    Print("myNumber is zero");
}
```

Description	
if	Keyword indicating the start of the if statement
(myNumber == 0)	The condition of the if statement enclosed by parentheses

{
Print("myNumber is zero");
}
	The if statement block enclosed between curly brackets. The block is executed only if the condition evaluates to true, otherwise it is skipped.

The following code checks if a number is between 0 and 100:

```
if (myNumber >= 0 && myNumber <= 100)
{
    Print("myNumber is between 0 and 100");
}
```

C# Flow Control : If … Else Statements

Sometimes the script needs to execute one block when some condition is satisfied, and another if not. Here is an example that prints the word ***positive*** when the given number is greater than zero, and prints ***less than 1*** if not. It uses the ***if… else*** statement.

```
if (myNumber > 0)
{
    Print("positive");
}
else
{
    Print( "less than 1" );
}
```

>>

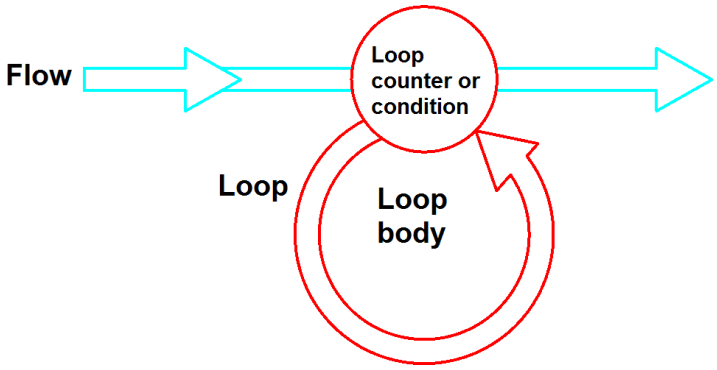
You can use as many conditions as you need as shown in the following example.

```
if (myNumber > 0)
{
    Print("myNumber is positive");
}
else if (myNumber < 0)
{
    Print( "myNumber is negative" );
}
else
{
    Print( "myNumber is zero" );
}
```

C# Language : Loops

2_7_2: Loops

Loops allows you to run the body of your loop a fixed number of times, or until the loop condition is no longer true.



Figure(18): Loops in the context of the overall flow of the program

There are two kinds of loops. The first is iterative where you can repeat code a defined number of times, and the second is conditional where you repeat until some condition is no longer satisfied.

Iterative loops: for loop

This is a common way of looping when you need to run some lock of code a specific number of times. Here is a simple example that prints numbers from 1 to 10. You first declare a counter and then increment in the loop to run the code specific number of times. Notice that the code statements that you would like to repeat are bound by the block after the for statement.

```
for (int i = 1; i <= 10; i++)
{
  //Convert a number to string
  Print( i.ToString() );
}
```

Description	
`for (int i = 1; i <= 10; i++)`	The for statement. It has 4 parts: **for(... ; … ; ...)**: the for keyword and loop condition and counter **i=1**: the counter initial value **i<=10**: the condition: if evaluates to true, then execute the body of the for loop **i++**: increment the counter (by 1 in this case)

`{`
`  //Convert a number to string`
`  Print( i.ToString() );`
`}`
	The body of the for loop. This is the code that is executed as long as the condition is met.
	Note that if the condition of the for loop is always true, then the program will enter what is called an “infinite loop” that leads to a crash. For example if the condition was “i>0” instead of “i<=10” then it will cause an infinite loop.

```
for (int i = 10; i >= -10; i = i-2)
{
   Print( i.ToString() );
}
```

If you happen to have an array that you need to iterate through, then you can set your counter to loop through the indices of your array. Just remember that the indices of arrays are zero based and therefore you need to remember to loop from index=0 to the length of the array minus 1, or else you will get an out-of-bound error.

```
string alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
char[ ] letters = alphabet.ToCharArray();
for (int i = 0; i < letters.Length; i++)
{
  Print( i.ToString() + " = " + letters[i] );
}
```

Here is another example that iterates through a list of places.

```
//List of places
List< string > placesList = new List< string >();
placesList.Add( "Paris" );
placesList.Add( "NY" );
placesList.Add( "Beijing" );

int count = placesList.Count();
//Loop starting from 0 to count -1 (count = 3, but last index of the placesList is 2)
for (int i = 0; i < count; i++)
{
   Print( "I have been to " + placesList[i] );
}
```

The loop counter does not have to be increasing, or change by 1. The following example prints the even numbers between 10 and -10. You start with a counter value equal to 10, and then change by -2 thereafter until the counter becomes smaller than -10.

Iterative Loops: foreach loop

You can use the ***foreach*** loop to iterate through the elements of an array or a list without using a counter or index. This is a less error prone way to avoid out of bound error. The above example can be rewritten as follows to use ***foreach*** instead of the ***for*** loop:

```
//List of places
List< string > placesList = new List< string >();
placesList.Add( "Paris" );
placesList.Add( "NY" );
placesList.Add( "Beijing" );

//Loop
foreach (string place in placesList)
{
    Print(place);
}
```

>>

Conditional Loops: while loop

>>

Conditional loops are ones that keep repeating until certain conditions are no longer true. You have to be very careful when you use conditional loops because you can be trapped looping infinitely until you crash when your condition continues to be true.

Some problems can be solved iteratively but others cannot. For example if you need to find the sum of 10 consecutive positive even integers starting with 2, then this can be solved iteratively with a ***for*** loop. Why? This is because you have a specified number of times to loop (10 in this case). This is how you might write your loop to solve your problem.

```
int sum = 0;
for (int i = 1; i <= 10; i++)
{
    sum = sum + ( i*2 );
}
Print( sum.ToString() );
```

On the other hand, if you need to add consecutive positive even integers starting with 2 until the sum exceeds 1000, then you cannot simply use an iterative ***for*** loop because you do not know how many times you will have to loop. You only have a condition to stop looping. Here is how you might solve this problem using the conditional ***while*** loop:

```
int sum = 0;
int counter = 0;
int number = 2;

//Loop
while (sum < 1000)
{
   sum = sum + number;
   //Make sure you increment the counter and the number
   counter = counter + 1;
   number = number + 2;
}

//Remove last number
sum -= number;
counter --;

Print("Count = " + counter);
Print("Sum = " + sum);
```

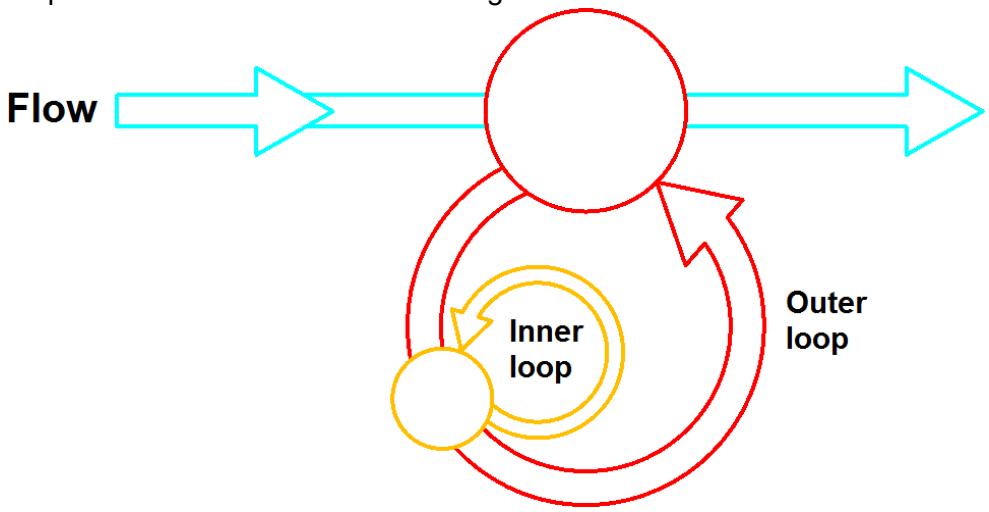
Here is another example from mathematics where you can solve the ***Collatz conjecture***. It basically says that you can start with any natural positive integer, and if even then divide by 2, but if odd, then multiply by 3 and add 1. If you repeat the process long enough, then you always converge to 1[5]. The following example prints all the numbers in a ***Collatz conjecture*** for a given number:

```
int x = 46;
while(x > 1)
{
  Print( x.ToString() );
  if( x % 2 == 0 )
    x = x / 2;
  else
    x = x * 3 + 1;
}
Print( x.ToString() );
```

[5] For details about the Collatz conjecture, check http://en.wikipedia.org/wiki/Collatz_conjecture

Nested loops

A nested loop is a loop that contains another loop inside it. You can nest as many loops as you need, but keep in mind that this can make the logic harder to follow.



Figure(19): Nested loops

For example, suppose you have the following list of words as input: {apple, orange, banana, strawberry} and you need to calculate the number of letters ***a***. You will have to do the following:

1. Loop through each one of the fruits. This is the outer loop.
2. For each fruit, loop through all the letters. This is the inner, or the nested, loop.
3. Whenever you find the letter “a” increment your counter by 1.
4. Print the counter.

```
//Declare and initialize the fruits
string[ ] fruits =  {“apple”, “orange”, “banana”, “strawberry”};
int count = 0;

//Loop through the fruits
foreach (string fruit in fruits)
{
  //loop through the letters in each fruit
  foreach (char letter in fruit)
  {
    //Check if the letter is ‘a"
    If (letter == ‘a’)      {
      count = count + 1;
    }
  }
}
Print( “Letter a is repeated “ + count + “ times“ );
```

Using break and continue inside loops

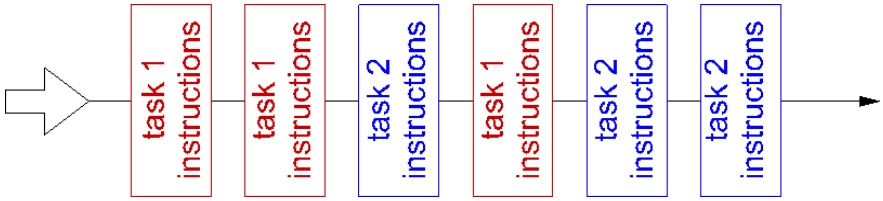
The ***break*** statement is used to terminate or exit the loop while the ***continue*** statement helps skip one loop iteration. Here is an example to show the use of both. It checks if there is an orange in a list of fruits, and counts how many fruits have at least one letter r in their name.

```
//Declare and initialize the fruits
string[ ] fruits =  {“apple”, “orange”, “banana”, “strawberry”};

//Check if there is at least one orange
foreach (string fruit in fruits)
{
   //Check if it is an orange
   if (fruit == "orange")
   {
      //Check if the letter is ‘a"
      Print( "Found an orange" );

      //No need to check the rest of the list, exit the loop
      break;
   }
}

//Check how many fruits has at least one letter ‘r’ in the name
int count = 0;
//Loop through the fruits
foreach (string fruit in fruits)
{
   //loop through the letters in each fruit
   foreach (char letter in fruit)
   {
      //Check if the letter is ‘r"
      if (letter == ‘r’)      {
         count = count + 1;

         //No need to check the rest of the letters, and skip to the next fruit
         continue;
      }
   }
}
Print( "Number of fruits that has at least one letter r is: " + count );
```

2_8: Methods

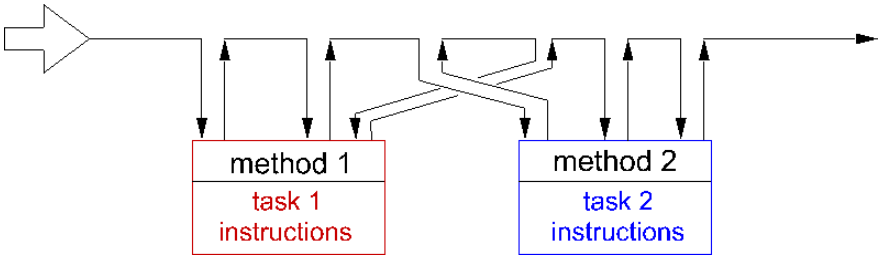
2_8_1: Overview

A method[6] is a self-contained entity that performs a specific task. The program can call it any number of times. Functions are basically used to organize your program into sub-tasks. The following is an example of the flow of a program that uses two distinct tasks repeatedly, but does not implement as a separate function, and hence has to retype at each encounter.



Figure(20): Sequential flow of the script where tasks are repeated

Most programming languages allow you to organize your tasks into separate modules or what is called functions or methods. You can think of Grasshopper components each as a function. This allows you to write the instructions of each task once and give it an identifying name. Once you do that, anytime you need to perform the task in your program, you simply call the name of that task. Your program flow will look like the following:



Figure(21): Methods can be called and executed as many times as the script requires

Whenever you need to perform the same task multiple times in different places in your code, you should consider isolating it in a separate function. Here are some of the benefits you will get from using functions:

1. Break your program into manageable pieces. It is much easier to understand, develop, test and maintain smaller chunks of code, than one big chunk.
2. Write and test the code once, and reuse instead of writing it again. Also update in one place if a modification is required at any time.

Writing functions might not always be easy and it needs some planning. There are four general steps that you need to satisfy when writing a function:

1. Define the purpose of your function and give it a descriptive name.
2. Identify the data that comes into the function when it is called.
3. Identify the data that the function needs to hand back or return.
4. Consider what you need to include inside the function to perform your task.

[6] The term “method” refers to functions associated with classes, but this guide may use them interchangeably.

For example, suppose you would like to develop a function that checks whether a given natural number is a prime number or not. Here are the four things you need to do:

1. **Purpose & name:** Check if a given number is prime. Find a descriptive name for your function, for example "***IsPrimeNumber***".
2. **Input parameters:** Pass your number as type integer.
3. **Return:** True if the number is prime, otherwise return false.
4. **Implementation**: Use a loop of integers between 2 and half the number. If the number is not divisible by any of those integers, then it must be a prime number. Note that there might be other more efficient ways to test for a prime number that can expedite the execution.

```
bool IsPrimeNumber( int x)
{
   bool primeFlag = true;

   for( int i = 2; i < (x / 2); i++)
   {
     if( x % i == 0)
     {
       primeFlag = false;
       break; //exit the loop
     }
   }
   return primeFlag;
}
```

Let us dissect the different parts of the ***IsPrimeNumber*** function to understand the different parts.

The function or method part	Description

`bool IsPrimeNumber(…)`
`{`
`  …`
`}`
	The name of the function proceeded by the return type. The function body is enclosed inside the curly parenthesis.

`( int x )`	Function parameters and their types.

`bool primeFlag = true;`
 	
`for( int i = 2; i < (x / 2); i++)`
`{`
`  if( x % i == 0)`
`  {`
`  primeFlag = false;`
`  break; //exit the loop`
`  }`
`}`
	This is called the *body* of the function. It includes a list of instructions to examine whether the given number is prime or not, and append all divisible numbers.

`return primeFlag;`	Return value. Use return keyword to indicate the value the function hands back.

2_8_2: Method parameters

Input parameters are enclosed within parentheses after the method name. The parameters are a list of variables and their types. They represent the data the function receives from the code that calls it. Each parameter is passed either ***by value*** or ***by reference***. Keep in mind that variables themselves can be a value-type such as primitive data (int, double, etc.), or reference-types such as lists and user-defined classes. The following table explains what it means to pass value or reference types by value or by reference.

Parameters	Description
Pass by value a **value-type** data such as int, double, bool	The caller passes a copy of the actual value of the parameter. Changes to the value of a variable inside the function will not affect the original variable passed from the calling part of the code.
Pass by reference a **value-type** data	Indicates that the address of the original variable is passed and any changes made to the value of that variable inside the function will be seen by the caller.
Pass by value a **reference-type** data such as lists, arrays, object, and all classes	Reference Type data holds the address of the data in the memory. Passing a Reference Type data by value simply copy the address, so changes inside the function will change original data. If the caller cares that ita data is not affected by the function, it should duplicate the data before passing it to any function.
Pass by reference a **reference-type** data	Indicates that the original reference of the variable is passed. Again, any changes made to the variable inside the function are also seen by the caller.

As you can see from the table, whether the parameter is passed as a copy (by value) or as a reference (by reference), is mostly relevant to ***value-type*** variables such as ***int***, ***double***, ***bool***, etc. If you need to pass a **reference-type** such as objects and lists, and you do not wish the function to change your original data (act as though the variable is passed by value), then you need to duplicate your data before passing it. All input values to the GH ***RunScript*** function (the main function in the scripting component) are duplicated before being used inside the component so that input data is not affected.

Let's expand the ***IsPrimeNumber*** function to return all the numbers that the input is divisible by. We can pass a list of numbers to be set inside the function (***List*** is a reference type). Notice it will not matter if you pass the list with the ***ref*** keyword or not. In both cases, the caller changes the original data inside the function.

```
private void RunScript(int num, ref object IsPrime, ref object Factors)
{
    //Assign variables to output
    List<int> factors = new List<int>();
    IsPrime= isPrimeNumber(num, factors);
    Factors = factors;
}
//Note: "List" is a reference type and hence you can pass it by value (without "ref" keyword)
//          and the caller still get any changes the function makes to the list
bool IsPrimeNumber( int num, ref List<int> factors )
{
    //all numbers are divisible by 1
    factors.Add(1);
    bool primeFlag = true;
    for( int i = 2; i < (num / 2); i++)
    {
      if( num % i == 0)
      {
        primeFlag = false;
        factors.Add( i); //append number
      }
    }
    //all numbers are divisible by the number itself
    factors.Add(num);

    return primeFlag;
}
```

Passing parameters ***by reference*** using keyword ***ref*** is very useful when you need to get more than one ***value-type*** back from your function. Since functions are allowed to return only one value, programmers typically pass more parameters by reference as a way to return more values. Here is an example of a division function that returns success if the calculation is successful, but also returns the result of the division using the ***rc*** parameter that is passed ***by-reference***.

```
public bool Divide(double x, double y, ref double rc)
{
    if (Math.Abs(y) < 1e-100)
        return false;
    else
        rc = x / y;
    return true;
}
```

As we explained before, the ***RunScript*** is the main function that is available is the ***C#*** and other script components. This is where the user main code lives. Notice that the scripting component output is passed by reference. This is what you see when you open a default ***C#*** script component in Grasshopper.


```
55    private void RunScript(object x, object y, ref object A)
56    {
57
58    }
59
60    // <Custom additional code>
61
62    // </Custom additional code>
63  }
```

Code part	Description

{
 	
}
	Curly parentheses enclose the function body of code.

RunScript	This is the name of the main function.
RunScript(…)	Parentheses after the function name enclose the input parameters. This is the list of variables passed to the function.
object x	*x* is the first input parameter. It is passed *by value* and its type is *object*. That means changes to "x" inside the ***RunScript*** does not change the original value.
object y	*y* is the second input parameter. It is passed *ByVal* and its type is *Object*.
ref object A	*A* is the third input parameter. It is passed *by reference* and its type is *object*. It is also an output because you can assign it a value inside your script and the change will be carried outside the ***RunScript*** function.
//Custom additional code	*Comment indicating where you should place your additional functions, if needed.*

2_9: User-defined data types

We mentioned above that there are built-in data types and come with and are supported by the programming language such as int, double, string and object. However, users can create their own custom types with custom functionality that suits the application. There are a few ways to create custom data types. We will explain the most common ones: enumerations, structures and classes.

User Defined Data Types

2_9: User-defined data types

We mentioned above that there are built-in data types and come with and are supported by the programming language such as int, double, string and object. However, users can create their own custom types with custom functionality that suits the application. There are a few ways to create custom data types. We will explain the most common ones: enumerations, structures and classes.

2_9_1: Enumerations

Enumerations help make the code more readable. An enumeration "provides an efficient way to define a set of named integral constants that may be assigned to a variable"[7]. You can use enumerations to group a family of options under one category and use descriptive names. For example, there are only three values in a traffic light signal.

```
enum Traffic
{
  Red = 1,
  Yellow = 2,
  Green = 3
}
Traffic signal = Traffic.Yellow;
if( signal == Traffic.Red)
    Print("STOP");
else if(signal == Traffic.Yellow)
    Print("YIELD");
else if(signal == Traffic.Green)
    Print("GO");
else
    Print("This is not a traffic signal color!");
```

[7] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/enumeration-types

User Defined Data Types - Structs

2_9_2: Structures

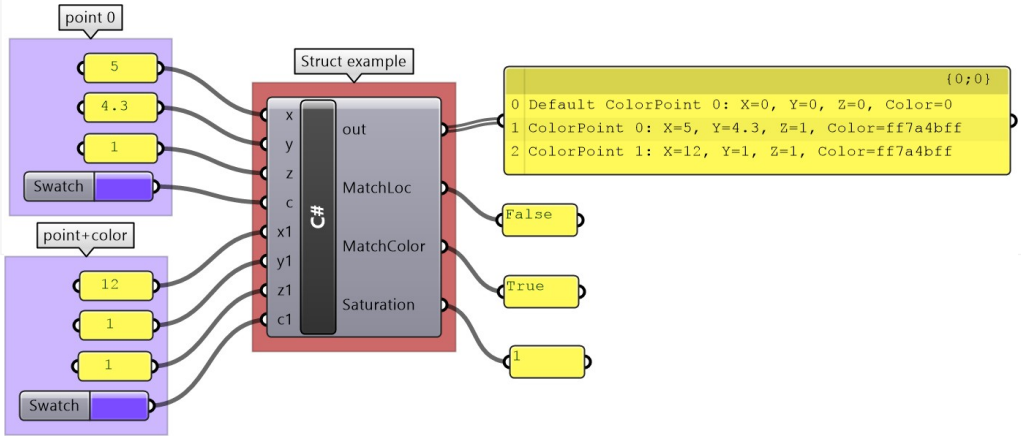
A structure is used to define a new ***value-type***. In C# programming , we use the keyword ***struct*** to define new structure. The following is an example of a simple structure that defines a number of variables (fields) to create a custom type of a colored 3D point. We use ***private*** access to the fields, and use ***properties*** to ***get*** and ***set*** the fields.

```
struct ColorPoint{
    //fields for the point XYZ location and color
    private double _x;
    private double _y;
    private double _z;
    private System.Drawing.Color _c;
    //properties to get and set the location and color
    public double X { get { return _x; } set { _x = value; } }
    public double Y { get { return _y; } set { _y = value; } }
    public double Z { get { return _z; } set { _z = value; } }
    public System.Drawing.Color C { get { return _c; } set { _c = value; } }
}
```

As an example, you might have two instances of the ***ColorPoint*** type, and you need to compare their location and color. Notice that when you instantiate a new instance of ***ColorPoint*** object, the object uses a default constructor that sets all fields to “0”:

```
ColorPoint cp0 = new ColorPoint();
//Using default constructor sets the fields to zero
Print("Default ColorPoint 0: X=" + cp0.X + ", Y=" + cp0.Y + ", Z=" + cp0.Z + ", Color=" + cp0.C.Name);
//set  fields
cp0.X = x;
cp0.Y = y;
cp0.Z = z;
cp0.C = c;
Print("ColorPoint 0: X=" + cp0.X + ", Y=" + cp0.Y + ", Z=" + cp0.Z + ", Color=" + cp0.C.Name);

ColorPoint cp1 = new ColorPoint();
//set fields
cp1.X = x1;
cp1.Y = y1;
cp1.Z = z1;
cp1.C = c1;
Print("ColorPoint 1: X=" + cp1.X + ", Y=" + cp1.Y + ", Z=" + cp1.Z + ", Color=" + cp1.C.Name);

//compare location
MatchLoc = false;
if(cp0.X == cp1.X && cp0.Y == cp1.Y && cp0.Z == cp1.Z)
  MatchLoc = true;

//compare color
MatchColor = cp0.C.Equals(cp1.C);
```

This is the output you get when implementing the above struct and function in a C# component:



User Defined Data Types - Structs

Structs typically define one or more constructors to allow setting the fields. Here is how we expand the example above to include a constructor.

```
struct ColorPoint{
    //fields
    private double _x;
    private double _y;
    private double _z;
    private System.Drawing.Color _c;
```

```
    //constructor
    public ColorPoint(double x, double y, double z, System.Drawing.Color c)
    {
      _x = x;
      _y = y;
      _z = z;
      _c = c;
    }
    //properties
    public double X { get { return _x; } set { _x = value; } }
    public double Y { get { return _y; } set { _y = value; } }
    public double Z { get { return _z; } set { _z = value; } }
    public System.Drawing.Color C { get { return _c; } set { _c = value; } }
}
```

You can use properties to only set or get data. For example you can write a property that get the color saturation:

```
    //property
    public double Saturation { get { return _c.GetSaturation(); } }
```

We can rewrite the ***Saturation*** property as a method as in the following:

```
    //method
    public double Saturation() { return _c.GetSaturation(); }
```

However, methods typically include more complex functionality, multiple input or possible exceptions. There are no fixed rules about when to use either, but it is generally acceptable that properties involve data, while methods involve actions. We can write a method to calculate the average location and color of two input ColorPoints.

```
    //method
    public static Average (ColorPoint a, ColorPoint b)
    {
      ColorPoint avPt = new ColorPoint();
      avPt.X = (a.X + b.X) / 2;
      avPt.Y = (a.Y + b.Y) / 2;
      avPt.Z = (a.Z + b.Z) / 2;
      avPt.C = Color.FromArgb((a.C.A + b.C.A) / 2, (a.C.R + b.C.R) / 2, (a.C.G + b.C.G) / 2, (a.C.B + b.C.B) / 2);
      return avPt;
    }
```

User Defined Data Types - Structs

Structures can include members and methods that can be called even if there is no instance created of that type. Those use the keyword ***static***. For example, we can add a member called OriginBlack that creates a black point located at the origin. We can also include a ***static*** method to compare if two existing points have the same color, as in the following.

```
//static methods
public static bool IsEqualLocation(ColorPoint p1, ColorPoint p2)
{
  return (p1.X == p2.X && p1.Y == p2.Y && p1.Z == p2.Z);
}
```

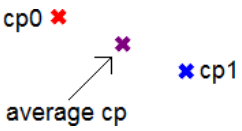
```
public static bool IsEqualColor(ColorPoint p1, ColorPoint p2)
{
  return p1.C.Equals(p2.C);
}
```

Because structs are value types, if you pass your colored points to a function, and change the location (x,y,z), it will only be changed inside or within the scope of that function, but will not affect the original data, unless explicitly passed by reference as in the following example:

```
void ChangeLocationByValue(ColorPoint cp)
{
  cp.X += 10;
  cp.Y += 10;
  cp.Z += 10;
  Print(" Inside change location= (" + cp.X + "," + cp.Y + "," + cp.Z + ")");
}
void ChangeLocationByReference(ref ColorPoint cp)
{
  cp.X += 10;
  cp.Y += 10;
  cp.Z += 10;
  Print(" Inside change location= (" + cp.X + "," + cp.Y + "," + cp.Z + ")");
}
```

Using our ***ColorPoint*** struct, the following is a program that generates 2 colored points, compares their location and color, then finds their average:

```
//create 2 instances of ColorPoints type
ColorPoint cp0 = new ColorPoint(1, 1, 1, System.Drawing.Color.Red);
ColorPoint cp1 = new ColorPoint(1, 1, 1, System.Drawing.Color.Blue);
//compare location
bool matchLoc = ColorPoint.isEqualLocation(cp0, cp1);
//compare color
Bool matchColor = ColorPoint.isEqualColor(cp0, cp1);
//Output average point and color
ColorPoint avPt = ColorPoint.Average(cp0, cp1);
```



Structures have limitations, most notable is the inability to build a hierarchy of objects. Also, they are not preferred for objects that are large in their size in bytes.

Assignment 01

For this assignment we are going to revisit the initial example from the first class, and consider “grid-like” algorithms.

A grid-like algorithm is an algorithm that generates shapes based on a regular grid of

- i points (1D),
- i * j points (2D), or
- i * j * k points (3D)

The “axes” of the grid do not need to be straight, but a “regular” grid:

- will not self interect
- Each point in the grid as a neighbor in each of the (+/- i, +/- j, +/- k) directions, except for the points at the edge (“the edge cases”).

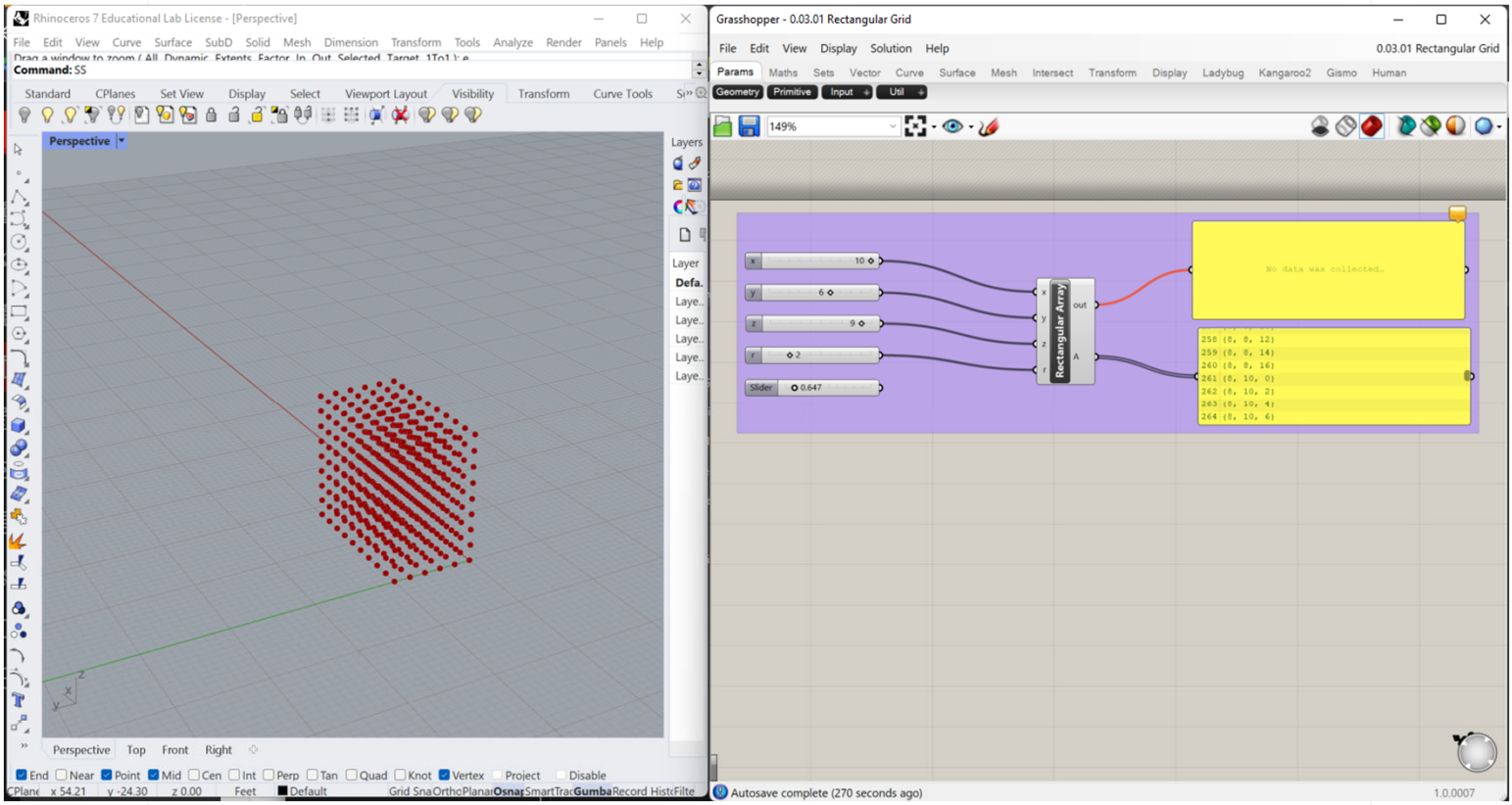
Create a 1-dimensional algorithm and then extend this algorithm to a 2D or 3D algorithm, using the template. Your algorithm may:

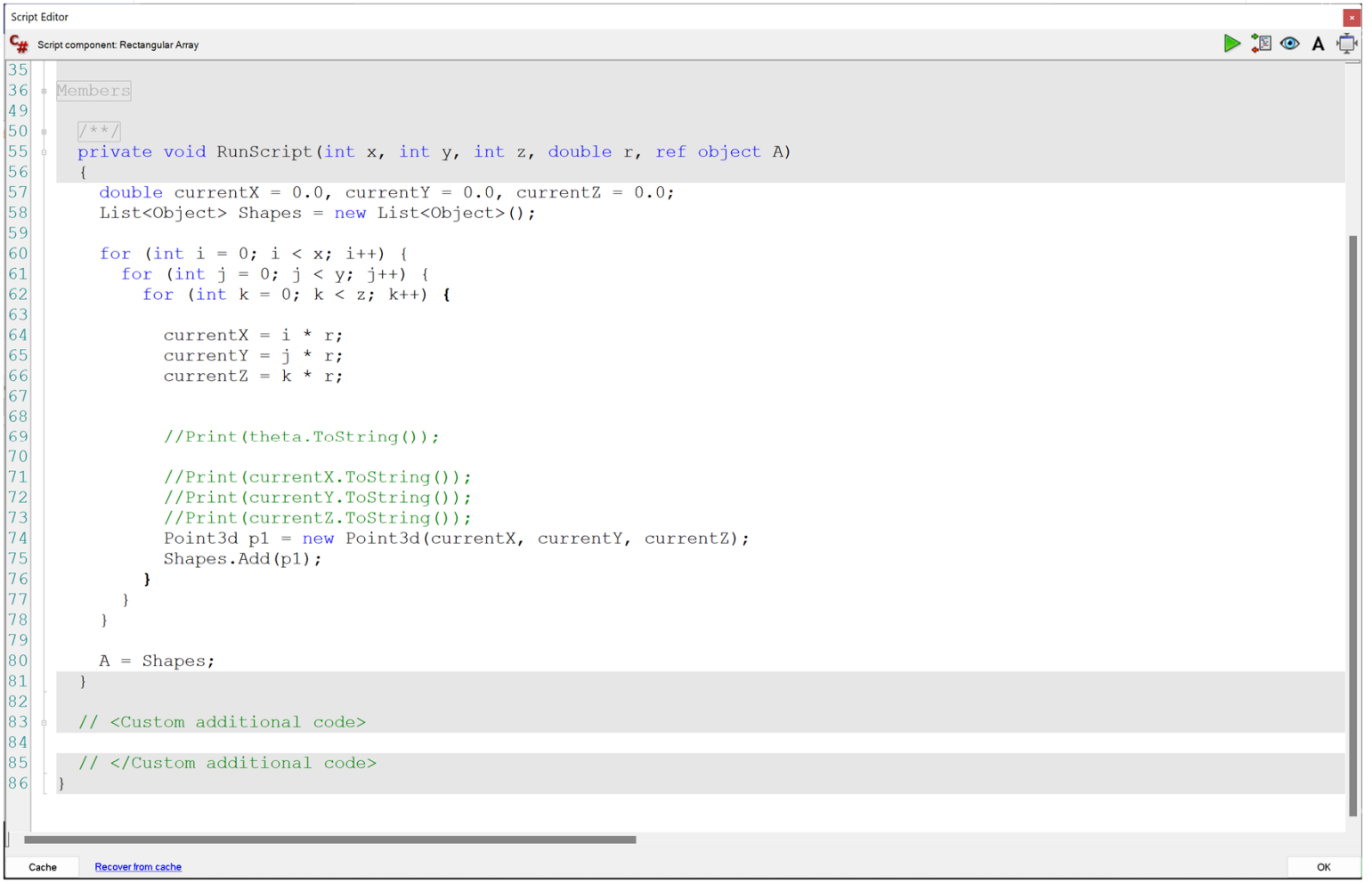
- Modify the placement, spacing, etc of Point3d locations over the field of the grid,
- Place objects at the location of the grid points, or more complex objects may be constructed with multiple grid or other points.
- Place or not place objects, replace objects, etc based on the indices of the points, random numbers, etc.

Include screen captures of the resulting spatial configurations of the algorithm and the code used to generate your two versions. Paste these screen captures in this presentation, with 1 slide per algorithm. Include your name on each slide.

Use this template to start the project. 1, 2 and 3D arrays are included.

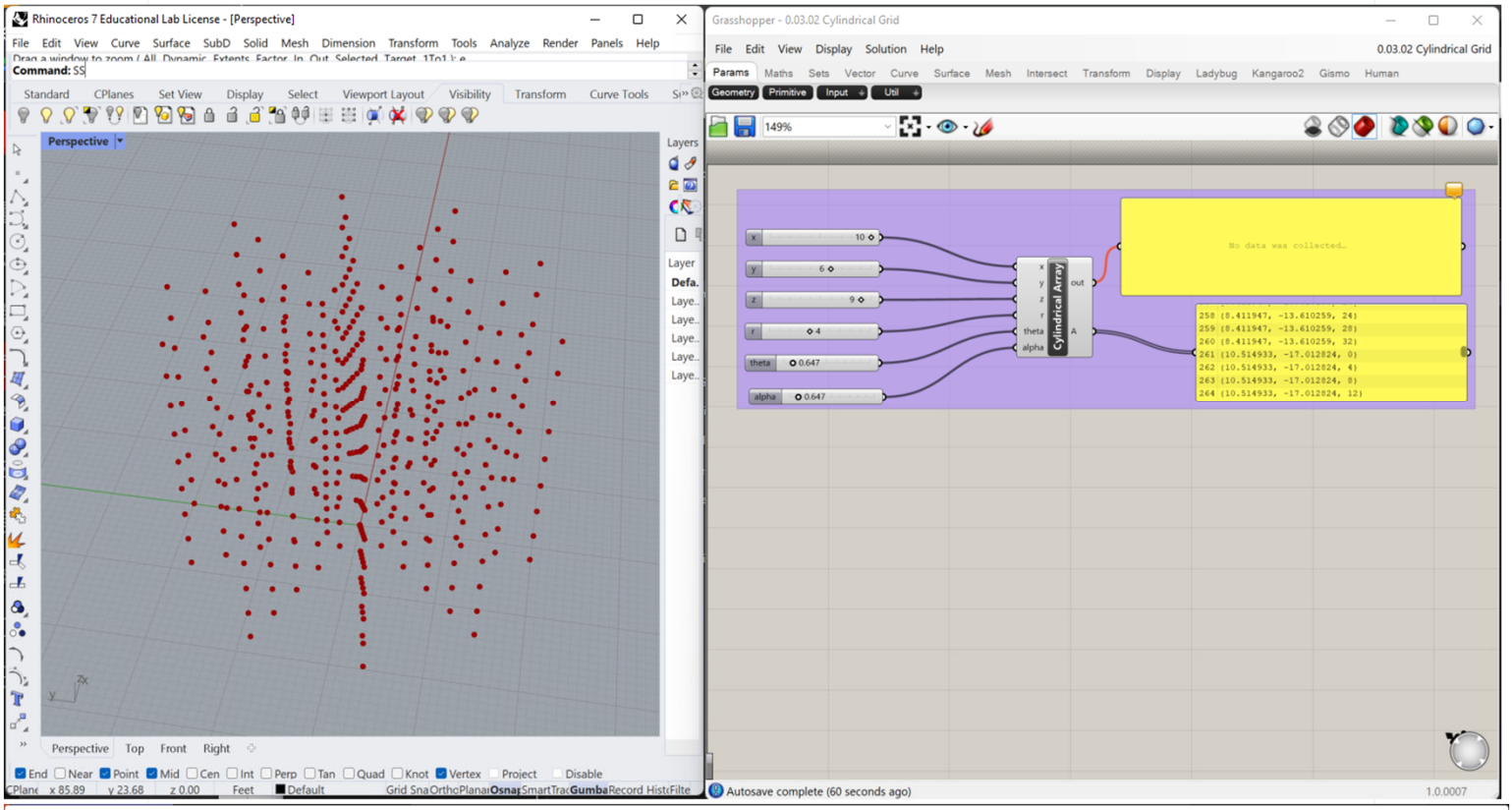
0.03.01 Basic rectangular grid

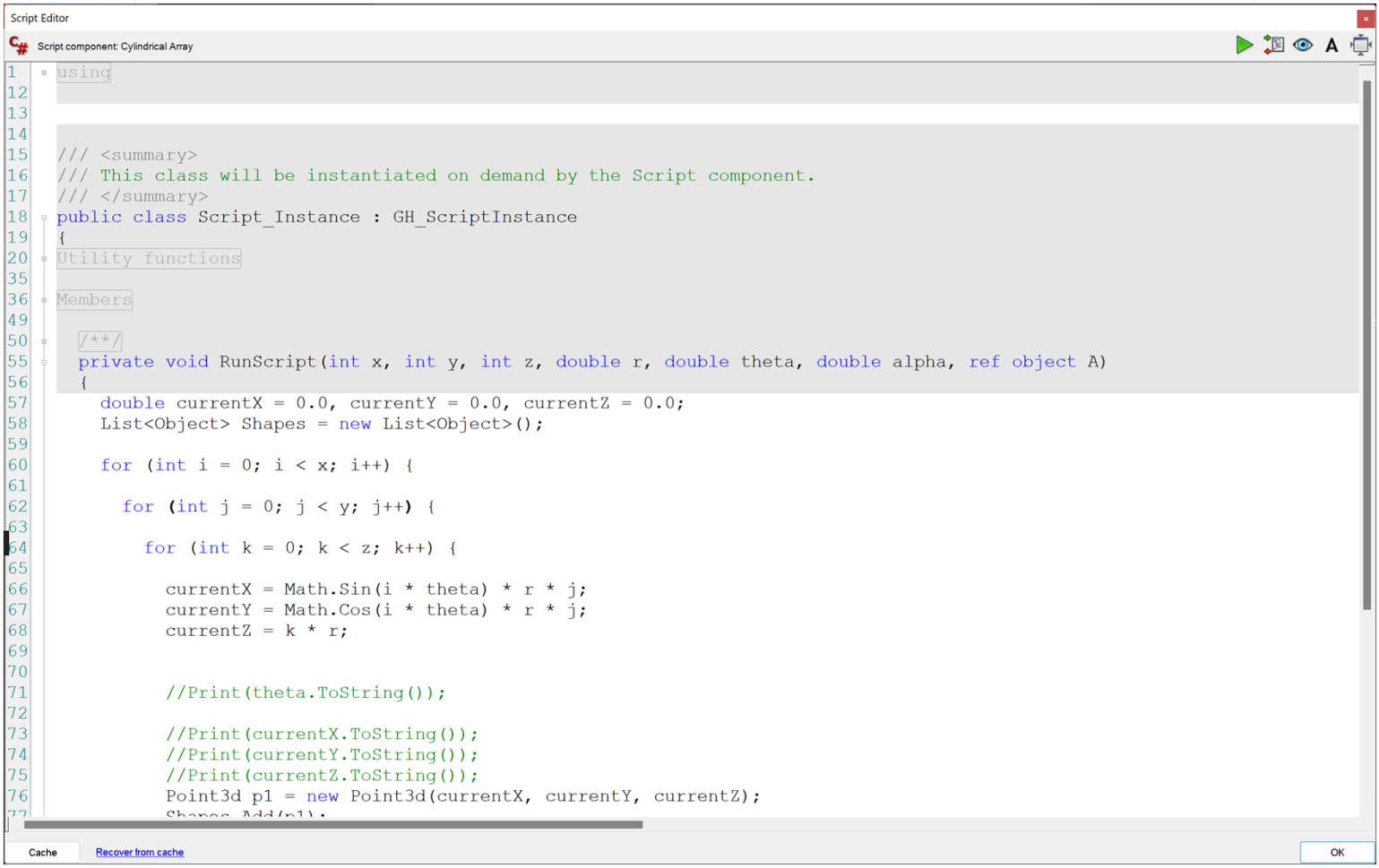




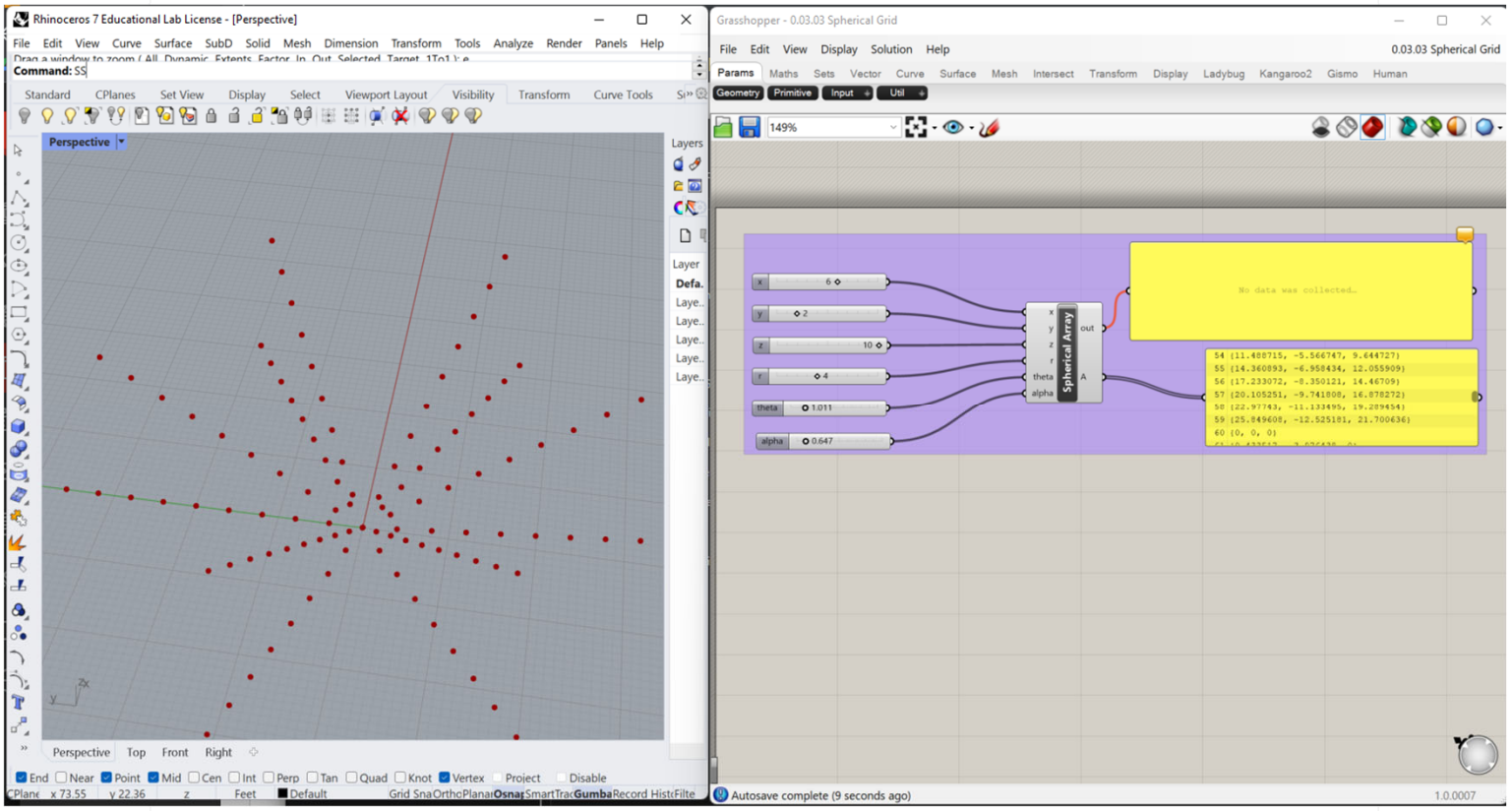
0.03.02 Basic cylindrical grid

>>





0.03.02 Basic spherical grid



Script Editor

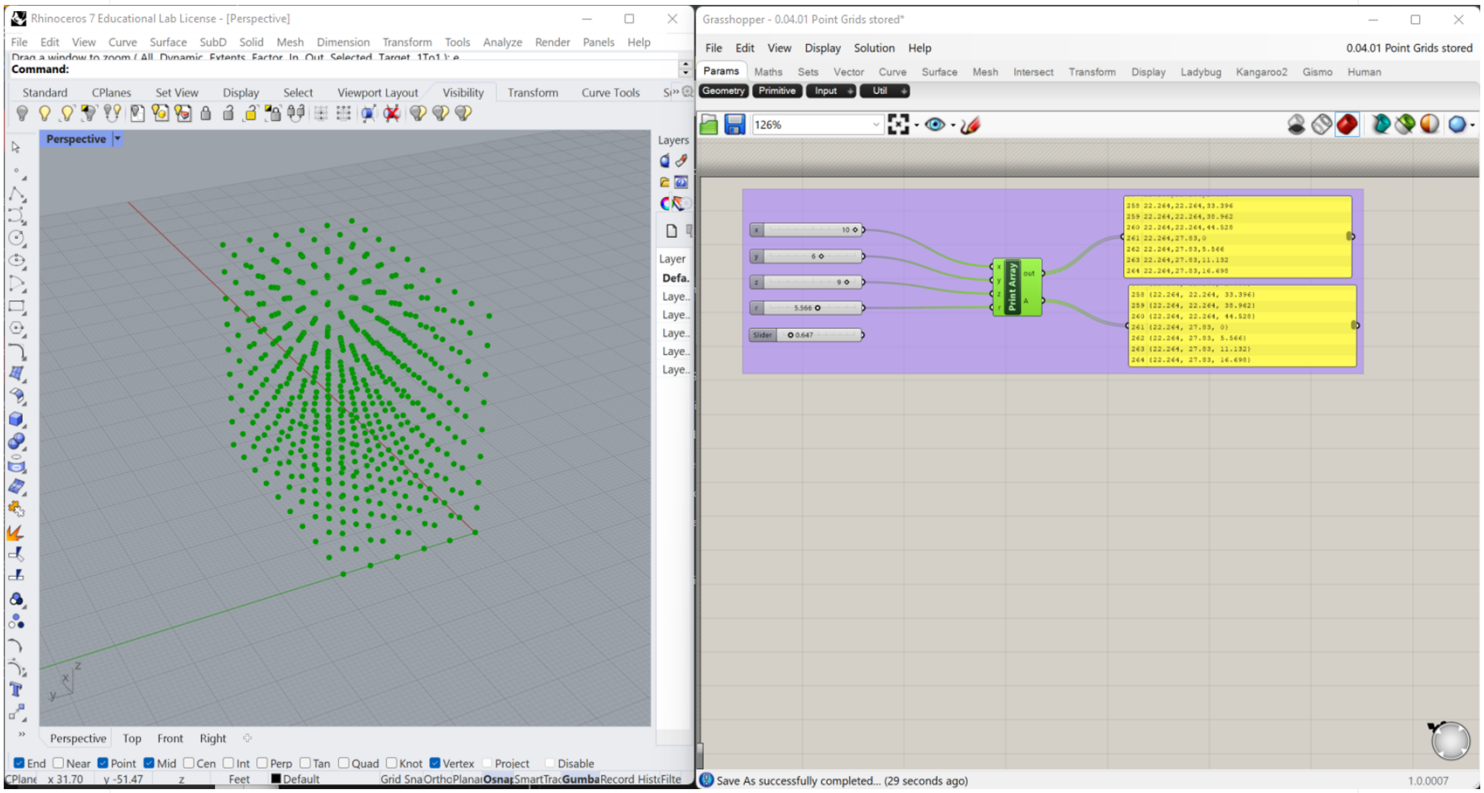
Script component: Cylindrical Array

```
using


/// <summary>
/// This class will be instantiated on demand by the Script component.
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
Utility functions

Members

  /**/
  private void RunScript(int x, int y, int z, double r, double theta, double alpha, ref object A)
  {
    double currentX = 0.0, currentY = 0.0, currentZ = 0.0;
    List<Object> Shapes = new List<Object>();

    for (int i = 0; i < x; i++) {

      for (int j = 0; j < y; j++) {

        for (int k = 0; k < z; k++) {

          currentX = Math.Sin(i * theta) * r * j;
          currentY = Math.Cos(i * theta) * r * j;
          currentZ = k * r;


          //Print(theta.ToString());

          //Print(currentX.ToString());
          //Print(currentY.ToString());
          //Print(currentZ.ToString());
          Point3d p1 = new Point3d(currentX, currentY, currentZ);
          Shapes.Add(p1);
```

Cache Recover from cache OK

0.04.01 Basic grid with stored points

>>

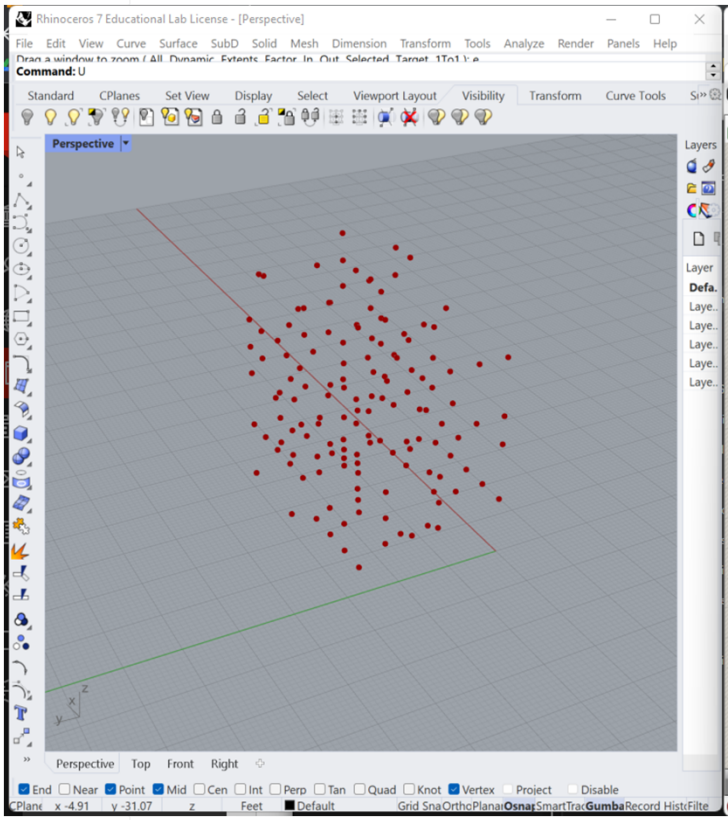


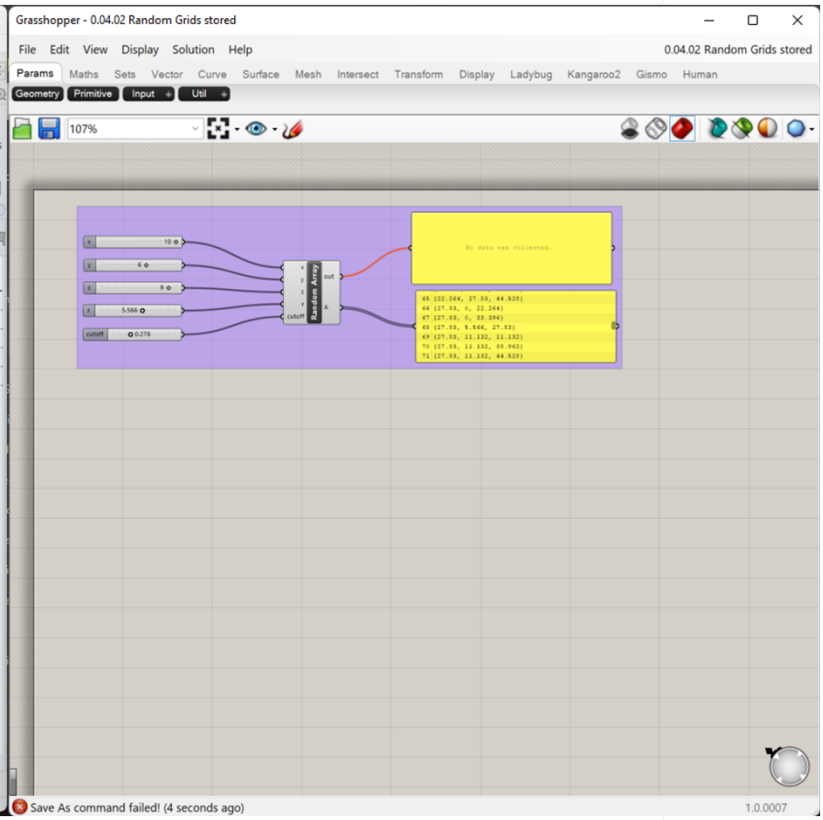
Script Editor

Script component: Print Array

```
49
50    /**/
55    private void RunScript(int x, int y, int z, double r, ref object A)
56    {
57      double currentX = 0.0, currentY = 0.0, currentZ = 0.0;
58      List<Object> Shapes = new List<Object>();
59      Object[,,] ShapeArray = new Object[x, y, z];
60
61      for (int i = 0; i < x; i++) {
62        for (int j = 0; j < y; j++) {
63          for (int k = 0; k < z; k++) {
64
65            currentX = i * r;
66            currentY = j * r;
67            currentZ = k * r;
68
69
70            //Print(theta.ToString());
71
72            //Print(currentX.ToString());
73            //Print(currentY.ToString());
74            //Print(currentZ.ToString());
75            Point3d p1 = new Point3d(currentX, currentY, currentZ);
76            ShapeArray[i, j, k] = p1;
77            Shapes.Add(p1);
78          } // end k
79        } // end j
80      } // end i
81
82      for (int i = 0; i < x; i++) {
83        for (int j = 0; j < y; j++) {
84          for (int k = 0; k < z; k++) {
85            Print(ShapeArray[i, j, k].ToString());
86          } // end k
87        } // end j
88      } // end i
89
90      A = Shapes;
91    }
92
```

0.04.02 Random Grids stored.gh



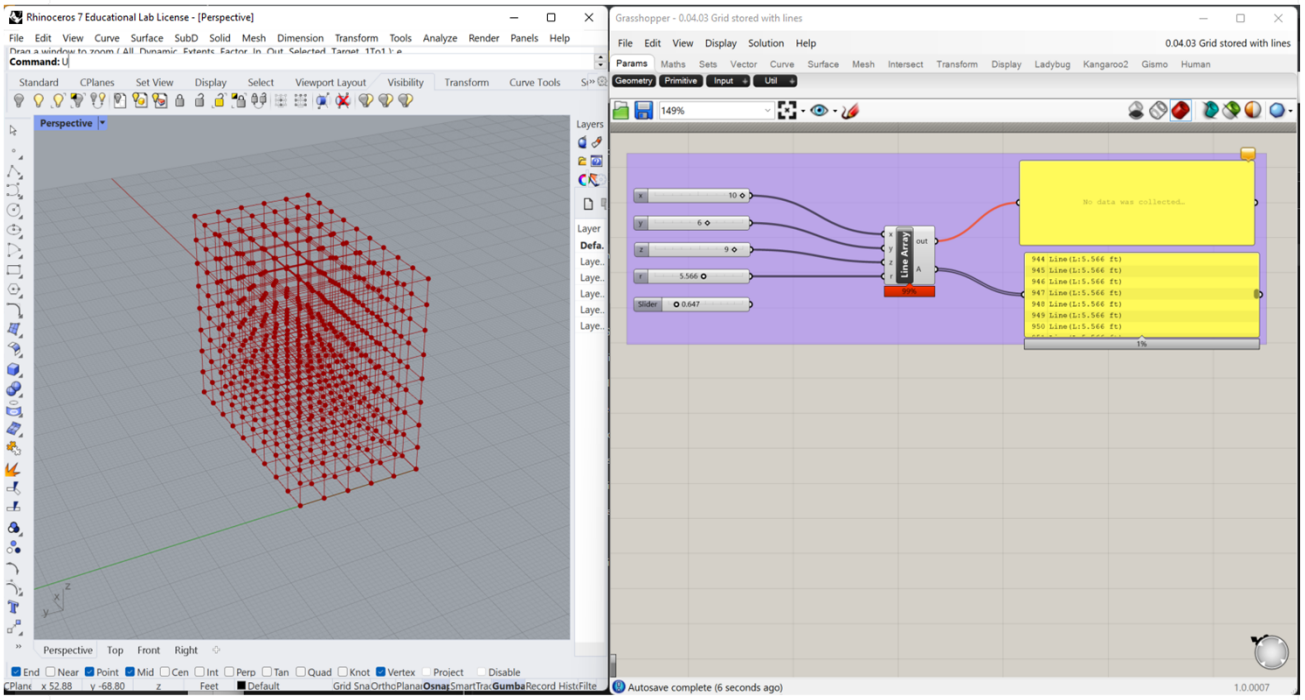


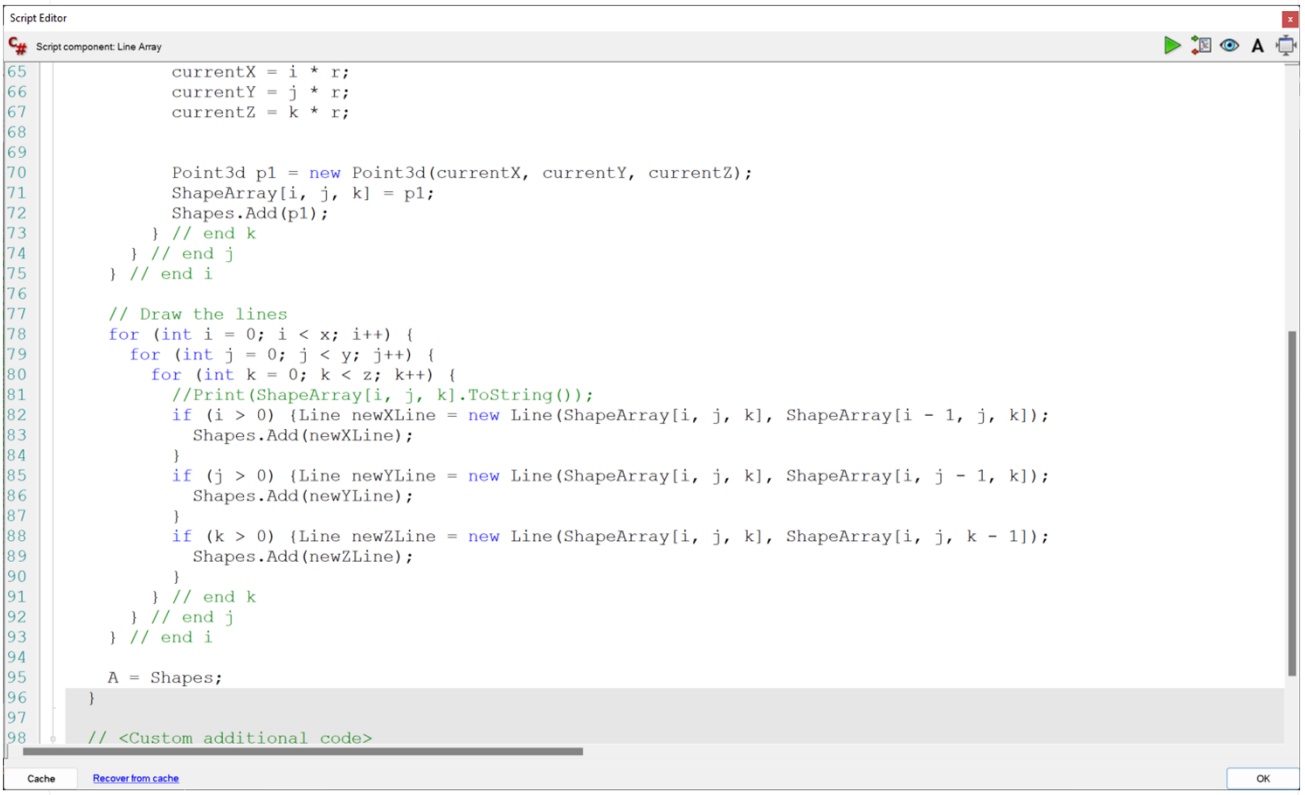
Script Editor

Script component: Random Array

```
62
63      for (int i = 0; i < x; i++) {
64        for (int j = 0; j < y; j++) {
65          for (int k = 0; k < z; k++) {
66
67            currentX = i * r;
68            currentY = j * r;
69            currentZ = k * r;
70
71
72            //Print(theta.ToString());
73
74            //Print(currentX.ToString());
75            //Print(currentY.ToString());
76            //Print(currentZ.ToString());
77            Point3d p1 = new Point3d(currentX, currentY, currentZ);
78            ShapeArray[i, j, k] = p1;
79
80          } // end k
81        } // end j
82      } // end i
83
84      for (int i = 0; i < x; i++) {
85        for (int j = 0; j < y; j++) {
86          for (int k = 0; k < z; k++) {
87            if (myRand.NextDouble() < cutoff) Shapes.Add(ShapeArray[i, j, k]);
88          } // end k
89        } // end j
90      } // end i
91
92      A = Shapes;
93    }
94
95    // <Custom additional code>
96
97    // </Custom additional code>
98  }
```

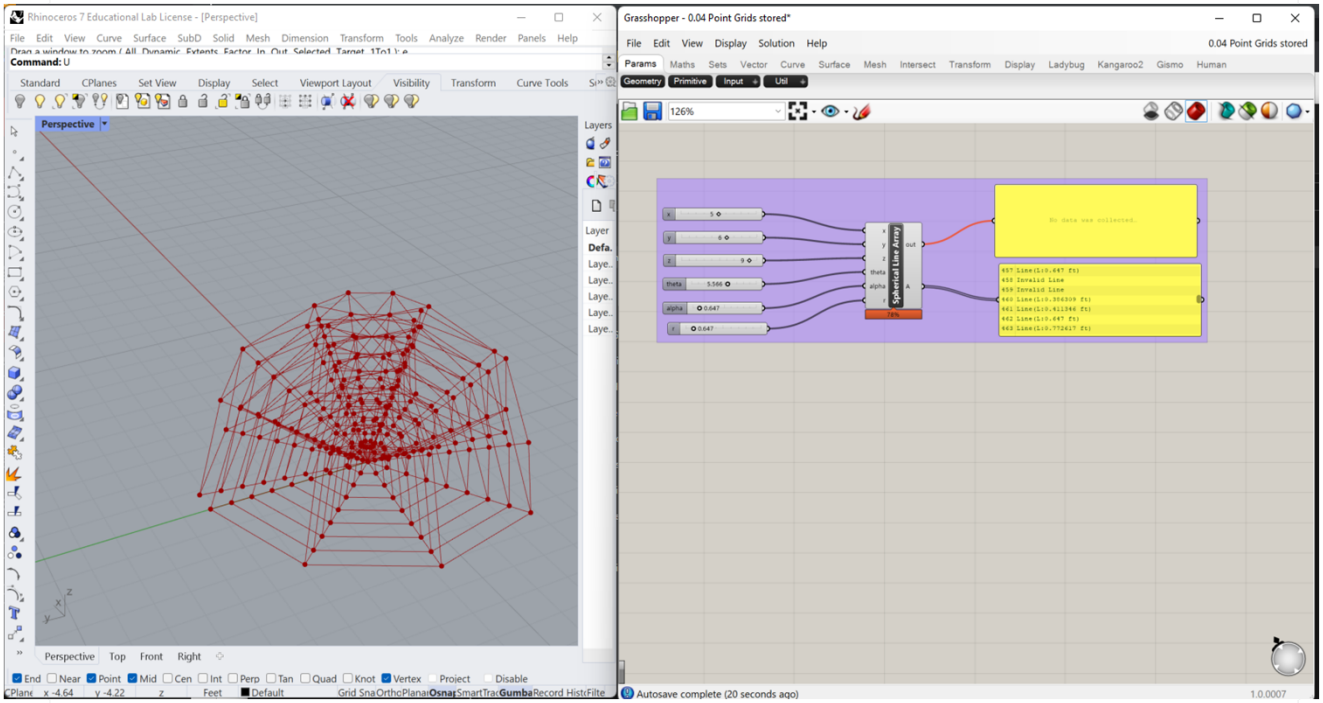
0.04.03 Grid stored with lines.gh






```
            currentX = i * r;
            currentY = j * r;
            currentZ = k * r;


            Point3d p1 = new Point3d(currentX, currentY, currentZ);
            ShapeArray[i, j, k] = p1;
            Shapes.Add(p1);
          } // end k
        } // end j
      } // end i

      // Draw the lines
      for (int i = 0; i < x; i++) {
        for (int j = 0; j < y; j++) {
          for (int k = 0; k < z; k++) {
            //Print(ShapeArray[i, j, k].ToString());
            if (i > 0) {Line newXLine = new Line(ShapeArray[i, j, k], ShapeArray[i - 1, j, k]);
              Shapes.Add(newXLine);
            }
            if (j > 0) {Line newYLine = new Line(ShapeArray[i, j, k], ShapeArray[i, j - 1, k]);
              Shapes.Add(newYLine);
            }
            if (k > 0) {Line newZLine = new Line(ShapeArray[i, j, k], ShapeArray[i, j, k - 1]);
              Shapes.Add(newZLine);
            }
          } // end k
        } // end j
      } // end i

      A = Shapes;
    }

    // <Custom additional code>
```

0.04.04 Spherical Grid stored with lines.gh



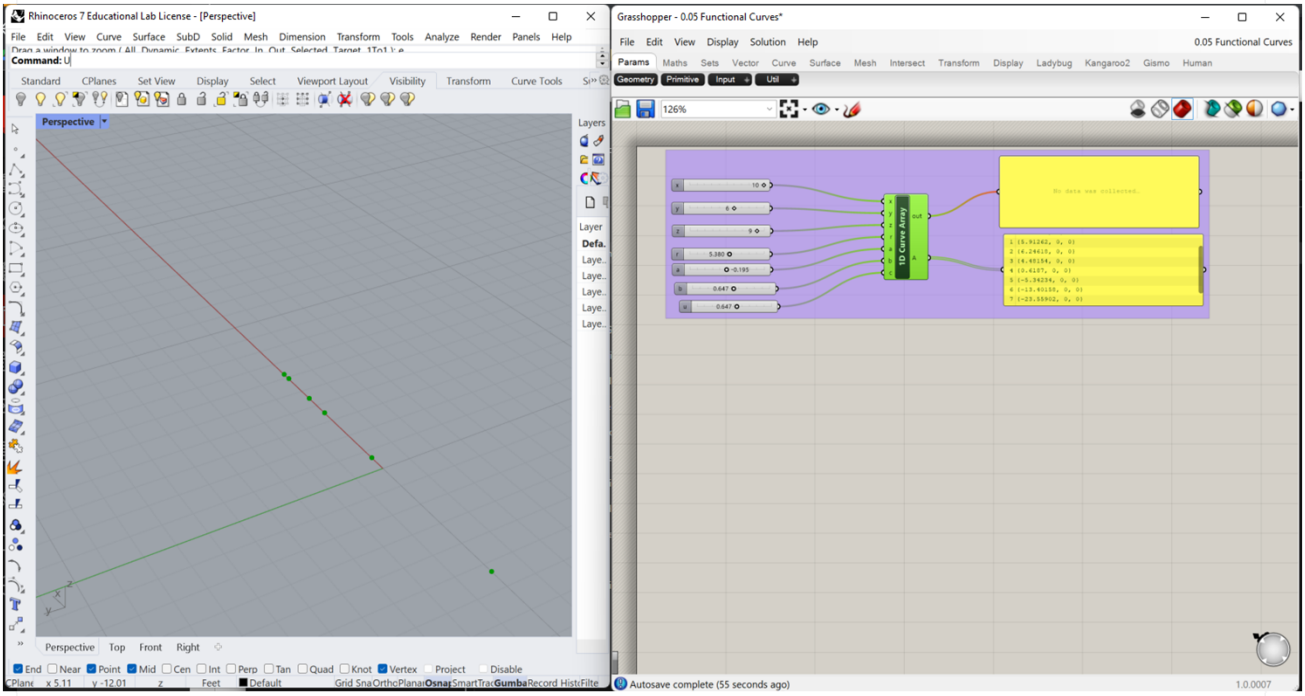
Script Editor

Script component: Spherical Line Array

```
57      double currentX = 0.0, currentY = 0.0, currentZ = 0.0;
58      List<Object> Shapes = new List<Object>();
59      Point3d[,,] ShapeArray = new Point3d[x, y, z];
60
61      for (int i = 0; i < x; i++) {
62        for (int j = 0; j < y; j++) {
63          for (int k = 0; k < z; k++) {
64
65            currentX = Math.Sin(i * theta) * Math.Cos(j * alpha) * r * k;
66            currentY = Math.Cos(i * theta) * Math.Cos(j * alpha) * r * k;
67            currentZ = Math.Sin(j * alpha) * r * k;
68
69
70            //Print(theta.ToString());
71
72            //Print(currentX.ToString());
73            //Print(currentY.ToString());
74            //Print(currentZ.ToString());
75            Point3d p1 = new Point3d(currentX, currentY, currentZ);
76            ShapeArray[i, j, k] = p1;
77            Shapes.Add(p1);
78          } // end k
79        } // end j
80      } // end i
81
82      // Draw the lines
83      for (int i = 0; i < x; i++) {
84        for (int j = 0; j < y; j++) {
85          for (int k = 0; k < z; k++) {
86            //Print(ShapeArray[i, j, k].ToString());
87            if (i > 0) {Line newXLine = new Line(ShapeArray[i, j, k], ShapeArray[i - 1, j, k]);
88              Shapes.Add(newXLine);
89            }
90            if (j > 0) {Line newYLine = new Line(ShapeArray[i, j, k], ShapeArray[i, j - 1, k]);
91              Shapes.Add(newYLine);
92            }
93            if (k > 0) {Line newZLine = new Line(ShapeArray[i, j, k], ShapeArray[i, j, k - 1]);
94              Shapes.Add(newZLine);
95            }
96          } // end k
```

Cache Recover from cache OK

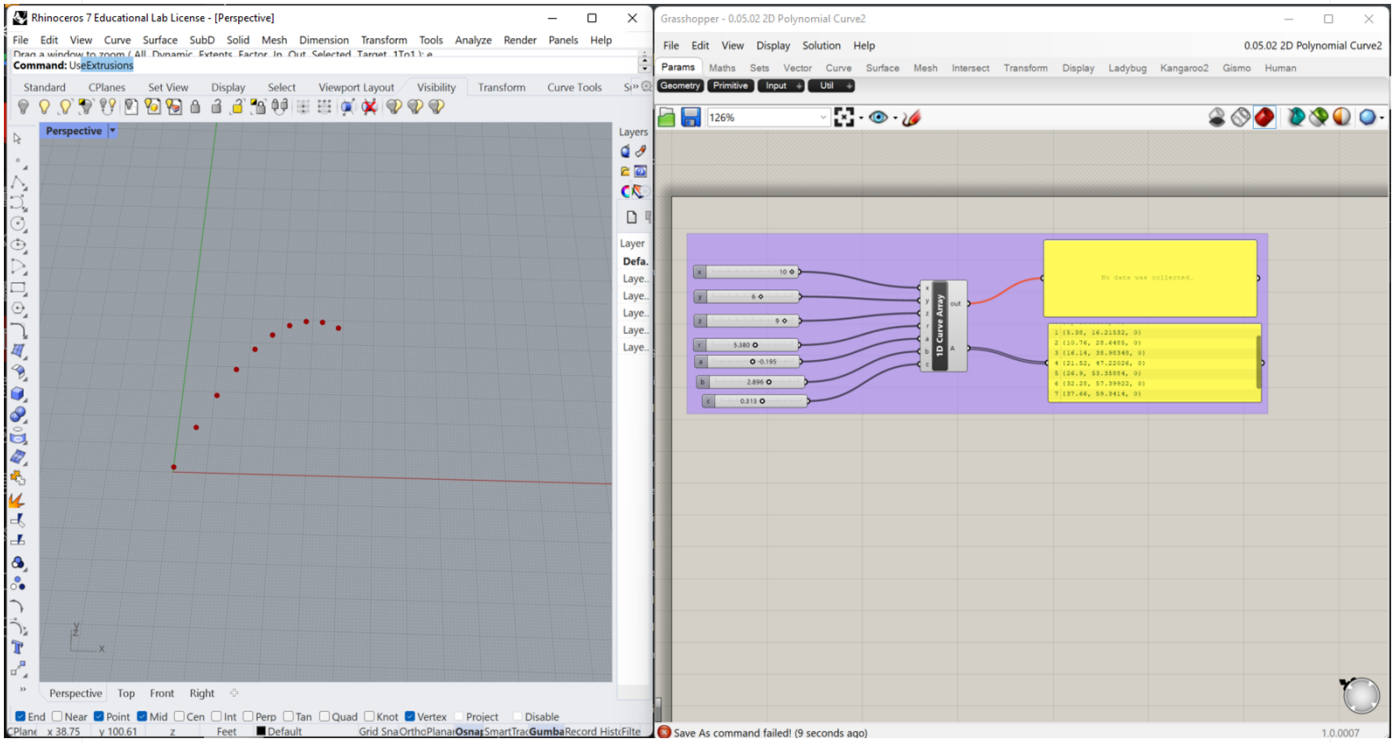
0.05.01 1D Functional Curves.gh




```
Script Editor
Script component: 1D Curve Array
```

```
using


/// <summary>
/// This class will be instantiated on demand by the Script component.
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
Utility functions

Members

  /**/
  private void RunScript(int x, int y, int z, double r, double a, double b, double c, ref object A)
  {
    double currentX = 0.0, currentY = 0.0, currentZ = 0.0;
    List<Object> Shapes = new List<Object>();

    for (int i = 0; i < x; i++) {

      //for (int j = 0; j < y; j++) {

      // for (int k = 0; k < z; k++) {

      currentX = (a * (i * i) + b * i + c) * r;


      Point3d p1 = new Point3d(currentX, currentY, currentZ);
      Shapes.Add(p1);
      // } // end k
      // } // end j
    } // end i

    A = Shapes;
  }

  // <Custom additional code>
```

0.05.02 2D Polynomial Curve2.gh

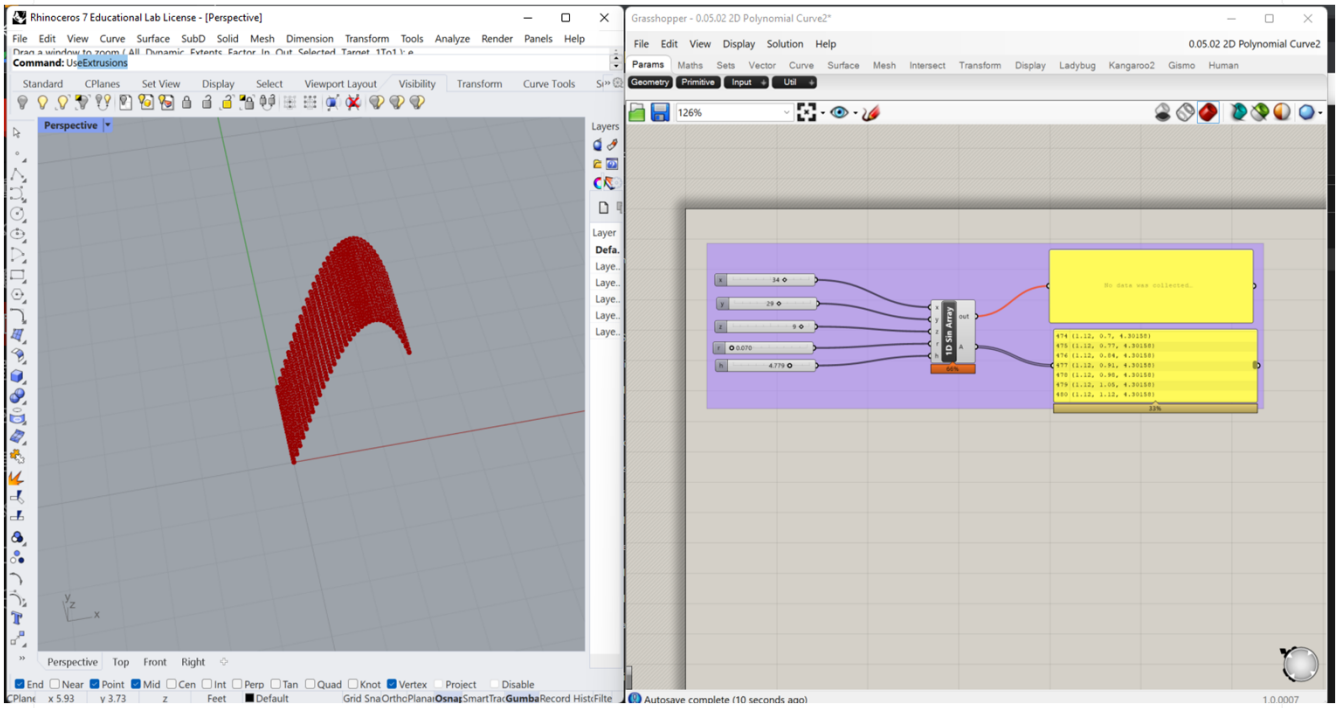


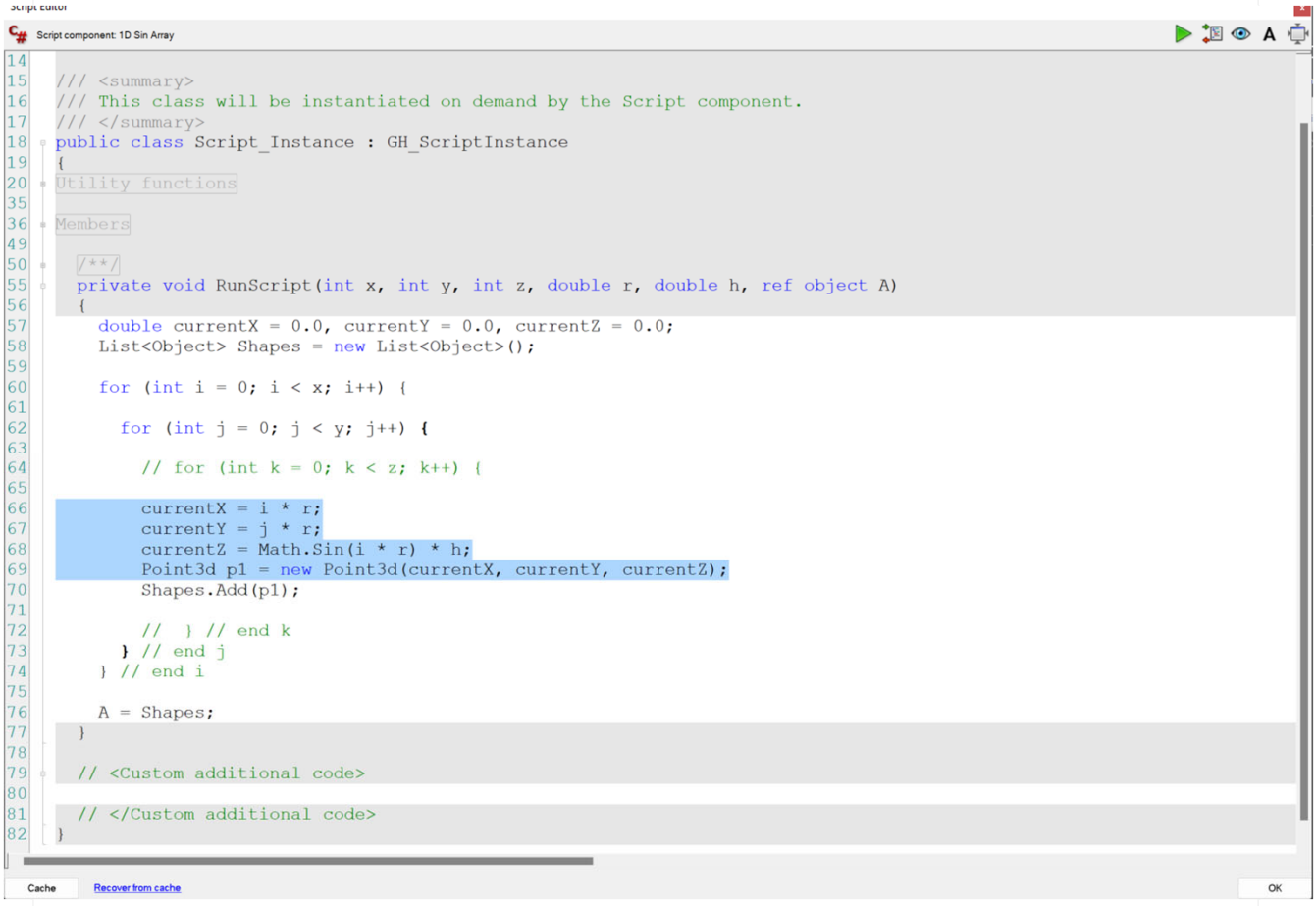
Script Editor

Script component: 1D Curve Array

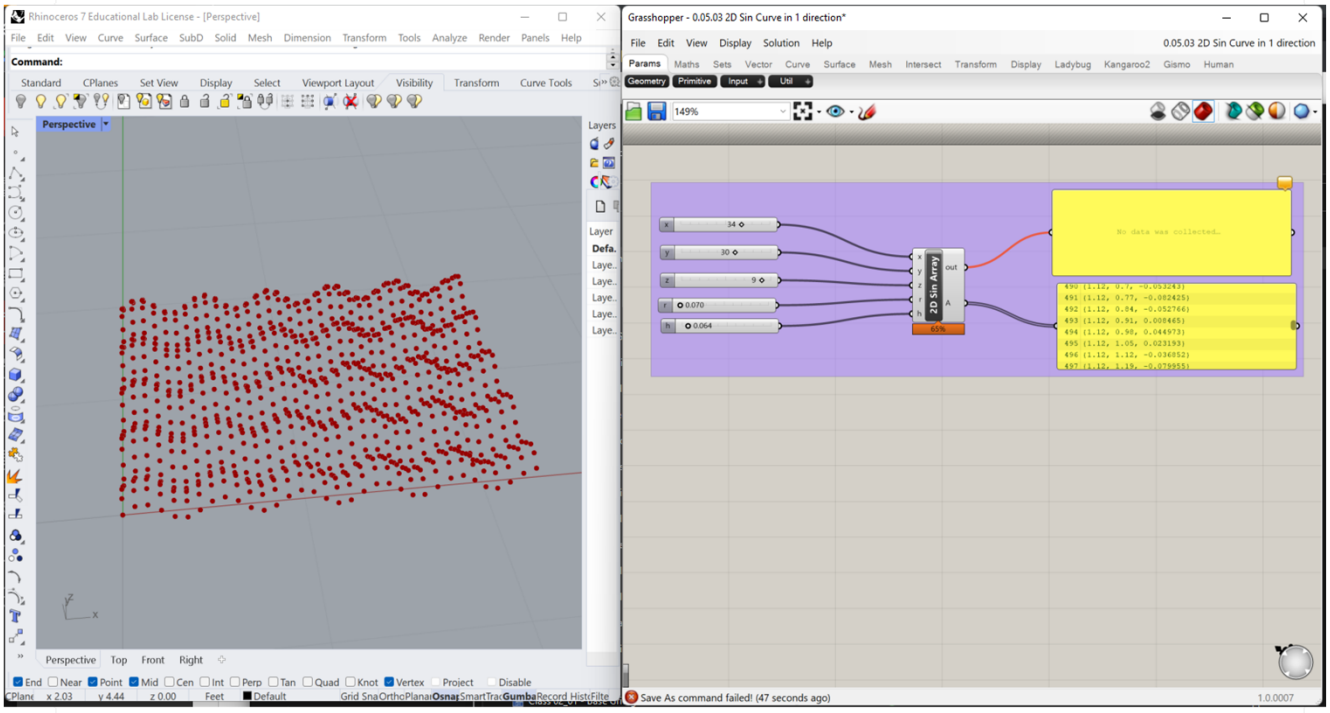
```
/// <summary>
/// This class will be instantiated on demand by the Script component.
/// </summary>
public class Script_Instance : GH_ScriptInstance
{
Utility functions

Members

  /**/
  private void RunScript(int x, int y, int z, double r, double a, double b, double c, ref object A)
  {
    double currentX = 0.0, currentY = 0.0, currentZ = 0.0;
    List<Object> Shapes = new List<Object>();

    for (int i = 0; i < x; i++) {

      //for (int j = 0; j < y; j++) {

      // for (int k = 0; k < z; k++) {

      currentX = i * r;
      currentY = (a * (i * i) + b * i + c) * r;

      Point3d p1 = new Point3d(currentX, currentY, currentZ);
      Shapes.Add(p1);
      // } // end k
      // } // end j
    } // end i

    A = Shapes;
  }

  // <Custom additional code>

  // </Custom additional code>
}
```

0.05.03 2D Sin Curve in 1 direction.gh





0.05.04 2D Sin Curve in 2 directions.gh



Script Editor

Script component: 2D Sin Array

```
14
15  /// <summary>
16  /// This class will be instantiated on demand by the Script component.
17  /// </summary>
18  public class Script_Instance : GH_ScriptInstance
19  {
20  #region Utility functions
35
36  #region Members
49
50    /**/
55    private void RunScript(int x, int y, int z, double r, double h, ref object A)
56    {
57      double currentX = 0.0, currentY = 0.0, currentZ = 0.0;
58      List<Object> Shapes = new List<Object>();
59
60      for (int i = 0; i < x; i++) {
61
62        for (int j = 0; j < y; j++) {
63
64          // for (int k = 0; k < z; k++) {
65
66            currentX = i * r;
67            currentY = j * r;
68            currentZ = (Math.Sin(i) + Math.Sin(j)) * h;
69
70            Point3d p1 = new Point3d(currentX, currentY, currentZ);
71            Shapes.Add(p1);
72
73          //  } // end k
74        } // end j
75      } // end i
76
77      A = Shapes;
78    }
79
80    // <Custom additional code>
81
82    // </Custom additional code>
83  }
```