

# ALIBRE SCRIPT

MANUAL VERSION 1.5



## DISCLAIMER

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the software purchaser's use, without prior written permission.

© Alibre, LLC 2018-2019, All Rights Reserved

Microsoft® and Windows™ are trademarks or registered trademarks of Microsoft Corporation.

PC® is a registered trademark of International Business Machines Corporation.

# TABLE OF CONTENTS

## CONTENTS

Disclaimer.....	2
Table of Contents.....	3
Chapter 1: Introduction .....	5
Chapter 2: Quick Start.....	6
Chapter 3: Units .....	15
Chapter 4: Polylines .....	16
Creation.....	17
Intersection of Two Polylines.....	18
Rotation .....	18
Joining .....	18
Translation .....	18
Copying .....	18
Chapter 5: Sketch Manipulation .....	20
Copying .....	20
Rotation .....	21
Translation .....	21
Scaling .....	22
Combination.....	22
Chapter 6: Involute Gears .....	24
Theory .....	24
Creating Gears.....	24
Advanced Functionality.....	27
Chapter 7: Modifying Existing Parts & Sketches .....	32
Chapter 8: Hints & Tips .....	34
Chapter 9: Running Scripts – Advanced .....	35
Chapter 10: Built-In Functionality .....	36
Details of the Current Script .....	36
Current Part or Assembly.....	36

Currently Opened Parts and Assemblies .....	36
The Alibre Script Version.....	36
Graphical User Interface .....	37
Getting User Input .....	37
Utility Libraries .....	37
Chapter 11: 2D Sketch Mapping .....	38
Face Mapping.....	38
Generic Mapping.....	41
Chapter 12: Editor.....	45
Undo And Redo .....	45
Select, Copy and Paste .....	45
Find and Replace.....	45
Quick Find .....	48
Go To.....	48
Run the Script.....	49
Save .....	49
Code Folding .....	49

## CHAPTER 1: INTRODUCTION

Alibre Script brings scripting to Alibre Design. Scripting provides a powerful means to create sketches and parts based on variables, repetition and algorithms. For example a single script could create a set of 50 parts that are all similar but have slight variations. Creating each part by hand would be tedious and time consuming. Scripts are good at generating precise mathematical shapes, for example the involute curve on the side of a gear tooth.

Alibre Script uses the Python language, which is widely supported and is ideal for rapid script development by non-programmers. Python comes with a large library of functionality that is ready to use out-of-the-box.

Here is a simple example script:

```
Units.Current = UnitTypes.Millimeters
Test = Part("Test")
XYPlane = Test.GetPlane("XY-Plane")
MySketch = Test.AddSketch("MySketch", XYPlane)
MySketch.AddCircle(0, 0, 10, False)
Cylinder = Test.AddExtrudeBoss("Object", MySketch, 5, False)
```

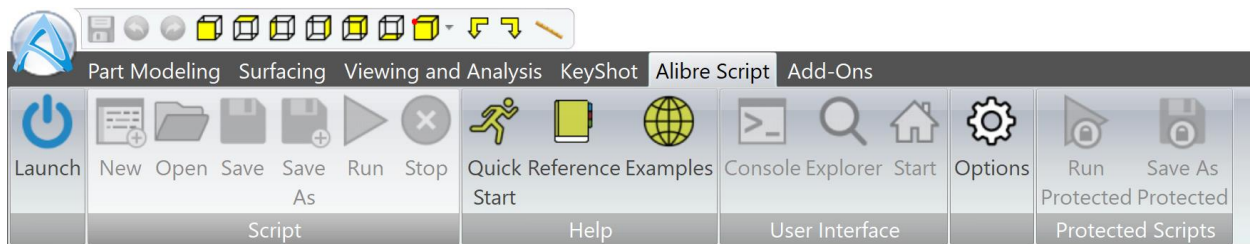
This script creates a new part called “Test” that contains a cylinder which is 10mm in diameter and 5mm in depth.

Familiarity with Alibre Design is required. Familiarity with Python is also required. More details about Python can be found from <http://www.python.org/>. We recommend “Python Essential Reference” published by Addison Wesley.

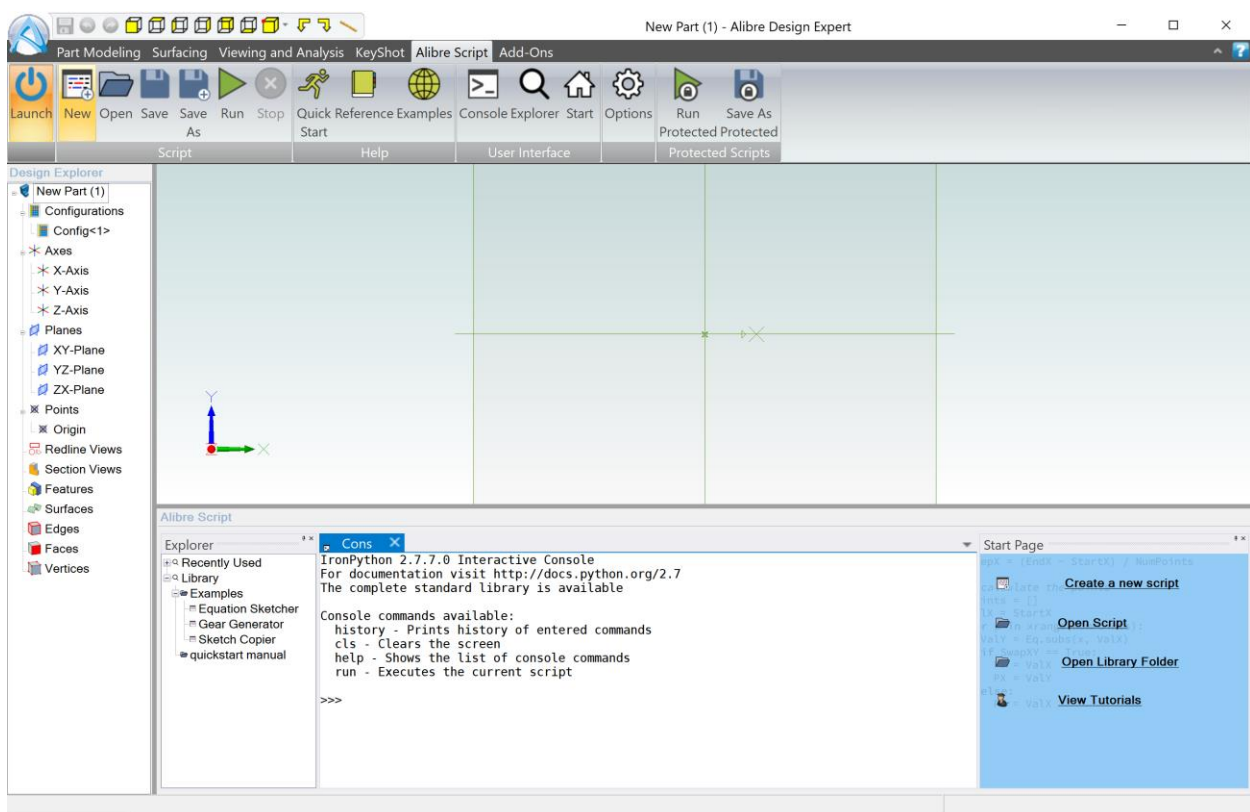
*For a complete list of all functionality provided by Alibre Script please see the separate reference manual.*

## CHAPTER 2: QUICK START

1. Start Alibre Design and open the Alibre Script tab. Click 'Launch' to open the Alibre Script window.



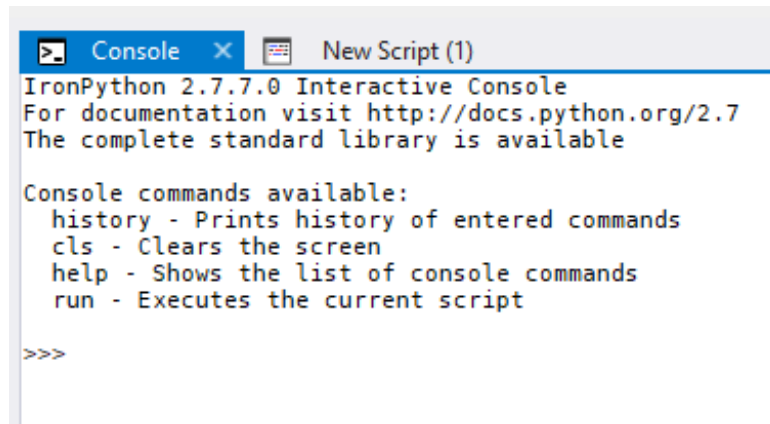
The Alibre Script window looks like this:



Across the top is the toolbar providing access to file, editing and execution features. The main part of the window has two tabs, New Script (1) and Console.

The script tab allows editing of the current script. The console tab provides immediately access to the Python environment.

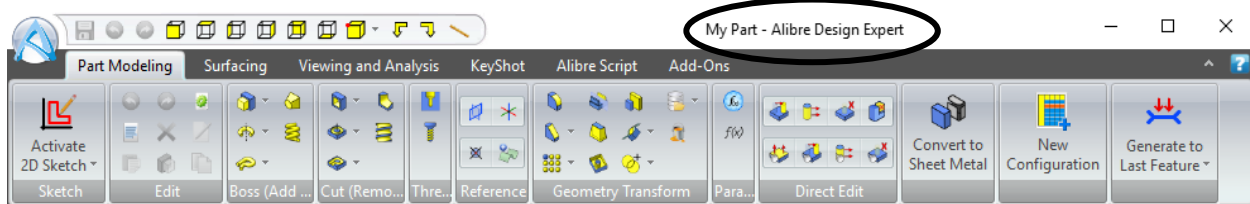
- Click on the Console tab to view the scripting console.



- Click on the script tab and type in:

```
MyPart = Part('My Part')
```

- Press Enter. Alibre Design will spring into action and create a new part ready for editing.



Notice that the name of the part is “My Part” which is the name you entered at the prompt.

- In order to create a sketch we need a plane to create it on. Enter the following in the script window to get access to the X-Y plane:

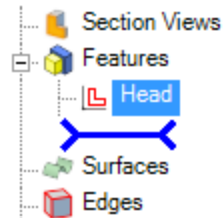
```
XYPlane = MyPart.GetPlane('XY-Plane')
```

Any design plane can be accessed this way by providing the name of the plane shown in the design explorer in Alibre Design.

- Create a new sketch on the XY plane by entering:

```
HeadSketch = MyPart.AddSketch('Head', XYPlane)
```

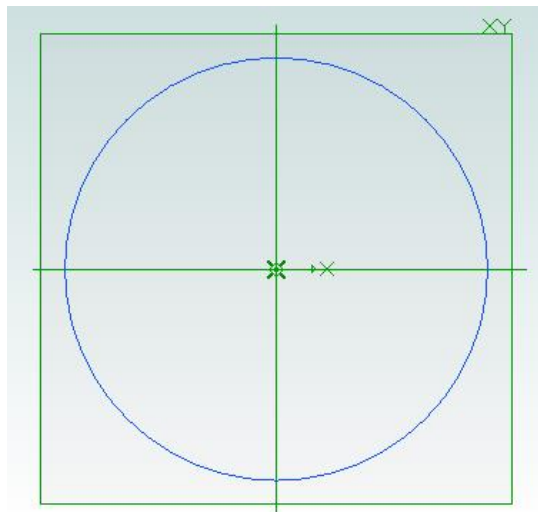
The sketch created can be seen in the design explorer.



7. Now we will create a circle in the sketch centered on the origin. We can do this by entering:

```
HeadSketch.AddCircle(0, 0, 10, False)
```

This command adds a circle to the HeadSketch 10mm in diameter centered at (0,0). The final parameter is set to False. If we set it to True instead, then it would create a reference circle.

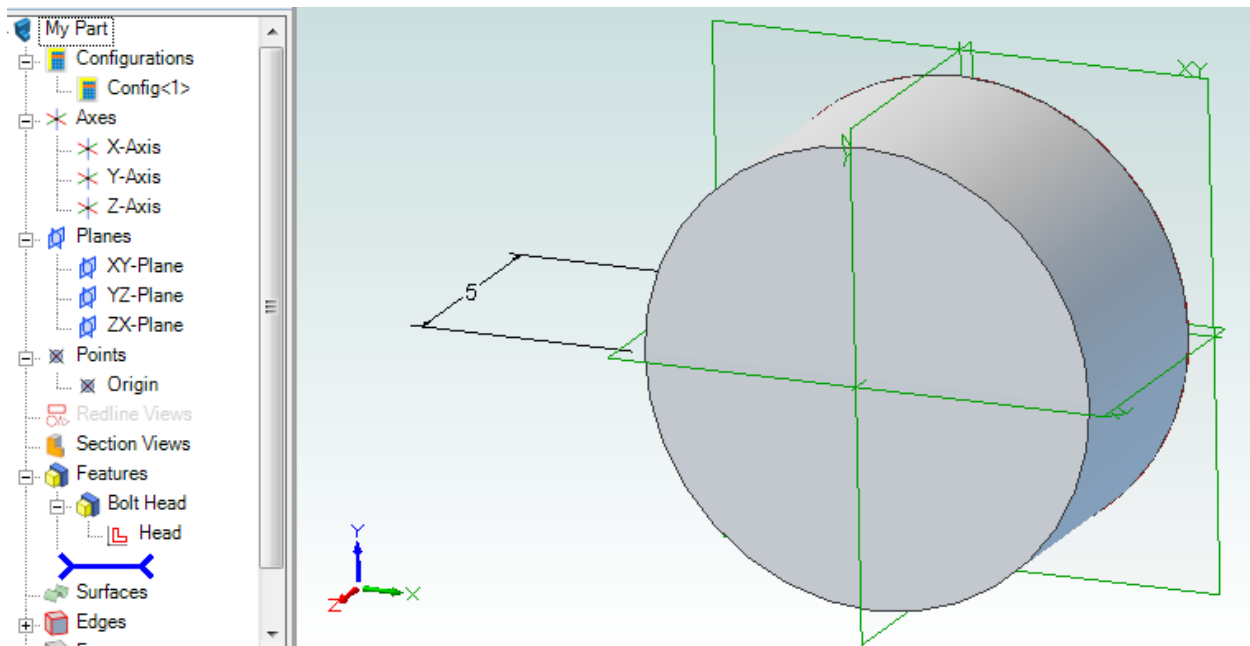


8. Now we have a sketch we can extrude. Enter the following line in the script window:

```
BoltHead = MyPart.AddExtrudeBoss('Bolt Head', HeadSketch, 5, False)
```

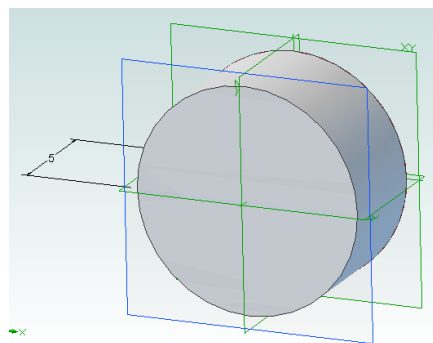
This extrudes the sketch HeadSketch by 5mm. The final parameter is set to False. If we set it to True instead, then the extrusion direction would be reversed.





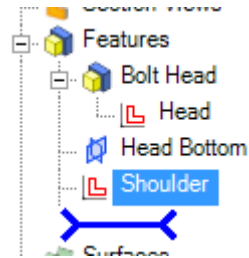
9. We will now create a reference plane 5mm from the XY plane. Enter the following line in the script window:

```
HeadBottomPlane = MyPart.AddPlane('Head Bottom', XYPlane, 5)
```



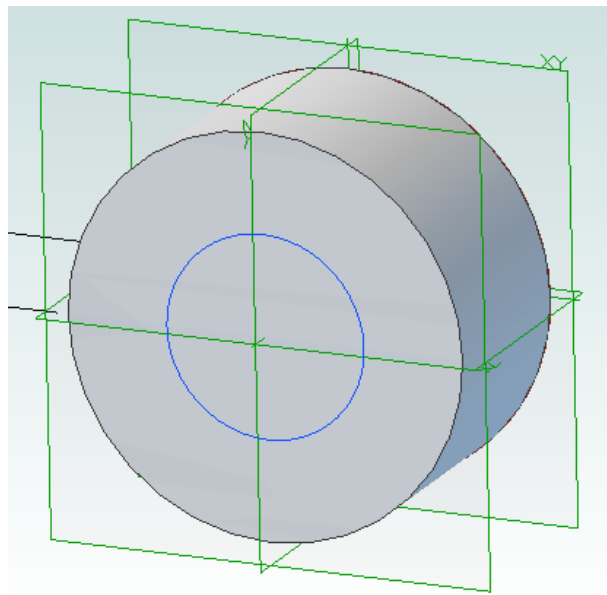
10. Now that we have a reference plane we can create a new sketch on it.

```
ShoulderSketch = MyPart.AddSketch('Shoulder', HeadBottomPlane)
```



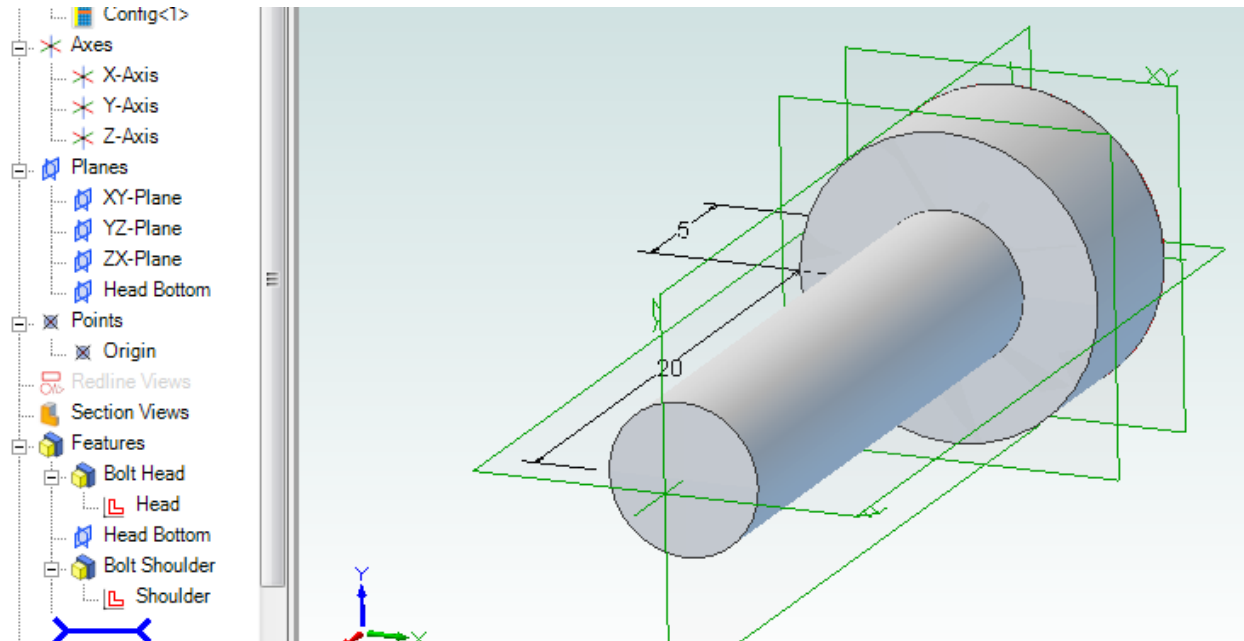
11. Draw a circle on the sketch 5mm in diameter.

```
ShoulderSketch.AddCircle(0, 0, 5, False)
```



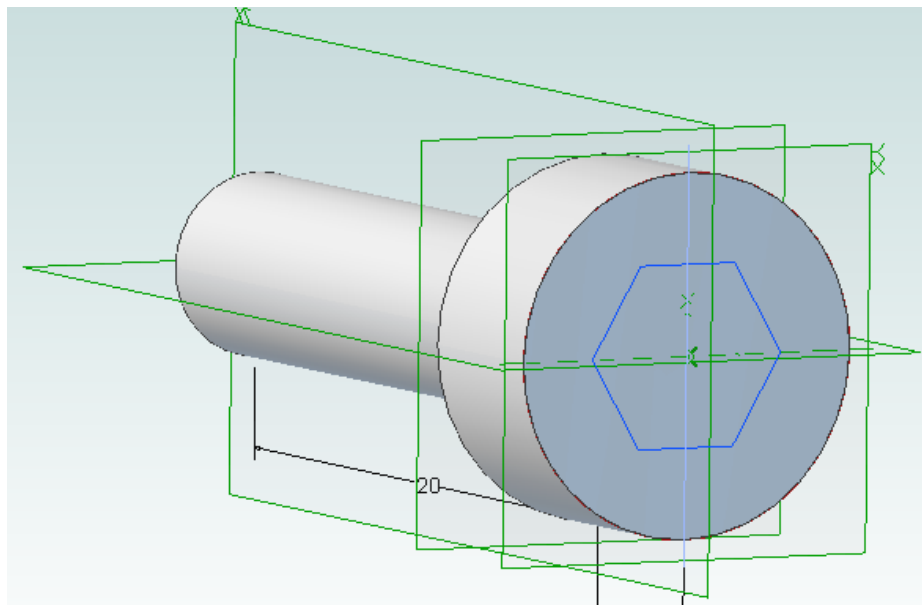
12. Now extrude the sketch 20mm to create the shoulder of the bolt.

```
BoltShoulder = MyPart.AddExtrudeBoss('Bolt Shoulder', ShoulderSketch, 20, False)
```



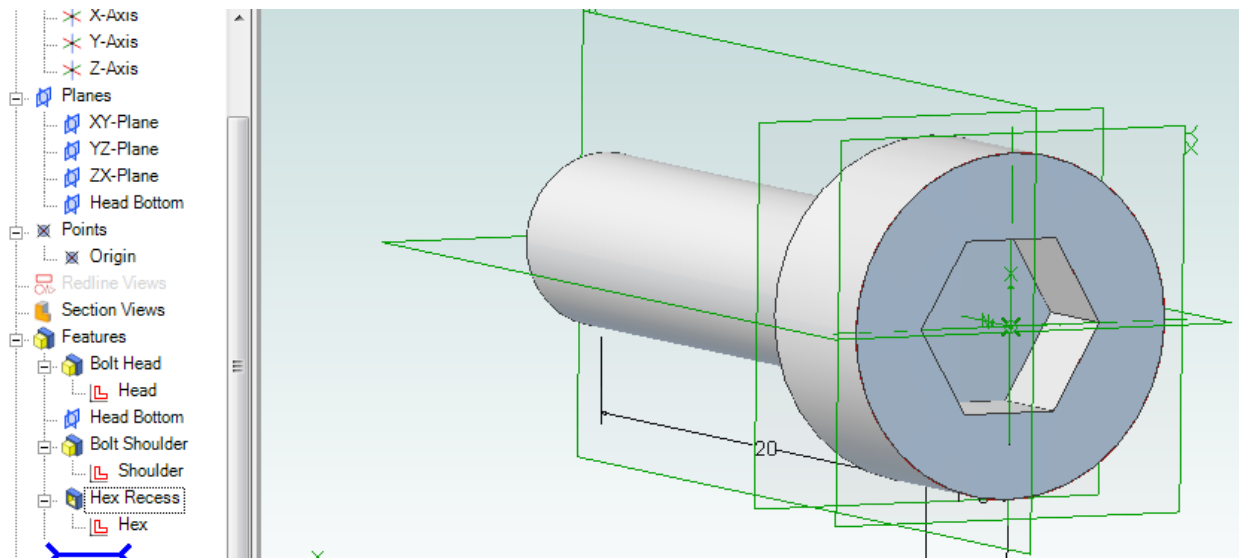
13. Now we will create the allen key recess in the head of the bolt. We start by adding a new sketch to the XY plane that contains a hexagon 5mm in diameter.

```
HexSketch = MyPart.AddSketch('Hex', XYPlane)
HexSketch.AddPolygon(0, 0, 5, 6, False)
```



14. Add an extrude cut to create the recess in the head of the bolt.

```
HexRecess = MyPart.AddExtrudeCut('Hex Recess', HexSketch, 3, False)
```



15. Finally we can save our new part, export it as an STL and then close the window. Replace the following paths with your own.

```
MyPart.Save("C:\Users\Andy\Desktop")
MyPart.ExportSTL("C:\Users\Andy\Desktop\My Part.stl")
MyPart.Close()
```

16. Here is the entire script:

```
MyPart = Part("My Part")

XYPlane = MyPart.GetPlane("XY-Plane")
HeadSketch = MyPart.AddSketch("Head", XYPlane)
HeadSketch.AddCircle(0, 0, 10, False)
BoltHead = MyPart.AddExtrudeBoss("Bolt Head", HeadSketch, 5, False)

HeadBottomPlane = MyPart.AddPlane("Head Bottom", XYPlane, 5)
ShoulderSketch = MyPart.AddSketch("Shoulder", HeadBottomPlane)
```

```

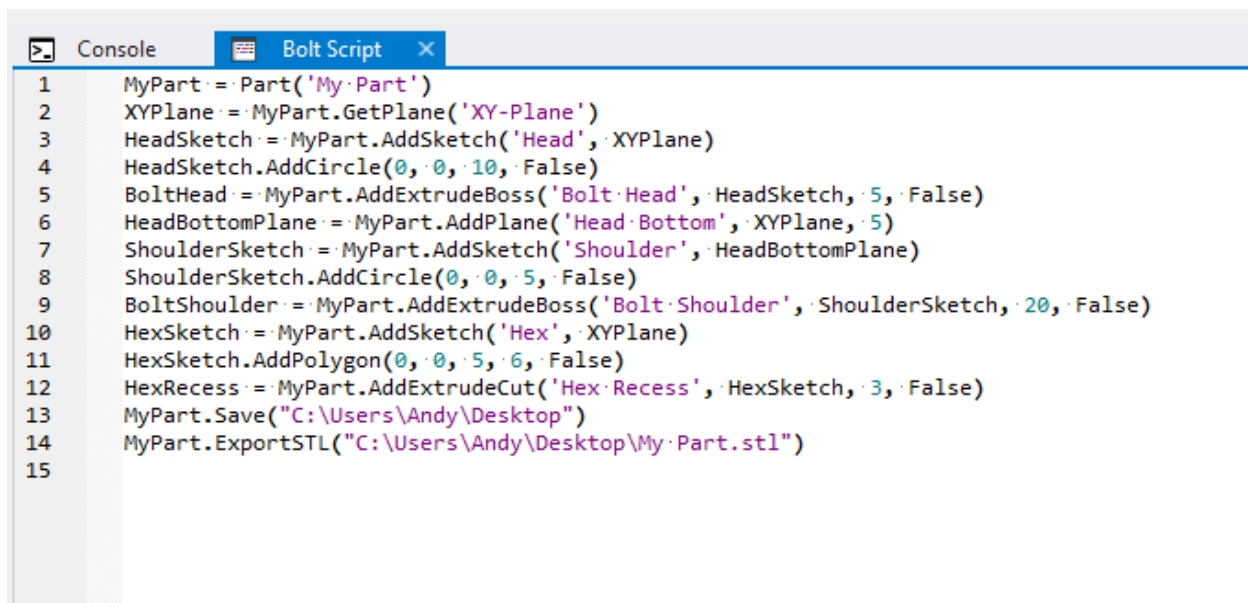
ShoulderSketch.AddCircle(0, 0, 5, False)

BoltShoulder = MyPart.AddExtrudeBoss("Bolt Shoulder", ShoulderSketch, 20,
False)

HexSketch = MyPart.AddSketch("Hex", XYPlane)
HexSketch.AddPolygon(0, 0, 5, 6, False)
HexRecess = MyPart.AddExtrudeCut("Hex Recess", HexSketch, 3, False)

MyPart.Save("C:\Users\Andy\Desktop")
MyPart.ExportSTL("C:\Users\Andy\Desktop\My Part.stl")
MyPart.Close()

```



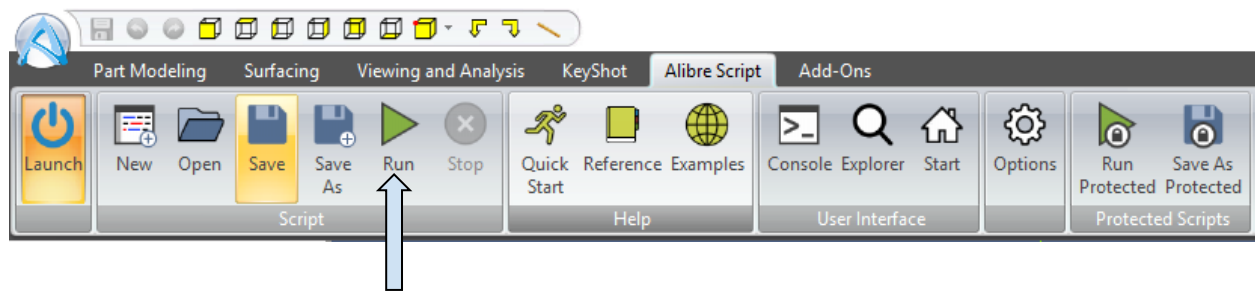
The screenshot shows a software interface with two tabs: 'Console' and 'Bolt Script'. The 'Bolt Script' tab is active and displays a script with 15 lines of code. The code is color-coded: keywords like 'Part', 'GetPlane', 'AddSketch', 'AddCircle', 'AddExtrudeBoss', 'AddPlane', 'AddExtrudeCut', 'Save', 'ExportSTL', and 'Close' are in purple; strings are in pink; and variables and numbers are in black. The script defines a part named 'My Part', creates a head with a circular boss, adds a shoulder with a circular boss, and creates a hexagonal recess. The part is then saved to the desktop and exported as an STL file.

```

1  MyPart = Part('My Part')
2  XYPlane = MyPart.GetPlane('XY-Plane')
3  HeadSketch = MyPart.AddSketch('Head', XYPlane)
4  HeadSketch.AddCircle(0, 0, 10, False)
5  BoltHead = MyPart.AddExtrudeBoss('Bolt Head', HeadSketch, 5, False)
6  HeadBottomPlane = MyPart.AddPlane('Head Bottom', XYPlane, 5)
7  ShoulderSketch = MyPart.AddSketch('Shoulder', HeadBottomPlane)
8  ShoulderSketch.AddCircle(0, 0, 5, False)
9  BoltShoulder = MyPart.AddExtrudeBoss('Bolt Shoulder', ShoulderSketch, 20, False)
10 HexSketch = MyPart.AddSketch('Hex', XYPlane)
11 HexSketch.AddPolygon(0, 0, 5, 6, False)
12 HexRecess = MyPart.AddExtrudeCut('Hex Recess', HexSketch, 3, False)
13 MyPart.Save("C:\Users\Andy\Desktop")
14 MyPart.ExportSTL("C:\Users\Andy\Desktop\My Part.stl")
15

```

17. Save the script and then run it by clicking on the Run Script button. The part will be created, saved, exported and closed in one step.



## CHAPTER 3: UNITS

The units used in scripts are separate from the units used in Alibre Design. For example Alibre Design can be configured for inches but a script uses millimeters, or vice versa.

The units used in a script can be set by adding one of the following lines to the start of the script:

```
Units.Current = UnitTypes.Millimeters  
Units.Current = UnitTypes.Centimeters  
Units.Current = UnitTypes.Inches
```

At any point in a script the units used can be changed. All values after the change will be in the new units. For example:

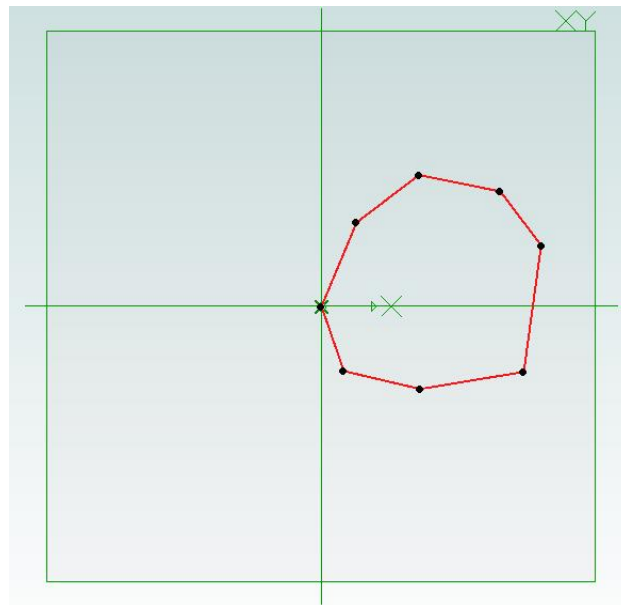
```
Units.Current = UnitTypes.Millimeters  
HeadSketch.AddCircle(0, 0, 10.2, False)  
Units.Current = UnitTypes.Inches  
HeadSketch.AddCircle(3, 2.6, 4.1, False)
```

This draws two circles on a sketch. The first is located at 0, 0 and is 10.2mm in diameter. The second is located at 3", 2.6" and is 4.1" in diameter.

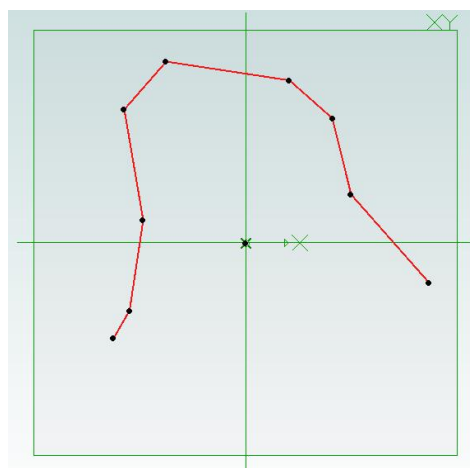
Angles are always given in degrees.

## CHAPTER 4: POLYLINES

A line consists of a start point and an endpoint. Typically sketches consist of a number of lines connected together. Here is an example of a sketch that consists of eight lines.



Alibre Script introduces the concept of polylines. A polyline is a set of lines chained together. For example the above sketch can be represented by a single polyline. Here is another polyline:





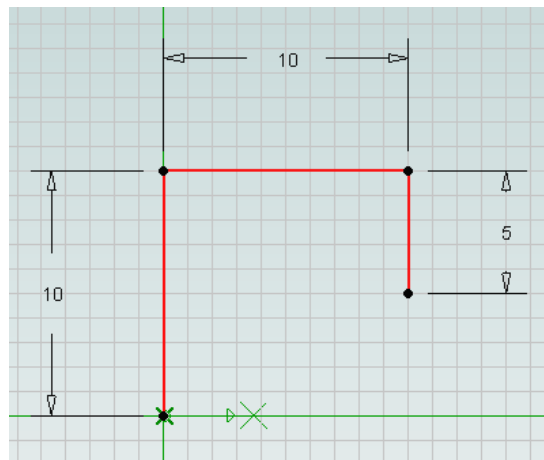
A polyline is defined by the set of points listed from one end to the other. For a polygon the first and last points are at the same location. Here is an example of a polyline defined by points followed by how it looks on the screen:

Point 1 = 0, 0

Point 2 = 0, 10

Point 3 = 10, 10

Point 4 = 10, 5



## CREATION

Creating polyline in Alibre Script is easy. Here is an example that creates the above polyline:

```
MyLine = Polyline()  
MyLine.AddPoint(PolylinePoint(0, 0))  
MyLine.AddPoint(PolylinePoint(0, 10))  
MyLine.AddPoint(PolylinePoint(10, 10))  
MyLine.AddPoint(PolylinePoint(10, 5))
```

Once created the polyline can be added to a sketch:

```
MySketch.AddPolyline(MyLine, False)
```

Polylines have some useful properties that help with creating sketches based on mathematics and algorithms. For example the intersection of two polylines can be found. A polyline can be trimmed at a point. Polylines can be rotated, translated, merged and duplicated. Points can be inserted into anywhere along a polyline.

The key point to remember is that polylines can be manipulated multiple times before committing to a sketch.

## INTERSECTION OF TWO POLYLINES

Here is an example of finding the intersection of two polylines and then trimming the first polyline up to that point:

```
Intersection = Polyline.FindIntersection(MyLine, MyOtherLine)
TrimmedLine = MyLine.SplitAtPoint(Intersection, 0.001)[0]
```

## ROTATION

To rotate a line around location 4, 7 by 15.3 degrees:

```
MyLine.RotateZ(4, 7, 15.3)
```

## JOINING

Joining two polylines together makes a new polyline:

```
LongPolyline = MyLine.Append(MyOtherLine)
```

## TRANSLATION

Translating a line is simple. For example to move it 4.7mm in the X direction and -8.9mm in the Y direction:

```
MyLine.Offset(4.7, -8.9)
```

## COPYING

A line can be duplicated:

```
NewLine = MyLine.Clone()
```

Now NewLine can be manipulated without changing the original MyLine it was based on.

## CHAPTER 5: SKETCH MANIPULATION

Alibre Script contains functionality for manipulating sketches such as copying, scaling, rotation and translation. This opens possibilities for reuse of sketches in novel ways.

The manipulation function is provided by the function CopyFrom. Here is how the function is defined:

```
CopyFrom(Sketch SketchtoCopy)
```

and:

```
CopyFrom(Sketch SketchtoCopy, double Angle, double RotationCenterX, double  
RotationCenterY, double TranslateX, double TranslateY, double ScaleOriginX,  
double ScaleOriginY, double ScaleFactor)
```

Angle is a rotation angle in degrees. Positive values result in clockwise rotation. Rotation can be around any point defined by RotationCenterX and RotationCenterY.

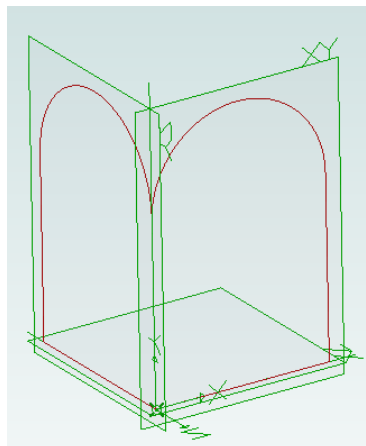
TranslateX and TranslateY allow the sketch to be moved.

A scale factor of 50.0 reduces the size of the sketch by 50% and a scale factor of 150.0 increases it by 50%. Scaling is based on a point defined by ScaleOriginX and ScaleOriginY.

### COPYING

Creating a copy of a sketch for a different plane:

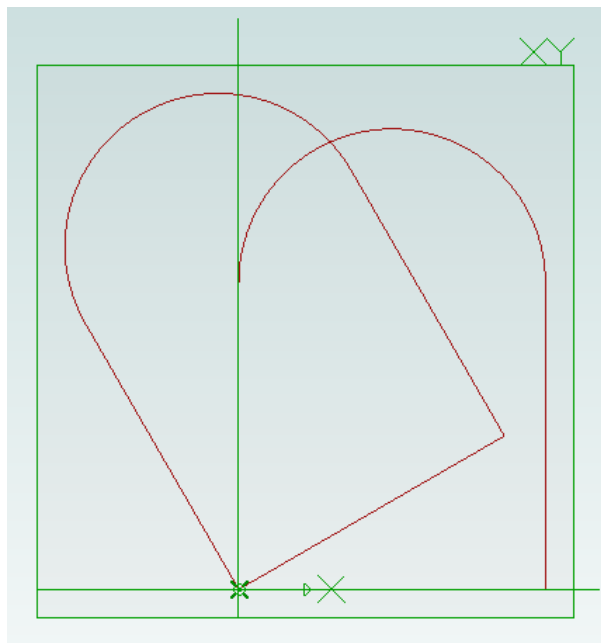
```
NewSketch = MyPart.AddSketch("New", MyPart.GetPlane("YZ-Plane"))  
NewSketch.CopyFrom(OtherSketch)
```



## ROTATION

Creating a copy of a sketch that is rotated 30 degrees clockwise around the origin:

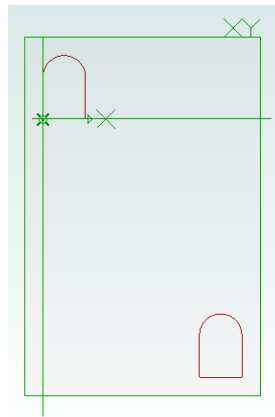
```
NewSketch = MyPart.AddSketch("New", MyPart.GetPlane("XY-Plane"))  
NewSketch.CopyFrom(OtherSketch, 30, 0, 0, 0, 0, 0, 0, 100.0)
```



## TRANSLATION

Creating a copy of a sketch that is offset by 3.7 in the X direction and -6.1 in the Y direction:

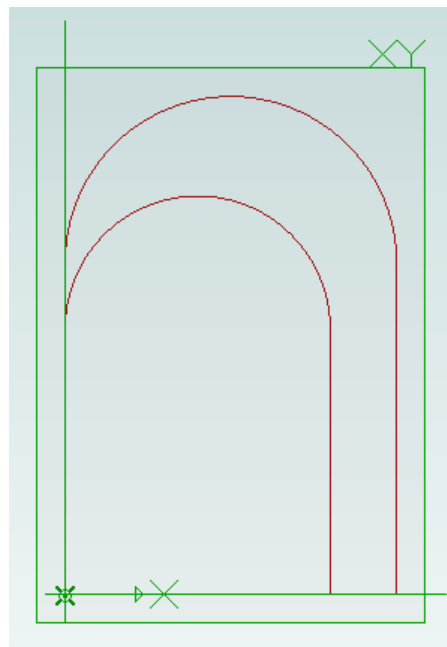
```
NewSketch = MyPart.AddSketch("New", MyPart.GetPlane("XY-Plane"))  
NewSketch.CopyFrom(OtherSketch, 0, 0, 0, 3.7, -6.1, 0, 0, 100.0)
```



## SCALING

Creating a copy of a sketch that is increased in size by 25%, scaling from the origin:

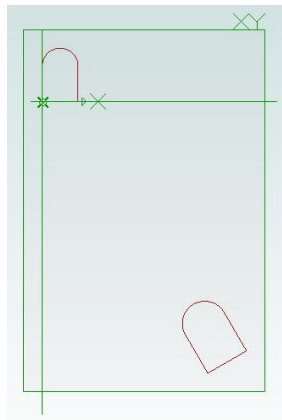
```
NewSketch = MyPart.AddSketch("New", MyPart.GetPlane("XY-Plane"))
NewSketch.CopyFrom(OtherSketch, 0, 0, 0, 0, 0, 0, 0, 0, 125.0)
```



## COMBINATION

Creating a copy of a sketch that is rotated, offset and scaled:

```
NewSketch = MyPart.AddSketch("New", MyPart.GetPlane("XY-Plane"))  
NewSketch.CopyFrom(OtherSketch, 30, 0, 0, 3.7, -6.1, 0, 0, 125.0)
```



## CHAPTER 6: INVOLUTE GEARS

Alibre Script provides basic functionality for creating involute gears, which are gears with involute curves on the edges of the teeth. An involute curve ensures constant force and direction of force throughout the meshing of two teeth, which results in smooth and efficient operation.

### THEORY

Gears are defined by three related parameters:

- Diametral pitch or Module (tooth size) (D)
- Pitch diameter (gear size) (P)
- Number of teeth (N)

Diametral pitch is used in Imperial/English measurement systems (teeth per inch of pitch diameter) and Module is used in metric measurement systems (mm per tooth of pitch diameter). Alibre Script supports diametral pitch however the module value can easily be converted:

$$\text{Diametral Pitch} = 25.4 / \text{Module}$$

All units in a script are configurable – see chapter 3 – with the exception of diametral pitch. This value is always given in teeth per inch.

The relationship between the three parameters is:

$$\text{Number of teeth} = \text{Pitch diameter (in inches)} \times \text{Diametral Pitch}$$

A fourth parameter is also needed, called the pressure angle. This is the angle that the force from one gear is exerted on the other gear. Typical values are 20 degrees and 25 degrees.

In order for two gears (A and B) to mesh properly the following must be true:

- Diametral pitch A = Diametral pitch B
- Pressure angle A = Pressure Angle B
- Distance between gear centers = (Pitch diameter A + Pitch diameter B) / 2

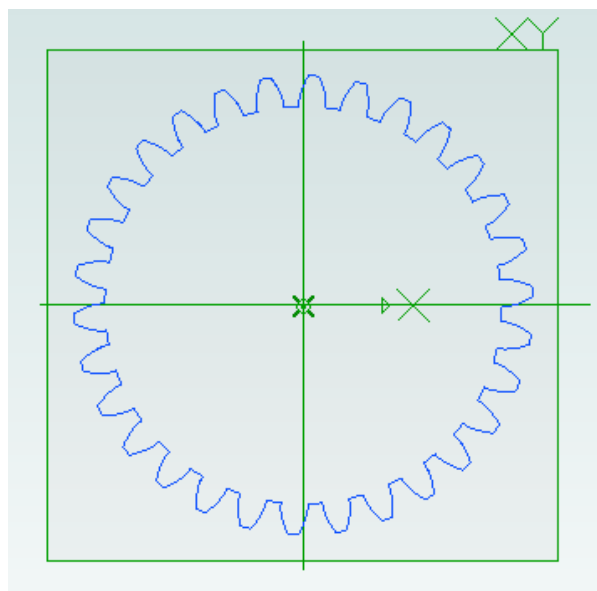
### CREATING GEARS

Alibre Script treats a gear profile as a specialized type of sketch. Creating a gear is therefore like creating a sketch with the profile already generated and added. Here is an example:



```
GearSketch = MyPart.AddGearNP("MyGear", 30, 38, 20, 0, 0,
MyPart.GetPlane("XY-Plane"))
```

This creates a gear profile that is 38mm in diameter and has 30 teeth, with a pressure angle of 20 degrees and is centered on the origin.



We used two of the three parameters to define the gear – number of teeth (N) and pitch diameter (P). The third parameter can be read out from the gear sketch:

```
D = GearSketch.DiametralPitch
print "Diametral Pitch = %f" % D
```

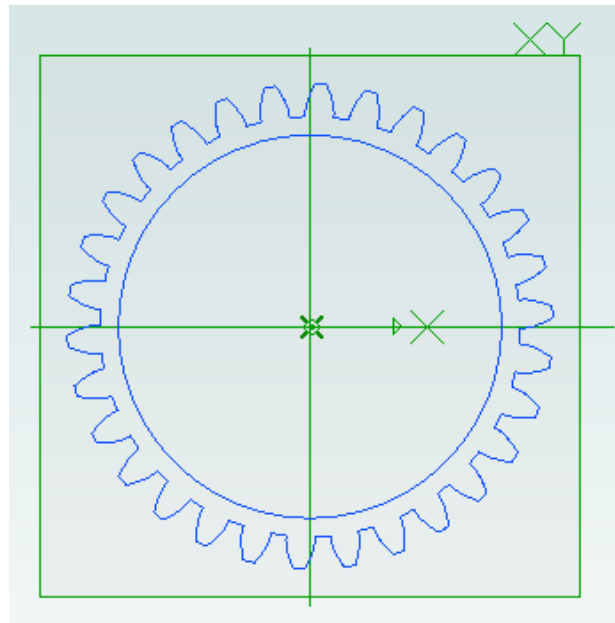
Alibre Script defines a total of four functions for creating gears:

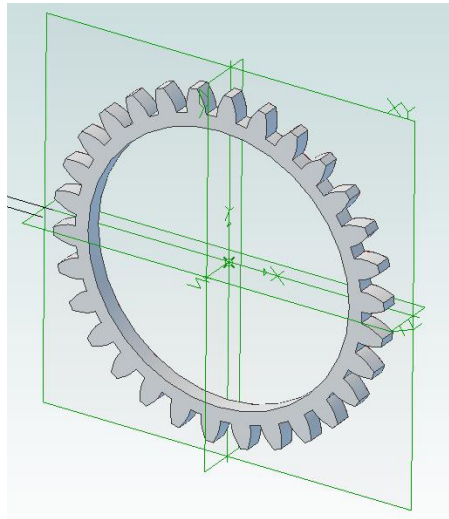
Function	Parameters	Calculates For You
<b>AddGearNP</b>	Number of teeth (N) Pitch diameter (P)	Diametral Pitch (D)
<b>AddGearDP</b>	Diametral pitch (D) Pitch diameter (P)	Number of teeth (N)
<b>AddGearDN</b>	Diametral pitch (D) Number of teeth (N)	Pitch diameter (P)

<b>AddGear</b>	Number of teeth (N)	None
	Pitch diameter (P)	
	Diametral pitch (D)	

Once the gear sketch has been created it can be treated just like any other sketch, for example adding circles:

```
GearSketch.AddCircle(0, 0, 32, False)
```



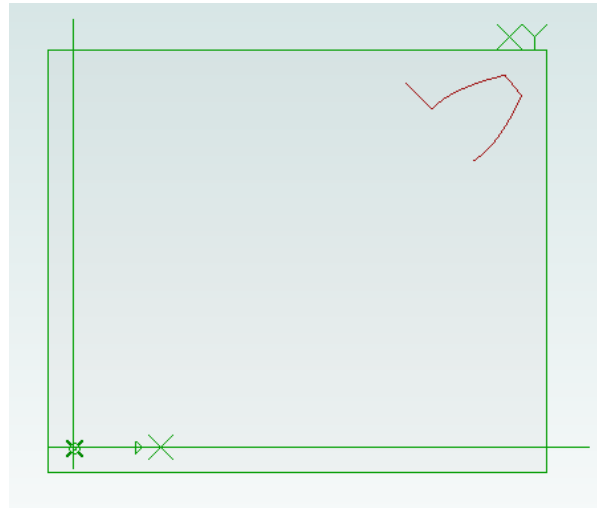


## ADVANCED FUNCTIONALITY

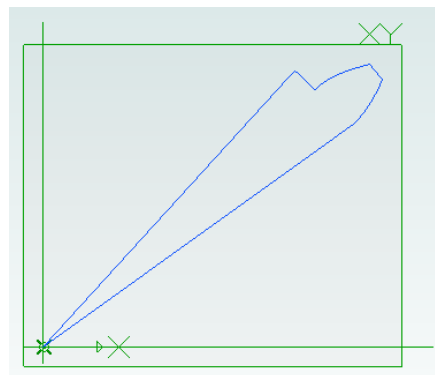
Sometimes it may be necessary to modify the shape of a tooth because of project requirements. For example Alibre Script does not add undercutting. To help with this it is possible to generate a single tooth which can be modified and then used in a circular pattern. Here is how it is done.

We start by using the AddGear function to generate a single tooth with the center of the gear on the origin. In this example the pitch diameter is 25.4mm and the number of teeth is 20, with a pressure angle of 20 degrees:

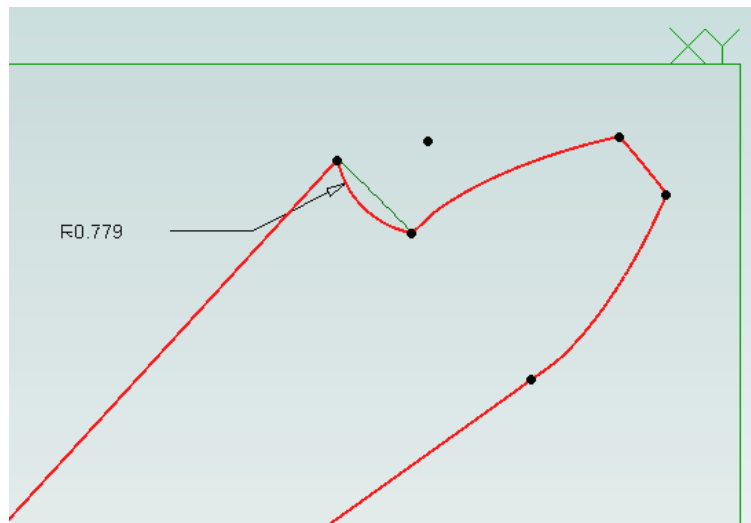
```
GearSketch = MyPart.AddGear("MyGear", 30, 30, 25.4, 20, True, 0, 0,  
MyPart.GetPlane("XY-Plane"))
```



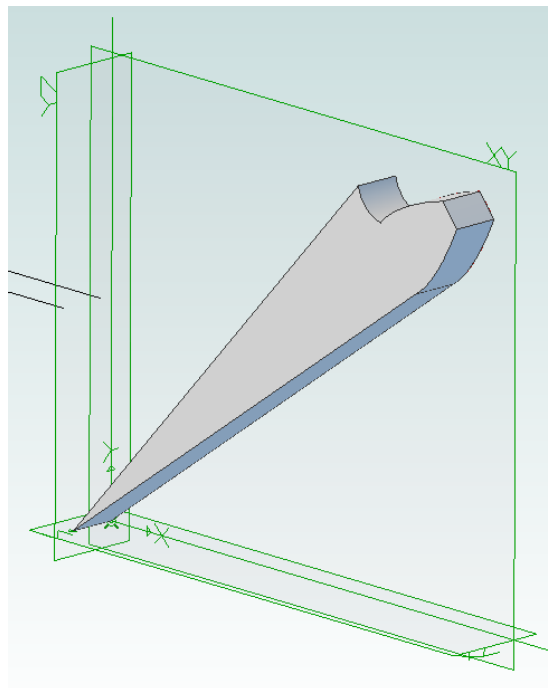
Now switch to using Alibre Design directly. Edit the sketch and connect the end points to the origin with straight lines:



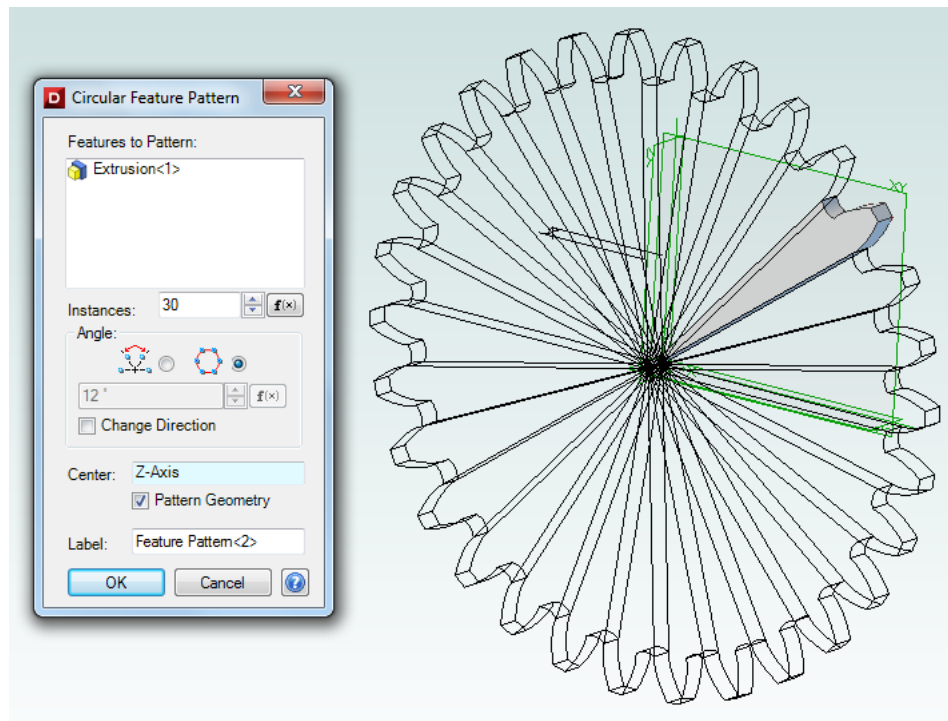
Edit the sketch to add the desired undercut:



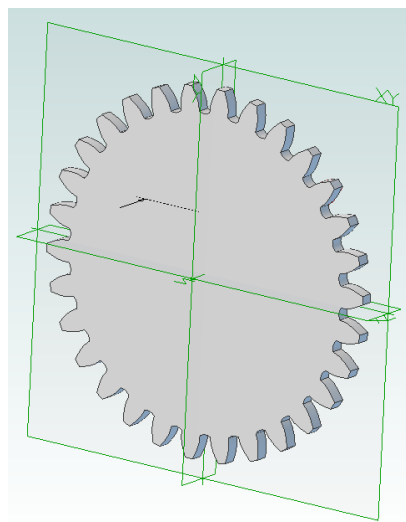
Extrude the sketch to the desired gear thickness:



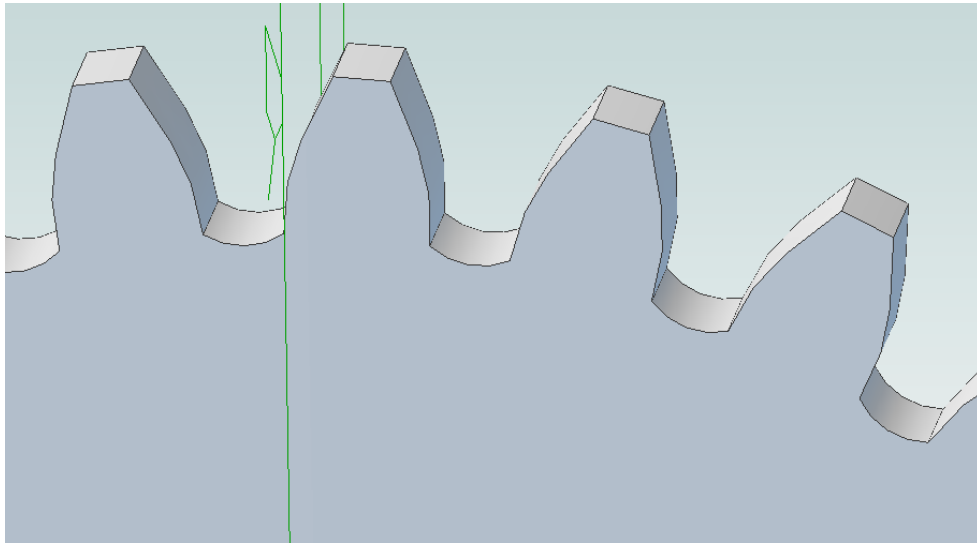
Create a circular feature pattern around the Z axis. We know that there are 30 teeth in this gear so we need to create 30 copies.



The completed gear:



A close up of the customized teeth:



## CHAPTER 7: MODIFYING EXISTING PARTS & SKETCHES

Sometimes it can be useful to use a script to modify a part or sketch that has been already created. For example a script could be written to perform a repetitive task that is not available in the Alibre Design user interface.

To obtain access to an existing part open the part in Alibre Design and then:

```
MyPart = Part("New Part (1)", False)
```

Insert the name of the part as shown in the design explorer. The second parameter is set to False, which tells Alibre Script to use a part that has already been created or opened.

The part can now be accessed as normal by the rest of the script.

To access a sketch on the part:

```
MySketch = Part.GetSketch("Sketch<1>")
```

Insert the name of the sketch as shown in the design explorer. The sketch can now be modified by the rest of the script.

The lines, circles, arcs, etc. defined on a sketch are called “figures”. The figures are available as a list in Python. For example:

```
print len(MySketch.Figures)
```

will print out the total number of figures on the sketch. We can view the details of a specific figure:

```
print MySketch.Figures[0]
```

```
Circle centered at 7.5,-35 with radius 1.75
```

We can get access to that figure and use it in other sketches and other parts:

```
Fig = MySketch.Figures[0]
NewPart = Part("New")
NewSketch = NewPart.AddSketch("Sketch", NewPart.GetPlane("XY-Plane"))
NewSketch.AddFigure(Fig)
```

Using this technique sketches can be created, accessed and reused across multiple parts, perhaps even with slight changes for each new part it is used in:

```
MyCircle = MySketch.Figures[0]
MyCircle.Center = [4, -3]
```



```
NewSketch.AddCircle(MyCircle)
MyBspline = MySketch.Figures[3]
MyBspline.IsReference = True
NewSketch.AddBspline(MyBspline)
```

Note that currently ellipses and elliptical arcs are not supported due to limitations in Alibre Design.

## CHAPTER 8: HINTS & TIPS

1. It is possible to create and edit multiple parts at once. For example:

```
Frame = Part("Frame")
Beam = Part("Beam")
BaseSketch = Frame.AddSketch("Base", Frame.GetPlane("XY-Plane"))
ProfileSketch = Beam.AddSketch("Profile", Beam.GetPlane("XY-Plane"))
```

2. It is not necessary to save a script before running it. This makes it easier to rapidly edit and test scripts.
3. A temporary sketch can be created and used as a template for sketches on other parts, then discarded. For example:

```
# create temporary sketch
TempPart = Part("Temp")
TempSketch = TempPart.AddSketch("Temp", TempPart.GetPlane("XY-Plane"))
TempSketch.AddCircle(...)
TempSketch.AddLine(...)

# copy to sketch on part A
SketchOnPartA.CopyFrom(TempSketch)

# copy to sketch on part B increasing size by 25%
SketchOnPartB.CopyFrom(TempSketch, 0, 0, 0, 0, 0, 0, 0, 125.0)

# copy one figure to sketch on part C modifying the figure first
Figure = TempSketch.Figures[1]
Figure.IsReference = True
SketchOnPartC.AddFigure(Figure)

# delete temporary sketch
TempPart.Close()
```

## CHAPTER 9: RUNNING SCRIPTS – ADVANCED

If there is only one script open then it can be run from the console by typing the run command:

```
>>>run
```

If there is more than one script currently open then a specific script can be run by specifying the script name. if the name has a space in it then use quote marks around the name:

```
>>>run "New Script (1)"
```

```
>>>run "New Script (2)"
```

Arguments can be passed to a script. Here is an example script that uses the special built-in variable 'Arguments' which is a list of arguments passed to the script:

```
print Arguments
```

Example usage of this script:

```
>>>run  
[]  
>>>run foo  
['foo']  
>>>run foo bar  
['foo', 'bar']  
>>>run "foo bar" baz  
['foo bar', 'baz']  
>>>run "New Script (1)" foo bar  
['foo', 'bar']
```

## CHAPTER 10: BUILT-IN FUNCTIONALITY

Some useful functionality is built directly into Alibre Script and is not part of the API. This functionality is detailed in this chapter.

### DETAILS OF THE CURRENT SCRIPT

The file name and the folder where the current script is stored can be obtained using the following variables. If the script has not been saved then these variables will be empty.

```
print ScriptFileName  
print ScriptFolder
```

### CURRENT PART OR ASSEMBLY

A script always runs inside a part workspace or an assembly workspace. The current part or assembly can be obtained by calling the following built-in functions.

```
MyPart = CurrentPart()
```

and:

```
MyAssy = CurrentAssembly()
```

### CURRENTLY OPENED PARTS AND ASSEMBLIES

It is possible to obtain a list of all currently opened parts and assemblies:

```
Parts = CurrentParts()  
print Parts[0]  
Assys = CurrentAssemblies()  
Print Assys[0]
```

### THE ALIBRE SCRIPT VERSION

If you write a script that requires a minimum version of Alibre Script, e.g. because of a new feature that was introduced, then the current version (build number) can be read out via a built-in variable:

```
print AlibreScriptVersion
```

## GRAPHICAL USER INTERFACE

Alibre Script can create dialogs, show messages to the user, etc. All graphical functionality can be found on the 'Windows' class. This is accessed as follows:

```
Win = Windows()
```

Only create one instance of the Windows class in a script.

## GETTING USER INPUT

Alibre Script can request input from the user on the command line.

The Read() function waits for the user to enter a line of text and then returns the text without the newline:

```
UserText = Read()
```

The ReadKey() function waits for the user to press a key and then returns the ASCII code for the pressed key:

```
PressedKey = ReadKey()
```

## UTILITY LIBRARIES

Some utility libraries are provided as built-in classes. Currently implemented are:

- TwoD – provides functions for two-dimensional calculations
- ThreeD – provides functions for three-dimensional calculations

Examples:

```
My2D = TwoD()
```

```
My3D = ThreeD()
```

See the reference manual for details of functions provided by these libraries.

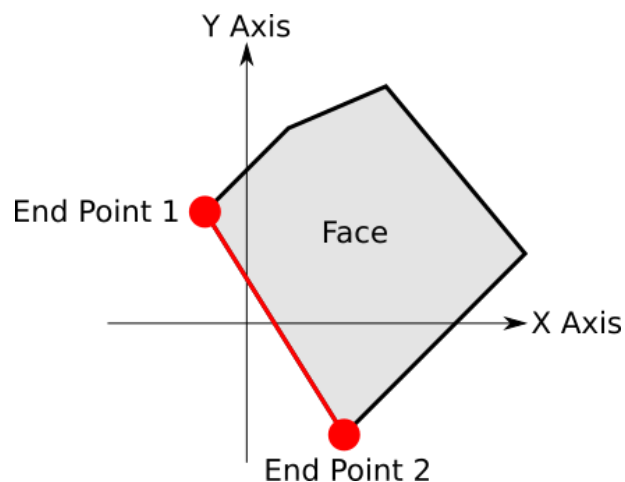
## CHAPTER 11: 2D SKETCH MAPPING

### FACE MAPPING

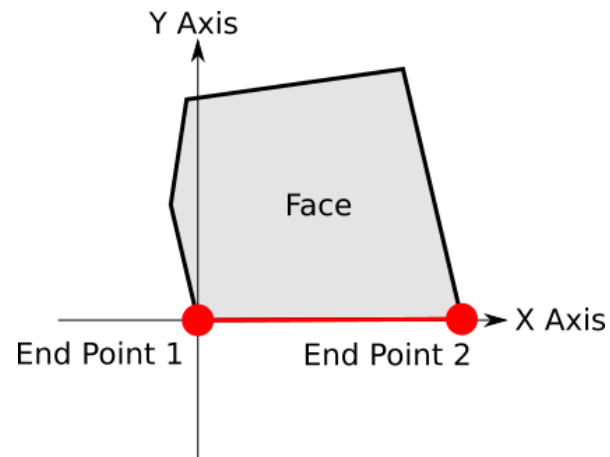
When creating a 2D sketch on a face using a script there are complications if the part was not created by the same script, for example created manually.

Consider the following 2D sketch view for a face. Because this part was created directly in an Assembly session it does not have any sides that are aligned with the X or Y axis. This makes creating sketches on the face complicated.

Sketch face mapping simplifies the problem for the script writing. To create a “mapping” two points on a face edge are required, highlighted in red in this example.

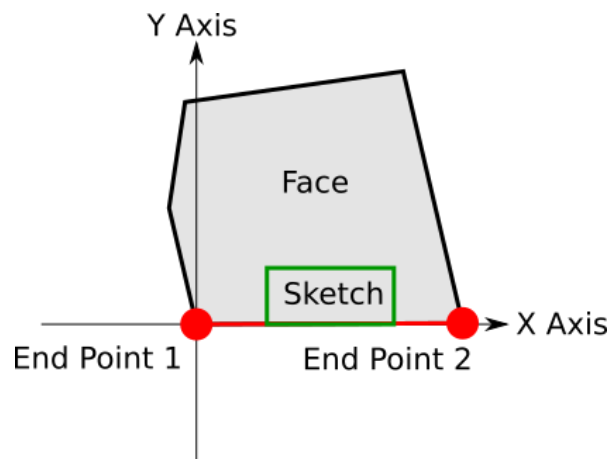


When mapping has been turned on the sketch coordinate system looks like this:

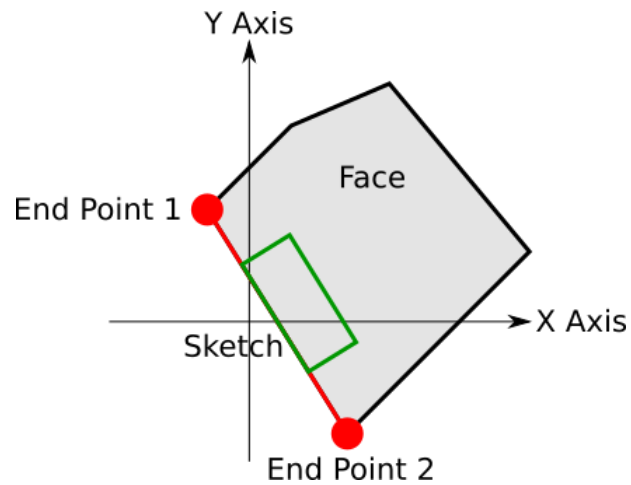


The chosen edge is on the X axis and the points on the edge are at the origin and somewhere along the positive X axis. The face is “sitting” on the X axis.

Now sketch figures can be drawn, such as this rectangle, in relation to the aligned edge. Drawing above the X axis ensures the sketch figures are on the face and drawing below the X axis ensures the sketch figures are alongside the face.

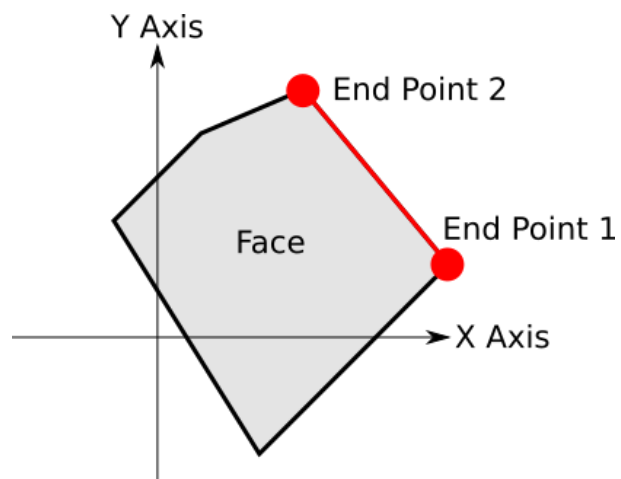


When mapping is then turned off the final sketch looks as follows.



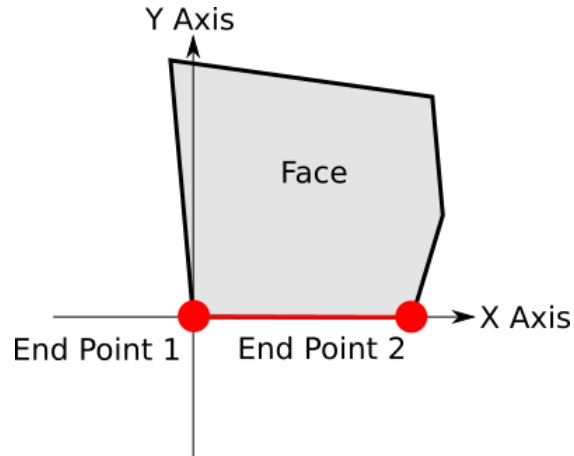
The relationship between the sketch figures and the aligned edge are maintained. Calculating the angles for this rectangle would be complicated to do without mapping the face to the X axis.

Here is another example, in this case a different edge has been chosen:



And this is the resultant mapping:





Any edge and any points on the edge can be chosen for the mapping. Here is what it looks like in a script.

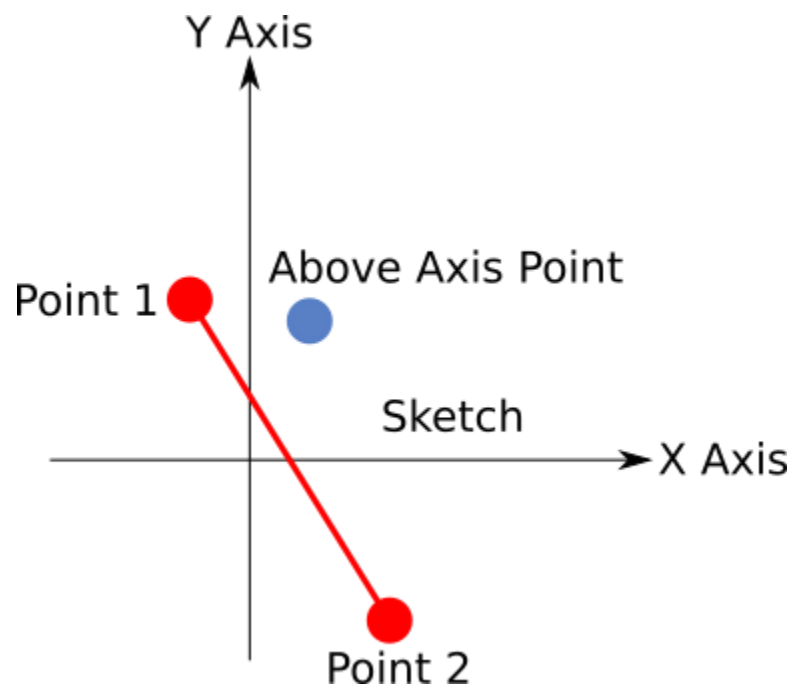
```
VertexA = MyPart.GetVertex("Vertex<4>")
VertexB = MyPart.GetVertex("Vertex<7>")
MySketch = MyPart.AddSketch("MySketch", MyPart.GetFace("Face<2>"))
MySketch.StartFaceMapping(VertexA, VertexB)
MySketch.AddRectangle(10, 0, 20, 5, False)
MySketch.StopFaceMapping()
```

## GENERIC MAPPING

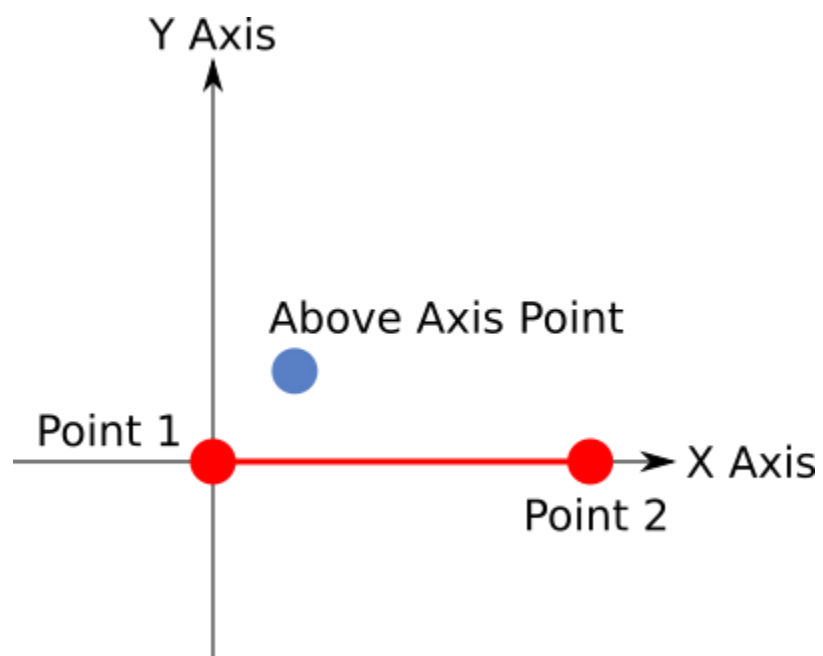
It is also possible in Alibre Design to create 2D sketches on planes instead of faces, however mapping of the coordinate system can still be useful to avoid the need for additional calculations in scripts.

With face mapping the transformation of the coordinate system is determined automatically based on the face, positioning it at (0, 0) and “sitting” on the X axis. For a plane this isn’t possible so some additional information has to be given.

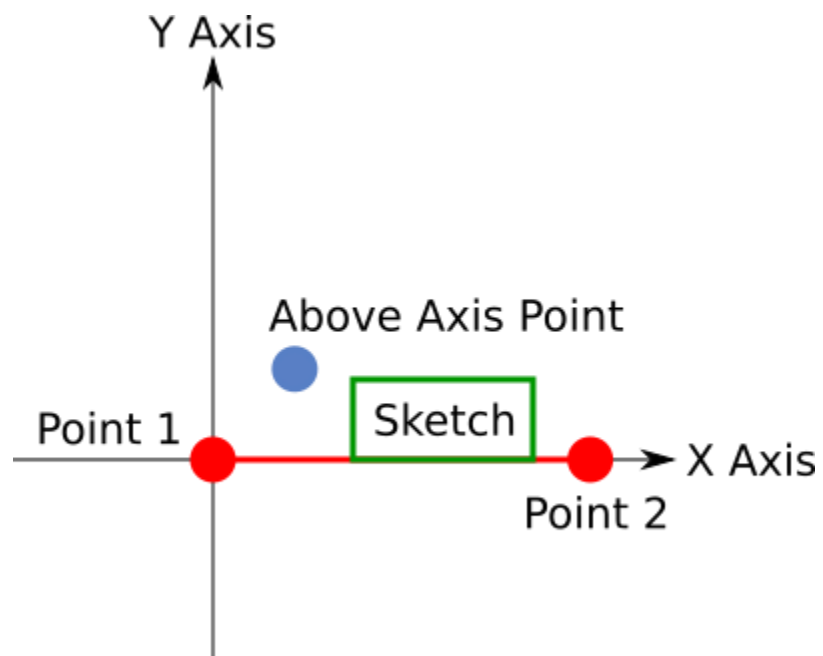
Here is a diagram for a sketch on a plane. Two arbitrary points are chosen along with a third point that indicates to Alibre Script how the mapping should be performed:



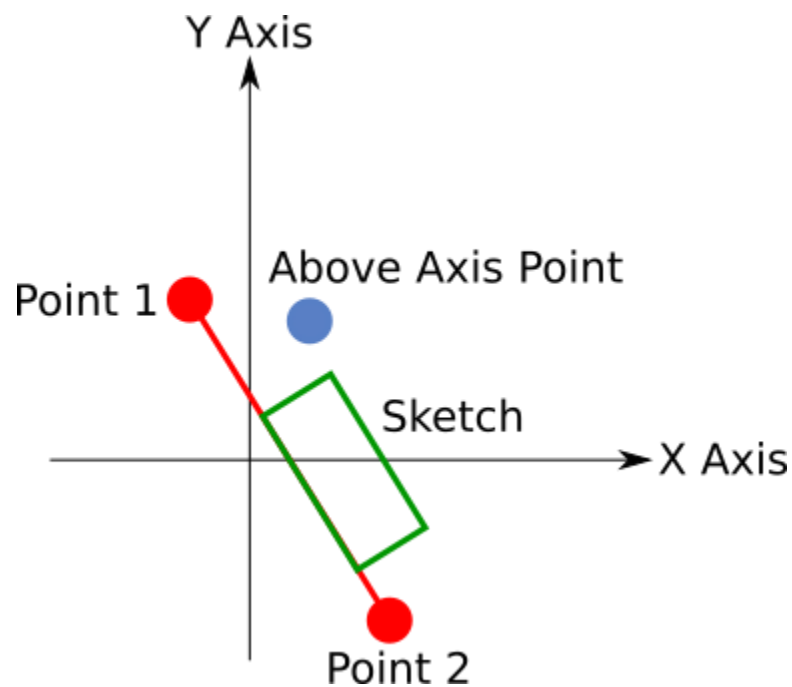
When mapping is enabled the coordinate system looks like this:



Point 1 is placed at the origin and rotation is such that the third point is always above the X axis. Now the sketch can be easily drawn:



When mapping is stopped the end result is as follows:



Here is what this looks like in a script:

```
Point1 = [5.0, 4.2, -6.3]
Point2 = [0.55, 0.0, 12.1]
PointAboveAxis = [7.2, 6.01, -3.2]
MySketch = MyPart.AddSketch("MySketch", MyPart.GetFace("Face<2>"))
MySketch.StartMapping(Point1, Point2, PointAboveAxis)
MySketch.AddRectangle(10, 0, 20, 5, False)
MySketch.StopMapping()
```

## CHAPTER 12: EDITOR

The editor has some features that make creating scripts easier.

### UNDO AND REDO

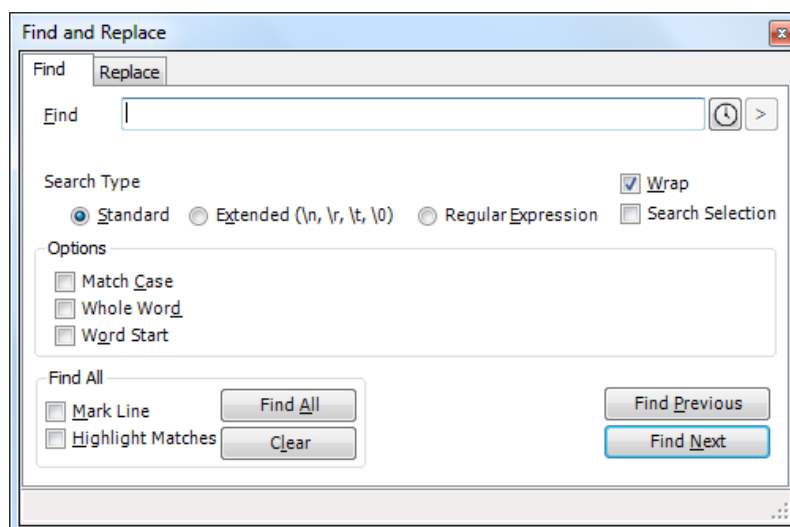
Press Ctrl+Z to undo the most recent change. Press Ctrl+Z to undo the change before that. Press Ctrl-Y to redo a change that has been undone.

### SELECT, COPY AND PASTE

Press Ctrl+A to select all of the text (or click and drag the cursor). Press Ctrl+C to copy the selected text to the Windows clipboard and press Ctrl+V to paste text from the Windows clipboard. Press Delete to delete the selected text.

### FIND AND REPLACE

Press Ctrl+F (find) or Ctrl+H (replace) to open the find and replace dialog.



Enter text to find into the Find box and click on Find Next to search from the current cursor location.

Click on the Replace tab to perform a find and replace. Press F3 to jump to the next match or Shift+F3 to jump to the previous match.

Regular expressions allow for complex find and replace operations. Details of the supported syntax:

- [1]      `char`      matches itself, unless it is a special character (metachar): `.` `\` `[ ]` `*` `+` `?` `^` `$` and `( )` if posix option.
  
- [2]      `.`           matches any character.
  
- [3]      `\`           matches the character following it, except:
  - `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v` match the corresponding C escape char, respectively BEL, BS, FF, LF, CR, TAB and VT; Note that `\r` and `\n` are never matched because Scintilla regex searches are made line per line (stripped of end-of-line chars).
  - if not in posix mode, when followed by a left or right round bracket (see [8]);
  - when followed by a digit 1 to 9 (see [9]);
  - when followed by a left or right angle bracket (see [10]);
  - when followed by `d`, `D`, `s`, `S`, `w` or `W` (see [11]);
  - when followed by `x` and two hexa digits (see [12]).
 Backslash is used as an escape character for all other meta-characters, and itself.
  
- [4]      `[set]`       matches one of the characters in the set. If the first character in the set is `^`, it matches the characters NOT in the set, i.e. complements the set. A shorthand S-E (start dash end) is used to specify a set of characters S up to E, inclusive. S and E must be characters, otherwise the dash is taken literally (eg. in expression `[\d-a]`). The special characters `]` and `-` have no special meaning if they appear as the first chars in the set. To include both, put `-` first: `[-]A-Z` (or just backslash them).  
examples:
  - match:
  - `[-]||`      matches these 3 chars,
  - `[|-|]`      matches from `|` to `|` chars
  - `[a-z]`      any lowercase alpha

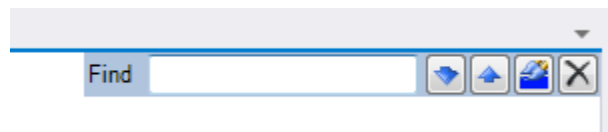
- [^ -] any char except - and ]
- [^A-Z] any char except uppercase alpha
- [a-zA-Z] any alpha
- [5] \* any regular expression form [1] to [4]  
(except [8], [9] and [10] forms of [3]),  
followed by closure char (\*)  
matches zero or more matches of that form.
- [6] + same as [5], except it matches one or more.
- [5-6] Both [5] and [6] are greedy (they match as much as possible).  
Unless they are followed by the 'lazy' quantifier (?)  
In which case both [5] and [6] try to match as little as possible
- [7] ? same as [5] except it matches zero or one.
- [8] a regular expression in the form [1] to [13], enclosed  
as \ (form\ ) (or (form) with posix flag) matches what  
form matches. The enclosure creates a set of tags,  
used for [9] and for pattern substitution.  
The tagged forms are numbered starting from 1.
- [9] a \ followed by a digit 1 to 9 matches whatever a  
previously tagged regular expression ([8]) matched.
- [10] \< a regular expression starting with a \< construct  
\> and/or ending with a \> construct, restricts the  
pattern matching to the beginning of a word, and/or  
the end of a word. A word is defined to be a character  
string beginning and/or ending with the characters  
A-Z a-z 0-9 and \_. Scintilla extends this definition  
by user setting. The word must also be preceded and/or  
followed by any character outside those mentioned.
- [11] \1 a backslash followed by d, D, s, S, w or W,  
becomes a character class (both inside and  
outside sets []).  
d: decimal digits  
D: any char except decimal digits  
s: whitespace (space, \t \n \r \f \v)  
S: any char except whitespace (see above)  
w: alphanumeric & underscore (changed by user setting)  
W: any char except alphanumeric & underscore (see above)
- [12] \xHH a backslash followed by x and two hexa digits,

becomes the character whose Ascii code is equal to these digits. If not followed by two digits, it is 'x' char itself.

- [13] a composite regular expression `xy` where `x` and `y` are in the form `[1]` to `[12]` matches the longest match of `x` followed by a match for `y`.
- [14] `^` a regular expression starting with a `^` character  
`$` and/or ending with a `$` character, restricts the pattern matching to the beginning of the line, or the end of line. [anchors] Elsewhere in the pattern, `^` and `$` are treated as ordinary characters.

## QUICK FIND

Press **Ctrl+I** to open the quick find panel at the top right of the editor:

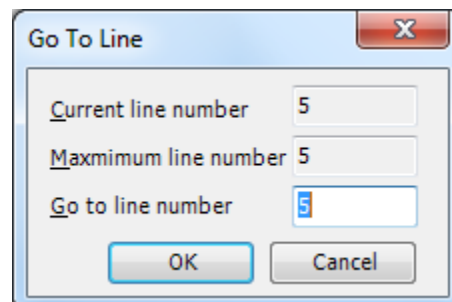


Enter text to find and click on the up and down arrows to search backwards and forwards.

Click anywhere in the editor to close the quick find panel.

## GO TO

Press **Ctrl+G** to open the GoTo dialog window:





Enter the line number to jump to in the editor.

## RUN THE SCRIPT

Press Ctrl+R to run the current script.

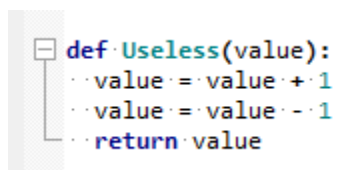
## SAVE

Press Ctrl+S to save the current script.

## CODE FOLDING

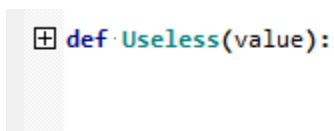
Code folding is a way of hiding pieces of code to reduce clutter, for example the contents of a large function that you are not currently working on.

When writing a function the code folding feature will automatically appear on the left.

A screenshot of a code editor showing a function definition. On the left, there is a light gray vertical bar containing a small square icon with a minus sign inside. To the right of this icon, the function code is displayed: 

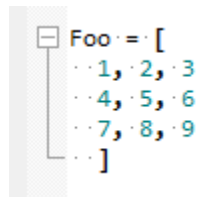
```
def Useless(value):  
    value = value + 1  
    value = value - 1  
    return value
```

Clicking on the box with the '-' will collapse the entire function, hiding it away:

A screenshot of the same code editor showing the function collapsed. The light gray vertical bar now contains a small square icon with a plus sign inside. To the right of this icon, only the first line of the function is visible: 

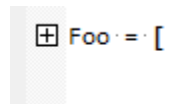
```
def Useless(value):
```

Clicking on the box with the '+' will expand the function again. This feature is not limited to functions, for example a multi-line list:

A code block with a light gray background. On the left side, there is a small square icon with a minus sign inside. A vertical line extends from the bottom of this icon, and a horizontal line connects it to the opening square bracket of the first line of code. The code is as follows:

```
Foo = [
  1, 2, 3
  4, 5, 6
  7, 8, 9
]
```

Collapsed:

A code block with a light gray background. On the left side, there is a small square icon with a plus sign inside. The code is collapsed to a single line:

```
⊕ Foo = [
```