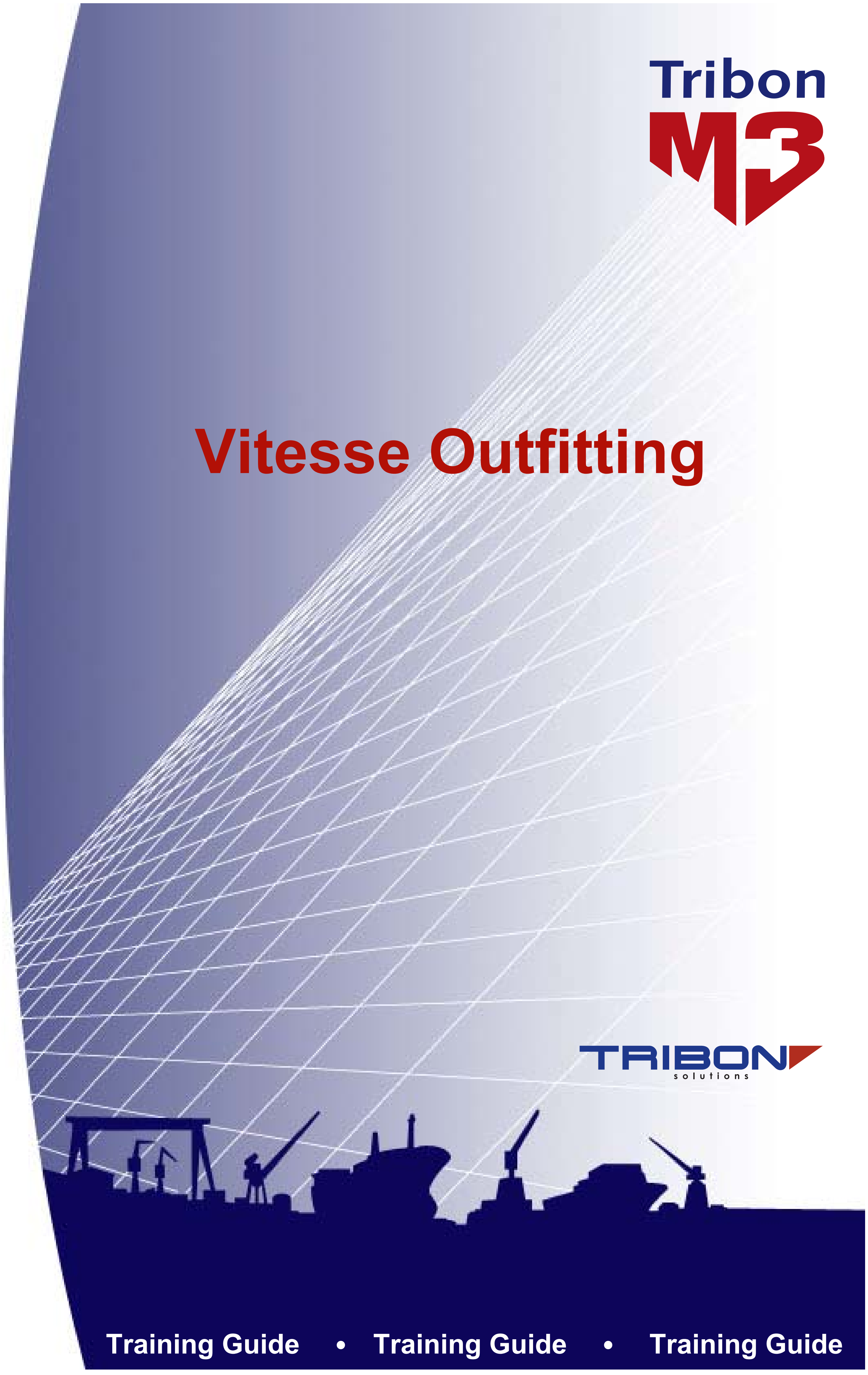
Tribon M3

Vitesse Outfitting

TRIBON solutions



Training Guide • Training Guide • Training Guide

Revision Log

Date	Page(s)	Revision	Description of Revision	Release
5/07/2004	All	T.Lisowski	General Update for M3	M3
21/07/2004	All	T.Lisowski	General Update for M3 SP1	M3 SP1

Updates

Updates to this manual will be issued as replacement pages and a new Update History Sheet complete with instructions on which pages to remove and destroy, and where to insert the new sheets. Please ensure that you have received all the updates shown on the History Sheet.

All updates are highlighted by a revision code marker, which appears to the left of new material.

Suggestion/Problems

If you have a suggestion about this manual, the system to which it refers, or are unfortunate enough to encounter a problem, please report it to the training department at

Fax +44 191 201 0001
Email training@tribon.com

Printed by Tribon Solutions (UK) Ltd on 17 January 2005

Contents

Chapter 1

1 Introduction

Tribon Vitesse contains a set of modules supporting the creation of outfitting objects. The flexibility of Vitesse macros can help to automate various modelling tasks concerning volumes, equipments, structures, cables, cableways, pipes and ventilation objects. The following chapters will guide you through the study of the Vitesse abilities concerning the modelling of the particular kind of outfitting objects.

1.1 Objectives

The aim of the course is to provide the knowledge required for creating Tribon Vitesse macros. After completing the course, the user should be in a position to immediately start using Vitesse system.

The objective is to become familiar with the Tribon Vitesse functions in the area of:

- modelling volumes;
- Structure Modelling;
- Pipe/Ventilation Modelling;
- Cable Modelling (cables, cableways and penetrations);
- modelling equipments.

1.2 Prerequisites for training course

During the training, the participants require access to a PC with an installation of the Tribon M3 system with the Python source editor (e.g. PythonWin or ConTEXT). All participants should have completed the Vitesse Basic training course.

The following skills are required from at least one person in each group:

- Working knowledge of Tribon Drafting system (including the modelling of volumes and equipments),
- Basic knowledge of Tribon Data Extraction system,
- Working knowledge of the Tribon Structure Modelling system,
- Working knowledge of the Tribon Cable Modelling system,
- Working knowledge of the Tribon Cable Modelling system,
- Working knowledge of the Pipe/Ventilation Modelling system,
- Understanding of the Tribon component concept,
- Basic programming experience.

A copy of the training project must be installed for this training prior to the trainer arriving.

1.3 Training methods

Presentations, demonstrations and practical exercises.

1.4 Overview

All outfitting modelling activities are based on the concept of components. The shape of components may be defined using volumes. Tribon M3 for the first time provides the Vitesse interface to the volume modelling abilities of Tribon. Unplaced volumes can be build from primitives. We can also set some basic properties of the primitives, and place copies of volumes in the ship's space.

Some components representing devices are placed in the model as equipments. Tribon Vitesse can create equipments, set its properties and place them in the ship's model. They provide an appropriate information for production, unlike placed volumes, which are 3D shapes only.

Tribon Vitesse supports also fully the creation of structures. They can be initialised and built by adding various kinds of parts (plates, profiles, bent plates, bent profiles, miscellaneous components). Production information can be supplied, and the structure can be marked as ready and split. The ability to use Data Extraction, and the modelling environment of Tribon system makes possible to create automatically various foundations, pipe supports or cableway hangers. A separate product, called Pipe Support, has been developed entirely in Vitesse to enable the creation of standardised pipe supports and hangers.

Similar scope of functionality is available for creating pipe and ventilation objects, although the Pipe Modelling interface is significantly more complex. Not only the typical parts can be added or inserted, but also the non-physical parts (like joints or weld gaps) can be handled. There are functions dealing with connections between pipe parts, or handling pipe spools. Additionally, pipe routes can be generated, the frame parts can be dressed with material, and the complete production information can be prepared and generated.

Finally, a similar support can be found for cables and cableways, including the penetrations. An effort has been made to keep the approach to the functionalities available in outfitting Vitesse modules as similar, as possible, which makes it easier to learn for the students.

1.5 Duration

3 days

1.6 Using this document

Certain text styles are used to indicate special situations throughout this document; here is a summary;

All examples will be displayed as bold text in the **`Courier New font`**, and the program output will be indented to the right with respect to the user input. System prompts should be bold and italic in single quotes, i.e. '***Choose function***'.

Annotation for trainees benefit:

Additional information

Refer to other documentation

Larger examples and solutions to the exercises have not been included in the Training Guide, but can be found in the folder '**Vitesse Outfitting Training**' under **SB_PYTHON** of the training project. References to these examples are annotated with:

Refer to the training examples

In order to keep the examples short, it is assumed, that the proper modules have been imported using the **`import`** statement. Missing parts of the code are often replaced by ellipsis '…' and a comment describing what has been omitted.

2 Placed and unplaced volumes

2.1 Introduction

Volumes are 3D shapes, composed from the predefined, parameterised 3D primitives, such as: the parallelepiped, the truncated cone, the spherical segment, the torus segment, etc. They are used for defining the shape of various Tribon components, that in turn, define equipments or the parts of structures and pipes. Additionally, the volumes can be placed in the ship's space, to reserve space for some equipment. It must be noted, however, that such placed volumes do not carry any production information about the equipment – they are just 3D shapes, nothing more.

The volumes are stored in three Tribon databanks:

- SBE_GENVOLDB – the library of unplaced volumes, independent from the project (component volumes)
- SBD_VOLUME – the library of unplaced volumes for the current project
- SBD_VOL_PLAC – the databank of placed volumes for the current project

Tribon Vitesse in Tribon M3 provides a set of new modules for modelling volumes. The basic functionality is offered by the `kcs_vol` module, which is accompanied by a set of modules containing definitions of classes representing various 3D primitives.

ⓘ *Current functionality of the `kcs_vol` module handles only the unplaced volumes stored in SBD_VOLUME.*

Following the convention used in the whole Vitesse API, the variable `kcs_vol.error` is set to a string describing an error, when an exception is raised while the volume modelling functions are used.

2.2 Handling unplaced volumes

All outfitting Vitesse modules understand the concept of the "***current model object***". Some functions will assume, that they are operating on the model object, that has been previously "***activated***" or "***made current***". The model object may become current by ***creating*** it (a new object) or by ***activating*** it (existing object). After making the modifications to the current model object it has to be ***deactivated*** (released), possibly after saving the changes made to the model.

The volume handling Vitesse API follows the approach presented above. For creating a new volume, the following pattern should be used:

```
⇒  kcs_vol.vol_new('MY_NEW_VOLUME', 2000) #optional max extensions given
   try:
        … #model the volume
        kcs_vol.vol_save()   #store the volume on the databank
   finally:
        kcs_vol.vol_close()  #deactivate 'current' volume
```

When updating an existing volume, just replace the first line with

```
kcs_vol.vol_open('EXISTING_VOLUME', 2000) #optional max extensions given
```

As you know, the Drafting application works either in the Drawing mode or in the Volume modelling mode. Activation of a volume using one of the above methods automatically puts the application in the Volume modelling mode. Cancelling the volume, using the `kcs_vol.vol_close()` function, additionally restores the application to the Drawing mode. The same can be done by calling

```
kcs_vol.vol_cancel()
```

Once a volume is activated, you can work on it, building its shape by adding volume primitives, setting up connections, etc. If you try to activate another volume, when a volume is already active, you will get an exception. That's why the finally section of our pattern contains the statement `kcs_vol.vol_close()` (or `kcs_vol.vol_cancel()`), which deactivates the current volume. After modelling a volume, you store it on the SBD_VOLUME databank by calling the function

```
kcs_vol.vol_save()
```

which saves the volume under the current name, or

```
kcs_vol.vol_save_as(newVolumeName)
```

which saves the current volume under the given name, unless this name is occupied. It is possible to check the existence of the given volume in the SBD_VOLUME databank by calling the function

```
kcs_vol.vol_exist(volumeName)
```

which returns ***1***, if the given volume exist in the databank, or ***0***, if it does not. If you still want to overwrite an existing volume, using the function **`kcs_vol.vol_save_as()`**, you may delete it from the databank by calling

```
kcs_vol.vol_delete(volumeName)
```

If the given volume is locked, an exception is raised.

2.3 Subvolumes and volume primitives

Volumes are composed of subvolumes, which in turn are built from 3D primitives. Before you start adding volume primitives, you have to obtain the subvolume ID, which is a simple integer number identifying the subvolume, to which the primitives are added. If the new volume has been just initialised, it contains no subvolumes yet. Then, you should add the first subvolume by calling

```
subVolID = kcs_vol.subvol_add()
```

The returned integer number will be then used for adding volume primitives. Of course, you may want to build your volume from more than a single subvolume. Then, just add another subvolume, and use its ID to add some primitives. If you activate an existing volume, it contains already some subvolumes. Then you can obtain the list of their IDs by calling

```
subVolList = kcs_vol.subvol_list()
```

Once you obtain the ID of a subvolume, you can add the volume primitive to the given subvolume using the following pattern

```
… #build an instance of the Volume Primitive Creation Class
primitiveID = kcs_vol.prim_add(subVolID, volPrimitiveInstance)
```

ⓘ *The available Volume Primitive Creation Classes are discussed in section 2.4*

The **`kcs_vol.prim_add()`** function returns an integer identifying the primitive within the subvolume. You will have to use this identifier for every later manipulation of this primitive. If you are adding the primitives yourself, then you know their IDs. If you want to update primitives of an existing volume, you need first to obtain the list of the existing primitives by calling

```
volPrimID_List = kcs_vol.prim_list(subVolID)
```

The returned list consists of integer numbers being the IDs of the volume primitives added to the subvolume identified by ***subVolID***. You can get the basic properties of the primitive following the example given below:

```
⇒   props = kcs_vol.vol_properties_get(subVolID, primitiveID)
    colour = props.GetColour() #Colour class instance
    density = props.GetDensity()
    softness = propt.GetSoftness()
```

The ***props*** variable is an instance of the **`VolPrimitiveBase`** class, handling the properties common to all types of volume primitives. You can update some properties, and set them to the primitive, as shown in the following example:

```
    props = KcsVolPrimitiveBase.VolPrimitiveBase()
    props.SetColour(KcsColour.Colour("Red"))
    props.SetDensity(8.0E-6)
    props.SetSoftness(10)
⇒   kcs_vol.vol_properties_set(subVolID, primitiveID, props)
```

Finally, the primitive may be deleted from the subvolume using the statement

```
kcs_vol.prim_delete(subVolID, primitiveID)
```

If you want to remove all primitives belonging to the given subvolume (and the subvolume itself), use the statement

```
kcs_vol.subvol_delete(subVolID)
```

2.4 Creation of volume primitives

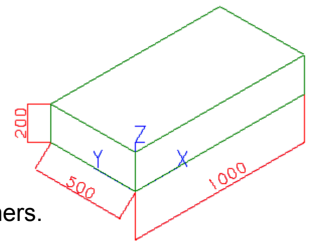
In the previous section we have seen the function **kcs_vol.prim_add()**, that adds a volume primitive to the subvolume. In order to use it, you have to create first an instance of the appropriate volume primitive creation class. Tribon M3 Vitesse provides the following volume primitive creation classes

Class	Description
VolPrimitiveBlock	Axis-parallel rectangular prism
VolPrimitiveGeneralCylinder	General cylinder with any closed contour as a basis. Can be also used for creating standard cylinders with circular basis
VolPrimitiveRevolution	A solid of revolution created by rotating the given contour around the specified axis
VolPrimitiveSphericalCap	The fragment of the sphere obtained by splitting the sphere with the plane
VolPrimitiveTorusSegment	For example, the pipeline elbow (arc segment)
VolPrimitiveTruncatedCone	The bottom part of the cone obtained by splitting the cone with the plane parallel to the cone's basis

All classes specified above have a common parent class, **VolPrimitiveBase**, which handles common volume primitive properties, like the colour, density, and softness. The examples given below, show typical patterns of creating instances of the above classes. Such instances can be then supplied to the function **kcs_vol.prim_add()** for adding the primitive to the subvolume.

VolPrimitiveBlock example

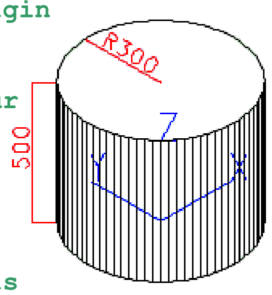
```
primitive = KcsVolPrimitiveBlock.VolPrimitiveBlock()
Corner1 = KcsPoint3D.Point3D(0, 0, 0)
Corner2 = KcsPoint3D.Point3D(1000, 500, 200)
primitive.SetBox(Corner1, Corner2)
```



which defines an axis-parallel rectangular prism expanding between the given opposite corners.

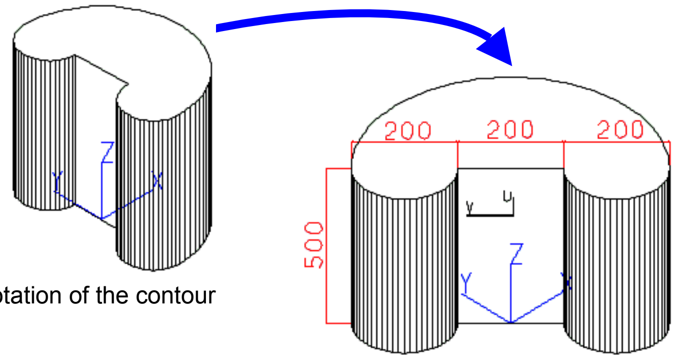
VolPrimitiveGeneralCylinder examples

```
primitive = KcsVolPrimitiveGeneralCylinder.VolPrimitiveGeneralCylinder()
origin = KcsPoint3D.Point3D(0, 0, 200) #bottom basis origin
primitive.SetOrigin(origin)
primitive.SetHeight(500.0) #height of the cylinder
point = KcsPoint2D.Point2D(300, 0)    #define the contour
basis = KcsContour2D.Contour2D(point) #of the basis
point.X = -300
basis.AddArc(point, 300)
point.X = 300
basis.AddArc(point, 300)
primitive.SetContour(basis) #set the contour of the basis
```



which defines a standard cylinder with circular basis (radius: 300 mm), and the height of 500 mm. The centre of the bottom basis (the contour's (0,0) point) is located at (0, 0, 200) in the volume's coordinate system. For defining a non-standard cylinder (with the basis being an arbitrary closed contour), you need only to change the part of the above example, that defines the contour ***basis***.

```
point = KcsPoint2D.Point2D(0, 300)
basis = KcsContour2D.Contour2D(point)
point.Y = 100
basis.AddArc(point, 100)
point.Y = -100
basis.AddLine(point)
point.Y = -300
basis.AddArc(point, 100)
point.Y = 300
basis.AddArc(point, 300)
primitive.SetContour(basis)
```

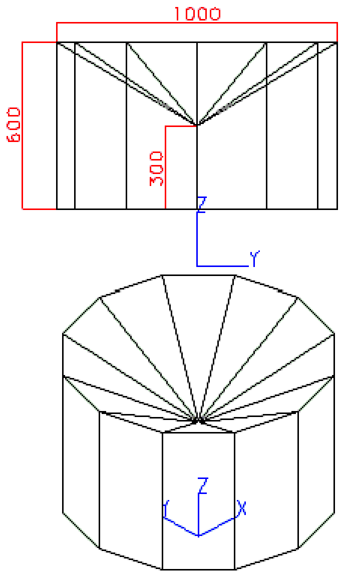


You can also specify the U and V axes, to simulate the rotation of the contour

```
primitive.SetUAxis(KcsVector3D.Vector3D(1, 1, 0))
primitive.SetVAxis(KcsVector3D.Vector3D(-1, 1, 0))
```

ⓘ *The ability to specify the U and V axis direction vectors is not available for the* `VolPrimitiveBlock` *class – the rectangular prism will be always created in an axis-parallel position.*

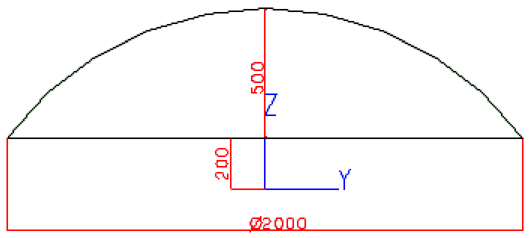
VolPrimitiveRevolution example



```
origin = KcsPoint3D.Point3D(0, 0, 200)
primitive.SetOrigin(origin)
point = KcsPoint2D.Point2D(0, 0)
cont = KcsContour2D.Contour2D(point)
point.Y = 300
cont.AddLine(point)
point.SetCoordinates(500, 600)
cont.AddLine(point)
point.Y = 0
cont.AddLine(point)
point.X = 0
cont.AddLine(point)
primitive.SetContour(cont) #set the contour
primitive.SetUAxis(KcsVector3D.Vector3D(1, 0, 0))
primitive.SetVAxis(KcsVector3D.Vector3D(0, 0, 1))
```

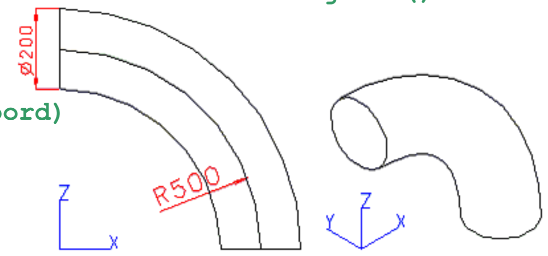
VolPrimitiveSphericalCap example

```
primitive = KcsVolPrimitiveSphericalCap.VolPrimitiveSphericalCap()
origin = KcsPoint3D.Point3D(0, 0, 200)
primitive.SetOrigin(origin)
primitive.SetAmplitude(500.0)
primitive.SetRadius(1000.0)
U = KcsVector3D.Vector3D(0, 0, 1)
primitive.SetUAxis(U)
```



VolPrimitiveTorusSegment example

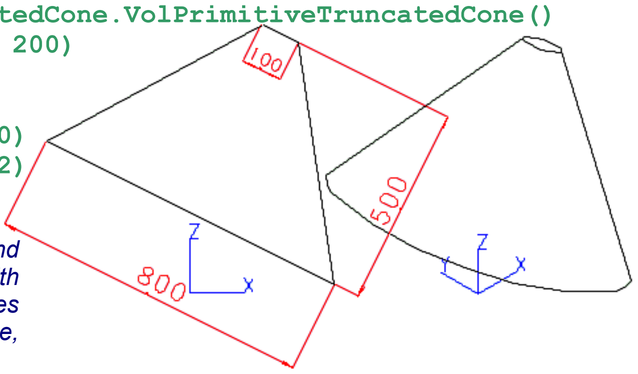
```
primitive = KcsVolPrimitiveTorusSegment.VolPrimitiveTorusSegment()
start = KcsPoint3D.Point3D(500, 0, 0)
end = KcsPoint3D.Point3D(0, 0, 500)
coord = 250.0*(math.sqrt(2.0) - 1)
amp = KcsVector3D.Vector3D(coord, 0, coord)
arc = KcsArc3D.Arc3D(start, end, amp)
primitive.SetArc(arc)
⇒ primitive.SetDiameter(100.0)
```



ⓘ *Measuring the pipe diameter reveals the fact, that the* `SetDiameter()` *method does not set the diameter, but the radius.*

VolPrimitiveTruncatedCone example

```
primitive = KcsVolPrimitiveTruncatedCone.VolPrimitiveTruncatedCone()
origin = KcsPoint3D.Point3D(0, 0, 200)
primitive.SetOrigin(origin)
primitive.SetHeight(500.0)
primitive.SetDiameter(800.0, 100.0)
vec = KcsVector3D.Vector3D(1, 0, 2)
primitive.SetUAxis(vec)
```

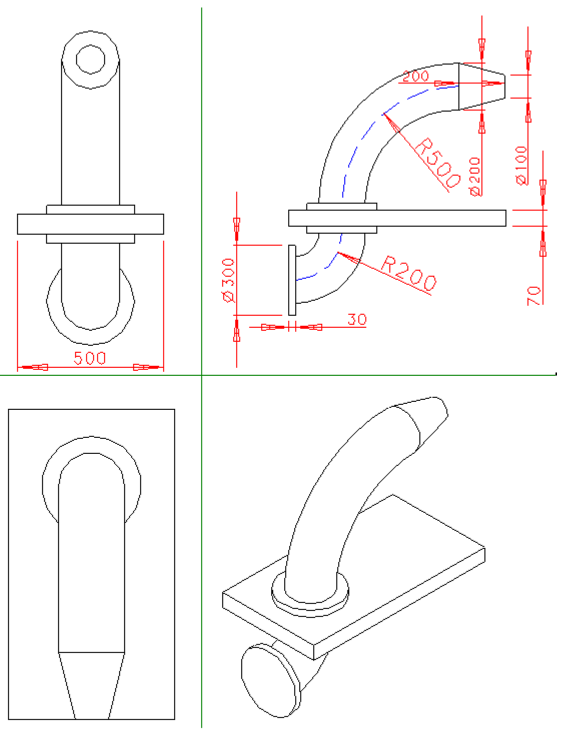


ⓘ *Use the* ***Truncated Cone*** *primitive with equal top and bottom diameters to create the standard cylinder with circular basis. It will be drawn without the reference lines connecting the bases, going along the side surface, which appear for the* ***General Cylinder*** *primitive.*

The above examples show, how to set up geometrical properties of the volume primitives. Before passing the created instance of the given volume primitive creation class to the **kcs_vol.prim_add()** function, you may define also additional properties, like: the colour, density, and softness, as shown below

```
primitive.SetColour(KcsColour.Colour("Red"))
primitive.SetDensity(8.0E-6)
primitive.SetSoftness(10)
```

Exercise 1: Creating a volume



Create a volume, like in the picture to the right. You may make the program more flexible by getting some dimensions from the user, and applying appropriate calculations to determine the related dimensions.

Set up two pipe connections at the ends of the pipe segments.

2.5 Connections

It is possible to define connections in the volume, which, for example, simplifies setting up connections for equipments. The connections are handled in Vitesse by instances of the **VolConnection** class. Each connection manages the following attributes:

- connection number (1 – 999, identifies the particular connection of the given type),
- connection type:
  - ★ 1 – pipe connection,
  - ★ 2 – electrical connection,
  - ★ 3 – ventilation connection,
- connection point (position),
- connection direction vector,
- connection description (max. 100 characters).

In order to add a connection, you may follow the example below:

```
    conn = KcsVolConnection.VolConnection(1, 10)  #pipe connection no. 10
    connPoint = KcsPoint3D.Point3D(0, 0, 500)     #connection point
    conn.SetPosition(connPoint)
    connDirection = KcsVector3D.Vector3D(0, 0, 1) #connection direction
    conn.SetDirection(connDirection)
    conn.SetDescription("Description of the connection")
⇒   kcs_vol.conn_add(conn)
```

When creating the **VolConnection** class instance you provide the connection type and connection number. The connection numbers must be unique within the given connection type, but it is quite possible to have two connections of different type having the same connection number. The list of currently defined connection for the active volume may be obtained by calling

```
connList = kcs_vol.conn_list()
```

The resulting list contains the **VolConnection** class instances, defining the volume's connections. The connections in the list are uniquely identified by the connection type and connection number. You have to provide these two connection attributes, if you want to remove an existing connection

```
conn = KcsVolConnection.VolConnection(1, 10) #pipe connection no. 10
kcs_vol.conn_delete(conn)
```

or obtain full information about the given connection

```
    conn = KcsVolConnection.VolConnection(1, 10) #pipe connection no. 10
⇒   kcs_vol.conn_properties_get(conn)
    print conn.GetPosition() #print out the connection position
```

After obtaining the connection's data, you can update them

```
    position = conn.GetPosition()
    position.X += 100 #move the connection by 100 mm along the X axis
    conn.SetPosition(position)
⇒   kcs_vol.conn_properties_set(conn)
```

2.6 Placed volumes

It is possible to place a copy of the volume in the ship's model. Such a model object has only the shape, weight (and COG) and connection information. It does not have any other production information. Most often such placed volumes eventually will become equipments. In order to place the volume, we use the following pattern

```
    origin = KcsPoint3D.Point3D(30000, 0, 5000)
    uAxis = KcsVector3D.Vector3D(1, 0, 0) #U vector
    vAxis = KcsVector3D.Vector3D(0, 0, 1) #V vector
⇒   placVolName = kcs_vol.placvol_new(unplacedVolName, origin, uAxis, vAxis)
```

where the placed volume name is generated automatically and returned as the function's result. You can also set your own name for the placed volume, by providing it as the last, additional argument:

```
⇒   kcs_vol.placvol_new(unplacedVolName, origin, uAxis, vAxis, placVolName)
```

The U and V vectors define the direction in the global ship's coordinate system of the X and Y axes of the volume's coordinate system. The origin is the location in the ship space of the origin of the volume's coordinate system.

ⓘ *For compatibility reasons, Tribon M3 still provides the* **`kcs_placvol`** *module, but its functionality has been included in the* **`kcs_vol`** *module.*

Chapter 3

3 Equipments

The Vitesse Equipment interface contains functions for creating, placing and manipulating equipment in the model.

ⓘ *The **kcs_equip** module is available from all Tribon modules supporting creation of equipment.*

When an error is encountered during execution of functions from **kcs_equip** module, an exception is raised, and the error identification string is stored in the **kcs_equip.error** variable.

3.1 Handling outfitting modules

The equipments, as well as the other outfitting objects, like: structures, pipes, ventilation objects, cables, and cableways, are assigned to the outfitting modules, defining groups of logically related outfitting model objects. A module is usually defined as a region in the ship's space; however, it can be as well an arbitrary collection of outfitting objects. The outfitting modules are usually created using the Design Manager application. It is also possible to handle them in Vitesse using the **kcs_modelstruct** module.

```
kcs_modelstruct.module_new(newModuleName)
```

This function creates a new outfitting module with the given name, that cannot be longer than 25 characters, and cannot be empty. In order to remove a module, the following function should be used:

```
kcs_modelstruct.module_delete(existingModuleName)
```

The **kcs_modelstruct** module contains also other functions handling hull blocks (discussed during the Vitesse Hull training) and outfitting systems (see section 5.1)

3.2 Activate/Build/Save process – the overview

Before you can create or update the equipment, you have to make it current. This is possible by either creating a new equipment item or activating an existing one. The example below shows how to work on the new equipment

```
    try:
⇒       kcs_equip.equip_new(equipmentName, moduleName) #create and activate
        try:
            … #work on the activated equipment
⇒           kcs_equip.equip_save()
        except:
⇒           kcs_equip.equip_cancel() #deactivate the equipment
⇒           kcs_equip.equip_delete(equipmentName) #incomplete – delete!
            print "Error modelling the equipment!"
    except:
        print "Error creating an equipment!"
```

When working on an EXISTING equipment, you would rather use the following pattern:

```
    try:
⇒       kcs_equip.equip_activate(equipmentName) #activate the equipment
        try:
            … #work on the activated equipment
⇒           kcs_equip.equip_save()
        except:
⇒           kcs_equip.equip_cancel() #deactivate the equipment
            print "Error modelling the equipment!"
    except:
        print "Error creating an equipment!"
```

In both cases the ***equipmentName*** should be given in presentation format (i.e. without the project prefix). Once you make an equipment current, you may work on it, defining its properties, and placing it at some position in the ship's model.

After making these modifications, the equipment should be saved on the databank using the function **`kcs_equip.equip_save()`**, which also deactivates the equipment, so that another equipment can be made current. If you don't want to store the modifications made to an equipment, you should deactivate it using the function **`kcs_equip.equip_cancel()`**.

While the equipment is current, you can learn its name, and the name of its module (useful for building messages, and Data Extraction command strings) by using the following functions:

```
equipmentName = kcs_equip.equip_name_get()
moduleName = kcs_equip.equip_module_get()
```

It is possible, that you will get an error, when trying to create a new equipment, because the given equipment item already exists. It is possible to verify that using the function **`kcs_equip.equip_exist()`**

```
⇒   if kcs_equip.equip_exist():
        … #work on an EXISTING equipment
```

which returns ***1***, if the given equipment already exists, or ***0***, if it does not. If you need to remove an existing equipment from the model, you can do it by calling **`kcs_equip.equip_delete()`**. At that time, no equipment should be current. All connections will be automatically removed.

3.3 Setting the attributes for the current equipment item

The example given below shows the available functions for setting the properties of the current equipment.

```
kcs_equip.equip_component_set(componentName)
kcs_equip.equip_description_set(description)
kcs_equip.equip_room_set(roomName)
kcs_equip.equip_alias_set(aliasName)
```

By using them, you may set or update the component reference for the current equipment item, its description, room, and alias name. All the arguments are strings. The component name should be valid, or an exception will be raised.

ⓘ *There are no specific functions returning these properties of the current equipment. Use Data Extraction interface to retrieve the current settings.*

3.4 Placing and transforming the equipment

After creation, the current equipment is not yet placed in the model. This can be achieved using the following pattern:

```
    Origin = KcsPoint3D.Point3D(20000, 0, 5000) #the equipment's origin
    U = KcsVector3D.Vector3D(1, 0, 0) #orientation of the equipment
    V = KcsVector3D.Vector3D(0, 1, 0)
⇒   kcs_equip.equip_place(Origin, U, V)
```

The ***Origin*** is the location in the ship's model, where the origin of the volume, defining the shape of the equipment, is placed. The ***U*** and ***V*** vectors define the direction of the X and Y axes of the local coordinate system of this volume.

Placing an equipment does not make it appear on the drawing. In order to see the placed equipment in the model views, you have to draw it explicitly, using the function **`kcs_draft.model_draw()`**.

```
    model = KcsModel.Model("equipment", projectName + "-" + equipmentName)
⇒   kcs_draft.model_draw(model, viewHandle)
```

ⓘ *The **`Model`** class instance should be provided with the **`presentation form`** of the equipment name, which is controlled by the EQUIP_NAME Drafting default. If this default is set to PROJ-NAME, then the project name (with a dash) is added as a prefix before the equipment name. Example: TTP-AIRCON3 is the presentation name of the equipment AIRCON3 on project TTP.*

If you want to change the location of an existing equipment, do not place it again, but rather transform it using the following pattern:

```
    … #activate the equipment
    trans = KcsTransformation3D.Transformation3D()
    vector = KcsVector3D.Vector3D(0, 0, 1000) #displacement vector
⇒   trans.Translate(vector) #apply the displacement
    origin = KcsPoint3D.Point3D(2000, 0, 7000) #origin and direction
    direction = KcsVector3D.Vector3D(0, 0, 1)  #of the rotation axis
⇒   trans.Rotate(origin, direction, 45) #apply the rotation by 45 degrees
⇒   kcs_equip.equip_transform(trans) #transform the equipment
```

Of course, you can choose the transformation to contain the displacement only, rotation only, or any other combination of these operations.

ⓘ *The equipment can be moved or rotated only. Other transformations are not allowed for this kind of outfitting objects.*

3.5 Releasing the equipment

When the equipment is 'ready', the information about it should be transferred to the production environment. The equipment is considered 'ready', if all required settings are defined. There is a Vitesse function, that makes this test for the current equipment, and if successful, passes the information about the equipment to the production environment.

```
⇒   status = kcs_equip.equip_ready()
    if status == 0: #SUCCESS!
         kcs_ui.message_confirm("The equipment is ready!")
    else: #FAILURE!
         kcs_ui.message_confirm("The equipment is not ready!\n"
                                "Status code: %d" % status)
```

The returned ***status*** is an integer indicating the outcome of the operation: ***0*** means, that the equipment has been marked as ready, and stored. The other values indicate various reasons for the failure of making the equipment ready:

1 – room not given	**2** – subproject not given	**3** – planning unit not given
4 – description not given	**5** – equipment not placed	

Please keep in mind, that this function also raises exceptions, when some problems are encountered. For example, it is possible, that the equipment is ready and stored, but the transfer to PDI failed. Then, the variable **kcs_equip.error** will have the value '**kcs_EquipPDITransferFailed**', and although an exception has been raised, you can consider the equipment to be marked as ready, and correctly stored.

3.6 Document references

Following the convention used in the other modelling modules of Tribon Vitesse, the **kcs_equip** module also offers the functions dealing with document references attached to the equipment items. Document references are represented in Vitesse by instances of the **DocumentReference** class.

```
    … #activate an equipment
    docRef = KcsDocumentReference.DocumentReference()
    docRef.SetType('drawing')
    docRef.SetDocument('EQUIP_DWG01')
    docRef.SetPurpose(kcs_draft.kcsDWGTYPE_GEN) #Databank (General drawing)
⇒   kcs_equip.document_reference_add(docRef)
⇒   docRefList = kcs_equip.document_reference_get()
    for docRef in docRefList[:]:
         if docRef.GetType() == 'vitesse': #Remove 'vitesse' references
⇒             kcs_equip.document_reference_remove(docRef)
```

The document references can have the following types:

- 'drawing' - reference to the Tribon drawing
- 'file' - reference to any external file
- 'vitesse' - reference to the Vitesse script
- 'document' - reference to the document stored in the external Document Management System

ⓘ *The 'document' type references are handled using Vitesse triggers. The 'drawing' type references must have the purpose set with the* **`SetPurpose()`** *method, describing the drawing databank, where the drawing is stored.*

Exercise 2: Placing an equipment

Create a macro, which automatically places equipment on a foundation (a structure), according to following rules:

- If the indicated foundation has standard reference to “FOUND_1”, then the component “VTO1” is chosen.
- If the indicated foundation has standard reference to “FOUND_2”, then the component “VTO2” is chosen.
- The user should be prompted to key in the equipment name.
- Equipment should belong to the same module, as the indicated structure.
- The equipment should be placed on top of the foundation, above its COG.

ⓘ *Example drawing “EQUIP_EXERCISE” contains views with structure items suitable for this exercise.*

Data extraction strings:
STRUCTURE('project').ITEM('struct').BOX
and/or
STRUCTURE('project').ITEM('struct').COG
can be used to find the proper 3D point for placing the equipment (e.g. above the COG, at the Z level defined by maximum Z of the extension box).

Chapter 4

4 Structures

4.1 Introduction

The purpose of Tribon Vitesse for Structure is to supply the programmer with simple tools to create and handle structures in an easy way. This tool, in combination with the Data Extraction interface, is very efficient at creating general structures that are dependent on other model objects, such as pipe or cableway hangers. The production may also be supported by automatic assembly drawings made by 2D-drafting Vitesse.

This module is available from all Tribon modules supporting creation of Structures

The structure-creating abilities are available from the **kcs_struct** module. In order to create structures from Tribon Vitesse, the program must contain the statement

```
import kcs_struct
```

Following the convention used in the whole Vitesse API, the module provides the variable **kcs_struct.error**, which is set to a string describing an error type, when one of the module's function raises an exception.

The functions from the **kcs_struct** module can be divided into the following main sections:

- functions acting on the complete structure object
- functions acting on the parts of the current structure

They will be discussed in details in the following sections.

Structures are assigned to outfitting modules. The available methods of handling the modules in Vitesse have been described in section 3.1

4.2 Manipulating the Complete Structure

Tribon Vitesse uses the same mechanism as the actual Tribon interactive applications. Among other things, there is a concept of the current structure, which is the implicit structure object used for many operations on the structure. You may notice, that all other outfitting applications follow the same approach: there is the current equipment, current volume, current pipe, current cableway, etc.

4.2.1 Activate/Build/Save process – the overview

The structure may become current either because it has been just created or activated. Below you can find the general pattern of working with structures in Vitesse

```
        activated = False #a flag – set to True, once the structure is activated
        try:
⇒           kcs_struct.struct_new("MYSTRUCT", "MODULE", "Cyan")
            activated = True
            … #add parts, set properties, etc.
⇒           kcs_struct.struct_save()
        except:
            if activated:
⇒                 kcs_struct.struct_cancel()
```

When creating the new structure, you need to provide the structure name, module name, and colour name

Instead of the string, the Colour class instance can be provided as the structure colour argument.

When the work on the structure is finished, the structure is saved by the **kcs_struct.struct_save()** function. When an error is encountered, after the structure has been successfully activated, it is deactivated by calling the function `kcs_struct.struct_cancel()` discarding all changes made to the structure. Both functions deactivate the structure.

ⓘ *Note that the function `kcs_struct.struct_save()` only saves the current structure, without performing tests for non-unique position numbers, or for compliance of the profile endcuts with the hull profile restriction file SBH_PROF_RESTRICT. For these purposes, separate functions are available.*

*Instead of using the **activated** variable, you may want to use nested `try:` … `except:` … statements. Just make sure, that you don't call `kcs_struct.struct_cancel()`, when Vitesse failed to activate the structure.*

Saving the structure does not make it appear on the model views. In order to show the newly created (or modified) structure, the program must explicitly call the `kcs_draft.model_draw()` function, providing the proper `Model` class instance defining the saved structure.

If, instead of creating a new structure, you want to update an existing one, just replace the call to the function `kcs_struct.struct_new()` with the line similar to

```
kcs_struct.struct_activate("MYSTRUCT")
```

where you need to provide the structure name only. Either way, the structure becomes current, and we can start adding parts, setting some properties, etc. It is an error to create or activate a structure, when another structure is current. Unfortunately, there is no function to check, whether a structure is current, or not. You may, however, try to use any function requiring, that a structure is current, and intercept the exception. Here is the example:

```
   try:
⇒      structName = kcs_struct.struct_name_get()
       structIsCurrent = True
   except:
       structIsCurrent = False
```

The function `kcs_struct.struct_name_get()`, used in the above example, returns the name of the current structure (useful for creating messages or Data Extraction strings), or raises an exception, if no structure is current.

When no structure is current, you can remove any given structure from the model, using the function

```
kcs_struct.struct_delete(structureName)
```

4.2.2 General Functions

All functions in this chapter require a current structure object to be established, and will raise the exception if it is not. As it will be shown in section 4.3.2, the location of the structure is determined by the location and orientation of its parts. It is possible to change the location (and orientation) of the current structure by using the following pattern

```
   trans = KcsTransformation3D.Transformation3D()
   … #build the transformation (see the equipment example in section 3.4)
⇒  kcs_struct.struct_transform(trans)
```

Parts of another structure may be copied into the current structure using the function

```
kcs_struct.struct_duplicate(structure)
```

The original structure is not affected by this operation, although you can provide the name of the current structure, and then all parts of the current structure will be duplicated. In order to create a copy of a structure, we have to use the following pattern:

1. Initialise the new structure (the copy).
2. Use `kcs_struct.struct_duplicate()`, providing the name of the original structure as the argument, to copy all parts from the original structure.
3. Use `kcs_struct.struct_transform()` to move the copy away from the original, and to place it at the right location.
4. Save the structure – the model will contain both the original and copy as two independent structures

By combining these functions in a different way, we can, for example, merge two structures together:

1. Activate the structure STRUCT_1.
2. Use `kcs_struct.struct_duplicate()`, providing the name of STRUCT_2 as the argument, to copy all parts from STRUCT_2.

3. Save the structure STRUCT_1
4. Use **kcs_struct.struct_delete()**, providing the name of STRUCT_2 as the argument, to remove the structure STRUCT_2 – only STRUCT_1 will remain, containing the parts from both structures.

ⓘ *The STRUCT_KEEP_INSERT_OBJ default is NOT taken into account. The inserted structure will not be removed from the databank if this default has the value of NO – the script must take care of it.*

A structure is not drawn automatically in the drawing, when created. If you want your structure to appear in the model views, you must call the function **kcs_draft.model_draw()** explicitly.

```
   model = KcsModel.Model("struct", structName)
⇒  kcs_draft.model_draw(model, viewHandle)
```

ⓘ *The **Model** class instance should be provided with the **presentation form** of the structure name, which is controlled by the STRUCT_NAME Drafting default. For example, if this default is set to MOD-NAME, then the module name (with a dash) should be added as a prefix before the structure name. Example: MOD1-FOUND would be the presentation name of the structure FOUND in the module MOD1. The default setting of the STRUCT_NAME default, however, is NAME – then no prefix is required.*

*__Note:__ This default controls also the names of the structures, when they are created, so the structure name provided to the **Model** class instance should take into account the value of the STRUCT_NAME default, which has been used, when the structure was created.*

4.2.3 Production Information

As mentioned before, the **kcs_struct.struct_save()** function does not perform any tests before storing the current structure. The program may request the following tests to be done:

1. Test of compatibility of all profile endcuts in the current structure with the restrictions imposed by the hull-profile restriction file SBH_PROF_RESTRICT.

```
   try:
⇒      kcs_struct.struct_check_restrict()
       print "Test passed!"
   except:
       print "At least one profile endcut did not pass the test!"
```

This function will raise the exception described as '**kcs_EndcutInvalid**' if at least one profile endcut does not meet the requirements set by the hull profile restriction file.

2. Test of uniqueness of the part position numbers

```
   try:
⇒      kcs_struct.struct_check_posno()
       print "Test passed!"
   except:
       print "At least one position number is not unique!"
```

This function will raise the exception described as '**kcs_PosnoDuplicate**' if at least one position number has been defined twice.

There are two functions setting the marking line on or off for the current structure:

```
   kcs_struct.struct_marking_lines_on()
   kcs_struct.struct_marking_lines_off()
```

By calling the function

```
   kcs_struct.struct_assembly(assemblyPathName)
```

it is possible to assign the whole current structure to an assembly or remove an existing assembly reference if an empty assembly name is given. The splitting of the current structure can be requested by calling

```
   kcs_struct.struct_split()
```

Then the whole structure is transferred to the production preparation environment. The structure is transferred also to PDI, if applicable, and remains current after the call.

4.3 Manipulating Parts of the Current Structure

The functions in this section help to handle structure parts. Structures are built from components, either 'real' (existing in GCDB databank) or pseudo components. The **kcs_struct** module contains functions that help to generate proper names for pseudo components, given its type and main cross-section dimensions. All other functions directly add or modify structure parts, and therefore require that a structure is current.

4.3.1 Names of Pseudo-Components

The functions below help to generate proper names of the pseudo components used as parts of structures. The syntax of the pseudo component names is specific to the actual component and contains the information about the type of the component, and the cross section dimensions. If the rules of generating the pseudo component names are well known to the programmer, he may avoid using the functions below and specify the names directly. All the functions below will generate an exception described as '**kcs_SyntaxInvalid**' if the given parameters do not allow the component name to be generated properly. These functions do not update the current structure object.

```
profileCompName = kcs_struct.pseudoname_profile('F', 200, 10)
plateCompName = kcs_struct.pseudoname_plate(3.0)
holeCompName = kcs_struct.pseudoname_hole(200, 100)
holeCompName2 = kcs_struct.pseudoname_hole(200, 100, 15)
```

The component name of the profile in the above example will be 'F#200*10'. The first parameter is always the string defining the profile type. Then, up to 6 additional parameters can be supplied, depending on the profile type. They will be interpreted as the dimensions ***a***, ***b***, ***s***, ***t***, ***c***, and ***u***.

*The information about the meaning of these dimensions can be found in the Tribon Hull documentation (**Tribon M3 Hull → Setup and Customisation → Profiles in Tribon → About Profile Standard in Tribon → Profile Types in Tribon**)*

The component name of the plate in the above example will be 'P#3'. The argument is the thickness of the plate. The component names for the standard holes from the above example are: 'H#200*100', and 'H#200*100*15'. The arguments are the width and height of the hole's rectangle. The last, optional argument, is the radius of the rounded corners. If not given, the hole's rectangle will have the sharp (not rounded) corners.

4.3.2 Generating Structure Parts

The functions described in this section create or modify parts of the current structure. All changes to the parts are saved on the databank when the whole structure is stored. Locations and directions are expressed using the Vitesse classes **Point3D**, and **Vector3D**. Be sure to include their definitions in the program if needed.

All functions creating the parts of a structure require the component name as the first argument. It may be either the name of a 'real' component, defined on the SB_GCDB databank, or the pseudo-component name (see section 4.3.1),

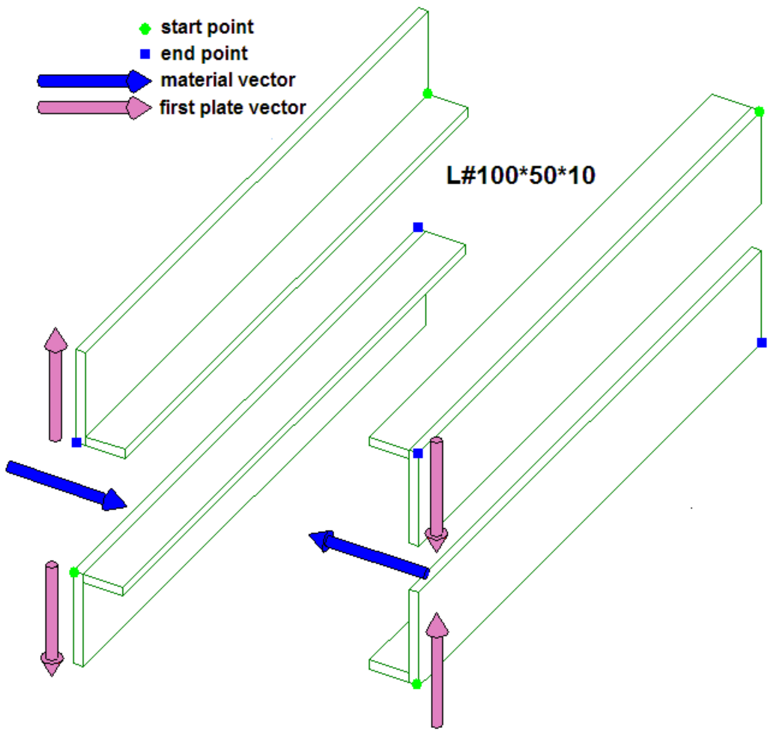
4.3.2.1 Profiles

The profile's location and orientation is determined by the material vector, and two points (start/end) defining the profile's theoretical line. For determining the orientation of a profile, Tribon uses the following rules:

RULES:

1. The **material vector** is perpendicular to the ***first*** plate of the profile (first dimension), and points into the sector occupied by the ***second*** plate (second dimension).
2. **start** and **end points** should be chosen so, that the right-handed screw moves from **start** towards the **end point**, when turned from the **material vector** towards the **first plate vector**.

The picture below shows four example orientations of an L-bar (isometric view). Use it for studying the rules given above. Verify each profile's position, and see if the direction of the material vector and the locations of **start** and **end** points are in accordance with the rules given above.



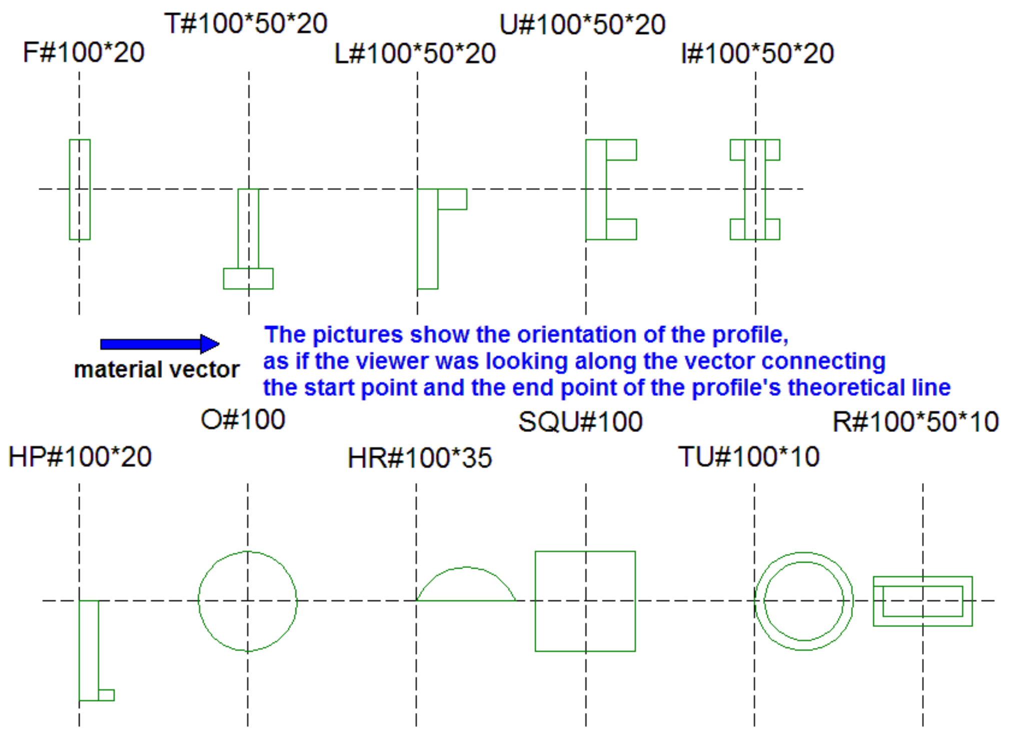
Which of the four variants shown on this picture could be generated by the example code given below?

```
        start = KcsPoint3D.Point3D(5000, 0, 0)
        end = KcsPoint3D.Point3D(3000, 0, 0)
        material = KcsVector3D.Vector3D(0, -1, 0)
        compName = "L#100*50*10" #part component name
  ⇒     id = kcs_struct.profile_new_2point3D(compName, start, end, material)
```

?

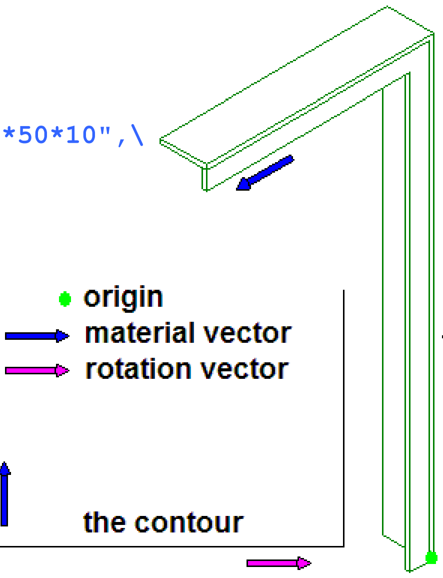
This example code is also the pattern for adding the profile part to the structure. Understanding of the positioning rules, presented above will help to set up the right orientation of the profile. The profile expands along the line connecting the provided ***start*** and ***end*** points. The function `kcs_struct.profile_new_2point3D()` creates the profile part using the provided component name, which can be the name of either a 'real' component, stored on GCDB, or of the pseudo-component. The integer number returned by this function is the ***part ID*** of the profile part in the current structure.

The picture to the right shows other examples of different profiles (pseudo-components) available in Tribon. Apart from verifying again the positioning rules, please notice also the placement of the theoretical line (connecting the start and end point), specific to the particular profile type. It always passes through the profile's cross-section at the points of intersection between the vertical and horizontal dashed lines.


F#100*20
T#100*50*20
L#100*50*20
U#100*50*20
I#100*50*20
material vector
The pictures show the orientation of the profile,
as if the viewer was looking along the vector connecting
the start point and the end point of the profile's theoretical line
HP#100*20
O#100
HR#100*35
SQU#100
TU#100*10
R#100*50*10

It is also possible to create the profile with a non-straight theoretical line, but following an arbitrary planar contour.

```
        point = KcsPoint2D.Point2D(0, 0) #contour's point placed at 'origin'
        cont = KcsContour2D.Contour2D(point)
        … #build the contour shape
        origin = KcsPoint3D.Point3D(20000, 0, 5000)
        material = KcsVector3D.Vector3D(-1, 0, 0)
        rotation = KcsVector3D.Vector3D(0, 0, 1)
   ⇒    id = kcs_struct.profile_new_contour2D("L#100*50*10",\
             cont, origin, material, rotation)
```



The ***rotation*** vector determines the direction of the X axis of the profile's contour coordinate system.

The ***material*** vector corresponds to the Y axis of the profile's contour coordinate system, and indicates the direction perpendicular to the first profile's plate (first dimension) – at the first contour's segment, starting at the ***origin*** point.

The function adds a profile part to the current structure, and returns the part identification (integer) number. The part's shape is defined by moving the profile's cross-section along the contour. This allows creating bent profiles. The contour can have both the line and arc segments.

At the endpoints of the profile, endcuts can be placed, providing the endcut type, and the corresponding parameters, which are defined by the appropriate Tribon Hull tools.

```
        kcs_struct.profile_endcut(profileID, endPoint, 1112, 80.0)
```

You need to provide the ID of the profile part, to which the endcut is added, the location (either start or end point), profile type, and up to 6 endcut parameters. Parameters not used by the specific endcut type can be omitted. An exception will be raised, if the given endcut type has not been predefined in the Tribon Hull environment. The end points' location of the profile can be updated using the function

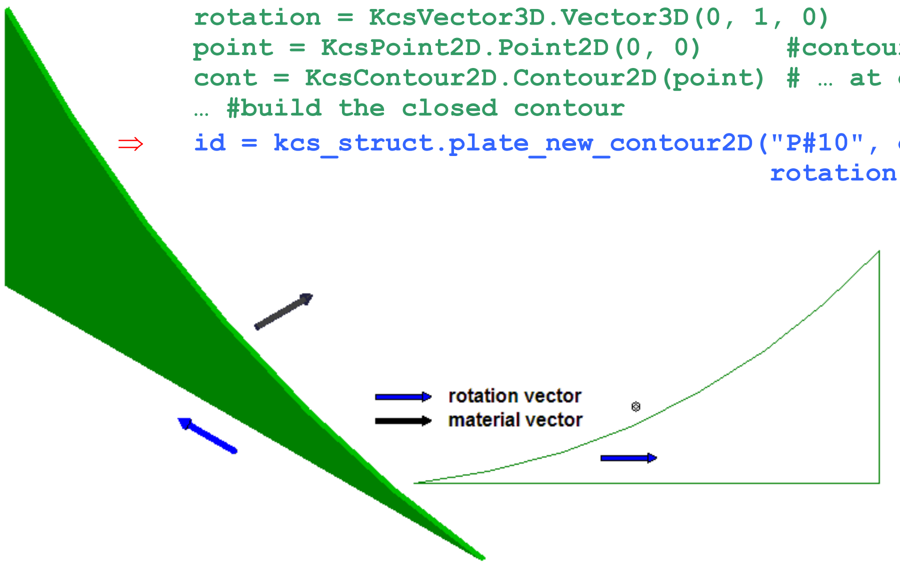
```
        kcs_struct.profile_endpoints(profileID, newStartPoint, newEndPoint)
```

The points given as the function's parameters are projected onto the line connecting the original profile's end points, so that the direction of the profile does not change. The original end point locations are known, if the Vitesse script has created the profile, or else they can be retrieved by Data Extraction.

4.3.2.2 Plates

Plate parts are always created from the 2D contour, as shown below.

```
             origin = KcsPoint3D.Point3D(20000, 0, 5000)
             material = KcsVector3D.Vector3D(1, 0, 0)
             rotation = KcsVector3D.Vector3D(0, 1, 0)
             point = KcsPoint2D.Point2D(0, 0)     #contour's point to be placed
             cont = KcsContour2D.Contour2D(point) # … at origin
             … #build the closed contour
        ⇒    id = kcs_struct.plate_new_contour2D("P#10", origin, material, \
                                                 rotation, contour)
```



The function **kcs_struct.plate_new_contour2D()** adds a plate part in the current structure, and returns its part identification number. The plate is defined by the 2D shape placed in the 3D model coordinate system. The ***material*** vector defines the direction perpendicular to the plate's plane. The ***rotation*** vector determines the direction of the X axis of the plate's contour 2D coordinate system.

After creating two adjacent plates it is possible to create a single bent plate from them by combining them with the bent fragment having the user-defined bending radius.

```
     … #create plates with IDs: id1, id2
⇒    id = kcs_struct.plate_bent_new(id1, point1, id2, point2, radius)
```

The function creates a bent plate group in the current structure, and returns its identification number. Both plates can belong to some bent plate groups, but not to the same group.

ⓘ *The plates being combined must have enough material along the side, where the bending occurs for the bend to be created. Otherwise an error occurs.*

4.3.2.3 Holes

Holes can be generated in either plates or profiles. The shape of the hole's boundary can be either standard (a rectangle with possibly rounded corners) or arbitrary defined by the provided Contour2D class instance defining a closed contour. First, let's see, how to create holes in plates:

```
     centre = KcsPoint3D.Point3D(20000, 0, 5000) #hole's centre
     rotation = KcsVector3D.Vector3D(1, 0, 0)    #hole's height direction
⇒    id = kcs_struct.hole_new("H#400*200*20", plateID, centre, rotation)
```

if the hole's contour can be defines as a rectangle with possibly rounded corners, and

```
     point = KcsPoint2D.Point2D(0, 0) #hole's contour origin
     cont = KcsContour2D.Contour2D(point)
     … #build the hole's closed contour
     origin = KcsPoint3D.Point3D(20000, 0, 5000) #hole's origin
     rotation = KcsVector3D.Vector3D(1, 0, 0)    #hole's contour X-axis
⇒    id = kcs_struct.hole_new(cont, plateID, origin, rotation)
```

for an arbitrary hole's contour. In both cases you have to provide the ID of the plate in which the hole is made, the reference point, at which the hole's contour is attached (origin/centre), and the rotation vector, describing the orientation of the hole's contour on the plate.

Each profile consists of at least two plates welded together. That's why for holes in profiles we have to add one more argument (***side***), indicating the profile's plate, in which the hole is made. It can take one of the following values:

- 0 – hole is created in Web side of a profile
- 1 – hole is created in Flange side of a profile
- 2 – hole is created in the opposite Flange side of a profile in the case of an I-bar

Then, the above patterns, rewritten for profiles will have the following form:

```
     … #set up the centre and rotation
⇒    id = kcs_struct.hole_new("H#400*200*20", \
          profileID, centre, rotation, side)
```

for standard holes, and

```
     … #set up the contour, centre, and rotation
⇒    id = kcs_struct.hole_new(cont, profileID, centre, rotation, side)
```

for arbitrary holes.

The **kcs_struct.hole_new()** function adds a hole to an existing plate or profile part in the current structure, and returns the hole identification number.

4.3.2.4 Miscellaneous components

Finally, miscellaneous components may be added to the current structure:

```
   origin = KcsPoint3D.Point3D(20000, 0, 5000)
   route = KcsVector3D.Vector3D(0, 0, 1)
   rotation = KcsVector3D.Vector3D(1, 0, 0)
⇒  id = kcs_struct.misc_comp_new(name, point, route, rotation)
```

This function adds a miscellaneous component part to the current structure, and returns the part identification number. The ***route*** vector defines the direction of the Z axis, and the ***rotation*** vector – the direction of the X axis of the coordinate system of the volume associated with the component being added.

ⓘ *To be used as the structure miscellaneous component, the component given as the first argument should have the type code of 1299XXX, and the associated volume should have the usage code of 30 or 31.*

4.3.3 General Modelling Functions

The functions below perform general modelling activities such as deleting, transforming or duplicating structure parts. The functions related to the production environment are described in the next section. All these functions require a structure to be current, and use the ID of the structure part as an argument.

It is possible to remove a part (including holes) from the current structure by calling

```
kcs_struct.part_delete(partID)
```

In a similar manner, like for the whole structure (see also the equipment example on page 15), it is possible to transform a single part

```
trans = KcsTransformation3D.Transformation3D()
… #set up the transformation
kcs_struct.part_transform(partID, trans)
```

or to duplicate a single part

```
kcs_struct.part_duplicate(anotherStructure, partID)
```

Then the original structure (***anotherStructure***) is not affected by this call. You can also duplicate parts from the current structure by providing its name. After duplicating, it is recommended to use `kcs_struct.part_transform()` to move away the copy from the original, unless the goal is to move (not copy) the parts from another structure to the current one. In this case, however, you will also use the `kcs_struct.part_delete()` function to delete the original parts from ***anotherStructure***.

If we need to replace one part with another, we could remove the original part first, then add the new one. Very often, however, such a replacement changes the component only, without modifying the location and orientation. In such case, it is better to call

```
kcs_struct.part_component(partID, newComponent)
```

The new component should be of the same 'family'; i.e. a profile should be replaced by a profile, a plate by a plate, etc. The function does not return any value.

4.3.4 Production Information

As mentioned in section 4.2.3, Tribon Vitesse can perform tests of compatibility of the profile endcuts in the current structure with the restrictions imposed by the hull-profile restriction file SBH_PROF_RESTRICT. The function `kcs_struct.struct_check_restrict()` performs this test on all profile parts. There is a function able to perform the same test on a specific profile part.

```
   try:
⇒      kcs_struct.part_check_restrict()
       print "Test passed!"
   except:
       print "At least one of the endcuts did not pass the test!"
```

Before the structure can be transferred to the production environment (“split”), the parts should be assigned specific position numbers, which may later be referred to in the material lists.

```
kcs_struct.part_posno(partID, "7") #Assign position number 7
```

The function **kcs_struct.struct_check_posno()** can be then used to verify, if the parts within the structure have unique position numbers. Before splitting the structure, we should also assign the structure (or individually – its parts) to the appropriate assembly. The function **kcs_struct.struct_assembly()** assigns the whole structure to the assembly. We can do the same on the part level

```
kcs_struct.part_assembly(partID, assemblyName)
```

Note, that by providing an empty assembly name, we can remove an existing assembly reference for the given part.

4.4 Standard Structures

The **kcs_struct** module provides a set of functions for handling standard structures. A new structure may be created from the standard structure by using the following pattern

```
   origin = KcsPoint3D.Point3D(20000, 0, 5000)
⇒  kcs_struct.standard_input(standardName, moduleName, \
                             newStructName, origin)
```

The function inputs a standard structure ***standardName*** into the model, creates a new structure ***newStructName*** and marks an existing standard reference. An optional parameter ***origin*** indicates the location, at which the origin of the standard structure is placed.

ⓘ *The Drafting default STRUCT_KEEP_STAND_REF decides, whether the created structure will keep its standard reference, or not. In order to modify the structure, you will have to remove the structure reference.*

Standard structures are created by creating (and storing!) first the normal structure, and then saving its copy as a standard structure using the function

```
kcs_struct.standard_output(normalStructName, stdStructName)
```

Once the standard reference has been established, you can use the function

```
kcs_struct.standard_replace(structName, stdStructName)
```

to update the standard reference for structure ***structName***. The argument ***stdStructName*** is the name of the standard structure, to which the standard reference should be set in the structure ***structName***. By providing the name of the same standard structure as already set you remake the structure. By providing the name of another standard structure you replace the structure with another, created from another standard. By providing an empty name you remove an existing standard reference – the structure ***structName*** becomes modifiable.

ⓘ *After using the function* ***kcs_struct.standard_replace()****, all assembly references for the structure* ***structName*** *are removed, and must be set again.*

When the current structure has a standard reference, it is possible to handle some properties related to this standard reference, including the origin

```
   origin = KcsPoint3D.Point3D()
⇒  kcs_struct.get_standard_origin(origin)
   origin.Z += 100 #update the origin location
⇒  kcs_struct.set_standard_origin(origin)
```

and description

```
⇒  descr = kcs_struct.get_standard_desc()
   print "Current description:", descr
   kcs_struct.set_standard_desc("New description")
```

4.5 Structure References (cableways)

This set of functions handles the cableways references to structures. They are available only from the Cable Modelling application. The reference can be created by using the following pattern

```
    … #model – Model class instance defining the cableway part
    point = KcsPoint3D.Point3D(20000, 0, 10000) #reference location
⇒   kcs_struct.struct_cway_connect(model.Name, model.PartId, \
                                   point, 'CW_SUPP01')
```

where the cableway reference to the structure '**CW_SUPP01**' is created for the cableway part indicated by the user (cableway name – model.Name, part ID – model.PartId). The reference is placed at the given point. The height and width of the structure is used in a fill level check.

ⓘ *The part ID of ZERO means, that the structure can refer to **ANY** part of the cableway. Any non-zero value of the part ID indicates the cableway part to be referred by the structure.*

An existing reference can be removed by calling

```
    kcs_struct.struct_cway_disconnect(model.Name, 'CW_SUPP01')
```

which requires only the cableway and structure names to be provided. If the reference exists, the cableway data can be updated for the current structure – they define the dimensions of the structure cableway component, used for fill level check. Additionally the node point coordinates and the direction and rotation vectors are given

```
    nodePoint = KcsPoint3D.Point3D(0, 375, 0)
    rotation = KcsVector3D.Vector3D(1, 0, 0)
    route = KcsVector3D.Vector3D(0, 1, 0)
    height, width, length = 1000.0, 300.0, 375.0
⇒   kcs_struct.struct_cway_data(nodePoint, route, rotation, \
                                length, width, height)
```

ⓘ *The cableway must be current.*

4.6 Document references

See section 3.6 for the explanation of the document references, and their representation in Vitesse. Here, we also have functions dealing with document references associated with structures.

```
    docRefList = kcs_struct.document_reference_get()
    kcs_struct.document_reference_add(docRef)
    kcs_struct.document_reference_remove(docRef)
```

where ***docRef*** is an instance of the `DocumentReference` class, and ***docRefList*** a Python list of such instances.

Exercise 3: Creation of the plate with bolt holes

Make a Vitesse program that creates a structure with the plate component in the form of a horizontal circle with eight bolt holes ∅12 around its perimeter. The plate is to be created in the Z-plane. The position of the plate together with the other parameters must be input interactively by the user.

Chapter 5

5 Pipes

There is a single Vitesse module handling both pipe and ventilation modelling interface. It is made available to the program by insertion of the statement

```
import kcs_pipe
```

The Vitesse Pipe / Ventilation interface includes the following functionality:

- Handling of Pipe Objects, Pipe Spools, Parts and Joints.
- Production functions like Checking, Ready and Splitting.
- Traversing functions to find Spools and Parts within Pipe.
- Routing and Material functions.
- Default handling.
- Pipe / Vent mode functions.

In a similar manner, as the other Vitesse modules, the `kcs_pipe` module defines the variable `kcs_pipe.error`. When an exception is raised during the execution of the module's functions, the string describing the type of an error will be stored in this variable, for analysis by the exception handlers.

Unless specified otherwise, whenever we mention Pipe Objects and its elements, the same applies to Ventilation Objects. Moreover, there are two functions able to switch between Pipe Modelling and Ventilation Modelling modes.

`kcs_pipe.pipe_mode_activate()`, and
`kcs_pipe.vent_mode_activate()`

Both functions will raise an exception described as '`kcs_ModelIsCurrent`', if the modelling work on an object of the opposite type is not finished, and the object is still current. This means, that it is not possible to switch to Ventilation Modelling mode, when a pipe is active. Trying to enter Pipe Modelling mode, while modelling a Ventilation Object also raises an exception.

The functions in this module often use the instances of some pipe-specific classes that store the attributes of the manipulated objects. One of the examples is `PipeName`, which contains methods for handling the conventional pipe name consisting of the module name, subsystem name and line number. The other classes can be found in the Tribon Vitesse User's Guide and in their source files.

5.1 Definition of pipe model structures

The pipes are categorised in modules and systems. Before a pipe is created, the corresponding module and system objects must exist in the pipe databank. Usually creation of these items is done in the Design Manager application, but it can be also made programmatically in Vitesse, using the `kcs_modelstruct` module. Functions managing outfitting modules are described in section 3.1. Below please find the functions dealing with pipe/ventilation/cable systems.

```
kcs_modelstruct.system_new(systemName, description, surfacePrepCode)
```

The function creates a new system using the provided arguments. ***systemName*** cannot be empty, and cannot be longer than 25 characters. The system can be removed by calling

```
kcs_modelstruct.system_delete(systemName)
```

provided, that it is empty (does not contain any pipe, ventilation or cable objects).

5.2 Pipe Object functions

The operations on Pipe Objects require that such an object is made 'current'. This is possible either by activating an existing pipe or by creating a new one. The outcome of both functions is that a Pipe Object is active and ready to be manipulated. This is exactly the same approach as for handling all other outfitting objects.

5.2.1 Activate/Build/Save process – the overview

Thus, in order to create a new pipe, we have to use the following pattern:

```
        activated = False #a flag - if True, the pipe has been activated
        try:
⇒           pipe_name = KcsPipeName.PipeName("KCS-WW17") #the new pipe name
⇒           kcs_pipe.pipe_new(pipe_name, "Cyan", "TLI", "PPBS3602SX2.9-10")
            activated = True
            … #build the pipe, set its properties, etc.
⇒           kcs_pipe.pipe_save() #save the current pipe
        except:
            if activated:
⇒               kcs_pipe.pipe_cancel() #cancel the current pipe
```

First new feature, that we notice, is that we cannot refer to the pipe names directly. Instead, we are using an instance of the **PipeName** class, that is taking care of checking the basic syntax rules of pipe names. Our ***pipe_name*** variable can be also initialised as:

```
        pipe_name = KcsPipeName.PipeName("KCS", "WW", 17)
```

where we provide individual parts of the pipe name: the module name, the system name, and the line number. It is also acceptable to provide the pipe name, which includes the project prefix, e.g. "ES-KCS-WW17", or four arguments "ES", "KCS", "WW", and 17. The **PipeName** class provides the method for getting the individual parts of the name, i.e. the module, system, and line.

The function **kcs_pipe.pipe_new()** initialises a new pipe. The following information must be supplied: the pipe name as the **PipeName** class instance, the pipe colour (string or Colour class instance), user identification, and the pipe's main component. The last two arguments are optional, so if the user identification is not given or is empty, default user identification string is used; if the pipe's main component is not given, the function tries to fetch the component name from the diagram list. When the component name is given, the user identification must be given too, however it can be set to an empty string, telling the function to use the default value.

If we are updating an existing pipe, then instead of calling **kcs_pipe.pipe_new()**, we should use

```
        kcs_pipe.pipe_activate(name)
```

ⓘ *In Tribon M3, the function* ***kcs_pipe.pipe_new()*** *accepts the fifth, optional argument, being an instance of the* ***SpecSearch*** *class, defining the search criteria (project, specification, function, nominal diameter, flow, and pressure class) and results. See an example on page 34.*

Regardless of the method of activation, the pipe becomes current, and may be built, updated, and various properties may be set. When you want to save the changes made to the pipe, you need to call **kcs_pipe.pipe_save()**, and if for some reasons, the changes should be discarded, the function **kcs_pipe.pipe_cancel()** should be called, provided, that the pipe has been successfully initialised or activated. After calling one of these functions, the pipe is no longer current.

You mail fail initialising the pipe, if such a pipe already exists in the model. This can be checked by calling

```
        pipe_name = KcsPipeName.PipeName("KCS-WW17")
⇒       if kcs_pipe.pipe_exist(pipe_name):
            … #choose another name or remove an existing pipe
```

Another reason of failure in activating a pipe could be, that there is already a pipe current, which has not been yet saved or cancelled. This can be checked by trying to execute a function, that requires a pipe to be current, and intercepting an exception, if no pipe is current.

```
        try:
⇒           activePipe = kcs_pipe.pipe_name_get()
            print "The pipe", activePipe.GetName(), "is current!"
        except:
            print "No pipe is current - we can activate a pipe!"
```

Please note, how to get the pipe name as a string from the **PipeName** class instance (***activePipe*** variable). This can be useful for creating messages or Data Extraction command strings.

If you have determined, that the given pipe exists in the model, you can delete it by calling

```
   pipe_name = KcsPipeName.PipeName("KCS-WW17")
⇒  kcs_pipe.pipe_delete(pipe_name)
```

ⓘ *For working with pipes you need to use the names in the form **<module>–<system&line>** (e.g. 'KCS-WW17'). This is also the syntax to be used for Data Extraction command strings.*

*On the other hand, in order to draw the pipe (`kcs_draft.model_draw()`), you may need to create a Model class instance, providing the name in the presentation form **<project>–<module>–<system&line>** (e.g. 'ES-KCS-WW17'). This form of the pipe name is also returned, when you identify the pipe, using `kcs_draft.model_identify()`. This is controlled by the PIPE_NAME Drafting keyword.*

5.2.2 General functions

When no structure is current, you can duplicate an existing pipe by calling

```
   kcs_pipe.pipe_duplicate(existingPipeName, newPipeName)
```

ⓘ *Note, that as an exception to the general rule, the arguments to this function are simple strings, not PipeName class instances.*

All remaining functions in this section require, that a Pipe Object is current. An exception will be raised, if it is not. In a similar way, like for equipments, and structures, it is possible to transform the active Pipe Object

```
   trans = KcsTransformation3D.Transformation3D()
   … #build the transformation (see equipment example on page 15)
⇒  kcs_pipe.pipe_transform(trans)
```

This function is especially useful in combination with `kcs_pipe.pipe_duplicate()`, so that after duplicating you may activate the copy and move it away from the original.

The detailed information about the active pipe can be written to the file named **<subsystem><module>.res** in the folder specified by the `SB_SHIPPRINT` Tribon environment variable by calling

```
   kcs_pipe.pipe_list()
```

This corresponds to the interactive function `Pipe` → `List`. Another listing, suitable for pipe modelling in batch, can be obtained by calling

```
   kcs_pipe.pipe_regenerate()
```

The function creates the PML file named **<subsystem><module>.prg** located in the folder specified by the **SB_PIPESCH** Tribon environment variable, containing the modelling information for the active pipe. This file can be then edited, and submitted for interpretation by the Pipe Batch Modelling facility.

Each pipe has the following production properties: the bending radius, pipe colour, sketch note, test pressure, planning unit, and the joint, weld, heat, and surface treatment codes. They are encapsulated in Vitesse in the `PipeProp` class.

*See the source file **KcsPipeProp.py** for details.*

See, how we may update the production properties of the current pipe.

```
   prop = KcsPipeProp.PipeProp()
   prop.SetPlanningUnit("PLU01")
   prop.SetTestPressure(22.0)
   prop.SetHeatCode(3)
   prop.SetBendingRadius(2.5)
⇒  kcs_pipe.pipe_properties_set(prop)
```

Only the attributes explicitly set by the appropriate methods of the `PipeProp` class will be set. The other attributes will be not updated. This is a common approach in Pipe Vitesse, that first a class instance is initialised and its attributes are set, then this instance is passed as an argument to a Vitesse function. Instead of having functions with really many arguments, describing the individual properties, our functions accept class instance arguments, that internally hold the information corresponding to the whole set of properties.

Exercise 4: Modifying the colour of pipes in the system

Ask for the pipe system name (verify it), and change the colour of all pipes in this system to the one chosen by the user.

Weldgaps can be added to the pipe spools. The function shown below sets the weldgap size to the whole pipe (to all its spools).

```
kcs_pipe.pipe_weldgap_set(weldGapSize)
```

The script 'Example 1.py' in the 'Vitesse Outfitting Training' subfolder under SB_PYTHON shows many of the functions discussed so far. Take your time to study it

More examples can be found in the Vitesse\Examples\Pipe subfolder in the Tribon M3 folder tree.

5.2.3 Pipe traversing functions

The hierarchical pipe structure can be analysed using the two functions given below. The first one retrieves the parts of the active pipe, whereas the second returns its spools. Both functions operate on an active pipe. Below you can find an example of traversing the **PART** sequence of the active pipe.

```
   #get the FIRST part ID
⇒  partInfo = kcs_pipe.pipe_part_find(1)
   while partInfo[0]: #as long, as status is 1 (success) …
     partID, connectionNumber = partInfo[1], partInfo[2]
     … #work on the given part
⇒    partInfo = kcs_pipe.pipe_part_find(2, partID) #get the NEXT part ID
```

The first argument of the `kcs_pipe.pipe_part_find()` function is the activity code:

- 1 – get the ID of the FIRST part (the second argument is then irrelevant – can be omitted)
- 2 – get the ID of the NEXT part (the second argument should be the ID of the CURRENT part)
- 3 – get the ID of the PREVIOUS part (the second argument should be the ID of the CURRENT part)

The ***partInfo*** variable, returned by the `kcs_pipe.pipe_part_find()` function, is a tuple, consisting of the following members:

[0] ***status***, either ***1*** (the requested part has been found), or ***0*** (the requested part has not been found)
[1] ***part ID*** of the requested part (if ***status*** is ***1***)
[2] ***connection number*** in the requested part (if ***status*** is ***1***), that is connected to the current part

So, in order to find the ID of the LAST part, you need to walk through the sequence of parts in the forward direction, until you get the ***status*** of ***ZERO***. For any pipe part you can try to call this function with activity codes of 2 and 3 to get the next and previous part. Remember to check the ***status***, before using the other elements of the returned tuple.

ⓘ *This function will return all kinds of pipe parts, including non-physical ones, like e.g. joints and connections. Currently Vitesse has no direct support for distinguishing between various types of parts. You may, however, check the existence or the value of some part data using Data Extraction to detect non-physical parts (e.g. component name and total building length not available, ZERO component type, specific TYPE_STRING and PART_TYPE values, etc.)*

The observations for an example pipe with the parts: -1002 (a bend), -1003 (straight segment), and –1005 (weld joint) show the following facts from running queries of the form PIPE.PIPM('pipe_name').PART(part_ID). … :

Data Extraction token	-1002	-1003	-1005
.COMP_NAME	existing component name	existing component name	no data available
.COMP_TYPE	> 0	> 0	0
.TYPE_STRING	Bend	Straight pipe	Weld joint
.PART_TYPE	4	3	73
.TOTBUILD_LENGTH	> 0.0	> 0.0	no data available

A similar pattern can be presented for traversing the spool sequence of the current pipe.

```
   #get the FIRST spool ID
⇒  status, spoolID = kcs_pipe.pipe_spool_find(1)
```

```
      while status: #as long, as there are spools in the pipe (status=1)
        … #work on the given spool
        #get the NEXT spool ID
⇒       status, spoolID = kcs_pipe.pipe_spool_find(2, spoolID)
```

The first argument of the **kcs_pipe.pipe_spool_find()** function is the activity code, having a similar meaning, like for the **kcs_pipe.pipe_part_find()** function, discussed above, only now spools IDs are considered, not part IDs.

ⓘ *In fact, a spool ID is an ID of one of the parts belonging to this spool.*

The result of the **kcs_pipe.pipe_spool_find()** function is again a tuple, consisting of the status code, and the requested spool ID. The status code indicates, whether the requested spool ID has been found (***1***), or not (***0***).

5.2.4 Document references

See section 3.6 for the explanation of the document references, and their representation in Vitesse. Here, we also have functions dealing with document references associated with pipes.

```
      docRefList = kcs_pipe.document_reference_get()
      kcs_pipe.document_reference_add(docRef)
      kcs_pipe.document_reference_remove(docRef)
```

where ***docRef*** is an instance of the **DocumentReference** class, and ***docRefList*** a Python list of such instances.

5.3 Spool functions

The functions in this section operate on pipe spools. The pipe spools are identified by spool IDs, being integer numbers. Many functions require providing the spool ID as one of these parameters. Before using them you have to make the pipe active.

ⓘ *The **spool ID** is the **part ID** of one of the spool parts.*

5.3.1 Handling spool properties

Tribon is able to handle automatic position names for spools. By calling

```
      kcs_pipe.pipe_auto_spool_name_delete()
```

you can remove automatically the position names from the spools in the current pipe. Then, you can request the position names to be automatically assigned by the system by calling

```
      kcs_pipe.pipe_auto_spool_name_set()
```

All spools without a position name receive an automatically assigned position name created using the syntax defined in the default file (SBP_MODEL_DEF Tribon environment variable). You can assign the specific position name to the particular spool by using the following pattern

```
      prop = KcsPipeSpoolProp.PipeSpoolProp()
      prop.SetPosNo('A1') #set user-defined position name
⇒     kcs_pipe.spool_properties_set(spoolID, prop)
```

The function **kcs_pipe.spool_properties_set()** can also update other spool settings, like: the sketch note, sketch type, heat code, surface treatment code, test pressure, and planning unit. All these properties are managed by the **PipeSpoolProp** class. See an example below

```
      prop = KcsPipeSpoolProp.PipeSpoolProp()
      prop.SetSketchNote("Sketch note")
      prop.SetHeatCode(1)
      prop.SetSurfaceTreatmentCode(40)
      prop.SetPlanningUnit("PLU01")
⇒     kcs_pipe.spool_properties_set(spoolID, prop)
```

The properties not set by the appropriate methods of the **PipeSpoolProp** class are not updated.

5.3.2 Handling spool weld gaps

The weld gaps can be set, removed or changed for individual spools in the current pipe.

```
kcs_pipe.spool_weldgap_delete(partID) #remove all weld gaps in the spool
kcs_pipe.spool_weldgap_set(partID, gapSize) #add weld gaps to a spool
kcs_pipe.spool_weldgap_edit(partID, newGapSize) #change weld gap size
```

The ***partID*** argument is the ID of any of the spool parts. The weld gap is removed, added, or its size changed in the whole spool (all its parts). If set or changed, all weld gaps in a spool will have the same size. This can be changed for individual parts, using the **kcs_pipe.part_weldgap_edit()** function (see section 5.4.6).

5.3.3 Traversing spool parts

In a similar way, like for parts or spools within a pipe (see section 5.2.3), we can traverse the parts within a spool

```
     #get the ID of the FIRST part in the spool 'spoolID'
⇒    status, partID = kcs_pipe.pipe_spool_part_find(1, spoolID)
     while status: #as long, as there are parts in the spool (status=1)
       … #work on the given part
       #get the ID of the NEXT part in the spool 'spoolID'
⇒      status, partID = kcs_pipe.pipe_spool_part_find(2, partID)
```

The first argument of the **kcs_pipe.pipe_spool_part_find()** function is the activity code:

- 1 – get the ID of the FIRST part in the spool (the second argument should be the ***spool ID***)
- 2 – get the ID of the NEXT part in the spool (the second argument should be the ID of the CURRENT part)
- 3 – get the ID of the PREVIOUS part (the second argument should be the ID of the CURRENT part)

ⓘ *Note, that to start the process, we have to obtain the ID of the first part in the spool by providing the **spool ID** as an argument. Later on, for moving forwards or backwards in the spool, we provide the **current part ID** as an argument.*

The result of the **kcs_pipe.pipe_spool_part_find()** function is a tuple, consisting of the status code, and the requested part ID. The status code indicates, whether the requested part ID has been found (***1***), or not (***0***).

ⓘ *This function will return all kinds of spool parts, including non-physical ones, like e.g. joints and connections. See the discussion in section 5.2.3 to learn, how to detect non-physical pipe parts.*

5.4 Pipe Part functions

This is the largest part of the **kcs_pipe** module. Almost all the details of pipe parts and their connections to the modelling environment can be handled by Vitesse using the functions described in this section. All of them require, that a pipe is made active, otherwise an exception is raised.

5.4.1 Adding (inserting) and deleting parts

The component may be added to the pipe part in several ways. Various settings, determining the method of adding the component to a pipe part are handled by the **PipePartAddCriteria** class. Study the examples below to understand the possible cases.

- Adding a new pipe part connected to the given pipe part

```
     criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
     criteria.SetConnType("part")           #connection type
     criteria.SetComponent("88.9-10-1330")  #component name
     criteria.SetLength(200)                #part length
⇒    newPartID = kcs_pipe.part_add(partID, connectionNumber, criteria)
```

The new part is added to the pipe part identified by the given ***partID*** at the given connection number, using the provided component, and part length. The function returns the ID of the created part.

When adding a pipe part to be connected to some external pipe part, we need to specify the ***partID*** and ***connectionNumber*** arguments referring to THAT external pipe part, and additionally, specify the name of the external pipe by adding one more line:

```
criteria.SetExternalPipe(externalPipeName)
```

before calling **kcs_pipe.part_add()** function. This modification applies also to all other methods of adding a component to an external pipe part, presented in this section.

ⓘ *The external pipe name can be given either as a string, or the* ***PipeName*** *class instance. Internally, the external pipe name is stored in the* ***PipePartAddCriteria*** *class as the* ***PipeName*** *class instance with the appropriate conversion, if string value is used.*

- If the new pipe part is angled, we need also specify the direction vector of this part – ***direction*** (**Vector3D**).

```
    criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
    criteria.SetConnType("part")
    criteria.SetComponent("B26-6")
    criteria.SetDirection(direction) #direction - Vector3D class instance
⇒   newPartID = kcs_pipe.part_add(partID, connectionNumber, criteria)
```

- Adding a surface part (***direction*** (**Vector3D**) is the direction of the added pipe part, ***point*** (**Point3D**) is the point on the surface)

```
    criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
    criteria.SetConnType("surface")
    criteria.SetComponent("88.9-10-1330")
    criteria.SetDirection(direction)
    criteria.SetSurfPoint(point)
    criteria.SetLength(200)
⇒   newPartID = kcs_pipe.part_add(partID, criteria)
```

- Adding an eccentric surface part (***direction*** (**Vector3D**) is the direction of the added pipe part, ***point*** (**Point3D**) is the point on the surface, and ***orientation*** (**Vector3D**) is the orientation of the added pipe part)

```
    criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
    criteria.SetConnType("surface")
    criteria.SetComponent("K90-3")
    criteria.SetDirection(direction)
    criteria.SetOrientation(orientation)
    criteria.SetSurfPoint(point)
⇒   newPartID = kcs_pipe.part_add(partID, criteria)
```

ⓘ *Note, that in the two last cases we don't provide the connection number as the second parameter – the part is added at the surface point.*

- Adding an eccentric part (***orientation*** (**Vector3D**) is the orientation of the added pipe part)

```
    criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
    criteria.SetConnType("part")
    criteria.SetComponent("K90-3")
    criteria.SetOrientation(orientation)
⇒   newPartID = kcs_pipe.part_add(partID, connectionNumber, criteria)
```

As already said, each of the examples presented above has also a variant, where the connection is made to an external pipe. Then the ***partID*** and ***connectionNumber*** should refer to the external pipe, and additionally the external pipe name should be given by calling

```
criteria.SetExternalPipe(externalPipeName)
```

In Tribon M3, the function **kcs_pipe.part_add()** can be also called with the additional, last argument, being the **SpecSearch** class instance.

```
newPartID = kcs_pipe.part_add(partID, connectionNumber, criteria, \
                              specSearch), or
newPartID = kcs_pipe.part_add(partID, criteria, specSearch)
```

This class holds the Specification information. This argument can be used as follows:

```
      import KcsSpecSearch
      specSearch = KcsSpecSearch.SpecSearch() #initialise
      specSearch.SetSCProject("SP")     #Sets the project criterion
      specSearch.SetSCSpec("WW")        #Sets the specification criterion
      specSearch.SetSCFunction("BALL") #Sets the function criterion
      specSearch.SetNomDia(100)         #Sets the nominal diameter criterion
      specSearch.Search()               #Perform the search …
      criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
      … #set the criteria for adding a pipe
   ⇒  newPartID = kcs_pipe.part_add(partID, connectionNumber, criteria, \
                                  specSearch)
```

In a similar way, we can **INSERT** a new part in an existing part

```
      criteria = KcsPipePartAddCriteria.PipePartAddCriteria()
      criteria.SetComponent("PFSODIN2501PN10-200")
      criteria.SetDistance(75.0)
   ⇒  newPartID = kcs_pipe.part_insert(partID, connectionNumber, criteria)
```

The criteria must specify the component name, and the distance from the connection point to the node point of inserted part. The function returns the ID of the inserted part.

ⓘ *As before, you may also use the last, optional argument **specSearch**, being the `SpecSearch` class instance.*

Any part, which is 'deletable', can be removed from the active pipe. If the part to be deleted is not 'deletable', the operation 'part to frame' must be performed, otherwise an exception will be raised. In order to actually remove the part, just call

```
      kcs_pipe.part_delete(partID)
```

5.4.2 Resizing and re-specifying parts

Sometimes, the existing parts need to be replaced by similar ones, with a different component, matching a specific diameter, flow or pressure class. A specification, represented by an instance of the `SpecSearch` class, is used to find suitable components. The specification may also be changed (`kcs_pipe.part_respec()`).

```
   ⇒  res = kcs_pipe.part_resize(0, partID, nominalDiameter, nominalDiameter2)
      if res[0] == 0: #success!
        prevPartID, newPartID, nextPartID, operationInfo = res[1:]
      else:
        print "Selection ambiguous!"
```

The first argument is the ***option***, which currently should be always set to ZERO. Then, we need to provide the ID the part being resized and the new nominal diameter ***nominalDiameter***. The CURRENT specification is searched to find the suitable component. If successful, the first element of the returned tuple has the value of ZERO. Otherwise, this element has the value ***1***, which means, that the component is multidimensional, and the second nominal diameter ***nominalDiameter2*** (optional), being the nominal diameter for the connection no. 2, is required to further limit the search.

What are the other elements of the returned tuple? Apart from the status code, we have also:

[1]	ID of the PREVIOUS part
[2]	new ID of the CURRENT part
[3]	ID of the NEXT part
[4]	operation information (string)

This information can be used to iterate over the sequence of parts in the pipe to resize appropriately all parts in the pipe. In a similar way we can re-specify the part, changing the specification for the given part.

```
   ⇒  res = kcs_pipe.part_respec(0, partID, specName, nominalDiameter2)
      if res[0] == 0: #success!
        prevPartID, newPartID, nextPartID, operationInfo = res[1:]
      else:
        print "Selection ambiguous!"
```

Instead of the nominal diameter (in the current specification) we provide the name of the new specification. The meaning of the other arguments and the returned values is the same, as for the `kcs_pipe.part_resize()` function.

ⓘ *These functions can return **–1** as the ID of the previous or next, which indicates the end of the part sequence in the particular direction.*

5.4.3 Transforming pipe parts

There are several transformations available for a pipe part. First, a part may be flipped by calling

```
kcs_pipe.part_flip(partID)
```

If this is not possible, an exception described as '`kcs_FlipNotPossible`' will be raised. Next, the part can be rotated or turned by calling

```
kcs_pipe.part_rotate(partID, connectionNumber, Angle), or
kcs_pipe.part_turn(partID, connectionNumber, Angle)
```

respectively. These functions rotate and turn the particular pipe part connection. The angles are given in degrees. Flanges of type 2601 can be also inclined using the function

```
kcs_pipe.part_incline(partID, direction)
```

where the ***direction*** vector (`Vector3D`) indicated the new direction of the part. Trying to use this function on another type of part raises an exception.

5.4.4 Handling part connections

The current part may be connected to another part of the same pipe by calling

```
kcs_pipe.part_connect(partID1, connNo1, partID2, connNo2)
```

If the part no. 2 belongs to an external pipe/equipment, then the correct syntax is:

```
kcs_pipe.part_connect(partID1, connNo1, extName, partID2, connNo2)
```

where ***extName*** is the name of the external pipe/equipment to be connected. An exception is raised, if the connection ***connNo1*** of the given part ***partID1*** is already connected. For disconnecting the parts you should use the function

```
kcs_pipe.part_disconnect(partID, connNo)
```

This is valid both for connections between two parts of the same pipe, and for connections between a part from the current pipe, and a part from some external pipe/equipment. An exception is raised, if the given connection is not connected, when this function is called.

If you want to learn the location of the given part connection, use the following pattern:

```
    point = KcsPoint3D.Point3D()
⇒   if kcs_pipe.part_conn_coord_get(partID, connNo, point):
        … #use the connection point coordinates
```

The function retrieves the coordinates of the given part connection point, and updates the ***point*** argument with the values found. The function result is the status code with the value of **1** (success) or **0** (failure). You can also determine, which of the connections of the given part is closest to the given point (for example, indicated by the user). An example:

```
    … #get the point from the user (Point3D)
⇒   connNo = kcs_pipe.part_conn_find(partID, point)
```

It is possible to learn the ID of the part connected at the given connection

```
connectedPartID = kcs_pipe.part_conn_part_get(partID, connNo)
```

If there is no connection established at the given connection number ***connNo***, an exception is raised, described as '`kcs_ConnectedPartNotFound`'.

For boss joint connections we can also change the boss connection code

```
kcs_pipe.part_boss_conn_type_set(partID, connNo, code)
```

where the ***code*** may take the following values:

1 – on surface **2** – insert **3** – extrude **4** – saddle **5** – none

We have also the possibility to control the spool limits at the connections of the pipe parts. To set the spool limit at the given part connection, we should call

```
kcs_pipe.part_spool_limit_set(partID, connNo, 1)
```

and to reset it …

```
kcs_pipe.part_spool_limit_set(partID, connNo, 0)
```

5.4.5 Handling the part excess information

We can control the end excess at the pipe part connections. The end excess is set by calling

```
kcs_pipe.part_end_excess_set(partID, connNo, excessLength, 1)
```

and reset by

```
kcs_pipe.part_end_excess_set(partID, connNo, excessLength, 0)
```

Setting an end excess is the default action, so you may omit the last argument of ***1***. To reset the excess, you have to provide the last argument of ***0***. Setting an end excess is allowed only for straight pipe parts, at the end of the pipe material.

The similar method applies for setting/resetting the feed excess

```
kcs_pipe.part_feed_excess_set(partID, 1) #set the feed excess
kcs_pipe.part_feed_excess_set(partID, 0) #reset the feed excess
```

and the minimum feed excess for a straight pipe.

```
kcs_pipe.part_feed_min_set(partID, 1) #set the minimum feed excess
kcs_pipe.part_feed_min_set(partID, 0) #reset the minimum feed excess
```

Feed excess are allowed on straight parts only. If the minimum feed excess are not set, the default value will be fetched from the bending machine object.

5.4.6 Handling pipe part weld gaps

The weld gaps can be set, removed or changed for individual parts in the current pipe.

```
kcs_pipe.part_weldgap_delete(gapPartID) #remove the given weld gap
kcs_pipe.part_weldgap_set(partID, connNo, gapSize) #add weld gap
kcs_pipe.part_weldgap_edit(gapPartID, newGapSize) #change weld gap size
```

The weld gap is added at the given part connection as an additional, non-physical pipe part, whose part ID (***gapPartID***) is then returned by the function **`kcs_pipe.part_conn_part_get(partID, connNo)`**. In Vitesse, you must be prepared to handle also such non-physical parts in the sequence of the pipe parts. See the discussion on page 3030, about the methods of distinguishing non-physical pipe parts from the normal pipe parts.

*Note, that for deleting or editing the weld gap, you need to provide the ID of the weld gap part itself (**gapPartID**), and not the ID of the physical part, to which the weld gap is added.*

5.4.7 Handling connections between pipe parts and structures

A connection between pipe parts and structures may be established by the function

```
kcs_pipe.part_structure_connect(partID, structureName, alias)
```

where ***alias*** is the structure alias name. Then, the connection may be disconnected by calling

```
kcs_pipe.part_structure_disconnect(partID, structureName)
```

Finally, the list of all connected structures may be obtained by using the following pattern:

```
⇒   structConnList = kcs_pipe.part_structure_get(partID)
    for conn in structConnList: #loop over the connections
         structureName = conn.GetStructure()
         aliasName = conn.GetAlias()
```

The returned list consists of the instances of the **ResultPipeStructConn** class, holding the information about the name and alias of the connected structures.

Tribon Vitesse offers another set of functions handling the connections between pipe parts and structures AT THE SPECIFIC POINT (**Point3D**), located on the structure, and on the part's frame. To establish the connection, we have to follow the pattern:

```
point = … #define the point, at which the connection is made
kcs_pipe.part_ext_structure_connect(partID, point, structureName)
```

The true 2-way connection is then stored in both pipe and structure. To disconnect the structure from the pipe part, use the function

```
kcs_pipe.part_ext_structure_disconnect(partID, structureName)
```

If you wish to disconnect ALL structures connected to the given pipe part, just omit the structure name

```
kcs_pipe.part_ext_structure_disconnect(partID)
```

5.4.8 Pipe part properties

The sequence of connected pipe parts forms a branch. It is possible to get the branch number of the particular part by calling

```
branchNo = kcs_pipe.part_branch_get(partID)
```

ⓘ *Note that in the next release the name of this function may be changed to* ***kcs_pipe.pipe_part_branch_get()***

The spool name (string) of the spool, that the given part belongs to, can be obtained by calling

```
spoolName = kcs_pipe.pipe_spool_get(partID)
```

Finally, production properties can be set or updated for the given part. They are represented in Vitesse by instances of the **PipePartProp** class, which handles the following information: material note, acquisition code, joint, weld, placing, and loose codes, insulation status and insulation component name.

```
    prop = KcsPipePartProp.PipePartProp()
    prop.SetMaterialNote("Material note")
    prop.SetAcquisitionCode(1)
    prop.SetJointCode(3)
    prop.SetInsulation("RW20") #Set insulation component
⇒   kcs_pipe.part_properties_set(partID, prop)
```

Only the attributes explicitly set by using the appropriate methods of the **PipePartProp** class are applied to the pipe part. The remaining attributes are not update. Note, that to remove the insulation component, you should call

```
prop.SetInsulation("RW20", 0) #Do not use the insulation
```

5.5 Joint functions

Adding a joint is very similar to adding a part. Various properties of the joint are here handled by instances of the **PipeJointAddCriteria** class. It determines the joint type (insert, thread, weld, mitre), direction (for angled joints), external pipe name (for external joints), and insert distance. Study the examples given below to understand possible cases of adding a joint.

- Adding a thread joint

```
     criteria = KcsPipeJointAddCriteria.PipeJointAddCriteria()
     criteria.SetJointType("thread")
⇒    JointPartID = kcs_pipe.joint_add(partID, connectionNumber, criteria)
```

- Adding a mitre joint (***direction*** (`Vector3D`) is the joint direction)

```
     criteria = KcsPipeJointAddCriteria.PipeJointAddCriteria()
     criteria.SetJointType("mitre")
     criteria.SetDirection(direction)
⇒    JointPartID = kcs_pipe.joint_add(partID, Conn, criteria)
```

- Adding an external joint (any kind)

```
     criteria = KcsPipeJointAddCriteria.PipeJointAddCriteria()
     … #set up joint adding criteria
⇒    criteria.SetExternalPipe(externalPipeName)
     JointPartID = kcs_pipe.joint_add(partID, Conn, criteria)
```

ⓘ *The external pipe name can be given either as a string, or the* ***`PipeName`*** *class instance. Internally, the external pipe name is stored in the* ***`PipeJointAddCriteria`*** *class as the* ***`PipeName`*** *class instance with the appropriate conversion, if string value is used.*

In the same way, we can **INSERT** the joint in an existing pipe part:

```
     criteria = KcsPipeJointAddCriteria.PipeJointAddCriteria()
     criteria.SetJointType("weld")
     criteria.SetDistance(100.0)
⇒    JointPartID = kcs_pipe.joint_insert(partID, connectionNumber, criteria)
```

Then, the ***criteria*** variable must specify the joint type ("insert" or "weld"), and the distance from the connection to the node point of the insert. Both functions return the ID of the added or inserted joint part.

5.6 Routing functions

The routing of pipes is always a 3-stage process:

- initiate routing, providing the starting route point
- continue routing, providing (repeatedly) the intermediate route points
- end routing, providing the end route point

At each stage, the information about the point in the routing sequence is provided by the specialised class **`PipeRoute`**. Apart from the point itself, it contains the additional attributes of the given route segment. The **`PipeRoute`** class stores the following information: route type, free point (route from point) or surface point (route from surface) coordinates, part ID and connection number (route from part), and the external pipe name (routing from/to external pipe).

To start the routing process, we should set up the **`PipeRoute`** class instance, and call the function **`kcs_pipe.pipe_route_start()`**. Below you can find some examples of various possible cases.

- routing from a point (***point*** (`Point3D`) – the free point)

```
     prop = KcsPipeRoute.PipeRoute()
     prop.SetRouteType('point')
     prop.SetFreePoint(point)
⇒    kcs_pipe.pipe_route_start(prop)
```

- routing from surface (***point*** (`Point3D`) – surface point, ***partID*** – ID of the pipe part)

```
     prop = KcsPipeRoute.PipeRoute()
     prop.SetRouteType('surface')
     prop.SetSurfPoint(point)
     prop.SetPartId(partID)
     kcs_pipe.pipe_route_start(prop)
```

- routing from part (***partID*** – ID of the pipe part, ***connNo*** – connection number)

```
prop = KcsPipeRoute.PipeRoute()
prop.SetRouteType('part')
prop.SetPartId(partID)
prop.SetConnection(connNo)
kcs_pipe.pipe_route_start(prop)
```

- routing from/to and external pipe part (any route type)

```
   prop = KcsPipeRoute.PipeRoute()
   … #set up the appropriate attributes of 'prop'
⇒  prop.SetExternalPipe(externalPipeName)
   kcs_pipe.pipe_route_start(prop)
```

ⓘ *The external pipe name can be given either as a string, or the* ***PipeName*** *class instance. Internally, the external pipe name is stored in the* ***PipeRoute*** *class as the* ***PipeName*** *class instance with the appropriate conversion, if string value is used.*

The **kcs_pipe.pipe_route_start()** function initiates the routing process and sets up the starting point. After starting the routing, we can add intermediate point in the route sequence by calling repeatedly the function

```
kcs_pipe.pipe_route_point(prop)
```

where we set up the variable ***prop*** (**PipeRoute**) in a similar way, as shown above. As a result, a new point is added to the route sequence. Finally, one more function has to be called to end the routing process.

```
firstPartID, lastPartID = kcs_pipe.pipe_route_end(prop)
```

The function also requires an instance of the **PipeRoute** class to be given as an argument, thus defining the final route point. It returns a tuple consisting of two part IDs, of the first and last frame part created. This is enough to retrieve the whole sequence of created parts (their IDs) by using the function **kcs_pipe.pipe_part_find()** (see page 3030) with the activity code of 1 or 2, to get the successive part IDs.

5.7 Material functions

After creating the pipe frame, material is added to it. The details of material to be added are defined as instances of the **PipeMaterial** class. This class stores the following information: method of adding the material (add material to pipe, add to straight part, add to bent part, add mitre material), component names for the straight and bent parts (separately).

To add the material to all parts of the current pipe, use the following pattern:

```
   prop = KcsPipeMaterial.PipeMaterial()
   prop.SetStraightMaterial("88.9-10-1330")  #material for straight parts
   prop.SetBendMaterial("88.9-10-1330")      #material for bent parts
⇒  kcs_pipe.pipe_material_set(prop)
```

Of course, you can set different component names for straight and bend parts. If you want to set the material for straight parts only, or for bent parts only, specify the material type by

```
prop.SetType("straight"), or
prop.SetType("bend")
```

The example above sets the appropriate material to ALL branches of the current pipe. If you want to dress a single branch with the given material, provide the branch number in the call to the function **kcs_pipe.pipe_material_set()**.

```
kcs_pipe.pipe_material_set(branchNumber, prop)
```

If the branch number is omitted, ALL branches are dressed with the given material. You can also provide the **SpecSearch** class instance to specify the restrictions imposed by the Specification

```
kcs_pipe.pipe_material_set(branchNumber, prop, specSearch)
```

ⓘ *See an example on page 34, about how to set up the* ***SpecSearch*** *class instance.*

You can also dress specific pipe parts with the material. The example below shows, how to set up the material for a single part

```
    prop = KcsPipeMaterial.PipeMaterial()
    … #set up the material (component names for straigh or bend parts)
⇒   newPartID = kcs_pipe.part_material_set(partID, prop)
```

The part can be here either a straight part or the frame bend part. If you have two straight frame parts, that connect to the bend, you should set up the bent material, and call

```
    newPartID = kcs_pipe.part_material_set(partID1, partID2, prop)
```

The similar syntax applies to setting up the mitred connection of two straight parts

```
    prop = KcsPipeMaterial.PipeMaterial()
⇒   prop.SetType("mitre")
    kcs_pipe.part_material_set(partID1, partID2, prop)
```

*Note, that in the last two cases you provide **TWO** part IDs of the straight parts, that connect, forming the bend or mitre. For the mitred connection you **DON'T** provide any material, since the straight parts, that are connected are already dressed with material – no new part is created.*

The result of the **`kcs_pipe.part_material_set()`** is the ID of the new part created in place of the frame part(s).

*This function also accepts the last argument, being the **`SpecSearch`** class instance.*

You will get an error, if you try to dress with material parts, that are already dressed. If you want to dress a different material, you should remove the existing material first. You may also consider using the functions **`kcs_pipe.part_resize()`** or **`kcs_pipe.part_respect()`**. The function shown below removes all material from the current pipe.

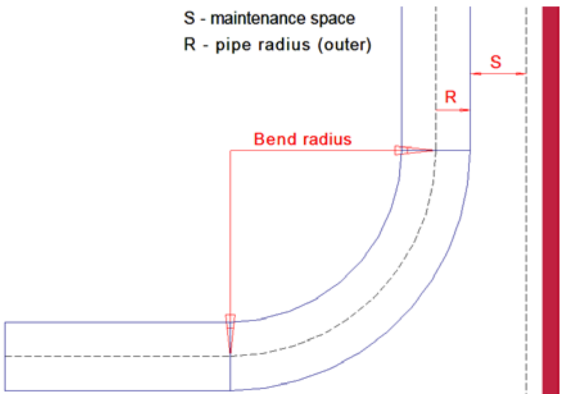
```
    kcs_pipe.pipe_material_remove()
```

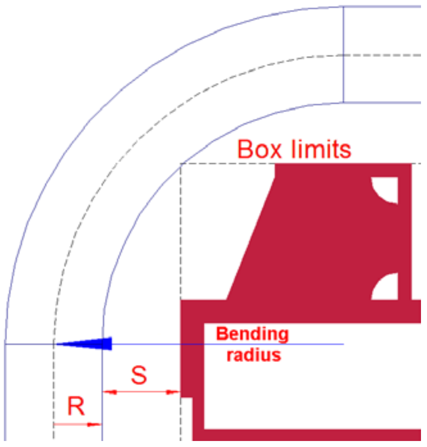
If you want to remove the material from the given part only, you should rather call

```
    kcs_pipe.part_material_remove(partID)
```

Exercise 5: Routing a pipe over a structure

Create an application that creates the pipe route going over an obstacle (user-indicated structure) in the X-Z plane. For determining the obstacle size and location, use the structure's extension box available through Data Extraction. The user should be prompted to indicate the structure, a point before the structure, and a point behind the structure. An appropriate locking should be set to guarantee, that both points lie on the same plane defined by the Y co-ordinate of the first point. Finally, dress the frame parts with the material provided by the user.





Provide a safety distance (S) between the pipe and the extension box. It should be chosen, so that the bent part of the pipe does not collide with the extension box of the structure.

5.8 Production functions

A pipe must meet many requirements, before it can be considered suitable for production. The functions in this section perform activities related to the preparation of a pipe for production purposes.

The function `kcs_pipe.pipe_check()` can perform various tests on the current pipe, and return the report about its findings. The tests are selected by calling appropriate methods of the `PipeCheck` class.

Test	`PipeCheck` class method	Test	`PipeCheck` class method
Position name checking	SetPosNoCheck()	Rotation checking	SetRotationCheck()
Excess checking	SetExcessCheck()	Joint checking	SetJointCheck()
Bending checking	SetBendingCheck()	Extrusion checking	SetExtrusionCheck()
Feed checking	SetFeedCheck()	Not connected connections checking	SetNonConnCheck()
Frame checking	SetFrameCheck()	Length checking	SetLengthCheck()
Loose checking	SetLooseCheck()	Weld gaps checking	SetWeldGapsCheck()

```
    check = KcsPipeCheck.PipeCheck()
    check.SetPosNoCheck() #request the position name check
    check.SetBendingCheck() #request the bending check
    … #request other checks
⇒   testResults = kcs_pipe.pipe_check(check)
```

The function returns a list consisting of various information about the test results. The most important is the first elements of this list – it contains the overall result of the checks:

```
⇒   status = testResults[0]
    if status == 1: print "OK"
    elif status == 2: print "OK with warnings"
    elif status == 3: print "Not OK, due to controls"
    elif status == 4: print "Not OK, due to other reasons"
```

If the ***status*** is greater than ***1***, we can inspect the remaining elements of the ***testResults*** list.

```
    nMessages = testResults[1]      #Number of messages
⇒   for msgInfo in testResults[2:]: #loop over the messages
      partID = msgInfo.GetPartId()      #part ID
      connNo = msgInfo.GetConnection() #connection number
      msgNo  = msgInfo.GetMessageNo()  #message number
      msg    = msgInfo.GetMessage()    #message text
      print msgNo, partID, connNo, msg
```

Starting from the index 2, the returned list contains the instances of the `ResultPipeCheck` class, holding the information about the given message (message number and text, part ID and connection number referred to by the message).

📖 *The list of message numbers and associated message texts can be found in the Tribon Vitesse User's Guide in the description of the* ***kcs_pipe.pipe_ready()*** *function.*

There are production checks, that are performed automatically by Tribon, when the user tries to make the pipe 'ready'. The function `kcs_pipe.pipe_check_settings()` activates or deactivates the specific production checks, using an instance of the `PipeCheckSettings` class.

```
    settings = KcsPipeCheckSettings.PipeCheckSettings()
    settings.SetPosNoCheck()
    settings.SetBendingCheck()
    settings.SetExtrusionCheck()
⇒   kcs_pipe.pipe_check_settings(settings, partID)
```

Certain production checks are made on the specific part (bending and extrusion checks) – be sure to provide the corresponding part ID. If you don't use these checks (position name check only) – then the function should be called with a single argument only

```
    kcs_pipe.pipe_check_settings(settings)
```

ⓘ *By providing an argument of **0** in the SetXXXCheck() methods of the* **`PipeCheck`** *and* **`PipeCheckSettings`** *classes, you can DISABLE the given check.*

Then, when you call the function

```
testResults = kcs_pipe.pipe_ready()
```

the checks activated for the given pipe (and parts) will be performed, and a list, similar to the one returned by the **`kcs_pipe.pipe_check()`** function, is returned. If everything is OK, the pipe is marked as 'ready' and saved on the databank.

The background splitting job for a pipe may be started by calling

```
kcs_pipe.pipe_split(pipeName)
```

where ***pipeName*** is a **`PipeName`** class instance, defining the pipe to be splitted. This pipe has to be 'ready', and no pipe should be current.

5.9 Miscellaneous functions

The pipe modelling system is controlled by a number of defaults stored in the file identified by the Tribon environment variable SBP_MODEL_DEF. The **`kcs_pipe`** module provides the functions for reading and setting the defaults.

```
value = kcs_pipe.default_value_get("AUTOSPLIT")
kcs_pipe.default_value_set("AUTOSPLIT=EDIT_SPLIT_AND_DELETE")
… #work with the new setting
kcs_pipe.default_value_set("AUTOSPLIT=" + value) #restore the original
```

ⓘ *The function* ***`kcs_pipe.default_value_set()`*** *sets the value of the given pipe default, for the current process only.*

While a pipe is current, we may ask for the ID of the bending machine object currently in use

```
bendMachObj_ID = kcs_pipe.default_bendobj_id_get()
```

and change the bending machine object by calling

```
kcs_pipe.default_bendobj_id_set(anotherBendMachObj_ID)
```

where we provide the ID of the new bending machine object to be used.

Chapter 6

6 Cables and cableways

The Tribon Vitesse Cable interface includes the functions for handling cables, cableways and cable penetrations. They are available in the module **kcs_cable**. In order to use these functions, the program must import the module:

```
        import kcs_cable
```

Following the convention used in the whole Vitesse API, when an exception is raised during the execution of the module's functions, the error description is stored in the variable **kcs_cable.error**. The programmer is encouraged to use **try ... except ...** statement, and check the value of **kcs_cable.error** variable in the **except** clause.

Cables are assigned to the outfitting systems, and cableways to outfitting modules. These modelling structures are usually managed by the Design Manager application, but they can be also handled by Vitesse. Functions handling outfitting modules are described in section 3.1, and functions concerning outfitting systems – in section 5.1.

6.1 Cable functions

The operations on Cable Objects require that such an object is made 'current'. This is possible either by activating an existing cable or by creating a new one. The outcome of both functions is that a Cable Object is active and ready to be manipulated. This is exactly the same approach as for handling all other outfitting objects.

6.1.1 Activate/Build/Save process – the overview

In order to work with an existing cable, the following pattern can be used

```
        activated = False #cable not active
        try:
  ⇒         kcs_cable.cable_activate(systemName, cableName) #activate cable
            activated = True #cable made active
            … #work with the cable
  ⇒         kcs_cable.cable_save() #save the cable, deactivate it
        except:
  ⇒         if activated: kcs_cable.cable_cancel()
            print "Error encountered:", kcs_cable.error
```

When activating a cable, you need to provide both the system name, and the cable name. All remaining functions work on the activated cable. If you want to create a new cable, then additional activities are required:

```
        status = 0 #not created, not active
        try:
  ⇒         kcs_cable.cable_new(systemName, cableName)      #create new cable
            status = 1 #created, but not active
  ⇒         kcs_cable.cable_activate(systemName, cableName) #and activate it
            status = 2 #created and active
            … #work with the cable
  ⇒         kcs_cable.cable_save() #save the cable, deactivate it
        except:
  ⇒         if status == 2: kcs_cable.cable_cancel() #deactivate the cable
  ⇒         if status >= 1: kcs_cable.cable_delete(systemName, cableName)
            print "Error encountered:", kcs_cable.error
```

The ***status*** variable controls the progress of the activities. Instead, nested **try: … except: …** statements could be used. After finishing the work on the cable you should save it by calling **kcs_cable.cable_save()**. In the case of an error, the cable should be deactivated first with the function **kcs_cable.cable_cancel()**, assuming, that it has been successfully activated. All modifications made to the cable will then be discarded. Furthermore, you can delete from the databank an incompletely defined cable by calling `kcs_cable.cable_delete()`. Then, any existing cable connections are automatically removed.

ⓘ *Please note, that creating a new cable does not activate it – you need to call both* `kcs_cable.cable_new()` *and* `kcs_cable.cable_activate()`.

You may fail creating a cable, if a cable with the same system and name already exists in the model. You can verify the existence of a given cable in the databank, as shown below

```
⇒   if kcs_cable.cable_exist(systemName, cableName):
         … #work with the EXISTING cable
```

You may also fail in activating a cable, if another cable is active. There is no direct method of checking, whether a cable is active, or not, but you may try to call a function, requiring that a cable is active, and intercepting an exception.

```
    try:
⇒        cableName = kcs_cable.cable_name_get()
⇒        systemName = kcs_cable.cable_system_get()
         print "The cable %s in the system %s is current" % \
               (cableName, systemName)
    except:
         print "No cable is current"
```

The functions `kcs_cable.cable_name_get()` and `kcs_cable.cable_system_get()` return the names of the cable and of the system for the current cable. They will raise an exception, if no cable is current.

The function `kcs_cable.cable_save()` saves the current cable without performing any additional checks. You can, however, request the ready checks to be performed before actually saving the cable, by calling

```
kcs_cable.cable_ready()
```

After successfully calling this function, the current cable is saved and deactivated. If the ready checks fail, an exception is raised, and depending on their results, the `kcs_cable.error` variable can take the following values: '`kcs_NoLength`' (cable length could not be calculated), '`kcs_CwayNotValid`' (status of the referenced cableway is not OK), '`kcs_ModelNotConnected`' (the cable is not connected), '`kcs_IllegalFunction`' (cable transfer to PDI failed).

6.1.2 General functions

When a cable is active, you may set (or update) the cable component reference

```
kcs_cable.cable_component_set(componentName)
```

Then, you can connect cables to the equipments or disconnect them. For establishing the connection, you need to provide the equipment name and the cable connection number: 1 – start point, 2 – end point

```
kcs_cable.cable_equipment_connect(equipmentName, connectionNumber)
```

An exception is raised, when the equipment is current or locked by another user. or if the given connection of the cable is already connected. To break the connection, call the function

```
kcs_cable.cable_equipment_disconnect(connectionNumber)
```

where ***connectionNumber*** is again ***1*** for the start point, or ***2*** for the end point of the cable. The equipment should not be current nor locked.

6.1.3 Document references

See section 3.6 for the explanation of the document references, and their representation in Vitesse. Here, we also have functions dealing with document references associated with cables.

```
docRefList = kcs_cable.cable_document_reference_get()
kcs_cable.cable_document_reference_add(docRef)
kcs_cable.cable_document_reference_remove(docRef)
```

where ***docRef*** is an instance of the `DocumentReference` class, and ***docRefList*** a Python list of such instances.

6.2 Cableway Object functions

6.2.1 Activate/Build/Save process – the overview

Before manipulating the cableway object, it must be made 'current'. It can be done either by activating an existing cableway, or by creating a new one. The example below shows how to work on an existing cableway

```
        activated = False
        try:
⇒           kcs_cable.cway_activate(cablewayName) #activate the cableway
            activated = True
            … #work on the cableway
⇒           kcs_cable.cway_save() #Save the changes
        except:
⇒           if activated: kcs_cable.cway_cancel() #Discard the changes
            print "Error encountered:", kcs_cable.error
```

When activating a cableway, you need to provide the cableway name. All remaining functions work on the activated cableway. If you want to create a new cableway, then additional activities are required:

```
        status = 0 #not created, not activated
        try:
⇒           kcs_cable.cway_new(cablewayName, moduleName, colour)
            status = 1 #created, but not activated yet
⇒           kcs_cable.cway_activate(cablewayName)
            status = 2 #created and activated
            … #work on the cableway
⇒           kcs_cable.cway_save()
        except:
⇒           if status == 2: kcs_cable.cway_cancel()
⇒           if status >= 1: kcs_cable.cway_delete(cablewayName)
            print "Error encountered:", kcs_cable.error
```

The function `kcs_cable.cway_new()` initialises a new cableway in the given module, and assigns it the given colour, which can be either a string (e.g. "White") or the `Colour` class instance, representing the chosen colour. The ***status*** variable controls the progress of the activities. Instead, nested `try:` … `except:` … statements could be used.

After finishing the work on the cableway you should save it by calling `kcs_cable.cway_save()`, which additionally deactivates it. In the case of an error, the changes made to the current cableway should be discarded, and the cableway deactivated with the function `kcs_cable.cway_cancel()`.

Furthermore, you can delete from the databank an incompletely defined cableway by calling `kcs_cable.cway_delete()`. Remember to remove first any cables from the cableway. Penetrations defined on the cableway will be automatically removed.

ⓘ *Note, that creating a new cableway does not activate it – you need to call both `kcs_cable.cway_new()` and `kcs_cable.cway_activate()`.*

You may fail creating a cableway, if a cableway with the same name already exists in the model. You can verify the existence of a given cableway in the databank, as shown below

```
⇒       if kcs_cable.cway_exist(cablewayName):
            … #work with the EXISTING cableway
```

You may also fail in activating a cableway, if another cableway is active. There is no direct method of checking, whether a cableway is active, or not, but you may try to call a function, requiring that a cableway is active, and intercepting an exception.

```
    try:
⇒       cablewayName = kcs_cable.cway_name_get()
        print "The cableway %s is current" % cablewayName
    except:
        print "No cableway is current"
```

The function **kcs_cable.cway_name_get()** returns the name of the current cableway. It will raise an exception, if no cableway is current.

The function **kcs_cable.cway_save()** saves the current cableway without performing any additional checks. You can, however, request the ready checks to be performed before actually saving the cableway, by calling

```
    kcs_cable.cway_ready()
```

After successfully calling this function, the current cableway is saved (and transferred to PDI, if applicable), and deactivated.

If the ready checks fail, an exception is raised, and depending on their result, the **kcs_cable.error** variable can take the following values: '**kcs_NoPlanningUnit**' – invalid planning unit, '**kcs_NoPositionNumber**' – position numbers in the cableway not assigned, or duplicated, '**kcs_NoAssembly**' – assembly reference not defined, '**kcs_IllegalFunction**' – the cableway should be transferred to PDI, but it failed.

6.2.2 Document references

See section 3.6 for the explanation of the document references, and their representation in Vitesse. Here, we also have functions dealing with document references associated with cableways.

```
    docRefList = kcs_cable.cway_document_reference_get()
    kcs_cable.cway_document_reference_add(docRef)
    kcs_cable.cway_document_reference_remove(docRef)
```

where ***docRef*** is an instance of the **DocumentReference** class, and ***docRefList*** a Python list of such instances.

6.2.3 Routing cableways

As for pipes, routing cableways is a 3-stage process:

- initiate routing, providing (optionally) the starting route point
- continue routing, providing (repeatedly) the intermediate route points
- end routing, providing (optionally) the end route point

All three functions given below expect a cableway to be active. Comparing to the similar activity in Pipe Vitesse, the task is here much simpler. The general pattern is shown below:

```
    point = KcsPoint3D.Point3D(startX, startY, startZ)
⇒   kcs_cable.cway_route_start_point(point) #initialise and set point no. 0
    for n in range(1, nPoints-1): # for all intermediate route points
        … #update 'point'
⇒       kcs_cable.cway_route_point(point)
    … #update 'point' - last route point
⇒   partID1, partID2, direction = kcs_cable.cway_route_end_point(point)
```

where ***partID1***, and ***partID2*** are the IDs of the first and last cableway part, that has been created. ***direction*** indicates the order of parts, as stored in the model:

- +1 – ***partID1*** is stored BEFORE ***partID2*** in the cableway model
- -1 – ***partID1*** is stored AFTER ***partID2*** in the cableway model
- 0 – only one part has been created (***partID1 == partID2***)

The ***point*** arguments are optional for the functions **kcs_cable.cway_route_start_point()**, and **kcs_cable.cway_route_end_point()**. When omitted, the route is only initialised/finalised, without adding a point to the route sequence. Then the function **kcs_cable.cway_route_point()** is responsible for adding consecutive route points. Our pattern can be then rewritten, as follows:

```
   point = KcsPoint3D.Point3D()
⇒  kcs_cable.cway_route_start_point() #initialise the route sequence
   for n in range(nPoints): # for ALL route points!
       … #update 'point'
⇒      kcs_cable.cway_route_point(point)
⇒  partID1, partID2, direction = kcs_cable.cway_route_end_point()
```

Please note, that any use of the `kcs_cable.cway_route_start_point()` function clears the list of the route points, that could possibly contain some points. Other functions simply add new points to this list. After the cableway route is stored in the current cableway, it is possible to delete some parts of it using the function

```
kcs_cable.cway_route_delete(startPartID, endPartID)
```

The function removes the part of the route in the current cableway defined as the fragment limited by the parts ***startPartID*** and ***endPartID***.

6.2.4 Managing cableway material

The function `kcs_cable.cway_material_set()` is responsible for dressing the cableway with the material. In the simplest form, it can be used as follows:

```
id0, id1 = kcs_cable.cway_material_set(startPartID, endPartID, \
           straightMaterial)
```

Then the cableway branch fragment limited with the parts having the IDs: ***startPartID***, and ***endPartID*** is dressed with the ***straightMaterial*** component. If you need to specify separate components for the straight parts and the bent parts (the corners), use the syntax:

```
id0, id1 = kcs_cable.cway_material_set(startPartID, endPartID, \
           straightMaterial, bendMaterial)
```

where ***bendMaterial*** specifies the component to be used for the corners, and the rest of the branch fragment will be dressed with ***straightMaterial*** component. In the above cases, no space is reserved for the frame between the parts. If you want to reserve such a space, you must supply additional arguments:

```
id0, id1 = kcs_cable.cway_material_set(startPartID, endPartID, \
           straightMaterial, bendMaterial, \
           startDistance, intermediateDistance, endDistance)
```

The ***startDistance*** argument specifies the reserved length of the frame part at the FIRST end, ***intermediateDistance*** argument determines the space reserved for a frame after each straight part, and ***endDistance*** is the reserved length of the frame part at the LAST end of the cableway branch fragment. These values are optional, and default to ZERO.

The material will be removed from the cableway branch fragment, if it has been dressed previously, and this function is called with the component name given as an empty string. This function returns a tuple, consisting of the first and last part IDs, that have been dressed. Normally, they are equal to ***startPartID***, and ***endPartID***, however, new parts can be created, if the length exceeds the material length, as given in the component definition. Then the returned values will differ from the IDs given as the function's arguments.

The straight part material can be rotated, by calling

```
kcs_cable.cway_material_rotate(partID, angle)
```

where the ***angle*** is given in degrees. Alternatively, you may specify the new ***rotation*** vector for the part, and call

```
   rotation = KcsVector3D.Vector3D(newX, newY, newZ)
⇒  kcs_cable.cway_material_rotate(partID, rotation)
```

Often you need to rotate the material of all straight parts in a branch. Then, the following code may be used:

```
   kcs_cable.cway_material_rotate_branch(partID, angle)
```

or

```
   rotation = KcsVector3D.Vector3D(newX, newY, newZ)
⇒  kcs_cable.cway_material_rotate_branch(partID, rotation)
```

which guarantees, that all parts of a branch are in the same plane. The ***partID*** argument is the ID of one of the parts belonging to the given branch, and identifies to branch, whose material is to be rotated.

6.2.5 Handling connections between cableways

The connections between cableways can be established by preparing a list of cableways allowed to connect to the current cableway, and then requesting a connection to be made. The general pattern of this process is given below:

```
⇒   kcs_cable.cway_cwenv_clear() #clear the list of cableways
⇒   kcs_cable.cway_cwenv_incl("CWXX2") #add a cableway to the list
    kcs_cable.cway_cwenv_incl("CWXX3") #add a cableway to the list
    … #continue adding other cableways to the list
⇒   kcs_cable.cway_cwenv_connect() #establish a connection
```

6.2.6 Handling cables on the cableway

It is possible to get the number of cables routed on the current cableway by calling

```
    nCables = kcs_cable.cway_check_cables()
```

This information can be helpful, for example, to verify it no cable is routed on the current cableway, before trying to delete it from the databank. It is possible to remove individual cables from the current cableway:

```
    kcs_cable.cway_remove_cable(systemName, cableName)
```

and if you want to remove all cables routed on the current cableway, and belonging to the given system, omit the cable name

```
    kcs_cable.cway_remove_cable(systemName)
```

Finally, if you want to remove ALL routed cables from the current cableway, omit both arguments

```
    kcs_cable.cway_remove_cable()
```

ⓘ *Currently, there is no function doing the reverse operation, i.e. adding the cable to the cableway.*

6.2.7 Handling the whole cableway

It is possible to duplicate all parts from another cableway to the current one, which should have NO PARTS at all, when this function is called.

```
    kcs_cable.cway_duplicate(anotherCablewayName)
```

After making a copy, you probably would transform the current cableway, so that it moves away from the original. The pattern for transforming the current cableway is given below:

```
    transf = KcsTransformation3D.Transformation3D()
    … #set up the transformation (see the equipment example on page 15)
⇒   kcs_cable.cway_transform(transf)
```

6.3 Default value handling

The Cable Modelling system is controlled by a number of defaults stored in the files identified by the Tribon environment variables SBC_DEF and SBC_DEF5.

```
⇒   stmt = kcs_cable.default_value_get("PUT_GLAND") #get the statement
    ind = stmt.find('=') #separate the string 'KEYWORD  = VALUE'
    value = stmt[ind+1:].strip()
    print "The keyword PUT_GLAND has the value", value
⇒   kcs_cable.default_value_set("PUT_GLAND=0") #set the new value
```

The function `kcs_cable.default_value_get()` returns the whole statement in the form **KEYWORD = VALUE**. Any updates made with the function `kcs_cable.default_value_set()` affect the current process only.

ⓘ *The statements should not be longer than 80 characters.*

6.4 Production Functions

Position names can be assigned to the parts of the current cableway by calling

```
kcs_cable.cway_part_posno_set("S1", partID) #assign position name 'S1'
```

The ***partID*** can be omitted. Then, the position names will be assigned automatically to ALL parts of the current cableway

```
kcs_cable.cway_part_posno_set("S") #assign position names 'S1', 'S2', …
```

Then, the first argument is treated as a PREFIX, and the position names will be created by concatenating this prefix with consecutive integer numbers, starting from 1.

Sometimes, not all kinds of cables can be routed on the given cableway. This restriction can be defined by indicating interference classes of cables, that may or may not be routed on the current cableway:

```
kcs_cable.cway_interference_exclude(interferenceClass1)
kcs_cable.cway_interference_permit(interferenceClass2)
```

You can also allow ALL interference classes of cables to be routed by calling

```
kcs_cable.cway_interference_permit()
```

It is an error to exclude the given interference class, when there are already cables with this interference class routed on the current cableway. You should not exclude an interference class, that has been already excluded.

It is possible to assign the planning unit to the current cableway by calling

```
kcs_cable.cway_planning_unit_set(plu)
```

The function sets or replaces the planning unit assigned for the current cableway. If ***plu*** is an empty string, the planning unit assignment will be removed.

Exercise 6: Making a cableway ready

Create an application making the user-selected cableway ready, and handling the problems such as:

- missing or invalid position names (assigns incremental position names with the user-defined prefix)
- missing or invalid planning unit (sets the user-defined planning unit)
- missing assembly reference (sets an assembly reference)

6.5 Cable Penetration Functions

Real and imaginary penetrations can be created in the cableway. Real penetrations need the height and width of the penetration to be provided. They can be specified either explicitly, or by providing the component name. Then the contour and dimensions stored in the component will be used. This means, that a non-rectangular penetration can be created ONLY by providing a component name. Below please find two possible syntaxes for creating a real penetration

```
    locationPoint = KcsPoint3D.Point3D(locX, locY, locZ)
⇒   kcs_cable.cpen_real_new(penetrationName, cablewayName, locationPoint, \
                            component)
```

or

```
    locationPoint = KcsPoint3D.Point3D(locX, locY, locZ)
⇒   kcs_cable.cpen_real_new(penetrationName, cablewayName, locationPoint, \
                            height, width)
```

ⓘ *Insert blocks are handled automatically, if the Cable Modelling default PUT_GLAND is set to 1*

Once placed, the real penetration can be transformed in the similar way, as all other outfitting objects

```
    transf = KcsTransformation3D.Transformation3D()
    … #set up the transformation (see the equipment example on page 15)
⇒   kcs_cable.cpen_real_transform(penetrationName, transf)
```

Creating an imaginary penetration is simpler – you only need to specify the location point.

```
     locationPoint = KcsPoint3D.Point3D(locX, locY, locZ)
⇒    kcs_cable.cpen_imag_new(penetrationName, cablewayName, locationPoint)
```

Finally, any kind of penetration can be deleted from the databank by calling

```
     kcs_cable.cpen_delete(penetrationName)
```