|  |
| --- |
| Technical Documentation |
|  |
| Last saved:  Author: Christian Tillich |
| autodesk_logo_bk |

Table of Contents

[3D Part Content Parametric Construction Development 1](#_Toc212135448)

[Table of Contents 2](#_Toc212135449)

[1 Administration 4](#_Toc212135450)

[1.1 Document History 4](#_Toc212135451)

[1.2 Team Members 4](#_Toc212135452)

[1.3 Related Documents 4](#_Toc212135453)

[2 Introduction 5](#_Toc212135454)

[2.1 Terminology 5](#_Toc212135455)

[3 Parametric Construction Scripts (Variant) 6](#_Toc212135456)

[3.1 Primitives 8](#_Toc212135457)

[3.1.1 ARC3D 10](#_Toc212135458)

[3.1.2 ARC3D2 11](#_Toc212135459)

[3.1.3 ARC3DS 12](#_Toc212135460)

[3.1.4 BOX 13](#_Toc212135461)

[3.1.5 CONE 14](#_Toc212135462)

[3.1.6 CYLINDER 15](#_Toc212135463)

[3.1.7 ELLIPSOIDHEAD 16](#_Toc212135464)

[3.1.8 ELLIPSOIDHEAD2 17](#_Toc212135465)

[3.1.9 ELLIPSOIDSEGMENT 18](#_Toc212135466)

[3.1.10 HALFSPHERE 19](#_Toc212135467)

[3.1.11 PYRAMID 20](#_Toc212135468)

[3.1.12 ROUNDRECT 21](#_Toc212135469)

[3.1.13 SPHERESEGMENT 22](#_Toc212135470)

[3.1.14 TORISPHERICHEAD 23](#_Toc212135471)

[3.1.15 TORISPHERICHEAD2 24](#_Toc212135472)

[3.1.16 TORISPHERICHEADH 25](#_Toc212135473)

[3.1.17 TORUS 26](#_Toc212135474)

[3.2 Member Functions 27](#_Toc212135475)

[3.2.1 .uniteWith 29](#_Toc212135476)

[3.2.2 .subtractFrom 30](#_Toc212135477)

[3.2.3 .intersectWith 31](#_Toc212135478)

[3.2.4 .erase 32](#_Toc212135479)

[3.2.5 .rotateX 33](#_Toc212135480)

[3.2.6 .rotateY 34](#_Toc212135481)

[3.2.7 .rotateZ 35](#_Toc212135482)

[3.2.8 .translate 36](#_Toc212135483)

[3.2.9 .setTransformationMatrix 37](#_Toc212135484)

[3.2.10 .setPoint 38](#_Toc212135485)

[3.2.11 .parameters 39](#_Toc212135486)

[3.2.12 .transformationMatrix 40](#_Toc212135487)

[3.2.13 .numberOfPoints 41](#_Toc212135488)

[3.2.14 .pointAt 42](#_Toc212135489)

[3.2.15 .directionAt 43](#_Toc212135490)

[3.3 Functions 44](#_Toc212135491)

[3.3.1 activate 45](#_Toc212135492)

[3.3.1.1 dbtype 46](#_Toc212135493)

[3.3.2 TESTACPSCRIPT / TESTACPSCRIPT1 48](#_Toc212135494)

[3.3.3 demand loader 49](#_Toc212135495)

[4 variants.zip 50](#_Toc212135496)

[4.1 amake.bat 51](#_Toc212135497)

[5 Anything else 53](#_Toc212135498)

# Administration

## Document History

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Name | Status | Comments |
| 18 Oct. 2008 | Christian Tillich | Preliminary draft | Initial pass |
| Date Revised |  | Enter status |  |

## Team Members

|  |  |  |
| --- | --- | --- |
| Role | Name | Date of Review and Sign-off |
| Development Lead | Christian Tillich | Date of Review |
| Product Management |  | Date of Review |
| Project Management |  | Date of Review |
| SWD Management |  | Date of Review |
| TD |  | Date of Review |
| TD Management |  | Date of Review |
| User Assistance |  | Date of Review |

## Related Documents

|  |  |
| --- | --- |
| File Name | Description |
|  |  |

# Introduction

This document describes the Autodesk Internal ‘Content Parametric Script Design Process’ for generating Plant 3D parts.

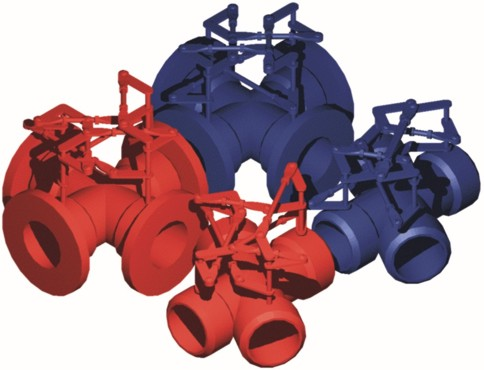
This document outlines the design of parametric construction scripts and covers the detailed steps to complete a Content Build.

The scope of this document is limited to 3D Part Content, as it will be published in a Part Catalog database.

## Terminology

| **Term** | **Definition** |
| --- | --- |
| Catalogs | A repository of parts which are referenced into the 3D model for various reasons. The parts are organized in a general fashion so that a user can browse through them for their various applications. |
| Variant | Synonym for parametric construction script |

# Parametric Construction Scripts (Variant)

**Plant 3D** is delivered with a large catalog of predefined parts. To get 3D representations of these parts into 3D drawings "**variant programs**" are used. A "**variant**" is a small (**Python**) subroutine that takes the dimensions of a specific part as input and creates a 3D representation (typically a solid within a block) as output.

Approximately 20.000 different **variant** subroutines are part of the package, and additional ones can easily be added. The available **variants** cover nearly all types of parts commonly used in plant design: pipes, elbows, flanges, tees, crosses, nozzles, olets, different types of valves and many more.

The **variant** are created by using default, C predefined, [**primitives**](#_Primitives) but each **variant** can also be used by another **variant**.

The [**primitives**](#_Primitives) can be positioned, combined, subtracted,… with different [**member functions**](#_Member_Functions).

And we use some [**functions**](#_Functions) to setup using the **variant** inside **Plant 3D**.

A typical **variant** looks like this:

def CPFLR(s, L=22.0, D1=220.0, D2=114.3, ID="CPFLR", \*\*kw):

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=L, O=D2/2.0)[.rotateY](#_Modifier_function:_.rotateY)(90.0)

s[.setPoint](#_Modifier_function:_.setPoint)((0.0, 0.0, 0.0), (-1.0, 0.0, 0.0), 0.0)

s[.setPoint](#_Modifier_function:_.setPoint)((0.0, 0.0, 0.0), ( 1.0, 0.0, 0.0), 0.0)

This one creates a simple lapped flange with two connection points (ports). Connection points have a position and a direction vector that allows for automatic alignment of connected parts.

To get the data from database we need also the database field information. That is also done inside the **variant** file at the construction routine (normally direct after the **variant** definition end) and looks like, for the example above:

[activate](#_activate)(CPFLR,

"""

[Type AQA-VCPFLR]

VID=STRING,32

DN=STRING

L=LENGTH

D1=LENGTH

D2=LENGTH

Units=STRING,8

;

uniqId=CALC =$self.VID$ $self.DN$

;

boltCalcVal=CALC CPF\_GetNeededBoltParam(B=self.L.value(), L=self.L.value())

;

@key=VID,DN

""",

"@VarDataDefault0",

)

This defines for the lapped flange now the parameter of the parametric construction data that has to be stored inside database or needed for editors, the needed fields and some calculated fields.

The **variants** can also be tested with the **AutoLISP®** function [**TESTACPSCRIPT**](#_TESTACPSCRIPT) and/or [**TESTACPSCRIPT1**](#_TESTACPSCRIPT).

The **variants** are loaded **on demand** to save memory. **Plant 3D** has created his own [**demand loader**](#_demand_loader) for the **variants**.

The **variants** are stored after installation in a [zip archive](#_variants.zip) and get read from the archive. That saves some time and doesn't fill the hard disk with much directories and files.

## Primitives

**Plant 3D** offers a lot of **Primitives** (basic parametric constructions) that are written inside C++ and are the interface to the CAD system.

There are:

* [**ARC3D**](#_ARC3D)defines a 'normal' elbow
* [**ARC3D2**](#_ARC3D2)defines a reduced elbow
* [**ARC3DS**](#_ARC3DS)defines a segmented elbow
* [**BOX**](#_BOX)defines a box
* [**CONE**](#_CONE)defines a cone and also a frustum
* [**CYLINDER**](#_CYLINDER)defines a 'normal' or an elliptical cylinder
* [**ELLIPSOIDHEAD**](#_ELLIPSOIDHEAD)defines a normalized ellipsoid head
* [**ELLIPSOIDHEAD2**](#_ELLIPSOIDHEAD2)defines an ellipsoid head (0.5 \* R)
* [**ELLIPSOIDSEGMENT**](#_ELLIPSOIDSEGMENT)defines an ellipsoid (football/rugby ball like)
* [**HALFSPHERE**](#_HALFSPHERE)defines a half sphere
* [**PYRAMID**](#_PYRAMID)defines a pyramid or a frustum of pyramid
* [**ROUNDRECT**](#_ROUNDRECT)defines a transition from a rectangle to a circle
* [**SPHERESEGMENT**](#_SPHERESEGMENT)defines a sphere segment
* [**TORISPHERICHEAD**](#_TORISPHERICHEAD)defines a normalized torispheric head
* [**TORISPHERICHEAD2**](#_TORISPHERICHEAD2)defines a torispheric head (small radius = 25.00)
* [**TORISPHERICHEADH**](#_TORISPHERICHEADH)defines a normalized torispheric head with height
* [**TORUS**](#_TORUS)defines a torus

Each created **variant** can also be used as **primitive** while creating other geometry bodies

### ARC3D

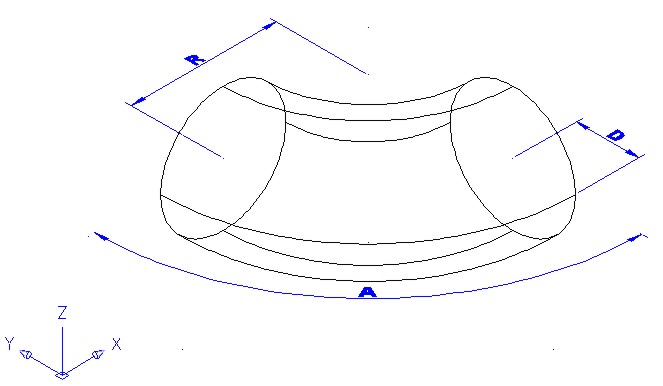
**ARC3D** defines a 'normal' elbow

a call from Python should be like

s=ARC3D(s, D, R, A)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **D** | the 1/2 diameter |
| **R** | the bend radius |
| **A** | the bend angle |



The base point is the intersection between the thought centerline trough booth ends

### ARC3D2

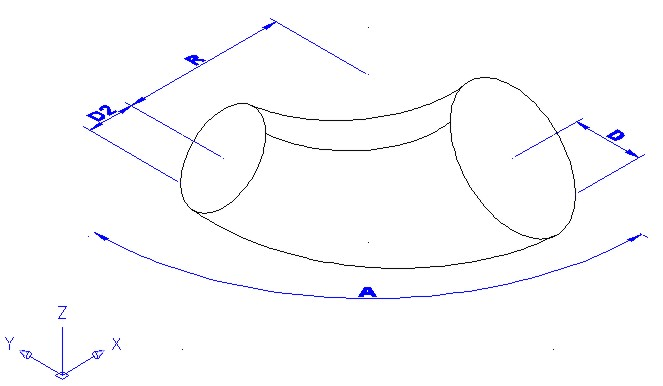
**ARC3D2** defines a reduced elbow (also a 'normal' elbow can be done with them)

a call from **Python** should be like

s=ARC3D2(s, D, D2, R, A)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **D** | the 1/2 diameter |
| **D2** | the second 1/2 diameter - if not set D ist used as D2 |
| **R** | the bend radius |
| **A** | the bend angle |



The base point is the intersection between the thought centerline trough booth ends

### ARC3DS

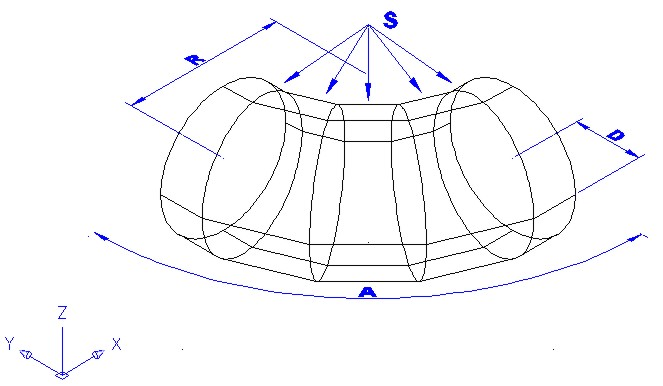
**ARC3DS** defines a segmented elbow

a call from **Python** should be like

s=ARC3DS(s, D, D2, R, A, S)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **D** | the 1/2 diameter |
| **R** | the bend radius |
| **A** | the bend angle |
| **S** | number of segments |



The base point is the intersection between the thought centerline trough booth ends

### BOX

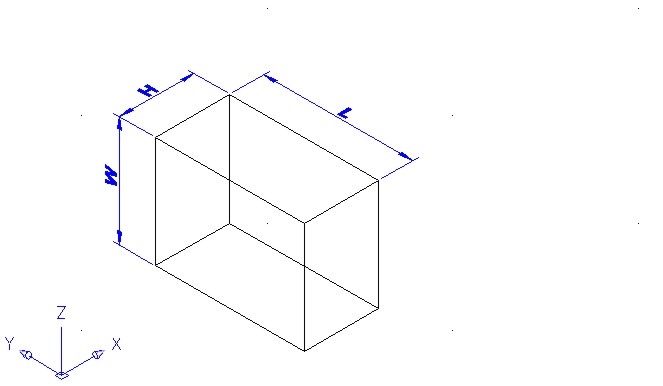
**BOX** defines a box

a call from **Python** should be like

s=BOX(s, L, W, H)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **L** | the box length |
| **W** | the box width |
| **H** | the box height |



The base point is the center of gravity

### CONE

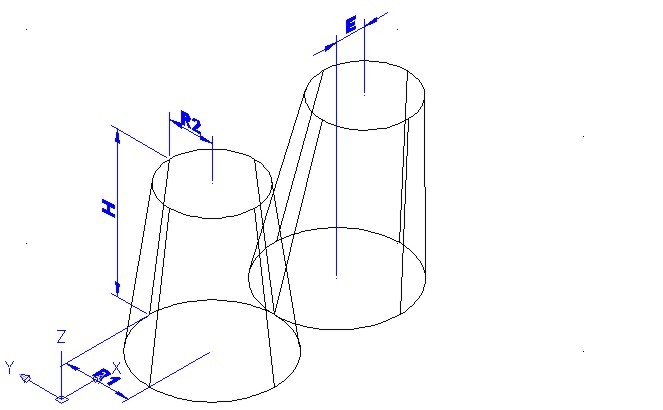
**CONE** defines a cone and also a frustum

a call from **Python** should be like

s=CONE(s, R1, R2, H, E)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R1** | the bottom radius |
| **R2** | the upper radius - if 0.0 a full cone,  if > 0.0 a frustum is created |
| **H** | the height |
| **E** | the eccentricity between upper and bottom center |



The base point is the center of bottom

### CYLINDER

**CYLINDER** defines a 'normal' or an elliptical cylinder

a call from Python should be like

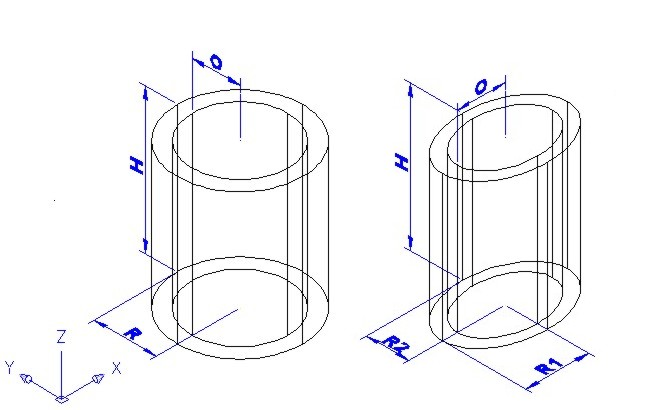
s=CYLINDER(s, R, H, O)

or

s=CYLINDER(s, R1, R2, H, O)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |
| **R1** | the 1. radius |
| **R2** | the 2. radius |
| **H** | the height |
| **O** | the hole radius |



The base point is the center of bottom

### ELLIPSOIDHEAD

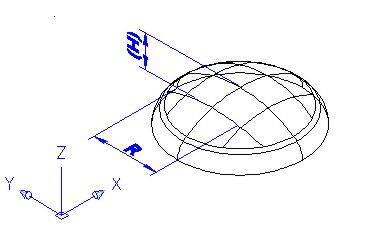
**ELLIPSOIDHEAD** defines a normalized ellipsoid head

a call from **Python** should be like

s=ELLIPSOIDHEAD(s, R)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |



The base point is the center of bottom

with obj[**.parameters**](#_.parameters)() we can get the height (H) of the primitive

Calculates the intersection iteratively. **ELLIPSOIDHEAD** is formed as an ellipsis. The body is interpolated with cones which are used to find the intersection points.

### ELLIPSOIDHEAD2

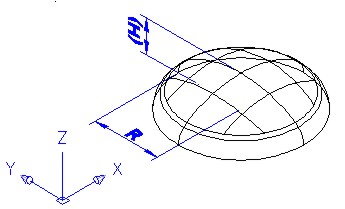
**ELLIPSOIDHEAD2** defines a ellipsoid head (0.5 \* R)

a call from **Python** should be like

s=ELLIPSOIDHEAD2(s, R)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |



The base point is the center of bottom

with obj[**.parameters**](#_Request_function:_.parameters)() we can get the height (H) of the primitive

Calculates the intersection iteratively. **ELLIPSOIDHEAD2** is formed as an ellipsis with the main axes a = radius and b = 0.5\*radius .The body is interpolated with cones which are used to find the intersection points.

### ELLIPSOIDSEGMENT

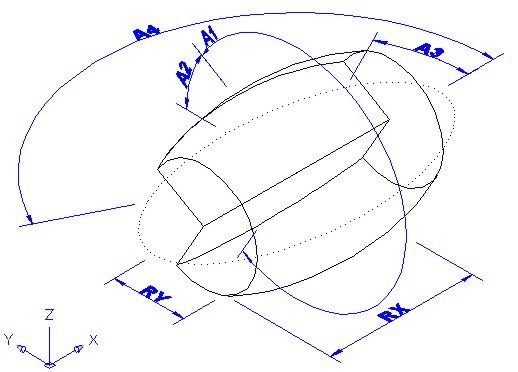
**ELLIPSOIDSEGMENT** defines an ellipsoid body (like a *football*/*rugby ball*)

a call from **Python** should be like

s=ELLIPSOIDSEGMENT(s, RX, RY, A1, A2, A3, A4)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **RX** | big ellipsoid axis |
| **RY** | small ellipsoid axis |
| **A1** | complete rotation angle |
| **A2** | start angle of rotation |
| **A3** | start angle of ellipse |
| **A4** | end angle of ellipse |



The base point is the center of gravity

### HALFSPHERE

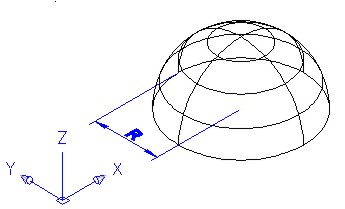
**HALFSPHERE** defines a half sphere

a call from **Python** should be like

s=HALFSPHERE(s, R)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |



The base point is the center of bottom

### PYRAMID

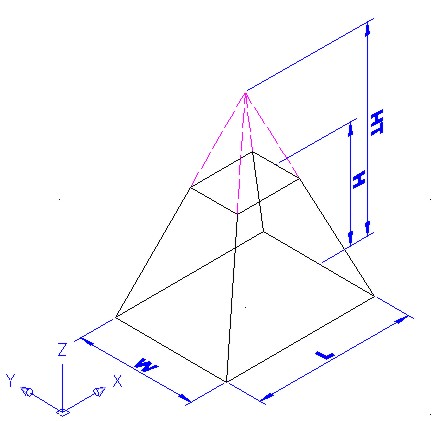
**PYRAMID** defines a pyramid or a frustum of pyramid

a call from **Python** should be like

s=PYRAMID(s, L, W, H, HT)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **L** | the length |
| **W** | the width |
| **H** | the frustum height if **HT** isn't set it's the total pyramid height |
| **HT** | the total pyramid height |



The base point is the center of the bottom rectangle

### ROUNDRECT

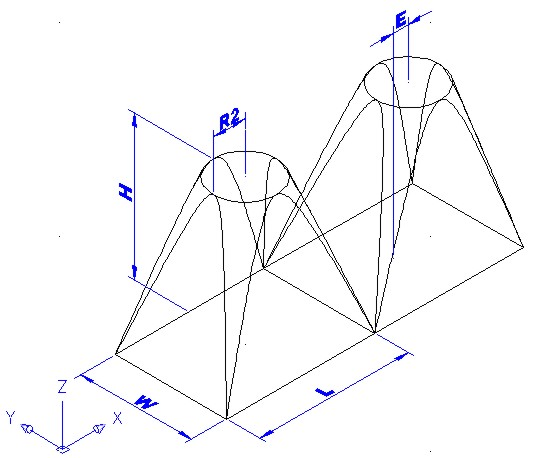
**ROUNDRECT** defines a transition from a rectangle to a circle

a call from **Python** should be like

s=ROUNDRECT(s, L, W, H, R2, E)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **L** | the length |
| **W** | the width |
| **H** | the height |
| **R2** | the circle radius |
| **E** | the eccentricity between upper and bottom center |



The base point is the center of the rectangle

### SPHERESEGMENT

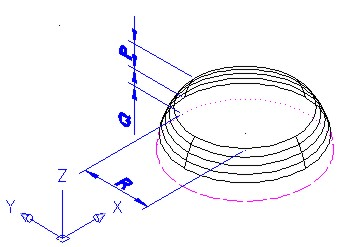
**SPHERESEGMENT** defines a sphere segment

a call from Python should be like

s=SPHERESEGMENT(s, R, P, Q)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the sphere radius |
| **P** | the segment height |
| **Q** | the height where the segment starts from center |



The base point is the center of the lower part of the segment.

### TORISPHERICHEAD

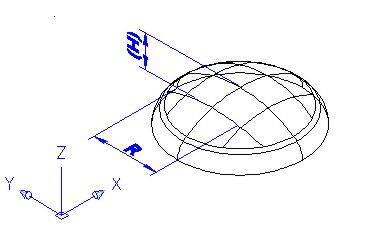
**TORISPHERICHEAD** defines a normalized torispheric head

a call from **Python** should be like

s=TORISPERICHEAD(s, R)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |



The base point is the center of bottom

with obj[**.parameters**](#_Request_function:_.parameters)() we can get the height (H) of the primitive.

### TORISPHERICHEAD2

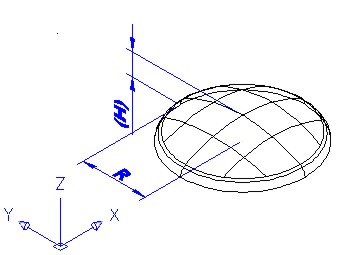
**TORISPHERICHEAD2** defines a torispheric head (small radius = 25.00)

a call from **Python** should be like

s=TORISPERICHEAD2(s, R)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |



The base point is the center of bottom

with obj[**.parameters**](#_Request_function:_.parameters)() we can get the height (H) of the primitive.

### TORISPHERICHEADH

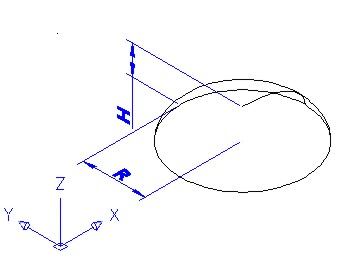
**TORISPHERICHEADH** defines a normalized torispheric head with height

a call from **Python** should be like

s=TORISPERICHEADH(s, R, H)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R** | the radius |
| **H** | the height |



The base point is the center of bottom.

### TORUS

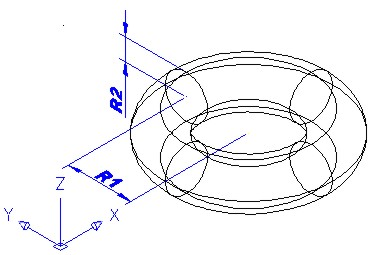
**TORUS** defines a torus

a call from **Python** should be like

s=TORUS(s, R1, R2)

The parameters means:

|  |  |
| --- | --- |
| **s** | the main object |
| **R1** | the outer radius |
| **R2** | the inner radius |



The base point is the intersection between the thought centerline trough booth ends.

## Member Functions

**Plant 3D** offers some member functions to modify objects (primitives) like rotate, move,...

There are also some member request functions to get additional information from the object.

* **Modifier functions**
  + obj[**.uniteWith**](#_Modifier_function:_.uniteWith)unites 2 objects
  + obj[**.subtractFrom**](#_Modifier_function:_.subtractFrom)subtracts 1 object from another 1
  + obj[**.intersectWith**](#_Modifier_function:_.intersectWith)creates an intersection of 2 objects
  + obj[**.erase**](#_Modifier_function:_.erase)removes an object from memory
  + obj[**.rotateX**](#_Modifier_function:_.rotateX)rotate the object round the X-axis
  + obj[**.rotateY**](#_Modifier_function:_.rotateY)rotate the object round the Y-axis
  + obj[**.rotateZ**](#_Modifier_function:_.rotateZ)rotate the object round the Z-axis
  + obj[**.translate**](#_Modifier_function:_.translate)moves the object
  + obj[**.setTransformationMatrix**](#_Modifier_function:_.setTransformati)set the object's transformation matrix
  + obj[**.setPoint**](#_Modifier_function:_.setPoint)append connection point to the body
* **Request functions**
  + obj[**.parameters**](#_Request_function:_.parameters)return the object's construction parameters
  + obj[**.transformationMatrix**](#_Request_function:_.transformationMa)return the object's current transformation matrix
  + obj[**.numberOfPoints**](#_Request_function:_.numberOfPoints)return number of (connection) points
  + obj[**.pointAt**](#_Request_function:_.pointAt)return the position of a connection point
  + obj[**.directionAt**](#_Request_function:_.directionAt)return the direction at a connection point

### .uniteWith

The modifier function **.uniteWith** unites the calling object ***obj*** with the second object ***oobj***.

The second object ***oobj*** is given to the function as argument.

The result is set to ***obj***.

After the unite the second object ***oobj*** has to be removed from memory with [**.erase**](#_.erase)

a call from **Python** should be like

obj.uniteWith(oobj)

It looks like:

def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", \*\*kw):

O=CON\_OF\_DIV(D2)/2.0

o1=[CYLINDER](#_CYLINDER)(s, R=D2/2.0, H=L-B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)

o0**.uniteWith(o1)**

o1[.erase](#_Modifier_function:_.erase)()

o2=[CYLINDER](#_CYLINDER)(s, R=O, H=L-B, O=0.0) )[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0[.subtractFrom](#_.subtractFrom)(o2)

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((L, 0, 0), ( 1, 0, 0))

### .subtractFrom

The modifier function **.subtractFrom** subtracts the second object ***oobj*** from the calling object ***obj***.

The second object ***oobj*** is given to the function as argument.

The result is set to ***obj***.

After the subtract the second object ***oobj*** has to be removed from memory with [**.erase**](#_Modifier_function:_.erase).

a call from **Python** should be like

obj.subtractFrom(oobj)

It looks like:

def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", \*\*kw):

O=CON\_OF\_DIV(D2)/2.0

o1=[CYLINDER](#_CYLINDER)(s, R=D2/2.0, H=L-B, O=O)[.rotateY](#_.rotateY)(90)[.translate](#_.translate)((B, 0, 0))

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)

o0[.uniteWith](#_.uniteWith)(o1)

o1[.erase](#_Modifier_function:_.erase)()

o2=[CYLINDER](#_CYLINDER)(s, R=O, H=L-B, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0**.subtractFrom(**o2**)**

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_Modifier_function:_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((L, 0, 0), ( 1, 0, 0))

### .intersectWith

The modifier function **.intersectWith** creates an intersect object with the calling object ***obj*** and the second object ***oobj***.

The second object ***oobj*** is given to the function as argument.

The result is set to ***obj***.

After the intersect the second object ***oobj*** has to be removed from memory with [**.erase**](#_Modifier_function:_.erase).

a call from **Python** should be like

obj.intersectWith(oobj)

It looks like:

def CADAPT\_SQ2RO\_Sub(s, LL=715.0, LW=700.0, R=225.0, OL=919.19, OW=698.28, H=2515.45, ID="CADAPT\_SQ2RO\_Sub", \*\*kw):

TL=R\*2.0

TW=R\*2.0

RB=sqrt(pow(LL, 2.0)+pow(LW, 2.0))/2.0

RA=90.0-asDegrees(atan2(OL, OW))

RE=sqrt(pow(OL, 2.0) + pow(OW, 2.0))

o1=[CONE](#_CONE)(s, R1=RB, R2=R, H=H, E=RE)[.rotateZ](#_.rotateZ)(RA)

o2=CADAPT\_SQ2SQ\_Sub(s, LL=LL, LW=LW, TL=TL, TW=TW, OL=OL, OW=OW, H=H)

o1**.intersectWith(**o2**)**

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_Modifier_function:_.setPoint)((0.0, 0.0, 0.0), (0.0, 0.0, -1.0))

s[.setPoint](#_Modifier_function:_.setPoint)((OL, OW, H ), (0.0, 0.0, 1.0))

### .erase

The modifier function **.erase** has always to be called after [**.uniteWith**](#_Modifier_function:_.uniteWith), [**.subtractFrom**](#_Modifier_function:_.subtractFrom) and [**.intersectWith**](#_.intersectWith).

**.erase** removes ***obj*** from the memory.

a call from **Python** should be like

obj.erase()

It looks like:

def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", \*\*kw):

O=CON\_OF\_DIV(D2)/2.0

o1=[CYLINDER](#_CYLINDER)(s, R=D2/2.0, H=L-B, O=O)[.rotateY](#_.rotateY)(90)[.translate](#_.translate)((B, 0, 0))

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)

o0[.uniteWith](#_.uniteWith)(o1)

o1**.erase()**

o2=[CYLINDER](#_CYLINDER)(s, R=O, H=L-B, O=0.0) )[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0[.subtractFrom](#_.subtractFrom)(o2)

o2**.erase()**

s[.setPoint](#_Modifier_function:_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((L, 0, 0), ( 1, 0, 0))

### .rotateX

The modifier function **.rotateX** rotates the calling object ***obj*** around the X-axis.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

obj.rotateX(a)

It looks like:

def CSGC004(s, D=116.0, L=30.0, W=4.0, K=0.0, H=95.0, M1=10.0, M2=0.0, ID="CSGC004", \*\*kw):

R1=D/2.0

R2=R1+W

H1=H-M2

if H1<(R2+W) or M2==0.0:

H1=R2+W

KWdt=M1\*tan(aqa.math.asRadiants(30.0))

H2=H-H1

o1a=[BOX](#_BOX)(s, L=M1, W=KWdt, H=H2)[.translate](#_Modifier_function:_.translate)((0.0-(H2/2.0)-H1, 0.0, 0.0))

o1b=[BOX](#_BOX)(s, L=M1, W=KWdt, H=H2)[.translate](#_Modifier_function:_.translate)((0.0-(H2/2.0)-H1, 0.0, 0.0))**.rotateX(** 60.0**)**

o1c=[BOX](#_BOX)(s, L=M1, W=KWdt, H=H2)[.translate](#_Modifier_function:_.translate)((0.0-(H2/2.0)-H1, 0.0, 0.0))**.rotateX(**-60.0**)**

o2=CSGC003(s, D=D, L=L, W=W, K=K, H=H1-R1)

o1a[.uniteWith](#_.uniteWith)(o1b)

o1b[.erase](#_Modifier_function:_.erase)()

o1a[.uniteWith](#_.uniteWith)(o1c)

o1c[.erase](#_Modifier_function:_.erase)()

o1a[.rotateZ](#_.rotateZ)(180.0)

o1a[.uniteWith](#_.uniteWith)(o2)

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_Modifier_function:_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((0, 0, 0), (1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((H, 0, 0), (1, 0, 0))

### .rotateY

The modifier function **.rotateY** rotates the calling object obj around the Y-axis.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

obj.rotateY(a)

It looks like:

def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", \*\*kw):

O=CON\_OF\_DIV(D2)/2.0

o1=[CYLINDER](#_CYLINDER)(s, R=D2/2.0, H=L-B, O=O)**.rotateY(**90**)**[.translate](#_.translate)((B, 0, 0))

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=B, O=O)**.rotateY(**90**)**

o0[.uniteWith](#_.uniteWith)(o1)

o1[.erase](#_Modifier_function:_.erase)()

o2=[CYLINDER](#_CYLINDER)(s, R=O, H=L-B, O=0.0)**.rotateY(**90**)**[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0[.subtractFrom](#_.subtractFrom)(o2)

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((L, 0, 0), ( 1, 0, 0))

### .rotateZ

The modifier function **.rotateZ** rotates the calling object obj around the Z-axis.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

obj.rotateZ(a)

It looks like:

def CSGC004(s, D=116.0, L=30.0, W=4.0, K=0.0, H=95.0, M1=10.0, M2=0.0, ID="CSGC004", \*\*kw):

R1=D/2.0

R2=R1+W

H1=H-M2

if H1<(R2+W) or M2==0.0:

H1=R2+W

KWdt=M1\*tan(aqa.math.asRadiants(30.0))

H2=H-H1

o1a=[BOX](#_BOX)(s, L=M1, W=KWdt, H=H2)[.translate](#_Modifier_function:_.translate)((0.0-(H2/2.0)-H1, 0.0, 0.0))

o1b=[BOX](#_BOX)(s, L=M1, W=KWdt, H=H2)[.translate](#_Modifier_function:_.translate)((0.0-(H2/2.0)-H1, 0.0, 0.0))[.rotateX](#_Modifier_function:_.rotateX)( 60.0)

o1c=[BOX](#_BOX)(s, L=M1, W=KWdt, H=H2)[.translate](#_Modifier_function:_.translate)((0.0-(H2/2.0)-H1, 0.0, 0.0))[.rotateX](#_Modifier_function:_.rotateX)(-60.0)

o2=CSGC003(s, D=D, L=L, W=W, K=K, H=H1-R1)

o1a[.uniteWith](#_.uniteWith)(o1b)

o1b[.erase](#_Modifier_function:_.erase)()

o1a[.uniteWith](#_.uniteWith)(o1c)

o1c[.erase](#_Modifier_function:_.erase)()

o1a.rotateZ(180.0)

o1a[.uniteWith](#_.uniteWith)(o2)

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_Modifier_function:_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((0, 0, 0), (1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((H, 0, 0), (1, 0, 0))

### .translate

The modifier function **.translate** moves the calling object ***obj*** along the given vector ***v***.

The argument ***v*** can be a **mPoint**, **mVector** or a **3-Tupel *(x, y, z)***.

a call from **Python** should be like

obj.translate(v)

It looks like:

def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", \*\*kw):

O=CON\_OF\_DIV(D2)/2.0

o1=[CYLINDER](#_CYLINDER)(s, R=D2/2.0, H=L-B, O=O)[.rotateY](#_.rotateY)(90)**.translate(**(B, 0, 0)**)**

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)

o0[.uniteWith](#_.uniteWith)(o1)

o1[.erase](#_Modifier_function:_.erase)()

o2=[CYLINDER](#_CYLINDER)(s, R=O, H=L-B, O=0.0) )[.rotateY](#_Modifier_function:_.rotateY)(90)**.translate(**(B, 0, 0)**)**

o0[.subtractFrom](#_.subtractFrom)(o2)

o2[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_.setPoint)((0, 0, 0), (-1, 0, 0))

s[.setPoint](#_Modifier_function:_.setPoint)((L, 0, 0), ( 1, 0, 0))

### .setTransformationMatrix

The modifier function **.setTransformationMatrix** sets the object's transformation matrix ***t***.

a call from **Python** should be like

obj.setTransformationMatrix(t)

It looks like:

def CPNO(s, D=114.3, R=300.0, L=100.0, D2=0.0, A=90.0, OF=0.0, ID="CPNO", \*\*kw):

vC=CPNO\_util(D=D, D2=D2, OF=OF, L=L, R=R, A=A)

R=vC[0]/2.0

R2=vC[1]/2.0

o1=[ARC3D](#_ARC3D)(s, D=R, R=vC[6], A=vC[2])

p2 = o1[.pointAt](#_.pointAt)(1)

if vC[2] == 90.0:

xMove = 0.0 - p2.x()

yMove = vC[7] - p2.y()

else:

xMove = 0.0 - p2.x() - (L \* cos(asRadiants(vC[2])))

yMove = vC[7] - p2.y() - (L - (L \* sin(asRadiants(vC[2]))))

t1 = mTransform()[.translate](#_Modifier_function:_.translate)((xMove, yMove, 0.0))

o1**.setTransformationMatrix(**t1**)**

s[.setPoint](#_.setPoint)((0, 0, 0), (0, -1, 0))

s[.setPoint](#_.setPoint)(t1 \* o1[.pointAt](#_.pointAt)(1), o1[.directionAt](#_.directionAt)(1))

if vC[5]:

d1 = [ARC3D](#_ARC3D)(s, D=R-vC[4], R=vC[6], A=vC[2])

d1**.setTransformationMatrix(**t1**)**

o1[.subtractFrom](#_.subtractFrom)(d1)

d1[.erase](#_Modifier_function:_.erase)()

o3=[CYLINDER](#_CYLINDER)(s, R=R2, H=vC[6]\*2.0, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(270)[.translate](#_Modifier_function:_.translate)((vC[6], 0, 0))

if vC[7] < vC[6]:

o2 = [BOX](#_BOX)(s, H=vC[6]\*2.0, W=vC[0]\*2.0, L=vC[6])[.translate](#_Modifier_function:_.translate)(((vC[6]/2.0), -(vC[6]/2.0), 0.0))

o3[.uniteWith](#_.uniteWith)(o2)

o2[.erase](#_Modifier_function:_.erase)()

o1[.subtractFrom](#_.subtractFrom)(o3)

o3[.erase](#_Modifier_function:_.erase)()

### .setPoint

The modifier function **.setPoint**  insert a point at position ***p*** and set the direction ***v*** to the next part inside this point.

Alternative we can also set a rotation angle ***a*** (most needed for elliptical flanges to get the right alignment)

The argument ***p*** and ***v*** can be a **mPoint**, **mVector** or a **3-Tupel *(x, y, z)***.

The rotation angle ***a*** has to be given as degree.

a call from **Python** should be like

obj.setPoint(p,v)

or

obj.setPoint(p,v,a)

It looks like:

def CPMB(s, L=54.0, B=22.0, D1=220.0, D2=114.3, ID="CPMB", \*\*kw):

O=CON\_OF\_DIV(D2)/2.0

o1=[CYLINDER](#_CYLINDER)(s, R=D2/2.0, H=L-B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0=[CYLINDER](#_CYLINDER)(s, R=D1/2.0, H=B, O=O)[.rotateY](#_Modifier_function:_.rotateY)(90)

o0[.uniteWith](#_.uniteWith)(o1)

o1[.erase](#_Modifier_function:_.erase)()

o2=[CYLINDER](#_CYLINDER)(s, R=O, H=L-B, O=0.0) )[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((B, 0, 0))

o0[.subtractFrom](#_.subtractFrom)(o2)

o2[.erase](#_Modifier_function:_.erase)()

s**.setPoint(**(0, 0, 0), (-1, 0, 0)**)**

s**.setPoint(**(L, 0, 0), ( 1, 0, 0), 0**)**

### .parameters

The request function **.parameters** returns the object's construction parameters

a call from **Python** should be like

obj.parameters()

It looks like:

def CPC(s, D=114.30, L=64.00, DISP=0, OF=-1.0, ID="CPC", \*\*kw):

if DISP == 0:

o2 = [TORISPHERICHEAD](#_TORISPHERICHEAD)(s, R=D/2.0)[.rotateY](#_Modifier_function:_.rotateY)(90.0)

vT = o2**.parameters()**

vTH = float(vT["H"])

if L - vTH < 0.0:

o2[.erase](#_Modifier_function:_.erase)()

o2 = [TORISPHERICHEAD](#_TORISPHERICHEAD)(s, R=D/2.0, H=L)[.rotateY](#_Modifier_function:_.rotateY)(90.0)

vTH = L

o2[.translate](#_Modifier_function:_.translate)((L-vTH, 0.0, 0.0))

else:

vTH = CON\_OF\_MULT(D)-D

o2 = [CYLINDER](#_CYLINDER)(s, R=(D+vTH)/2.0, H=vTH, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(90)[.translate](#_Modifier_function:_.translate)((L-vTH, 0.0, 0.0))

if L - vTH > 0.0:

o1 = [CYLINDER](#_CYLINDER)(s, R=D/2.0, H=L-vTH, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(90.0)

o1[.uniteWith](#_.uniteWith)(o2)

o2[.erase](#_Modifier_function:_.erase)()

if not OF == 0.0:

if OF == -1:

O = CON\_OF\_DIV(D)

else:

O = D-(OF \* 2.0)

L2 = L-(D-O)+OF

if DISP == 0:

o4 = [TORISPHERICHEAD](#_TORISPHERICHEAD)(s, R=O/2.0)[.rotateY](#_Modifier_function:_.rotateY)(90.0)

o4[.translate](#_Modifier_function:_.translate)((L-vTH, 0.0, 0.0))

else:

o4 = None

if L - vTH > 0.0:

o3 = [CYLINDER](#_CYLINDER)(s, R=O/2.0, H=L-vTH, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(90.0)

if o4:

o3[.uniteWith](#_.uniteWith)(o4)

o4[.erase](#_Modifier_function:_.erase)()

else:

o3 = o4

o1[.subtractFrom](#_.subtractFrom)(o3)

o3[.erase](#_Modifier_function:_.erase)()

s[.setPoint](#_.setPoint)((0.0, 0.0, 0.0), (-1.0, 0.0, 0.0), 0.0)

### .transformationMatrix

The request function **.**transformationMatrix returns the object's current transformation matrix

a call from **Python** should be like

obj.transformationMatrix()

(sorry, not used inside any of the existing parametric constructions so I can’t show a real used example)

### .numberOfPoints

The request function **.numberOfPoints** returns the object's number of ([connection](#_Modifier_function:_.setPoint)) points

a call from **Python** should be like

obj.numberOfPoints()

(sorry, not used inside any of the existing parametric constructions so I can’t show a real used example)

### .pointAt

The request function **.pointAt** returns the position of connection point ***n*** as mPoint.

If **inECS** is set to ***1*** the mPoint is not transformed to WCS - default for **inECS** is ***0***.

a call from **Python** should be like

obj.pointAt(n, inECS)

It looks like:

def CPNO(s, D=114.3, R=300.0, L=100.0, D2=0.0, A=90.0, OF=0.0, ID="CPNO", \*\*kw):

vC=CPNO\_util(D=D, D2=D2, OF=OF, L=L, R=R, A=A)

R=vC[0]/2.0

R2=vC[1]/2.0

o1=[ARC3D](#_ARC3D)(s, D=R, R=vC[6], A=vC[2])

p2 = o1**.pointAt(**1**)**

if vC[2] == 90.0:

xMove = 0.0 - p2.x()

yMove = vC[7] - p2.y()

else:

xMove = 0.0 - p2.x() - (L \* cos(asRadiants(vC[2])))

yMove = vC[7] - p2.y() - (L - (L \* sin(asRadiants(vC[2]))))

t1 = mTransform()[.translate](#_Modifier_function:_.translate)((xMove, yMove, 0.0))

o1[.setTransformationMatrix](#_.setTransformationMatrix)(t1)

s[.setPoint](#_.setPoint)((0, 0, 0), (0, -1, 0))

s[.setPoint](#_.setPoint)(t1 \* o1**.pointAt(**1**)**, o1[.directionAt](#_.directionAt)(1))

if vC[5]:

d1 = [ARC3D](#_ARC3D)(s, D=R-vC[4], R=vC[6], A=vC[2])

d1[.setTransformationMatrix](#_.setTransformationMatrix)(t1)

o1[.subtractFrom](#_.subtractFrom)(d1)

d1[.erase](#_Modifier_function:_.erase)()

o3=[CYLINDER](#_CYLINDER)(s, R=R2, H=vC[6]\*2.0, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(270)[.translate](#_Modifier_function:_.translate)((vC[6], 0, 0))

if vC[7] < vC[6]:

o2 = [BOX](#_BOX)(s, H=vC[6]\*2.0, W=vC[0]\*2.0, L=vC[6])[.translate](#_Modifier_function:_.translate)(((vC[6]/2.0), -(vC[6]/2.0), 0.0))

o3[.uniteWith](#_.uniteWith)(o2)

o2[.erase](#_Modifier_function:_.erase)()

o1[.subtractFrom](#_.subtractFrom)(o3)

o3[.erase](#_Modifier_function:_.erase)()

### .directionAt

The request function **.directionAt** returns the direction of connection point ***n*** as mVector.

If **inECS** is set to ***1*** the mVector is not transformed to WCS - default for **inECS** is ***0***.

a call from **Python** should be like

obj.directionAt(n, inECS)

It looks like:

**def CPNO(s, D=114.3, R=300.0, L=100.0, D2=0.0, A=90.0, OF=0.0, ID="CPNO", \*\*kw):**

**vC=CPNO\_util(D=D, D2=D2, OF=OF, L=L, R=R, A=A)**

**R=vC[0]/2.0**

R2=vC[1]/2.0

o1=[ARC3D](#_ARC3D)(s, D=R, R=vC[6], A=vC[2])

p2 = o1[.pointAt](#_.pointAt)(1)

if vC[2] == 90.0:

xMove = 0.0 - p2.x()

yMove = vC[7] - p2.y()

else:

xMove = 0.0 - p2.x() - (L \* cos(asRadiants(vC[2])))

yMove = vC[7] - p2.y() - (L - (L \* sin(asRadiants(vC[2]))))

t1 = mTransform()[.translate](#_Modifier_function:_.translate)((xMove, yMove, 0.0))

o1[.setTransformationMatrix](#_.setTransformationMatrix)(t1)

s[.setPoint](#_.setPoint)((0, 0, 0), (0, -1, 0))

s[.setPoint](#_.setPoint)(t1 \* o1[.pointAt](#_.pointAt)(1), o1**.directionAt(**1**)**)

if vC[5]:

d1 = [ARC3D](#_ARC3D)(s, D=R-vC[4], R=vC[6], A=vC[2])

d1[.setTransformationMatrix](#_.setTransformationMatrix)(t1)

o1[.subtractFrom](#_.subtractFrom)(d1)

d1[.erase](#_Modifier_function:_.erase)()

o3=[CYLINDER](#_CYLINDER)(s, R=R2, H=vC[6]\*2.0, O=0.0)[.rotateY](#_Modifier_function:_.rotateY)(270)[.translate](#_Modifier_function:_.translate)((vC[6], 0, 0))

if vC[7] < vC[6]:

o2 = [BOX](#_BOX)(s, H=vC[6]\*2.0, W=vC[0]\*2.0, L=vC[6])[.translate](#_Modifier_function:_.translate)(((vC[6]/2.0), -(vC[6]/2.0), 0.0))

o3[.uniteWith](#_.uniteWith)(o2)

o2[.erase](#_Modifier_function:_.erase)()

o1[.subtractFrom](#_.subtractFrom)(o3)

o3[.erase](#_Modifier_function:_.erase)()

## Functions

The **variants** need and offer some functions to link to **Plant 3D** or for testing:

There are:

* [**activate()**](#_activate)  
  activate the variant and defines the needed database fields
* [**TESTACPSCRIPT**](#_TESTACPSCRIPT)  
  an AutoLISP® function to see and test the variant parameters
* [**demand loader**](#_demand_loader)  
  Loader for the variants on demand

### activate

To use a variant definition as Plant 3D part we have to **activate** the function.

To get and convert the data from database we need also the field information. That is also done inside activate.

a call from **Python** should be like

activate(func, dbtype, defkey)

The parameters means:

|  |  |
| --- | --- |
| **func** | the **Python** function to activate (in this case a variant) |
| [**dbtype**](#_dbtype) | this is the database definition. this definition has to be done between **"""** and **"""** (each has 3 quotation marks) |
| **defkey** | database definition key (it’s always **"@VarDataDefault0"** inside **Plant 3D**) |

For example:

**activate(CPFLR,**

**"""**

**[Type AQA-VCPFLR]**

**VID=STRING,32**

**DN=STRING**

**L=LENGTH**

**D1=LENGTH**

**D2=LENGTH**

**Units=STRING,8**

**;**

**uniqId=CALC =$self.VID$ $self.DN$**

**;**

**boltCalcVal=CALC CPF\_GetNeededBoltParam(B=self.L.value(), L=self.L.value())**

**;**

**@key=VID,DN**

**""",**

**"@VarDataDefault0",**

**)**

This defines for the lapped flange now the needed fields and some calculated fields.

#### dbtype

This is the database / parameter definition.

This definition has to be done between **"""** and **"""** (each has 3 quotation marks)

And then it looks like:

**"""**

**[Type AQA-VCPFLR]**

**VID=STRING,32**

**DN=STRING**

**L=LENGTH**

**D1=LENGTH**

**D2=LENGTH**

**Units=STRING,8**

**;**

**uniqId=CALC =$self.VID$ $self.DN$**

**;**

**boltCalcVal=CALC CPF\_GetNeededBoltParam(B=self.L.value(), L=self.L.value())**

**;**

**@key=VID,DN**

**""",**

Its defined as:

1. Each dbtype definition starts with 3 quotation marks  
   **“””**
2. The next line describes the **variant** name and should start with **[AQA\_V** and end with **]**  
   **[Type AQA-V**CPFLR**]**  
   (the definition with **AQA-V** comes from the adapter which read ***ACPlant Designer*** parametric scripts and can’t changed now)
3. The next 2 lines are also always the same (come also from adapter and should be added for compatibility)  
   **VID=STRING,32  
   DN=STRING**
4. Now we define for each needed script argument an extra definition line  
   each line starts with the argument name then we use the = sign as delimiter and now we add the type.  
   After the definition we can also add char, numbers length , delimited with ‘**,**’, but not needed inside **Plant 3D** (**ACPlant Designer** used this definition to create database tables)  
   **DN=STRING  
   L=LENGTH  
   D1=LENGTH  
   D2=LENGTH  
   Units=STRING,8**  
   allowed values for the type are
   1. **LENGTH** for length values that also can be calculated to another unit (like mm -> in)
   2. **ANGLE** is used for set and real angle valve
   3. **INT** a real integer value
   4. **DOUBLE** a float value
   5. **STRING** a string
5. Now we can define some help fields like  
   **uniqId** (for calculating unique blocknames)  
   **boltCalcVal**,…  
   we can set here each parameter that we want and with **=CALC** we can call each function that we want and set the return value inside the wished parameter  
   **uniqId=CALC =$self.VID$ $self.DN$  
   boltCalcVal=CALC CPF\_GetNeededBoltParam(B=self.L.value(), L=self.L.value())**
6. The line **@key=** was also use inside **ACPlant Designer** for database creation and defined the table index  
   **@key=VID,DN**
7. Comments can be set with ‘**;**’
8. And now comes the ending 3 quotation marks  
   **“””**

### TESTACPSCRIPT / TESTACPSCRIPT1

To test and check the geometry body we have the **AutoLISP®** function **TESTACPSCRIPT** and **TESTACPSCRIPT1** created.

The first parameter at booth script types is the variants name (like "CPFLR")

The next parameters for **TESTACPSCRIPT** are couples of values: the fieldname and the needed value (like "D1" "300.5").

For **TESTACPSCRIPT1** we have to set all the parameters inside 1 string (like it’s done inside catalog/spec dbs: “D1=300.5”)

If a field isn't set the function takes the default values from the **variants** definition.

That looks like:

**(TESTACPSCRIPT "CPFLR")**

**(TESTACPSCRIPT "CPFLR" "D1" "300.5")**

**(TESTACPSCRIPT "CPFLR" "L" "40" "D1" "300.5" "D2" "88.9")**

Or

**(TESTACPSCRIPT1 "CPFLR")**

**(TESTACPSCRIPT1 "CPFLR" "D1=300.5")**

**(TESTACPSCRIPT1 "CPFLR" "L=40,D1=300.5,D2=88.9")**

Before we can use **TESTACPSCRIPT** or **TESTACPSCRIPT1** inside **Plant 3D** we have to load **PnP3dACPAdapter.arx**.

That can be done with

**(arxload "PnP3dACPAdapter.arx")**

or with the **appload** function

### demand loader

Approximately 20.000 different **variant** subroutines needs there time to load. To speed up the start procedure we created the **Variant Demand Loader**.

First we packed all the variants into an zip archive so we solved the operating system problems with the much directories - **Python** use this zip archive as package and get the files really fast.

At second we load the **variants** only if we need them. For that we created a *dummy* loader file that includes only links to the variants.

This file is compiled at build time to **variants.map**.

The contents of the file **dummy\_var.load** looks like (variant;python\_package\_like\_location):

**CPFBO;varmain.flangesub.cpfbo**

**CPFBR;varmain.flangesub.cpfbr**

**CPFLO;varmain.flangesub.cpflo**

**CPFLR;varmain.flangesub.cpflr**

**CPFWO;varmain.flangesub.cpfwo**

**CPFWO\_F\_SF;varmain.flangesub.cpfwo\_f\_sf**

**CPFWR;varmain.flangesub.cpfwr**

**CPFWR\_F\_SF;varmain.flangesub.cpfwr\_f\_sf**

**CPFWRI;varmain.flangesub.cpfwri**

**CPFWRI\_F\_SF;varmain.flangesub.cpfwri\_f\_sf**

**FLANGE;varmain.flangesub.cpf\_old**

**CPFW;varmain.flangesub.cpf\_old**

# variants.zip

All **variants** are organized inside a directory structure.

Normally each **variant** is defined inside 1 separate file.

Some **variant** have also sub files.

So we get very much directories and files that make some problems with slow hard disk (it slows down the machine).

To solve that we pack all the directories and variant files inside variant.zip. Python can handle this archive as package and is very fast.

What is all inside variant.zip:

1. the compile variant files (\*.pyc)
2. dimensioned images with resolution 640x640px per script (\*\_640.png)
3. basic thumbnail images per script with resolution
   1. 200x200px (\*\_200-png)
   2. 64x64px (\*\_64.png)
   3. 32x32px (\*\_32.png)
4. Script metadata (\*.xlm)

Variant.zip is build with **p3dmakeall.bat** (or you call [**amake.bat**](#_amake.bat) inside directory “**…\plant\Develop\Piping\3rdParty\ACP”**

## amake.bat

That **Plant 3D** can work with all the **variant** scripts we have to generate some files like [**variants.zip**](#_variants.zip), [**variants.map**](#_demand_loader) and **variants.xlm**.

**amake.bat** helps us to generate this files and looks like (batch content / comment not in batch):

@rem $Header: //depot/plant/Develop/PnID/Source/Components/amake.bat#2 $

@echo off

if (%ACTOP%) == () goto BADACADROOT

…set the base environment vars…

@set PYTHONPATH=%PIPING\_3RDPARTY\_ACP%\variants\aqa\varmap;%PIPING\_3RDPARTY\_PYTHON%\Source\Lib;%PIPING\_3RDPARTY\_PYTHON\_BIN%

@echo pnproot is %PNPROOT%

@echo PYTHON\_EXE is %PYTHON\_EXE%

@echo PIPING\_3RDPARTY\_ACP is %PIPING\_3RDPARTY\_ACP%

@rem -----------------------------------------------------------------------

@if not exist %PIPING\_3RDPARTY\_ACP%\buildzip.py goto BADBLDROOT

@cd %PIPING\_3RDPARTY\_ACP%

@set CONTENTTOOLPATH=%PIPING\_3RDPARTY\_ACP%\ContentTools

@set XLSREADERPATH=%PIPING\_3RDPARTY\_ACP%\ContentTools\XmlCreator

@if /i "%1" == "clean" goto makeClean

… remove existing error files

@rm -f %CONTENTTOOLPATH%\ContentTools.err

… build xlm creator tools

@devenv /useenv %CONTENTTOOLPATH%\ContentTools.sln /build %BLDTYPE% /out %CONTENTTOOLPATH%\ContentTools.err

@if exist %CONTENTTOOLPATH%\ContentTools.err typeerr.exe %CONTENTTOOLPATH%\ContentTools.err

%CBIN%\XmlCreator -img %PIPING\_CONTENT%\Templates\Piping.dcfx %PIPING\_CONTENT%\Templates\PipingClassToImgRel.xml

… compile all the python files, create also variants.map

%PYTHON\_EXE% %PIPING\_3RDPARTY\_ACP%\buildzip.py --build

… build all the needed xlm files

%CBIN%\XmlCreator -acp %PIPING\_3RDPARTY\_ACP%\variants\variant\_parameter\_doc.xlsx %PIPING\_3RDPARTY\_ACP%\dummy\_var.load %PIPING\_3RDPARTY\_ACP%\log.txt %PIPING\_3RDPARTY\_ACP%\variants.xml

… now create variants.zip

%PYTHON\_EXE% %PIPING\_3RDPARTY\_ACP%\buildzip.py --zip

@goto DONE

:makeClean

@cd %PIPING\_3RDPARTY\_ACP%

@%CBIN%\XlsToXml.exe -d %PIPING\_3RDPARTY\_ACP%\log.txt

@devenv /useenv %CONTENTTOOLPATH%\ContentTools.sln /clean %BLDTYPE%

@rm -f %PIPING\_3RDPARTY\_ACP%\intermediaryXml.xml %PIPING\_3RDPARTY\_ACP%\scriptInXls.map

%PYTHON\_EXE% %PIPING\_3RDPARTY\_ACP%\buildzip.py --clean

@rm -f %PIPING\_3RDPARTY\_ACP%\VARIANTS.zip %PIPING\_3RDPARTY\_ACP%\VARIANTS.map

@goto DONE

@rem -----------------------------------------------------------------------

:BADBLDROOT

@echo The PIPING\_3RDPARTY\_ACP environment variable has incorrect value

@goto DONE

@rem -----------------------------------------------------------------------

:BADACADROOT

echo The ACTOP environment variable has incorrect value

goto done

:DONE

@set PYTHONPATH=