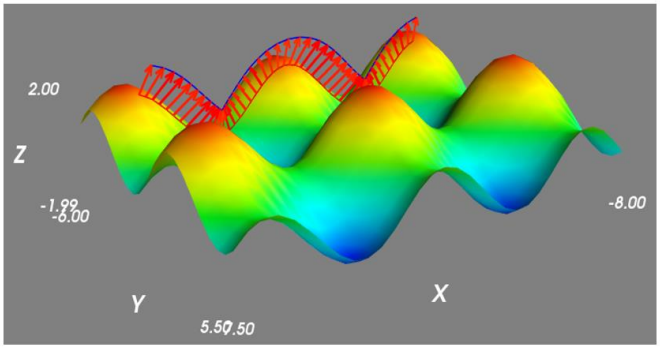
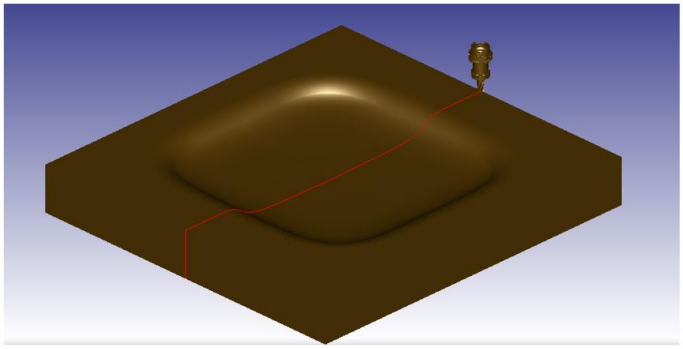
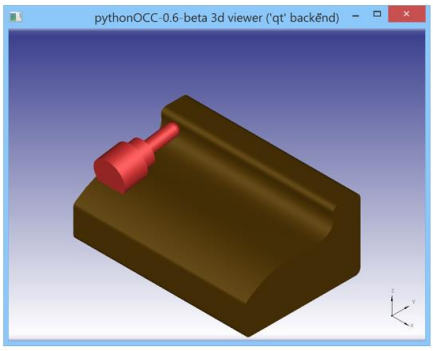
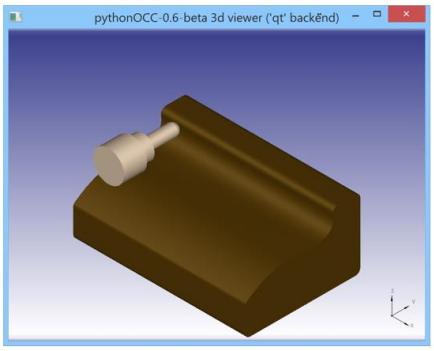
京都大学
KYOTO UNIVERSITY










```
(CNC HEADER)
(COMMAND 1)
G1 X-5.000 Y0.000 Z1.000
G1 X-3.889 Y0.000 Z1.000
G1 X-2.778 Y0.000 Z1.000
G1 X-1.667 Y0.000 Z1.000
G1 X-0.556 Y0.000 Z1.000
G1 X0.556 Y0.000 Z1.000
G1 X1.667 Y0.000 Z1.000
G1 X2.778 Y0.000 Z1.000
G1 X3.889 Y0.000 Z1.000
G1 X5.000 Y0.000 Z1.000
(CNC FOOTER)
(COMMAND 1)
```

CAD/CAM Programming in Python (v2.0)

Dr. Beaucamp Anthony, Senior Lecturer, Kyoto University

1 Table of Contents

Contents

1 Table of Contents 1
2 Introduction 2
3 Installation Guide 3
4 Spyder IDE 5
5 Tutorial 1 – Basic Data Handling in Python 6
6 Tutorial 2 – Curve Plotting and Interpolation 8
7 Tutorial 3 – Basic CAD handling 10
8 Tutorial 4 – Sectioning and Offsetting 12
9 Tutorial 5 - Collision Detection 14
10 Tutorial 6 – Tool Animation 16
11 Tutorial 7 – Collision Avoidance 17
12 Appendix 1: Designing a Bottle 19
12.1 Model Specifications 19
12.2 Import required Modules 19
12.3 Building the Profile 19
12.3.1 Defining Support Points 19
12.3.2 Profile: Defining the Geometry 20
12.3.3 Profile: Defining the Topology 21
12.3.4 Profile: Completing the Profile 22
12.4 Building the Body 24
12.4.1 Prism the Profile 24
12.4.2 Applying Fillets 25
12.4.3 Cut the inside the Bottle 26
12.4.4 Adding the Neck 27
12.4.5 Opening the Neck 27
12.5 Display and Export 28

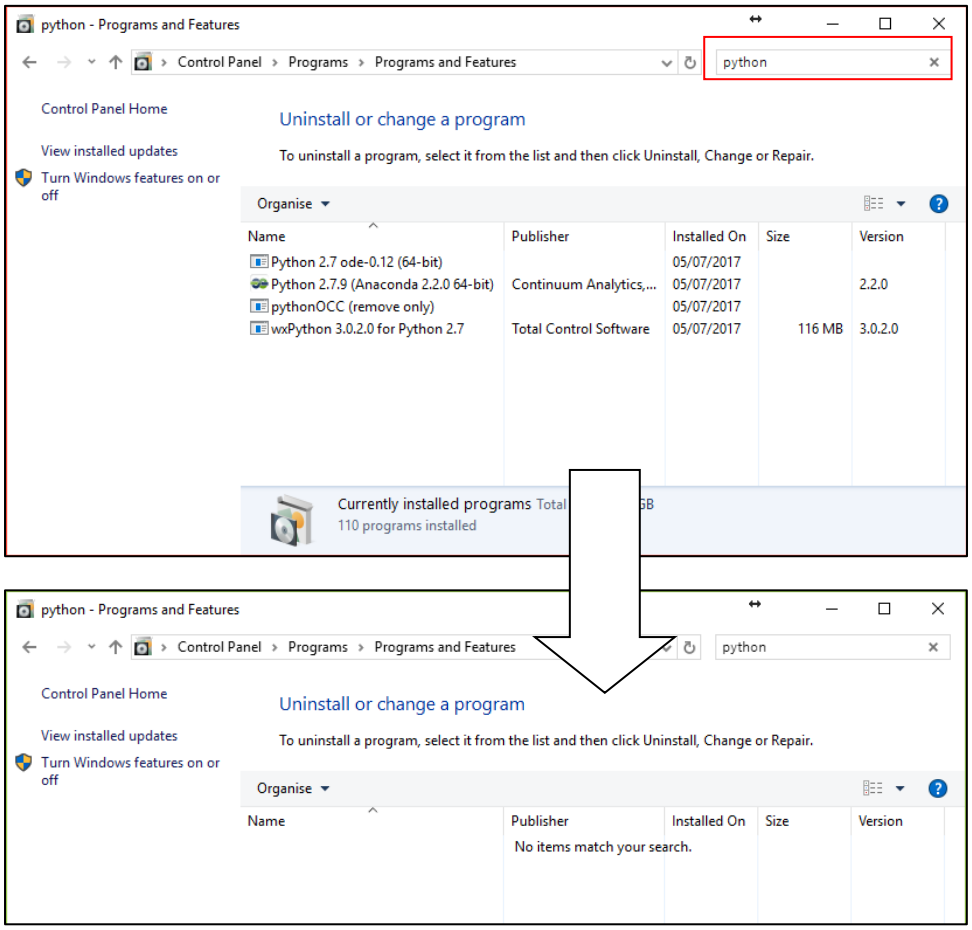
2 Introduction

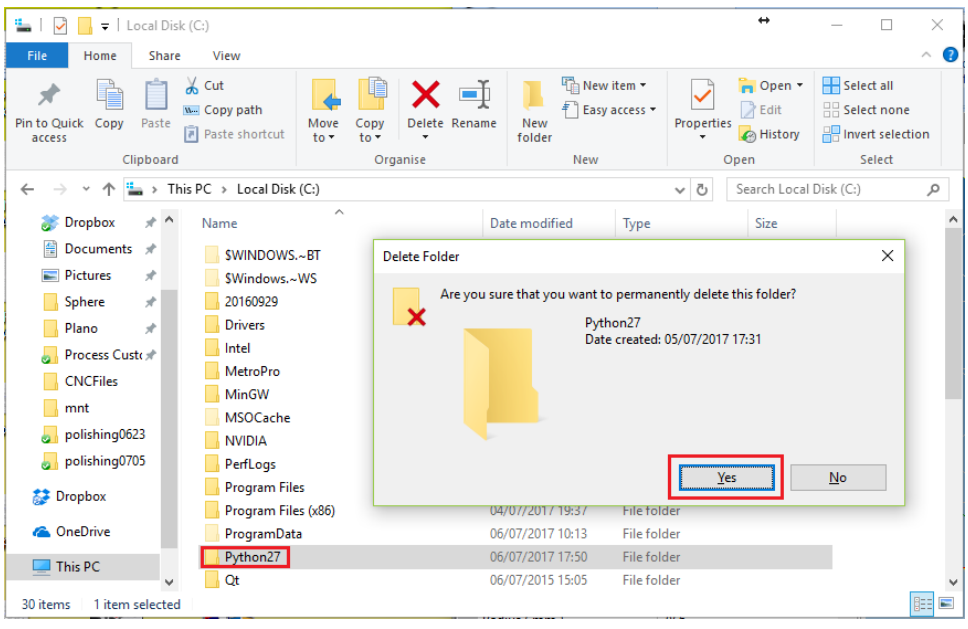
- Python is a high level programming language (i.e.: easy to understand and use).
- First released in 1991, it has grown into a huge collection of modules over the years.
- Python is used for:
    - Networking
    - Web development
    - Database programming
    - Graphical/Video animation
    - Game development
    - …
    - **Scientific programming!** (i.e: Matlab style programming)
    - **CAD/CAM programming!** (i.e: CATIA/Esprit style programming)
- Even better, Python is “opensource”: free to download and use!

Tutorials have been provided to understand how to use the various CAD/CAM functionalities in Python.

3 Installation Guide

STEP 1: A windows PC (XP, Vista, 7, 8, or 10) is required. **FIRST** remove any existing installation of Python: go to “Programs and Features” then search for “python”. Uninstall all python programs until the list is empty. Also **DELETE** the folder “**C:\Python27**”





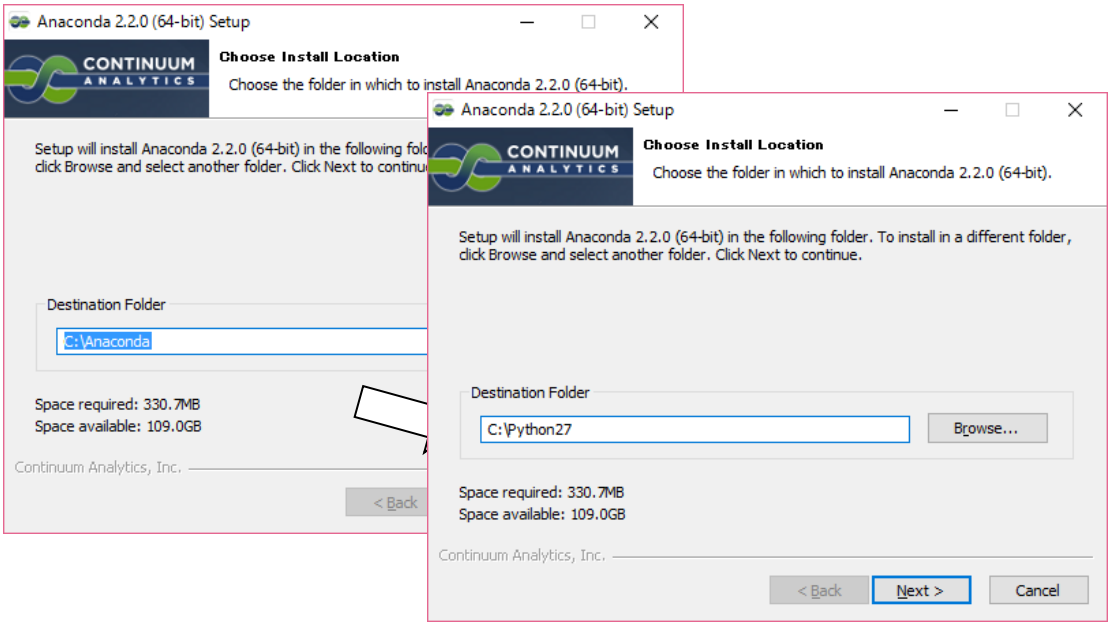
STEP 2:

Install the files included in the archive **Python-CAD-CAM (x86).zip**, in order (1) to (10):

(1) Anaconda2-2.5.0-Windows-x86.exe
(2) scipy-0.16.1-win32-superpack-python2.7.exe
(3) pythonOCC-0.16.2-win32-py27.exe
(4) pythonOCC-extras.exe
(5) pyode-0.12.win32-py27.exe
(6) wxPython3.0-3.0.2.0-win32-py27.exe
(7) WMI-1.4.9.win32-py27.exe
(8) Zeeko-CADCAM-2017-07-05-py27.exe
(9) Spyder IDE Layout.exe
(10) CAD-Examples.exe

IMPORTANT NOTE: When installing **(1) Anaconda2-2.5.0-Windows-x86.exe**:

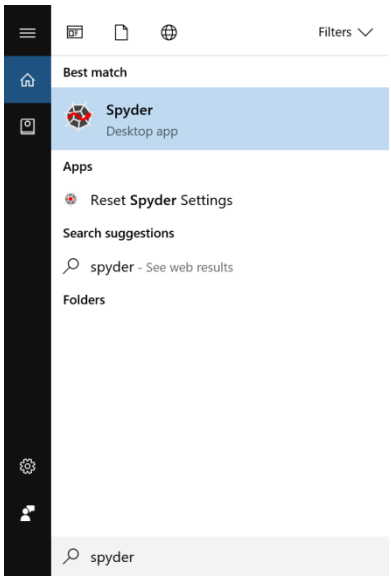
- Set the **Destination Folder** to "**C:\Python27**"
- In "Advanced Options" keep all boxes checked and click install to continue.



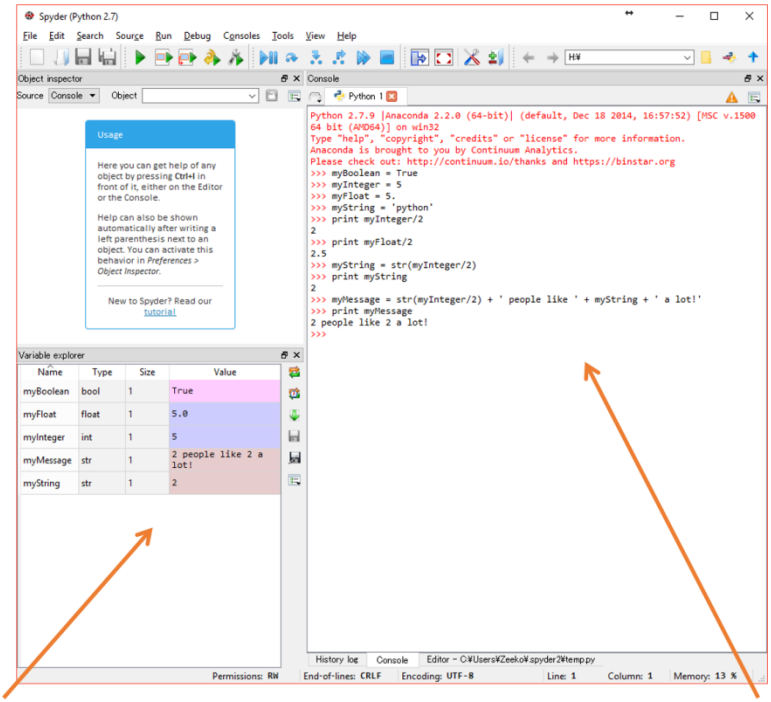
IF YOU HAVE TROUBLES WITH INSTALLING, EMAIL: beaucamp@me.kyoto-u.ac.jp

4 Spyder IDE

After installing Python as shown in Chapter 3, please launch the application called Spyder (you can find it using Windows search):



Spyder is an integrated development editor (IDE) for python. Please organize the layout as shown here, you should enter all the code shown in the following tutorials in the **Interactive Console**.



Variable currently in memory

Interactive Console (to test your code)

5 Tutorial 1 – Basic Data Handling in Python

```
########################################
# There are 4 basic data types in Python:
myBoolean = True
myInteger = 5
myFloat = 5.
myString = 'python'

# Do not confuse integers and floats, they behave differently!!!
print myInteger/2
print myFloat/2

######################
# Working with Strings:
myString = str(myInteger/2)
print myString

# You can easily concatenate strings from various data types
myMessage = str(myInteger/2) + ' people like ' + myString + ' a lot!'
print myMessage

########################################
# Working with Lists:
#  Lists are dynamically allocated arrays
#  They can store mixed data types
myList = [True, 5, 5., 'python']
print myList

# Indexes in Python start at 0 (unlike Matlab which starts at 1!!!)
print myList[0]
print myList[1]

# Use negative indexes to access elements from the end of the list
print myList[-1]
print myList[-2]

# You can fetch a subset from the list
subList = myList[1:3]
print subList

# You can append data only at the end of a list...
myList.append('extra')
print myList

# ...but pop elements at any index
element = myList.pop(-2)
print myList

# You can create a list of lists, and so on...
doubleList = [ [0], [0,1], [0,1,2] ]
print doubleList[2][2]

# Create numerical lists with the function range(start,stop,step)
myList = range(0,8,2)
print myList
```

```
#######################################################################
# Iterating through list (add "tab" before print, to indent the code)
for value in myList:
    print value
```

TAB

```
########################################
# Working with Dictionaries:
#  Dictionaries associate "keys" with "values"
axisLimits = dict()
axisLimits['X'] = [-150.,150.]
axisLimits['Y'] = [-100.,100.]
axisLimits['Z'] = [-50.,75.]
print axisLimits

# Accessing values using keys:
print axisLimits['X']

# Iterating through dictionaries:
for key in axisLimits:
    limits = axisLimits[key]
    print key + '-axis range is ' + str(limits)
```

MEMO: Python Lists

- `myList = []; myList = [0, 1, 2]` Create a new Python List
- `value = myList[index]` Access value at index (starts from 0)
- `myList.append(3)` Add new value at the end of a List
- `value = myList.pop(index)` Remove value at the specified index
- `myRange = range(start, stop, step)` Create numerical list

MEMO: Python Dictionaries

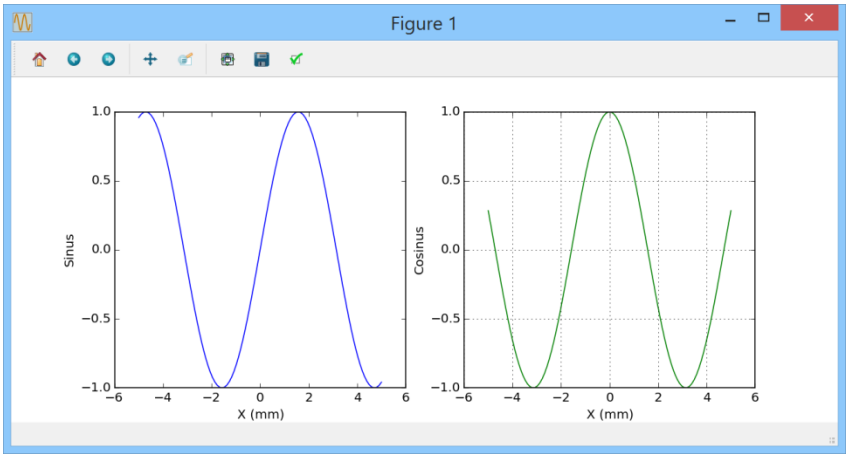
- `myDict = {}; myDict = {'X':24, 'JOE':90}` Create a Dictionary
- `myDict[newKey] = newValue` Add new key/value to Dictionary
- `value = myDict[key]` Access value linked to this key
- `myDict.pop(key)` Remove key (and linked value) from Dictionary

MEMO: Python Iterating

- `for value in myList:` Iterating over values in a list
- `for key in myDict:` Iterating over keys in a dictionary
  `value = dict[key]`

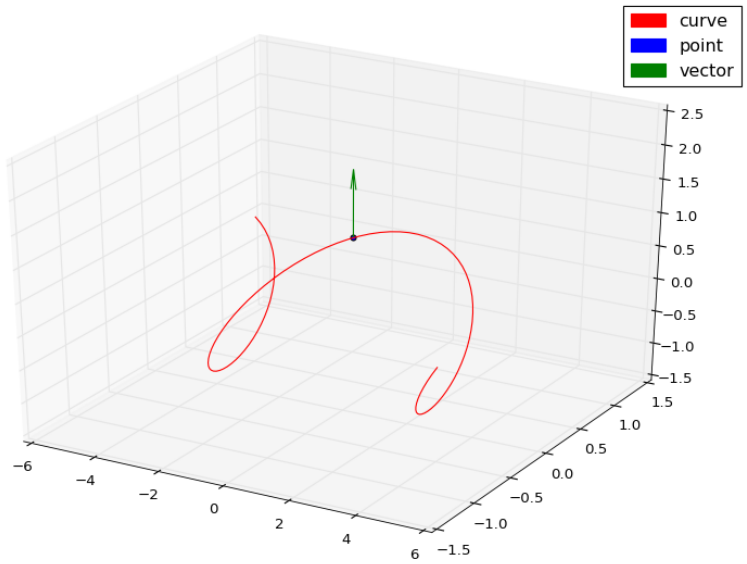
6 Tutorial 2 – Curve Plotting and Interpolation

```
# In this lesson, we will use the module numpy (linear algebra
plotting and interpolation (which allows you to write 'matlab' syntax
in Python).

######################
# Import some modules:
from numpy import *
from zeeko.occ.plotting import *
from zeeko.occ.interpolation import *

##############################################
# Working with 2D curves: similar to Matlab!
xCurve = linspace(-5, 5, 100)
yCurve = sin(xCurve)
zCurve = cos(xCurve)

#######################################################
# 2D curve plotting:
ax = figure(size=[1200,500], grid=[1,2])
plot(ax[0][0], xCurve, yCurve, color='b')
plot(ax[0][1], xCurve, zCurve, color='g')
labels(ax[0][0], 'X (mm)', 'Sinus')
labels(ax[0][1], 'X (mm)', 'Cosinus')
grid(ax[0][1])
```




```
#####################################################
# 3D curve plotting:
ax = figure(projection='3d', title='pig tail')
plot(ax[0][0], xCurve, yCurve, zCurve, color='r')

# Interpolate point from 3D curve
xPoint = 0.1
yPoint = interp1(xCurve, yCurve, xPoint)
zPoint = interp1(xCurve, zCurve, xPoint)
print xPoint, yPoint, zPoint

# Display the interpolated point
scatter(ax[0][0], xPoint, yPoint, zPoint, color='b')
```

```
# Display vector above interpolated point
quiver3d(ax[0][0], xPoint, yPoint, zPoint, 0, 0, 1, color='g')

# Add legend
legend(ax[0][0], labels=['curve','point','vector'], colors=['r','b','g'])
```



MEMO: Generating numpy Arrays

- `xRange = linspace(x1, x2, n)` Generate n points between x1 and x2
- `myArray = array([[0,1,2], [1,2,3]])` Convert list to numpy vector/array
- `myArray = zeros((rows,cols))` Create an array of zeros (also: ones(...))

MEMO: Interpolating over Arrays

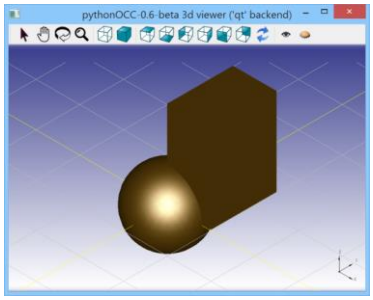
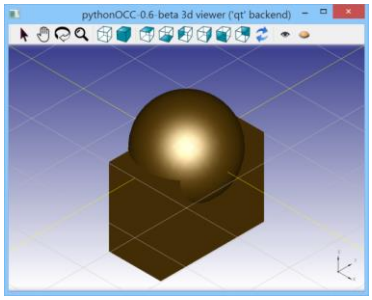
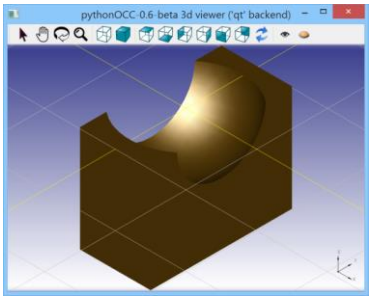
- `yPoints = interp1(xCurve, yCurve, xPoints, method='linear'/'cubic'...)`
  Interpolate points from specified 2D curve

MEMO: Plotting Functions (w:width, h:height, r:row, c:col)

- `axes = figure(size=[w,h], grid=[row,col], projection='2d'/'3d', title='')`
- `plot(axes[r][c], x, y, z, color='b')` Plot 2D/3D curve
- `scatter(axes[r][c], x, y, z, color='b', marker='o')` Plot markers
- `quiver2d(axes[r][c], xPts, yPts, xVec, yVec, ...)` Plot 2D vectors
- `quiver3d(axes[r][c], xPts, yPts, zPts, xVec, yVec, zVec, ...)` Plot 3D vectors
- `legend(axes[r][c], labels=[...], colors=[...])` Add legend box
- `grid(axes[r][c])` Show reference grids

7 Tutorial 3 – Basic CAD handling

```
# OpenCascade is a powerful C++ library for CAD/CAM programming. It
can be used to create primitives, perform boolean operations, and
import/export to CAD formats (IGES/STEP...). Python offers a binding
for OpenCascade that is much more user-friendly than C++.
# The OpenCascade API : http:/dev.opencascade.org/doc/refman/html
# Python OpenCASCADE API: http:/api.pythonocc.org/py-modindex.html

############################
# Import some custom modules
from zeeko.occ.opencascade import *

############################
# Create OpenCascade window:
occ, start_display = occviewer()

# Create box object
from OCC.BRepPrimAPI import *
box = BRepPrimAPI_MakeBox(10,15,20).Shape()
occ.Draw(box)
occ.FitAll()

# Create sphere objects
sphere = BRepPrimAPI_MakeSphere(8).Shape()
occ.Draw(sphere)
occ.FitAll()

# Define a redrawing function
def redraw(objects):
    occ.Erase()
    for object in objects:
        occ.Draw(object)
    occ.FitAll()

# Perform a redraw
redraw([box, sphere])

##################################
# Apply transformations to the Box:
#  First create a transform object
from OCC.gp import *
transform = gp_Trsf()

# Set Rotation component of the transform
angle = deg2rad(-90)
axis = gp_Ax1(gp_Pnt(0,0,0), gp_Dir(1,0,0))
transform.SetRotation(axis, angle)

# Apply Transform to Box
from OCC.TopLoc import *
location = TopLoc_Location(transform)
box.Location(location)
redraw([box, sphere])
```

```
# Set Translation component of the transform
transform.SetTranslationPart(gp_Vec(-5,-10,0))
location = TopLoc_Location(transform)
box.Location(location)
redraw([box, sphere])

############################
# Perform Boolean Operations:
#   Cut the box with the sphere
from OCC.BRepAlgoAPI import *
solid = BRepAlgoAPI_Cut(box, sphere).Shape()
redraw([solid])

############################
# Export Solid to STEP file:
occexport('C:\Python27\Tutorial3.stp', [solid])
```

MEMO: Primitive Functions

- `solid = BRepPrimAPI_MakeBox(length, width, height).Shape()`
- `solid = BRepPrimAPI_MakeCone(radius1, radius2, height).Shape()`
- `solid = BRepPrimAPI_MakeCylinder(radius, height).Shape()`
- `solid = BRepPrimAPI_MakeSphere(radius).Shape()`
- `solid = BRepPrimAPI_MakeTorus(radius1, radius2).Shape()`

MEMO: Transform Functions

- `transform = gp_Trsf()` Create new transform
- `axis = gp_Ax1(gp_Pnt(x,y,z), gp_Dir(x,y,z))` Create new axis
- `transform.SetRotation(axis, angle)`
- `transform.SetTranslationPart(gp_Vec(x,y,z))`
- `solid.Location(TopLoc_Location(transform))` Apply transform

MEMO: Boolean Operations

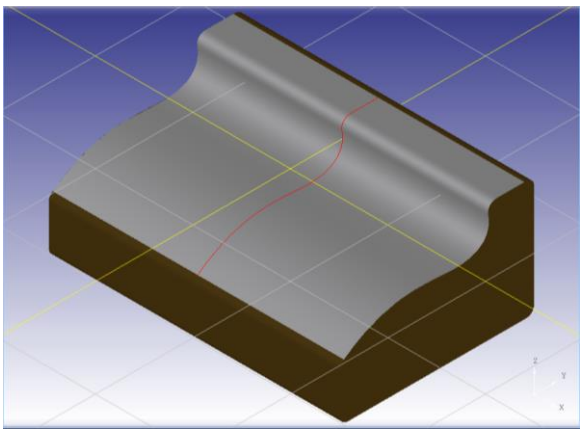
- `solid = BRepAlgoAPI_Cut(solid1, solid2).Shape()` Boolean cut
- `solid = BRepAlgoAPI_Common(solid1, solid2).Shape()` Boolean intersect
- `solid = BRepAlgoAPI_Fuse(solid1, solid2).Shape()` Boolean union

MEMO: STEP/IGES Import/Export

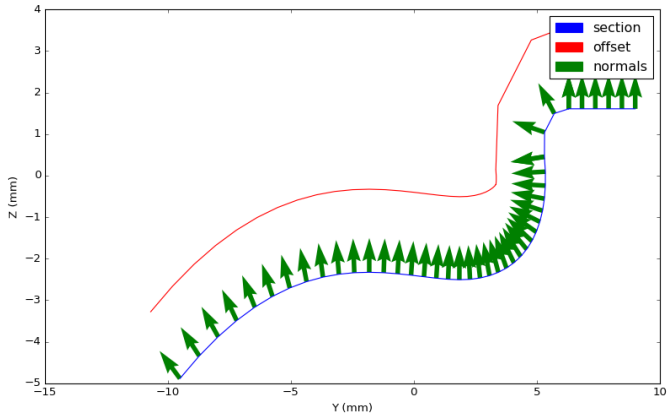
- `solids = occimport(filename)` File extension: *.iges or *.step
- `occexport(filename, [solids])` File extension: *.iges or *.step

8 Tutorial 4 – Sectioning and Offsetting

```
# In this tutorial, you will learn how to compute a section curve, and
offset it to generate a milling tool path curve.

############################
# Import some custom modules
from zeeko.occ.opencascade import *
from zeeko.occ.plotting import *

# Import example workpiece from STEP file
occ, start_display = occviewer()
work = occimport("C:/Python27/Examples/rail.stp")
occ.Draw(work[0])
occ.FitAll()

# We select some faces of the CAD object
occ.SelectFaces([2,6,7])

# Generate shell from the selected faces
faces = occ.GetSelection(work[0])
shell = occshell(faces)

# It is possible to convert a shell to a mesh
vertices,triangles = occmesh(shell, quality=1.)

# It is possible to convert a shell to a mesh
ax = figure(projection='3d')
trisurf(ax[0][0], vertices, triangles)
labels(ax[0][0], 'X (mm)', 'Y (mm)', 'Z (mm)')

########################################
# Define section plane
from OCC.gp import *
secPoint = gp_Pnt(0,0,0)
secDirection = gp_Dir(1,0,0)
secPlane = gp_Pln(secPoint, secDirection)

# Section workpiece with Plane, and draw result
from OCC.BRepAlgoAPI import *
section = BRepAlgoAPI_Section(shell, secPlane)
occ.Draw(section.Shape())
```



```
# Sample points every 0.5 mm along the section curve
from zeeko.occ.sectioning import *
points, normals, tangents = occsample(section, secDirection, 0.5)

# Compute and offset curve 2 mm away
toolRadius = 2.
offsets = points + toolRadius*normals

##########################################
# Plot the cross section
ax = figure()
plot(ax[0][0], points[1], points[2], color='b')
plot(ax[0][0], offsets[1], offsets[2], color='r')
quiver2d(ax[0][0], points[1], points[2], normals[1], normals[2], color='g')
legend(ax[0][0], labels=['section','offset','normals'], colors=['b','r','g'])
labels(ax[0][0], 'Y (mm)', 'Z (mm)')
```



MEMO: Shell funtions

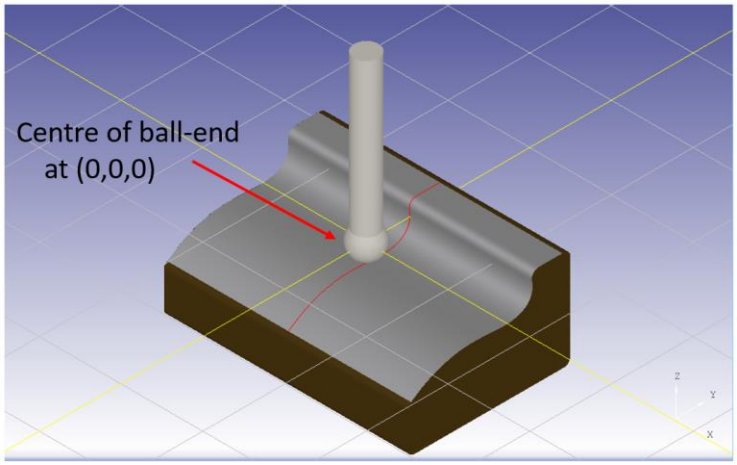
- `occ.SelectFaces([indices])` Select faces by index
- `faces = occ.GetSelection(solid)` Get list of selected faces
- `shell = occshell(faces)` Convert list of faces to shell
- `vertices,triangles = occmesh(shell, quality=1.)` Convert to mesh

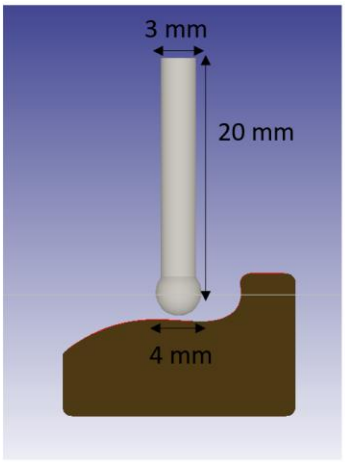
MEMO: Sectioning functions

- `plane = gp_Pln(gp_Pnt, gp_Dir)` Create section plane
- `section = BRepAlgoAPI_Section(shape, plane)` Perform section
- `points, normals, tangents = occsample(section, gp_dir, spacing)`

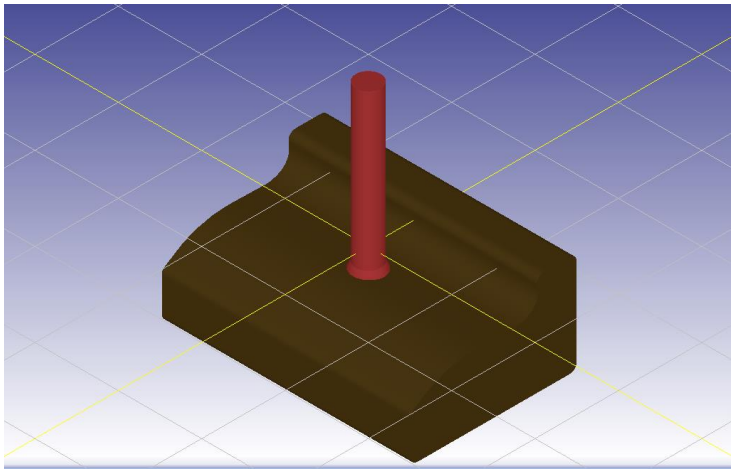
9 Tutorial 5 - Collision Detection

```
# In this tutorial, you will learn how to use ODE to check whether 2
meshes are colliding.

########################################
# Generate a milling tools from primitives
from OCC.BRepPrimAPI import *
sphere = BRepPrimAPI_MakeSphere(2.0).Shape()
cylinder = BRepPrimAPI_MakeCylinder(1.5, 20.0).Shape()
tool = BRepAlgoAPI_Fuse(sphere, cylinder).Shape()
drawTool = occ.Draw(tool, selectable=False, color=occ.white)
occ.FitAll()
```






```
############################
# Setup Collision Simulator
from zeeko.occ.colliders import *
from OCC.DYN.Context import *
from ode import collide

# Create Simulation World
world = DynamicSimulationContext()
world.enable_collision_detection()

# Create colliders for the work and tool
colWork = CreateCollider(work[0], world)
colTool = CreateCollider(tool, world)

################################################
# Compute collision state (0 collision detected!)
cols = collide(colWork.geometry, colTool.geometry)
print "Collisions: " + str(len(cols))

############################
# Move the tool into the work
from OCC.gp import *
transform = gp_Trsf()
transform.SetTranslationPart(gp_Vec(0, 0, -3))
UpdateCollider(colTool, transform)
occ.Update(drawTool, transform, color=occ.red)
```



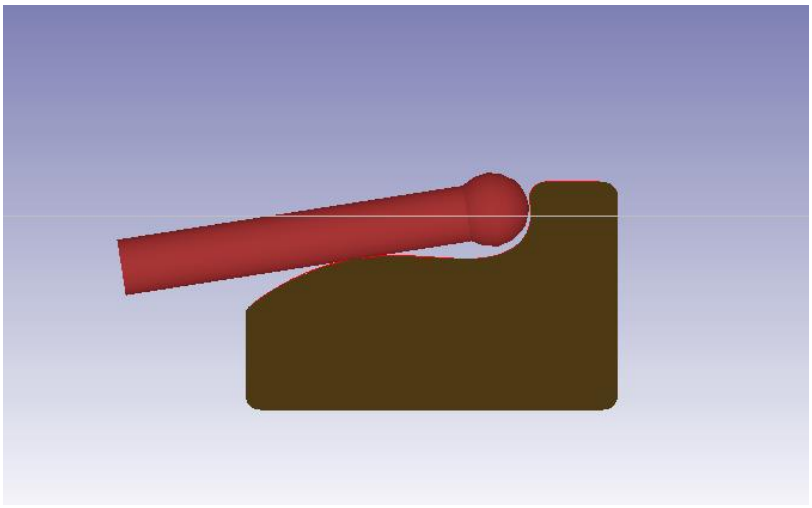
```
##################################
# Update collider and check again
cols = collide(colWork.geometry, colTool.geometry)
print "Collisions: " + str(len(cols))
```

MEMO: Collision functions

- `world = DynamicSimulationContext()`
- `collider = CreateCollider(shape, world)`
- `UpdateCollider(collider, transform)`
- `collide(collider1.geometry, collider2.geometry)`

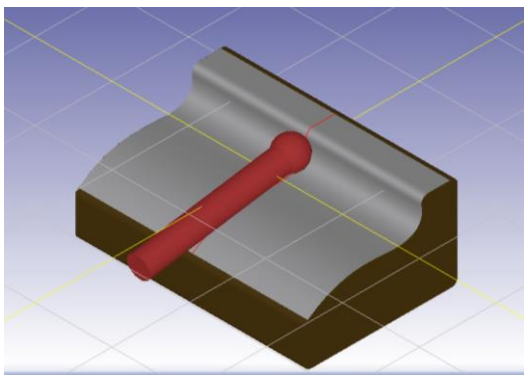
10 Tutorial 6 – Tool Animation

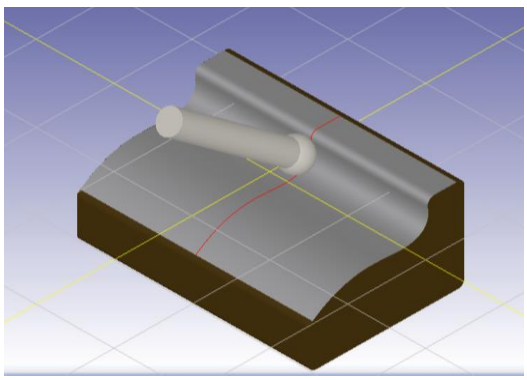
```
# In this tutorial, we provide a convenient routine for animating tool
paths, and switching the tool white/red depending on collisions.

# Define simulation routine
def Simulate(offset, normal):
    # Compute quaternion for tool rotation
    fromVector = gp_Vec(0,0,1)
    toVector = gp_Vec(normal[0], normal[1], normal[2])
    quaternion = gp_Quaternion(fromVector, toVector)
    # Create new Transform
    transform = gp_Trsf()
    transform.SetRotation(quaternion)
    transform.SetTranslationPart(gp_Vec(offset[0],offset[1], offset[2]))
    # Compute Collisions
    UpdateCollider(colTool, transform)
    cols = collide(colWork.geometry, colTool.geometry)
    # Update color and transform of drawn tool
    if len(cols) > 0:
        occ.Update(drawTool, transform, color=occ.red)
    else:
        occ.Update(drawTool, transform, color=occ.white)
    # Return number of collisions
    return len(cols)


# Simulate tool path, sleep() is used to regulate simulation speed.
# The 0.1 mm offset is used to prevent "proximity" collisions
from time import *
length = len(offsets[0])
for i in range(0, length):
    normal = normals[:,i]
    offset = offsets[:,i] + 0.1*normal
    nCols = Simulate(offset, normal)
    if nCols > 0:
        print "Index: %i (%i collisions)" % (i,nCols)
    sleep(0.05)
```



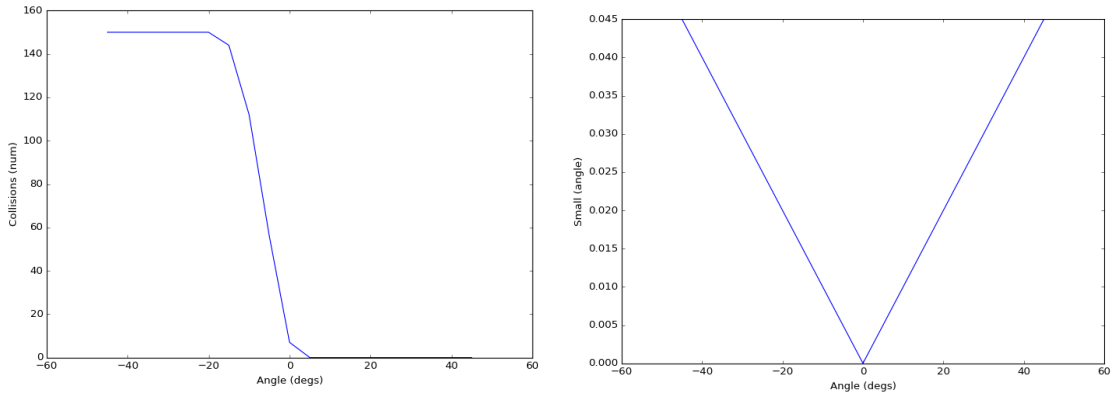
11 Tutorial 7 – Collision Avoidance

```
# In this tutorial, you will learn the technique for collision
avoidance using numerical optimization.

# We select a colliding point in the tool path (index 34).
normal = normals[:,34]
offset = offsets[:,34] + 0.1*normal
nCols = Simulate(offset, normal)
print nCols

# We define an attack axis, and some attack angles
from zeeko.occ.algebra import *
axis = array([1,0,0])
angles = linspace(-45, 45, 19)

# We simulates the various attack angles
from time import *
collisions = zeros(19)
for i in range(0, len(angles)):
    # Compute rotation matrix and apply to normal
    rotMat = RotationMatrix(axis, deg2rad(angles[i]))
    vector = dot(rotMat, normal)
    # Simulate collisions
    nCols = Simulate(offset, vector)
    print "Angle: %f (%i cols) " % (angles[i],nCols)
    collisions[i] = nCols
    sleep(0.05)
```

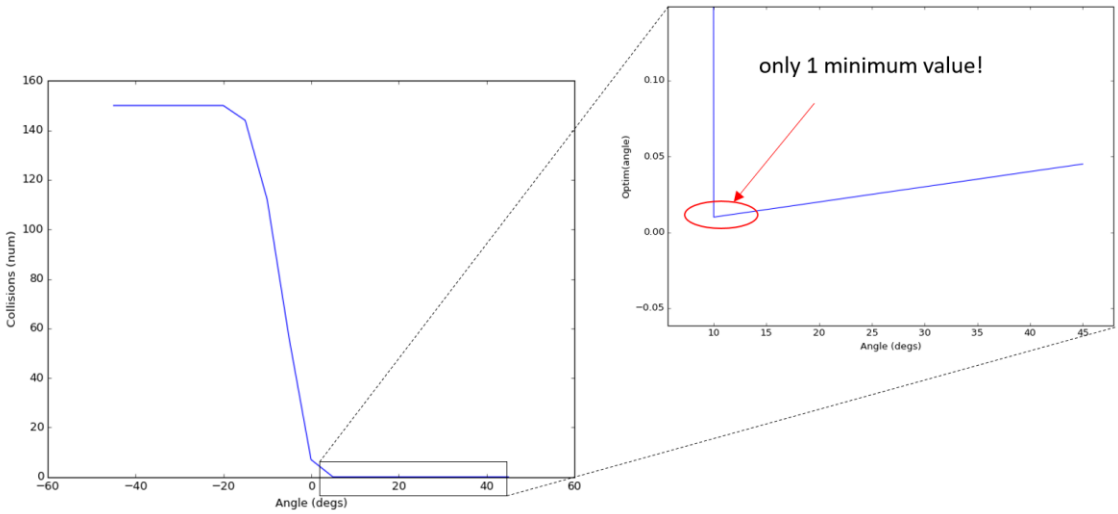




```
# Plot attack angles vs number of collisions
from zeeko.occ.plotting import *
ax = figure()
plot(ax[0][0], angles, collisions)
labels(ax[0][0], 'Angle (degs)', 'Collisions (num)')

# We generate a small deviation function
small = 0.001*abs(angles)
ax = figure()
plot(ax[0][0], angles, small)
labels(ax[0][0], 'Angle (degs)', 'Small(angle)')
```




```
# The optimization function is the combination of both
ax = figure()
plot(ax[0][0], angles, collisions+small)
labels(ax[0][0], 'Angle (degs)', 'Optim(angle)')
```




```
# Create Optimization Routine
def Optim(angle):
    # Compute rotation matrix, apply to normal, and simulate
    rotMat = RotationMatrix(axis, deg2rad(angle))
    vector = dot(rotMat, normal)
    nCols = Simulate(offset, vector)
    sleep(0.05)
    return nCols + 0.001*abs(angle)

# Optimize the Collision Avoidance
from scipy.optimize import brute
result = brute(Optim, ((-45, 45),))
print "Optimum angle: %f degs" % (result)
```

MEMO: Rotating vectors

- `rotMat = RotationMatrix(axis, angle)`
- `rotatedVector = dot(rotMat, originalVector)`

MEMO: Optimization functions

- `result = brute(Optim, ((minValue, maxValue),))`

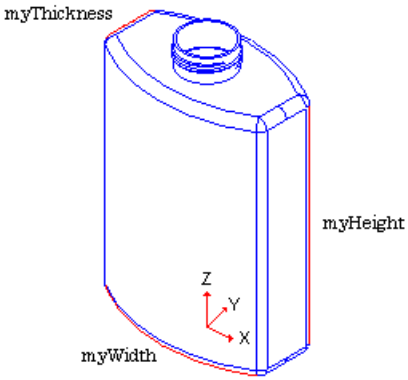
12 Appendix 1: Designing a Bottle

To illustrate usage of the various functions provided in python OCC, this appendix shows how to design a bottle (see script Bottle.py).

12.1 Model Specifications

We first define the bottle specifications as follows:

Object Parameter	Parameter Name	Parameter Value
Bottle height	myHeight	70mm
Bottle width	myWidth	50mm
Bottle thickness	myThickness	30mm



In addition, we decide that the bottle's profile (base) will be centered on the origin of the global Cartesian coordinate system.

12.2 Import required Modules

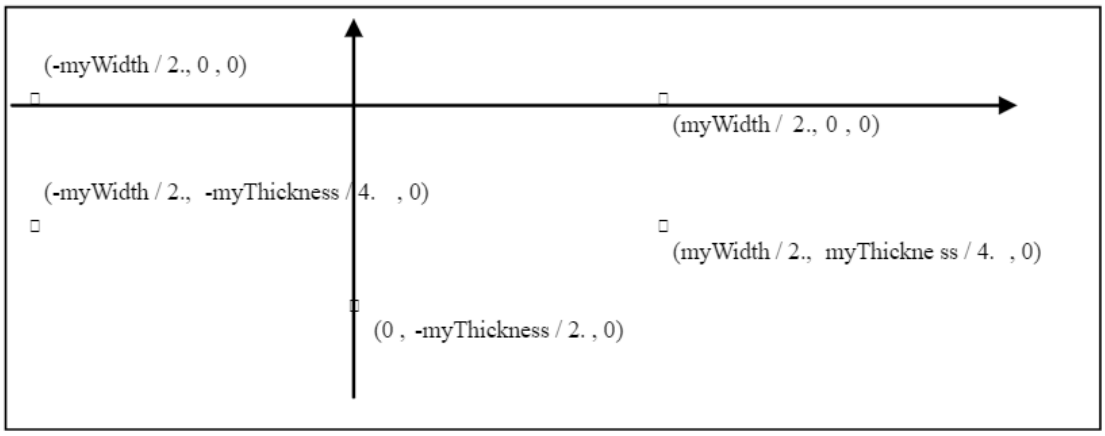
The following modules are imported in order to produce the bottle:

```
# Import modules
from OCC.gp import *
from OCC.GC import *
from OCC.BRep import *
from OCC.BRepAlgoAPI import *
from OCC.BRepBuilderAPI import *
from OCC.BRepFilletAPI import *
from OCC.BRepPrimAPI import *
from OCC.TopAbs import *
from OCC.TopoDS import *
from OCC.TopExp import *
from OCC.TopLoc import *
```

12.3 Building the Profile

12.3.1 Defining Support Points

To create the bottle's profile, you first create characteristic points with their coordinates as shown below in the (XOY) plane. These points will be the supports that define the geometry of the profile.



To instantiate a *gp_Pnt* point object, just specify the X, Y, and Z coordinates of the points in the global Cartesian coordinate system:

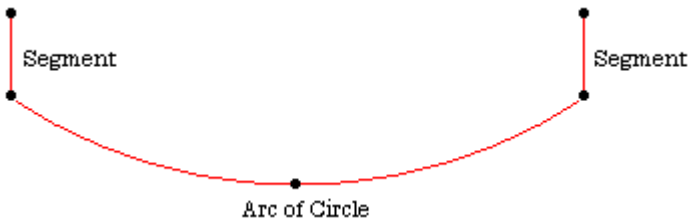
```
# Define the bottle dimensions
myHeight = 70.
myWidth = 50.
myThickness = 30.

# The points we'll use to create the profile of the bottle's body
aPnt1 = gp_Pnt(-myWidth / 2.0, 0, 0)
aPnt2 = gp_Pnt(-myWidth / 2.0, -myThickness / 4.0, 0)
aPnt3 = gp_Pnt(0, -myThickness / 2.0, 0)
aPnt4 = gp_Pnt(myWidth / 2.0, -myThickness / 4.0, 0)
aPnt5 = gp_Pnt(myWidth / 2.0, 0, 0)
```

Once your objects are instantiated, you can use methods provided by the class to access and modify its data. For example, to get the X coordinate of a point:

```
xVal = aPnt1.X();
```

12.3.2 Profile: Defining the Geometry

With the help of the previously defined points, you can compute a part of the bottle's profile geometry. As shown in the figure below, it will consist of two segments and one arc.



To create such entities, you need a specific data structure, which implements 3D geometric objects. You can compute elementary curves and surfaces by using the *GC* package, which provides two algorithm classes which are exactly what is required for our profile:

- Class *GC_MakeSegment* to create a segment. One of its constructors allows you to define a segment by two end points P1 and P2
- Class *GC_MakeArcOfCircle* to create an arc of a circle. A useful constructor creates an arc from two end points P1 and P3 and going through P2.

Both of these classes return a *Geom_TrimmedCurve*. This entity represents a base curve (line or circle, in our case), limited between two of its parameter values. For example, circle C is parameterized between 0 and 2PI. If you need to create a quarter of a circle, you create a *Geom_TrimmedCurve* on C limited between 0 and $\pi/2$.

```
34 # Generate arc and segments
35 aArcOfCircle = GC_MakeArcOfCircle(aPnt2, aPnt3, aPnt4)
36 aSegment1 = GC_MakeSegment(aPnt1, aPnt2)
37 aSegment2 = GC_MakeSegment(aPnt4, aPnt5)
```

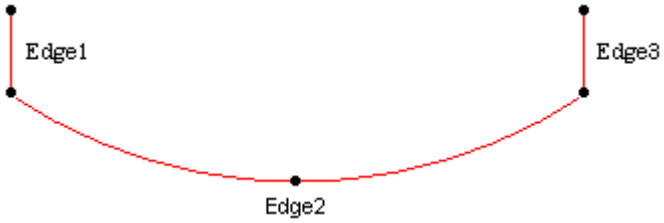
12.3.3 Profile: Defining the Topology

You have created the support geometry of one part of the profile but these curves are independent with no relations between each other. To simplify the modeling, it would be right to manipulate these three curves as a single entity. This can be done by using the topological data structure of Open CASCADE Technology defined in the *TopoDS* package: it defines relationships between geometric entities which can be linked together to represent complex shapes. Each object of the *TopoDS* package, inheriting from the *TopoDS_Shape* class, describes a topological shape as described below:

Shape	OCC Class	Description
Vertex	TopoDS_Vertex	Zero dimensional shape corresponding to a point in geometry.
Edge	TopoDS_Edge	One-dimensional shape corresponding to a curve and bounded by a vertex at each extremity.
Wire	TopoDS_Wire	Sequence of edges connected by vertices.
Face	TopoDS_Face	Part of a surface bounded by a closed wire(s).
Shell	TopoDS_Shell	Set of faces connected by edges.
Solid	TopoDS_Solid	Part of 3D space bounded by Shells.
CompSolid	TopoDS_CompSolid	Set of solids connected by their faces.
Compound	TopoDS_Compound	Set of any other shapes described above.

Referring to the previous table, to build the profile, you will create:

- Three edges out of the previously computed curves.
- One wire with these edges.



However, the *TopoDS* package provides only the data structure of the topological entities. Algorithm classes available to compute standard topological objects can be found in the *BRepBuilderAPI* package. To create an edge, you use the BRepBuilderAPI_MakeEdge class with the previously computed curves:

```
39 # Convert the arcs/segments to edges
40 aEdge1 = BRepBuilderAPI_MakeEdge(aSegment1.Value()).Edge()
41 aEdge2 = BRepBuilderAPI_MakeEdge(aArcOfCircle.Value()).Edge()
42 aEdge3 = BRepBuilderAPI_MakeEdge(aSegment2.Value()).Edge()
```

In Open CASCADE, you can create edges in several ways. One possibility is to create an edge directly from two points, in which case the underlying geometry of this edge is a line, bounded by two vertices being automatically computed from the two input points. For example, aEdge1 and aEdge3 could have been computed in a simpler way:

```
aEdge1 = BRepBuilderAPI_MakeEdge(aPnt1, aPnt3);
aEdge2 = BRepBuilderAPI_MakeEdge(aPnt4, aPnt5);
```

To connect the edges, you need to create a wire with *BRepBuilderAPI_MakeWire* class. There are two ways of building a wire with this class:

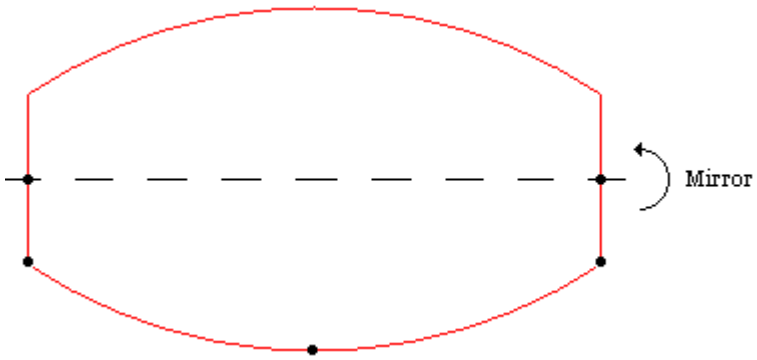
- directly from one to four edges
- by adding other wire(s) or edge(s) to an existing wire (see from line 61).

```
44 # Create a wire out of the edges
45 aWire = BRepBuilderAPI_MakeWire(aEdge1, aEdge2, aEdge3).Wire()
```

12.3.4 Profile: Completing the Profile

Once the first part of your wire is created you need to compute the complete profile. A simple way to do this is to:

- compute a new wire by reflecting the existing one.
- add the reflected wire to the initial one.



To apply a transformation on shapes (including wires), you first need to define the properties of a 3D geometric transformation by using the gp_Trsf class. This transformation can be a translation, a rotation, a scale, a reflection, or a combination of these. In our case, we need to define a reflection with respect to the X axis of the global coordinate system. An axis, defined with the gp_Ax1 class, is built out of a point and has a direction (3D unitary vector). There are two ways to define this axis. The first way is to define it from scratch, using its geometric definition:

- X axis is located at (0, 0, 0) - use the *gp_Pnt* class.
- X axis direction is (1, 0, 0) - use the *gp_Dir* class. A *gp_Dir* instance is created out of its X, Y and Z coordinates.

```
47 # Specify the X axis
48 aOrigin = gp_Pnt(0, 0, 0)
49 xDir = gp_Dir(1, 0, 0)
50 xAxis = gp_Ax1(aOrigin, xDir)
```

As previously explained, the 3D geometric transformation is defined with the *gp_Trsf* class. There are two different ways to use this class:

- by defining a transformation matrix by all its values
- by using the appropriate methods corresponding to the required transformation (SetTranslation for a translation, SetMirror for a reflection, etc.): the matrix is automatically computed.

Since the simplest approach is always the best one, you should use the SetMirror method with the axis as the center of symmetry.

```
52 # Set up the mirror
53 aTrsf = gp_Trsf()
54 aTrsf.SetMirror(xAxis)
```

You now have all necessary data to apply the transformation with the BRepBuilderAPI_Transform class by specifying:

- the shape on which the transformation must be applied.
- the geometric transformation

```
56 # Apply the mirror transformation
57 aMirroredShape = BRepBuilderAPI_Transform(aWire, aTrsf).Shape()
```

The returned object is a generic shape. What you need is a method to cast this generic shape as a wire. The *TopoDS* global functions provide this kind of service by casting a shape into its real type. To cast the transformed wire, use *topods_Wire*.

```
59 # Convert to wire instead of a generic shape now
60 aMirroredWire = topods.Wire(aMirroredShape)
```

The bottle's profile is almost finished. You have created: *aWire* and *aMirroredWire*. You need to concatenate them to compute a single shape. To do this, you use the *BRepBuilderAPI_MakeWire* class as follows:

- create an instance of *BRepBuilderAPI_MakeWire*.
- add all edges of the two wires by using the *Add* method on this object.

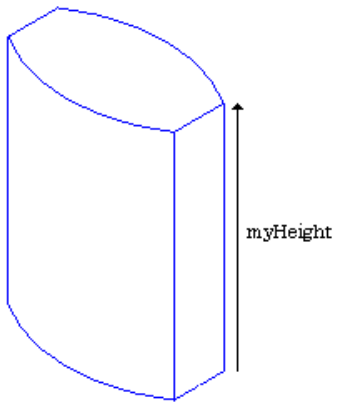
```
62 # Combine the two constituent wires
63 mkWire = BRepBuilderAPI_MakeWire()
64 mkWire.Add(aWire)
65 mkWire.Add(aMirroredWire)
66 myWireProfile = mkWire.Wire()
```

12.4 Building the Body

12.4.1 Prism the Profile

To compute the main body of the bottle, you need to create a solid shape. The simplest way is to use the previously created profile and to sweep it along a direction.
The *Prism* functionality is the most appropriate for that task. It accepts a shape and a direction as input and generates a new shape according to the following rules:

Shape	Generates
Vertex	Edge
Edge	Face
Wire	Shell
Face	Solid
Shell	Compound of Solids



Your current profile is a wire. Referring to the Shape/Generates table, you need to compute a face out of its wire to generate a solid. To create a face, use the *BRepBuilderAPI_MakeFace* class. As previously explained, a face is a part of a surface

bounded by a closed wire. Generally, *BRepBuilderAPI_MakeFace* computes a face out of a surface and one or more wires. When the wire lies on a plane, the surface is automatically computed.

```
68 # The face that we'll sweep to make the prism
69 myFaceProfile = BRepBuilderAPI_MakeFace(myWireProfile).Face()
```

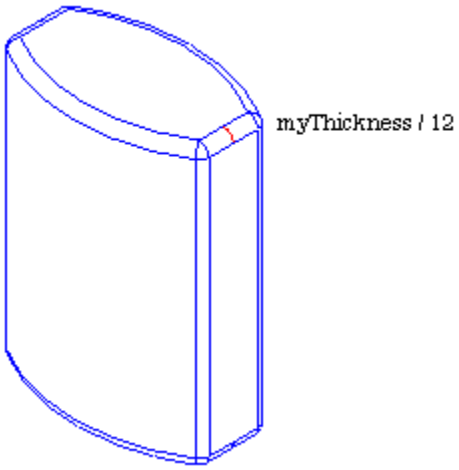
The *BRepPrimAPI* package provides all the classes to create topological primitive constructions: boxes, cones, cylinders, spheres, etc. Among them is the *BRepPrimAPI_MakePrism* class. As specified above, the prism is defined by:

- the basis shape to sweep;
- a vector for a finite prism or a direction for finite and infinite prisms.

You want the solid to be finite, swept along the Z axis and to be myHeight height. The vector, defined with the *gp_Vec* class on its X, Y and Z coordinates, is aPrismVec and then all the necessary data to create the main body of your bottle is now available. Just apply the *BRepPrimAPI_MakePrism* class to compute the solid:

```
71 # We want to sweep the face along the Z axis to the myHeight
72 aPrismVec = gp_Vec(0, 0, myHeight)
73 myBody = BRepPrimAPI_MakePrism(myFaceProfile, aPrismVec).Shape()
```

12.4.2 Applying Fillets



The edges of the bottle's body are very sharp. To replace them by rounded faces, you use the *Fillet* functionality. For our purposes, we will specify that fillets must be:

- applied on all edges of the shape
- have a radius of *myThickness* / 12

To apply fillets on the edges of a shape, you use the *BRepFilletAPI_MakeFillet* class. This class is normally used as follows:

- Specify the shape to be filleted in the *BRepFilletAPI_MakeFillet* constructor.
- Add the fillet descriptions (an edge and a radius) using the *Add* method (you can add as many edges as you need).
- Ask for the resulting filleted shape with the *Shape* method.

```
75 # Create filletting tool
76 mkFillet = BRepFilletAPI_MakeFillet(myBody)
```

To add the fillet description, you need to know the edges belonging to your shape. The best solution is to explore your solid to retrieve its edges. This kind of functionality is

provided with the *TopExp_Explorer* class, which explores the data structure described in a *TopoDS_Shape* and extracts the sub-shapes you specifically need. Generally, this explorer is created by providing the following information:

- the shape to explore (*myBody*)
- the type of sub-shapes to be found (*TopAbs_EDGE*).

An explorer is usually applied in a "while" loop by using its three main methods:

- *More()* to know if there are more sub-shapes to explore.
- *Current()* to know which is the currently explored sub-shape.
- *Next()* to move onto the next sub-shape to explore.

```
78 # Add fillets to all edges through the explorer
79 anEdgeExplorer = TopExp_Explorer(myBody, TopAbs_EDGE)
80 while anEdgeExplorer.More():
81     anEdge = topods.Edge(anEdgeExplorer.Current())
82     mkFillet.Add(myThickness / 12.0, anEdge)
83     anEdgeExplorer.Next()
84 myBody = mkFillet.Shape()
```

In the explorer loop, you have found all the edges of the bottle shape. Each one must then be added in *mkFillet* with the *.Add(radius, edge)* method. Once this is done, the last step of the procedure was asking for the filleted shape.

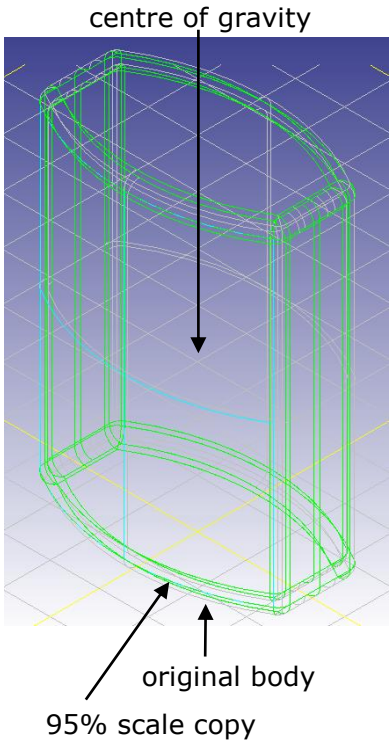
12.4.3 Cut the inside the Bottle

Right now, the body of the bottle is a solid shape. In order to fill it with liquid, we need to cut out the inside of the bottle. We can do this by first making a copy of the body using *BRepBuilderAPI_Copy*:

```
86 # Make a copy of the body
87 copier = BRepBuilderAPI_Copy()
88 copier.Perform(myBody)
89 myCopy = copier.Shape()
```

Next, we rescale the copy to make it 95% as big as the original, using the centre of gravity of the bottle (0, 0, *myHeight/2*) as the scaling centre.

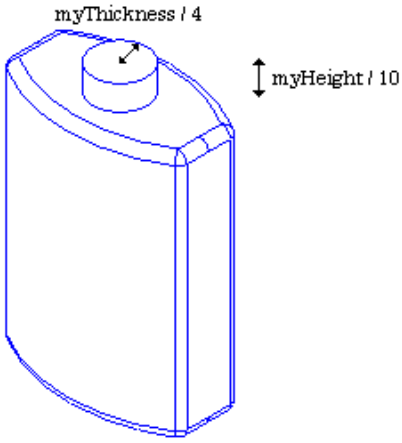
```
91 # Rescale the copy
92 transform = gp_Trsf()
93 scaleCentre = gp_Pnt(0, 0, myHeight/2)
94 transform.SetScale(scaleCentre, 0.95)
95 location = TopLoc_Location(transform)
96 myCopy.Location(location)
```



Finally, we cut the inside of *myBody* with the rescaled *myCopy* using the function *BRepAlgoAPI_Cut*:

```
# Cut inside the bottle using the rescaled copy
myBody = BRepAlgoAPI_Cut(myBody, myCopy).Shape()
```

12.4.4 Adding the Neck



To add a neck to the bottle, you will create a cylinder and fuse it to the body. The cylinder is to be positioned on the top face of the body with a radius of *myThickness* / 4. and a height of *myHeight* / 10.

To position the cylinder, you need to define a coordinate system with the *gp_Ax2* class defining a right-handed coordinate system from a point and two directions - the main (Z) axis direction and the X direction (the Y direction is computed from these two). To align the neck with the center of the top face, being in the global coordinate system (0, 0, *myHeight*), with its normal on the global Z axis, your local coordinate system can be defined as follows:

```
# Create axis for bottle neck
neckLocation = gp_Pnt(0, 0, myHeight)
neckAxis = gp_DZ()
neckAx2 = gp_Ax2(neckLocation, neckAxis)
```

To create a cylinder, use another class from the primitives construction package: the *BRepPrimAPI_MakeCylinder* class. The information you must provide is:

- the coordinate system where the cylinder will be located;
- the radius and height.

```
# Add bottle neck
neckRadius = myThickness / 4.0
neckHeight = myHeight / 10.0
neckCylinder = BRepPrimAPI_MakeCylinder(neckAx2, neckRadius, neckHeight).Shape()
```

You now have two separate parts: a main body and a neck that you need to fuse together. The *BRepAlgoAPI* package provides services to perform Boolean operations between shapes, and especially: *common* (Boolean intersection), *cut* (Boolean subtraction) and *fuse* (Boolean union). Use *BRepAlgoAPI_Fuse* to fuse the two shapes:

```
bottle = BRepAlgoAPI_Fuse(myBody, neckCylinder).Shape()
```

12.4.5 Opening the Neck

To open the neck, we will create a cylinder that is narrower (*neckRadius-0.2*) and longer (neckHeight+10.0). We also need to offset this cylinder slightly to (0, 0, *myHeight-5.0*), in order to cut through the inside of the bottle, using *BRepAlgoAPI_Cut*:

```
112 # Create axis for cutter
113 cutLocation = gp_Pnt(0, 0, myHeight-5.0)
114 cutAxis = gp_DZ()
115 cutAx2 = gp_Ax2(cutLocation, cutAxis)
116
117 # Cut inside of neck
118 cutRadius = neckRadius-0.1
119 cutHeight = neckHeight+10.0
120 cutCylinder = BRepPrimAPI_MakeCylinder(cutAx2, cutRadius, cutHeight).Shape()
121 bottle = BRepAlgoAPI_Cut(bottle, cutCylinder).Shape()
```

12.5 Display and Export

Finally, we can display and/or export our bottle model.

```
123 # Display the result in OCC viewer
124 from zeeko.occ.opencascade import *
125 occ, start_display = occviewer()
126 occ.Draw(bottle)
127 occ.FitAll()
128
129 # Export to STEP file
130 occexport('C:\Python27\Bottle.step', [bottle])
```

STEP file render from the Autodesk Website (https://a360.autodesk.com/viewer):