



Vitesse Hull



Revision Log

Date	Page(s)	Revision	Description of Revision	Release
17/05/2004	All	T.Lisowski	General Update for M3	M3
01/07/2004	14	T.Lisowski	Added section 2.6 concerning hull blocks	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

Copyright © 2004 Tribon Solutions

All rights reserved. No part of this publication may be reproduced or used in any form or by any means (graphic, electronic, mechanical, photocopying, recording, taping, or otherwise) without written permission of the publisher.

Python 2.3 is Copyright © 2001-2003 Python Software Foundation

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

1	Introduction.....	5
1.1	Objectives	5
1.2	Prerequisites for training course	5
1.3	Training methods	5
1.4	Overview.....	5
1.5	Duration.....	6
1.6	Using this document	6
2	Planar Hull Modelling.....	7
2.1	Introduction.....	7
2.2	Generating plane panels via the scheme file	7
	Exercise 1: Creating a bulkhead panel	8
	Exercise 2: Creating a side web panel	8
	Exercise 3: Adding a hole to the panel	10
	Exercise 4 (advanced): Adding holes to the panel	10
2.3	Using the PanelSchema class	11
2.4	Panel object functions	12
	Exercise 5: Renaming a panel	14
2.5	Direct modification of panel components	14
2.6	Managing hull blocks	15
2.7	View handling	15
2.8	Curve functions	15
2.9	Document references	16
3	Curved Hull Modelling.....	17
3.1	Introduction to XML.....	17
3.2	Handling XML files in Python	19
	Exercise 6: Modifying an existing XML file.....	21
3.3	Modelling curved hull objects using XML	21
	Exercise 7: Modifying the shell profile.....	21
3.4	Direct modelling of curved hull objects	22
3.5	View handling	24
4	Weld Planning.....	27
4.1	Python classes for storing the welding information.....	27
4.2	Weld Planning functions.....	28
	Exercise 8: Calculating total weld length of an assembly	28

1 Introduction

Tribon Vitesse contains a set of modules supporting the creation of hull structures. The flexibility of Vitesse macros can help to automate various modelling tasks not only in Planar Hull Modelling, but also in Curved Hull Modelling. The ability to handle curved hull objects and views is one of the new additions in Tribon M3, as well as the `kcs_weld` module, handling the modelling of welds of steel structures.

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:

- Planar Hull Modelling;
- Curved Hull Modelling;
- Weld Planning.

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
- Working knowledge of Tribon Data Extraction system.
- Working knowledge of the Tribon modelling subsystems:
 - Hull (Planar and Curved);
 - Weld Planning
- Basic programming experience;
- Basic knowledge of the Assembly Planning system and its Vitesse API is an additional benefit.

At least one participant should have completed the Interactive Curved Hull Modelling (XML Add-on) training. 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

The planar hull modelling abilities of Tribon Vitesse, well known since its first release, have been enhanced with new functions, performing some activities on multiple panels, and modifying directly the panel components without using the scheme file.

For the first time Tribon Vitesse is able to model curved hull objects, using their XML representation. By interpreting the properly formatted XML files, Vitesse programs are able to create or modify the curved hull objects.

Another new functionality in Tribon M3 Vitesse is the interface to the welding information for steel structures, provided by the `kcs_weld` module.

1.5 Duration

2 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 Hull 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 Planar Hull Modelling

2.1 Introduction

Tribon Vitesse features access to the modelling functionality in Hull Planar Modelling through the `kcs_hullpan` Vitesse module. This means that all types of planar panels can be created by the Vitesse program launched from within Tribon M3 Hull Planar (or Curved) Modelling or Tribon M3 Basic Design application.

❶ *The `kcs_hullpan` module provides only an interface between the Python language and the planar panel creation abilities of the Tribon application. Hence, this module is only available from applications such as Hull Planar Modelling, or Basic Design, which have such abilities. Such a Vitesse program run e.g. from the Drafting application would raise an `ImportError` exception.*

📖 *For more information on the modelling language used Hull Planar Modelling, see the section [Tribon M3 Hull → Planar Modelling → Design Language of Tribon Hull Modelling](#) in the documentation.*

In short, Tribon Vitesse is able to create and modify planar panels in the same way as Tribon itself – by executing statements written in the design language of the Hull modelling system. What makes it more powerful than the Hull modelling application? The ability to generate input schemes automatically, basing on a set of input parameters and well-defined yard rules incorporated in the program.

The use of hull modelling abilities of Tribon Vitesse requires insertion of the statement

```
import kcs_hullpan
```

As usual, the module provides an 'error' variable (`kcs_hullpan.error`), which is set to a string describing the reason of an error, if an exception is raised during the execution of the `kcs_hullpan` module functions.

2.2 Generating plane panels via the scheme file

Typically, the creation or modification of plane panels is done through creation or modification of its input scheme. When the scheme's statements are interpreted, the corresponding changes are made to the hull model.

The pattern for creating a new panel is shown below:

```
⇒ kcs_hullpan.pan_init(scheme, ident_statement) #initialise the scheme
   try:
⇒     kcs_hullpan.stmt_exec(0, stPAN) #execute PAN statement
       kcs_hullpan.stmt_exec(0, stBOU) #execute BOU statement
       ... #execute remaining statements
⇒     kcs_hullpan.pan_store() #store the panel
   finally:
⇒     kcs_hullpan.pan_skip() #skip (deactivate) the scheme
```

First, the scheme (and panel) are initialised (`kcs_hullpan.pan_init()`), and the identification statement is added to the scheme file. Then the remaining statements of the scheme are added and executed by a series of calls to the function `kcs_hullpan.stmt_exec()`. It is up to the programmer to create the correct statement strings – Tribon Vitesse does not check them, but passes them immediately to the hull modelling engine for execution. If no error was encountered so far, the panel is stored in the SB_OGDB databank (`kcs_hullpan.pan_store()`). As the final step, the scheme should be deactivated using the function `kcs_hullpan.pan_skip()`, because otherwise, it would not be possible to start creating another panel. It is guaranteed by the `try: ... finally: ...` statement.

Since many errors can be encountered during the component generation, we recommend to enclose this code in the `try: ... except: ...` statement and handle possible errors. When an error is encountered, the scheme editor is not launched anymore. You can, however, start the scheme editor explicitly, by calling

```
kcs_hullpan.editor(panelName)
```

The next example shows the new possibilities available in Tribon M3 of reporting errors encountered when using the `kcs_hullpan` module functions.

```
try:
    ... #hull modelling activity
except:
    print "Error encountered: %s!" % kcs_hullpan.error
    #For some error types we can report additional error information ...
    if kcs_hullpan.error in ("kcs_InterpretationError", \
        "kcs_GenerationError", "kcs_SystemError"):
⇒      nErr = kcs_hullpan.nerr() #Number of errors
        for ind in range(nErr):
⇒          errCode = kcs_hullpan.err_code(ind) #Error code
⇒          errMsg = kcs_hullpan.err_mess(ind) #Error message
            print "%d: %s" % (errCode, errMsg)
```

The `kcs_hullpan.nerr()` function returns the number of error messages collected. The functions `kcs_hullpan.err_code()` and `kcs_hullpan.err_mess()` access these error messages and return the error code and message string for the error message at the index given as an argument.

❗ Please note, that these functions are useful only, if `kcs_hullpan.error` variable is set to one of the following values: `"kcs_InterpretationError"`, `"kcs_GenerationError"`, or `"kcs_SystemError"`.

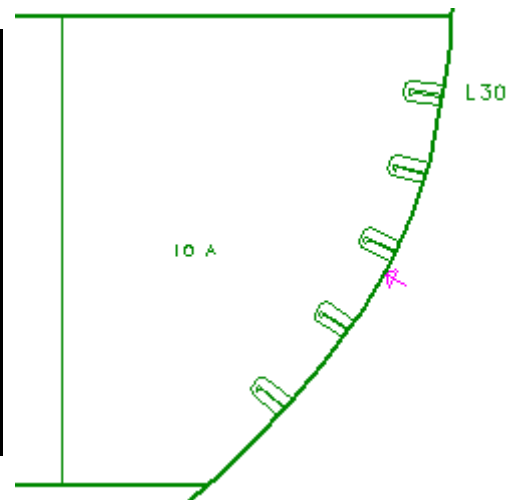
Exercise 1: Creating a bulkhead panel

Write the program adding a bulkhead panel located at the given frame position, and bounded by two decks, the shell and the given Y position, as shown on the picture to the right. All data mentioned above should be provided by the user.

The panel should have a plate with a thickness of 10 mm, and cutouts of type 5 for the shell profiles (longitudinals) indicated by the user.

Hint: The frame range of 30 – 40 is suitable for this exercise. Use the function `VTHull.getModel()` for picking model objects from the drawing.

Advanced: If you know, how to use the Perl regular expressions, you can use the standard Python `re` module to validate the user input of the Y coordinate, and accept either a real number, or an expression involving horizontal longitudinal positions (LP) with a possible offset.

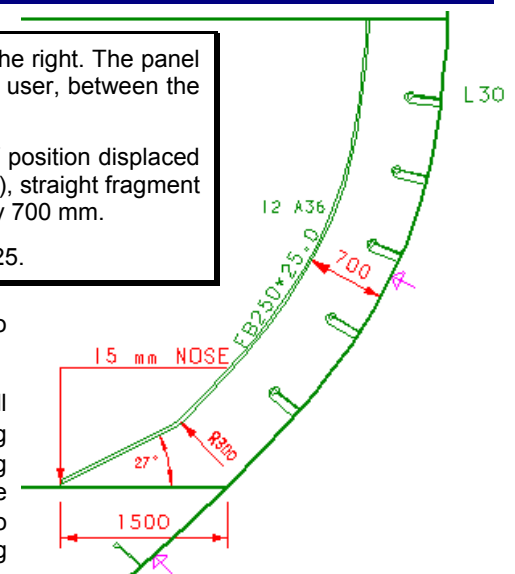


Exercise 2: Creating a side web panel

Write the program creating a side web panel, as shown on the picture to the right. The panel should be 12 mm thick, and be located at the frame position given by the user, between the lower and upper deck.

The left limit should be a curve consisting of a 15 mm high nose (at the Y position displaced by 1500 mm from the intersection between the lower deck and the surface), straight fragment inclined at 27°, and the curve fragment parallel to the surface, displaced by 700 mm.

A flange should be added along this limit with profile parameters 10, 250, 25.



So far, we have seen how to generate new panels. Now, let's see how to modify existing ones.

First of all, instead of calling `kcs_hullpan.pan_init()` we will call `kcs_hullpan.pan_modify()`, since we are updating an existing panel. Then we may want to recreate the panel by calling `kcs_hullpan.pan_recreate()` in order to make sure, that the drawing shows exactly the panel as it is stored in OGDB. Now we come to the core of the panel's modification process – to the problem of modifying panel's components.

Basically, there are three possible methods of modification:

1. **adding of new components** – this may be done using the call to `kcs_hullpan.stmt_exec(0, st)`, with `st` being the statement defining the new component, exactly like when adding statements to the new panel;
2. **removing a component** – here we can either replace the statement with a comment and re-execute it. Alternatively, we may initialise the panel again, re-executing ALL panel's statements except the one, that we want to remove. The third method consist in using the function `kcs_hullpan.pan_group_delete()` (see page 14). See also the note below ... ;
3. **modifying a component** – here we have to obtain the current text of the statement and re-execute it after making appropriate modification.

❗ Please note, that some statements can define more than one component. Example:

`STI, SID=1, V=1000(1000)9000, PRO=30,500,200,20, CUT=2100, CON=15/ CUT=2100, CON=15;`

This STI statement defines 9 stiffeners. If we want to remove only the first one at V=1000, then we should rather modify the statement, so that this position is excluded,

`STI, SID=1, V=2000(1000)9000, PRO=30,500,200,20, CUT=2100, CON=15/ CUT=2100, CON=15;`

than remove the statement completely, because otherwise ALL 9 stiffeners would be removed.

Activities no. 2 and 3 require, that we identify somehow the statements responsible for creation of the given component, that we want to remove or modify. Each statement has an associated number called the **group number**, which is the identification number of the group of components, that are defined by the given statement, and also of the statement itself. It must be provided as the first argument to the function `kcs_hullpan.stmt_exec()`, when this statement is re-executed. Tribon Vitesse provides functions that return the group number for the given panel component or for the given statement.

```
... #PID variable is the part ID of some component of the active panel
⇒ group_no = kcs_hullpan.group_get(panelName, PID) #get group number
⇒ st = kcs_hullpan.stmt_get(panelName, group_no) #get statement text
... #modify the statement text
⇒ kcs_hullpan.stmt_exec(group_no, st) #re-execute modified statement
```

The above example shows the pattern of modifying a panel component, whose part ID is known.

❗ The part ID of the panel component may be obtained either by indication by the user (it is reported as the **PartId** attribute of the **Model** class instance – see `kcs_draft.model_identify()` function), or by Data Extraction calls.

First we get the component's group number, then the original statement text, that created this component. After making appropriate modification to the statement, we re-execute it, preserving the component's group number.

❗ The first argument of the function `kcs_hullpan.stmt_exec()` is the group number of the component being modified. If set to 0 (zero), a new component is added to the panel.

The next example shows another method of modifying a panel. This time we analyse the scheme statement by statement, updating some of them, if necessary.

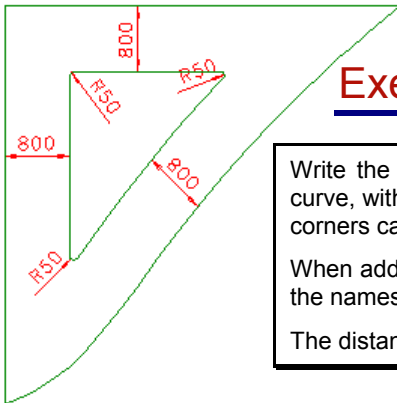
```
⇒ kcs_hullpan.pan_modify(panelName, 2) #Update existing panel
   try:
⇒   kcs_hullpan.pan_recreate(panelName) #Recreate the panel
       #Get the FIRST group number - second argument: -2
⇒   group_no = kcs_hullpan.group_next(panelName, -2, 0)
       while 1: #start a loop over the statements
           st = kcs_hullpan.stmt_get(panelName, group_no) #get statement text
           ... #analyse the statement, possibly update it ...
           if should_we_update_the_statement: #program's decision
               kcs_hullpan.stmt_exec(group_no, st) #re-execute statement
           #Get the NEXT group number - second argument: 1
⇒   try: group_no = kcs_hullpan.group_next(panelName, 1, group_no)
       except: break #break the loop
       kcs_hullpan.pan_store()
   finally:
       kcs_hullpan.pan_skip()
```

First, we activate the panel for modification, using the function `kcs_hullpan.pan_modify()`, providing the panel name and the update mode (2 – update existing panel). Then the panel is recreated, and we start getting the scheme statements, analysing them, and updating if necessary. For the retrieval of scheme statements we use the following function calls:

- `group_no = kcs_hullpan.group_next(panelName, -2, 0)` – to obtain the group number of the FIRST statement;
- `group_no = kcs_hullpan.group_next(panelName, 1, group_no)` – to obtain the group number of the NEXT statement;
- `kcs_hullpan.group_next(panelName, 0, group_no)` – to validate the given group number.

If this function fails (e.g. there is no FIRST/NEXT statement, or the given group number is not valid), an exception is raised. That's why in the above example we process the statements in an apparently infinite loop, using the `break` statement, when the function `kcs_hullpan.group_next()` raises an exception.

After executing all necessary statements, the panel is usually stored on the OGDB databank. It is interesting, that even before storing the panel some information about it is available through the Data Extraction interface (e.g. the boundary corners' co-ordinates). This gives the opportunity to create a 'dummy' panel, retrieve the necessary information (e.g. for future calculations), and terminate the scheme WITHOUT storing the panel on the databank. Of course, if the function `kcs_hullpan.pan_store()` is not called, changes made to the panel are lost!



Exercise 3: Adding a hole to the panel

Write the program adding a hole to the panel. The hole should be defined by the closed curve, with segments parallel to the panel's limits, keeping the given distance to the limit. The corners can be rounded with the radius given by the user.

When adding the curve to the panel, please choose its name so, that there is no conflict with the names of other possibly existing curves in the panel.

The distance from the hole to the panel's boundary should not be less than 300 mm.

Exercise 4 (advanced): Adding holes to the panel

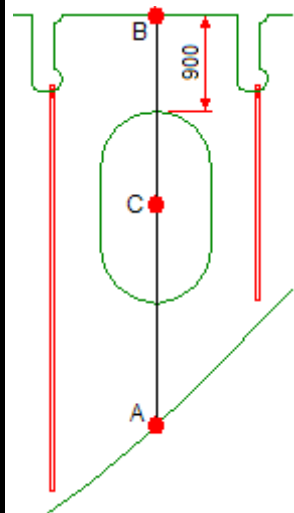
Write the program adding standard holes HO<height>*<width> to the frame panel indicated by the user. The user indicates the approximate position of the hole's centre. Then the program locates the hole's centre (point C) according to the following rules:

- the Y co-ordinate is at the middle position between two horizontal longitudinal positions (LPs), nearest to the point indicated by the user;
- the Z co-ordinate is at the middle position between the points A and B, where the vertical line going through the point C intersects the panel limits.

The hole's width should be 350 mm. The hole's height should be set so, that there is a space of 900 mm left between the panel's limit and the hole's boundary. The minimum height of the hole is 350 mm. If it is not possible to fulfil these requirements, the hole should NOT be created.

Hints: You may have to refer to the panel limit numbers. In order to obtain them, you can let the user indicate the panel's limit on a symbolic view.

If successful, `Model.PartType` is "boundary", `Model.SubPartType` is "limit", and `Model.SubPartId` is the limit number, which can be used for defining topological points A and B.



i If you encounter difficulties in indicating panel limits, you may temporarily hide everything except from the panel being modified (e.g. by changing the layer of the panel, and showing that layer only). This way you can avoid indicating something, that does not belong to your panel.

When the statement are executed, current settings of the scheme run mode options are taken into account. Tribon Vitesse handles these options with `RunModeOptions` class instance.

```

=> options = kcs_hullpan.pan_scheme_runmode_get() #current options
confirmGeneration = options.GetConfirmGeneration()
traceOn = options.GetTraceOn()
options.SetConfirmGeneration(1) #turn on generation confirmation
options.SetTraceOn(0)           #turn off Trace On setting
=> kcs_hullpan.pan_scheme_runmode_set(options) #set new options

```

2.3 Using the PanelSchema class

The updating of the scheme statements became much easier thanks to the **PanelSchema** class. This class understands the syntax of the statements, can split the statements into tokens, retrieve the values of the options present in the statement, and modify the statement, setting new values of the keywords. See the following example ...

```

import KcsPanelSchema
... #activate panel 'ES123-AY012' for modification
=> sch = KcsPanelSchema.PanelSchema("ES123-AY012")
... #group_no is the group number of some PLATE statement
=> side = sch.GetValue(group_no, "MSI") #get the value for MSI keyword
if side == "FOR":
=>     sch.SetValue(group_no, "MSI", "AFT") #set new value for MSI keyword

```

The **GetValue()** method returns the current value of the given keyword (e.g. "MSI") as specified in the scheme statement provided as the first argument. Here, the statement is identified by its group number **group_no**. If the keyword is not present in the statement, the function returns **None**.

The **SetValue()** method updates the value of the given keyword in the given statement, **executes the modified statement**, and returns it.

The class supports yet another method of updating scheme statements, where in **GetValue()** and **SetValue()** methods we provide statement texts explicitly instead of the group numbers, that identify them. This is a more efficient way to make several updates to the single statement.

```

import KcsPanelSchema
... #activate panel 'ES123-AY012' for modification
=> sch = KcsPanelSchema.PanelSchema("ES123-AY012")
... #group_no is the group number of some PLATE statement
st = kcs_hullpan.stmt_get('ES123-AY012', group_no)
=> side = sch.GetValue(st, "MSI") #get the value for MSI keyword
if side == "FOR":
=>     st = sch.SetValue(st, "MSI", "AFT") #set new value for MSI keyword
     st = sch.SetValue(st, "QUA", "A36") #set new value for QUA keyword
     kcs_hullpan.stmt_exec(group_no, st) #execute modified statement

```

❗ If the method **SetValue()** is called with the group number as the first argument, the modified statement is automatically executed.

*This **DOES NOT** happen, if the first argument is the statement (a string) to be modified. In both cases the method returns the modified statement string.*

The **PanelSchema** class provides one more interesting method, retrieving all panel's statements:

```

import KcsPanelSchema
... #activate panel 'ES123-AY012' for modification
sch = KcsPanelSchema.PanelSchema('ES123-AY012')
=> res = sch.GetStatements() #get ALL scheme statements
for group_no, st in res: #print out group numbers and statements
    print "%d: %s" % (group_no, st)

```

The result returned by the **GetStatements()** method is a list of 2-element tuples, consisting of the group number and the statement text. This can be helpful to perform some global analysis of the scheme, and to make the appropriate updates.

2.4 Panel object functions

The `kcs_hullpan` module provides also a set of functions, that work on plane panels, but without using the scheme file. The panel (one or more) must be first activated, before the other functions can be used.

```
panelNames = ["ES123-AY012", "ES123-AY013"] #panels to work on
⇒ kcs_hullpan.pan_activate(panelNames)
try:
    ... #work on the activated panels
⇒ kcs_hullpan.pan_store(panelNames) #store panels
finally:
⇒ kcs_hullpan.pan_skip(panelNames) #skip (deactivate) panels
```

You can see, that this pattern is very similar to the one, presented on page 7, concerning the creation of a panel via the scheme file. The difference is, that now we provide the list of names of the panels (or a single panel name), on which we are working, as an argument to the highlighted functions above.

- ❗ If the list of panel names, passed to the function `kcs_hullpan.pan_skip()` omits some names, that were passed initially to the function `kcs_hullpan.pan_activate()`, the panels with the omitted names are STILL active!

This allows us to decide, which activated (and modified) panels should be stored, and which ones – deactivated.

- ❗ The functions `kcs_hullpan.pan_store()` and `kcs_hullpan.pan_skip()` can be used WITHOUT the argument. Then ALL activated panels will be stored/skipped.

What can we do with the activated panels? We can delete them

```
kcs_hullpan.pan_delete(listOfPanelNames)
```

... or recreate them

```
kcs_hullpan.pan_recreate(listOfPanelNames)
```

- ❗ Please note, that in previous releases of Tribon Vitesse the function `kcs_hullpan.pan_recreate()` was supposed to be called WITHOUT any argument, and was working on the current panel, activated by the function `kcs_hullpan.pan_modify()`. Now the argument is mandatory – it may be either a single panel name (a string) or a list of activated panel names.

It is quite common, that there are topological dependencies between the panels. This function takes into account the topological order of panels when recreating them.

It is also possible to move, copy or split the panels. These operations, however, require additional settings, that are defined using instances of appropriate classes. First example shows the operation of moving the panel to the new position, defined by the principal plane

```
options = KcsMovePanOptions.MovePanOptions()
⇒ options.SetPrincipalPlane(1, False, 'FR20')
```

First argument defines the plane orientation (1 – X, 2 – Y, 3 – Z), and the third one – the coordinate of the plane as a string. The second argument determines, whether the coordinate (third argument) should be considered as relative to the current position (true), or absolute (false) – the above example would mean a move by 20 frames forward, if the second argument was set to **True**.

- ❗ Make sure, that the target position, to which you would like to move your panel, is defined in the system, when it is specified as a frame or longitudinal position expression! If it is not, the panel will not be moved.

Examples of incorrect settings:

- move to the position FR300, when such a high frame number has not been defined,
- move from vertical LP25 by LP-5 (relative move), when the vertical position LP20 has not been defined

If you want to define the new location as an arbitrary plane (Three Points method), replace the second line with

```
options.SetThreePoints(origin, uAxis, vAxis)
```

where all three arguments are **Point3D** class instances, that determine the origin, and location of the U and V axes of the new local coordinate system of the panel. Finally, we have the possibility to define the new location by providing the name of the object, from which the new local coordinate system should be derived. Then the last statement should have the form

```
options.SetPlaneObject('DECK_CURVE')
```

After setting up the **MovePanOptions** class instance, we call the function

```
kcs_hullpan.pan_move(listOfPanelNames, moveOptions)
```

to move the panels, whose names appear in the list given as the first argument, according to the given options. The panels being moved should be activated first. It is correct to provide a panel name (a string) instead of a list containing this single panel name.

The panel copy operation requires also similar settings, and should be regarded as the two-stage process, where first a copy is made at the original position, then the copy is moved in the same way, as explained above. The settings of a copy operation are defined with the **CopyPanOptions** class instance, which for defining the move stage uses the same methods, as shown above.

```
options = KcsCopyPanOptions.CopyPanOptions()
... #use SetPrincipalPlane(), SetThreePoints(), or SetPlaneObject() method
#dictionary of panel's oldName:newName pairs
options.SetNameMapping({'ES123-AY012':'ESNEW-AY012', \
                        'ES123-AY013':'ESNEW-AY013'})
options.SetBlockName('ESNEW') #New block name
⇒ kcs_hullpan.pan_copy(['ES123-AY013', 'ES123-AY013'], options)
```

The panel splitting function facilitates splitting the panel into TWO distinct panels, and also requires to provide the settings in the **SplitPanOptions** class instance.

```
options = KcsSplitPanOptions.SplitPanOptions()
origin = KcsPoint3D.Point3D(0, 0, 5000) #cutting plane's origin
normal = KcsVector3D.Vector3D(0, 0, 1) #cutting plane's normal vector
plane = KcsPlane3D.Plane3D(origin, normal) #cutting plane
⇒ options.SetCuttingPlane(plane)
#name mapping definition
⇒ options.AddPanelMapping('ESNEW-AY012A', 'ESNEW', 'SBPS') #first panel
options.AddPanelMapping('ESNEW-AY012B', 'ESNEW', 'P') #second panel
⇒ kcs_hullpan.pan_split('ES123-AY013', options)
```

You have to define the splitting plane, appropriate name dictionaries, linking the old and new panel and block names, and specify symmetry codes for resulting panels. Remember to activate first the panels to be moved, copied or split.

i The symmetry code of **SBP** is not recognised. Use **SBPS** instead.

You can split one panel at a time. Providing more than one panel name (a list) as an argument is not supported.

After copying or splitting, the new panels become active, and must be stored and skipped properly. At any time you can get the list of names of currently activated panels by calling the function

```
activePanelsList = kcs_hullpan.pan_list_active()
```

This can be useful to verify, if all panels have been successfully copied or split – then all new panel names should be present in the resulting list. If some of them are missing – they were NOT created due to some errors.

The last function in this section determines the topology relationships between the given panel, and surrounding panels.

```
model = KcsModel.Model("plane panel", "ES123-AY012")
⇒ res = kcs_hullpan.pan_topology(model, "Dependent primary")
```

The **kcs_hullpan.pan_topology()** function returns a list of Model class instances, defining model objects, that:

- DIRECTLY depend on the given **model** (second argument: "**Dependent primary**");
- DIRECTLY or INDIRECTLY depend on the given **model** (second argument: "**Dependent all**");
- the given **model** depends on (second argument: "**Defining**").

This function can be used, for example, to recreate dependent panels, after modifying the given panel. Please note, however, that this function can return also model objects, that are NOT plane panels (e.g. hull curves, surfaces, etc.), especially, when using the second argument of "Defining".

- ❗ *The function `kcs_hullpan.pan_topology()` does not guarantee, that the returned list of model objects that depend on the given panel is ordered in the right topological order for recreation of the dependent panels. This, however, should not be a problem, since the function `kcs_hullpan.pan_recreate()` will take care of the topological sorting of the panels.*

Exercise 5: Renaming a panel

Write the program that renames the panel indicated by the user, by creating a copy of the panel with the new name, and removing the original. The user should provide the new name of the panel, and select (confirm) the block name for the panel. The program must verify, if there is no name conflict with existing panels.

The program should take care of updating the views on the current drawing.

Advanced: Before removing the original panel, find all panels, that depend on the original panels. Then, modify these panels too, and exchange all references to the old panel name with references to the new panel name.

2.5 Direct modification of panel components

In this section we will discuss functions, that manipulate panel components, without using the panel's scheme file. It is assumed, that the panel, on which these functions are working, has been activated. It is now possible to remove the seam from the active panel.

```
kcs_hullpan.pan_remove_seam(panelName, seamPartID)
```

- ❗ *Component IDs of the seams are the integer numbers starting at 5001.*

After removing the seam, the neighbouring plates are combined. We can also remove any existing group of components from the active panel using the function `kcs_hullpan.pan_group_delete()`

```
... #activate the panel, indicate one of its components (part ID)
group_no = kcs_hullpan.group_get(panelName, partID)
⇒ kcs_hullpan.pan_group_delete(panelName, group_no)
kcs_hullpan.pan_store()
model = KcsModel.Model("plane panel", panelName)
kcs_draft.model_draw(model) #redraw (exchange) the panel
```

where **group_no** is the group number of the group of components (the scheme statement) to be removed. Please note, that after removing a group of components you need to redraw the panel to see the effect.

It is possible to split the component group by regrouping some of the components into the new group:

```
... #activate the panel, indicate one of its components (part ID)
group_no = kcs_hullpan.group_get(panelName, partID)
... #collect components to regroup as a list of Model class instances
⇒ newGroupNo = kcs_hullpan.pan_group_divide(panelName, group_no, \
    componentList)
kcs_hullpan.pan_store()
```

The function `kcs_hullpan.pan_group_divide()` takes out from the group **group_no** the components present in **componentList**, and puts them into a new group (new statement), whose group number is returned as a result.

Groups of stiffeners can be split using either the splitting component (e.g. a hole, another stiffener, a flange, a bracket or a seam) or the principal plane.

```
... #activate the panel, indicate one of the stiffeners (stiffener)
... #indicate the splitting component (splitComp)
⇒ kcs_hullpan.pan_sti_split_by_model(stiffener.PartId, splitComp.PartId)
```

where **stiffener** and **splitComp** are Model class instances – **stiffener** indicates one of the stiffeners belonging to the stiffeners group being split. If stiffeners are split by the principal plane, then the code looks as shown below:

```
... #activate the panel, indicate one of the stiffeners (stiffener)
⇒ kcs_hullpan.pan_sti_split_by_plane(stiffener.PartId, "Y=LP6")
```

As the plane definition, you may supply the string describing the cutting principal plane (e.g. "X=FR20+200", "Y=LP5-100", "Z=LP25").

2.6 Managing hull blocks

Hull panels are assigned to the hull blocks, defined as the rectangular prisms within the ship's space. Tribon Vitesse provides the **kcs_modelstruct** module, which allows to define or remove such hull blocks. Use the following pattern to define the new hull block:

```
corner1 = KcsPoint3D.Point3D(x1, y1, z1)
corner2 = KcsPoint3D.Point3D(x2, y2, z2)
⇒ kcs_modelstruct.block_new(newBlockName, corner1, corner2)
```

where **corner1** is the lower-left corner, and **corner2** – upper-right corner of the extension box. **newBlockName** should not be empty, and should not contain more than 25 characters. For removing an existing hull block, just call the function

```
kcs_modelstruct.block_delete(blockName)
```

2.7 View handling

The functions in this section complement the view manipulation functions from the **kcs_draft** module. In order to create the symbolic view, we have the function **kcs_draft.view_symbolic_new()**. If we want to modify an existing symbolic view, we need to get the current settings, update some options, and create the symbolic view with new settings

```
⇒ options = kcs_hullpan.view_symbolic_modify(viewHandle)
options.SetDepth(1000.0, 1000.0) #New before/behind depth settings
options.SetShellCurves(options.CURVE_CUT)
kcs_draft.element_delete(viewHandle) #Remove old symbolic view
viewHandle = kcs_draft.view_symbolic_new(options) #create the view anew
```

❶ *The new symbolic view is placed at the drawing's origin (0, 0), and has the scale of 1:1. If you want to place the new view in the original position in the drawing, you have to move and scale the view appropriately.*

If the symbolic view needs to be recreated, then call the function

```
kcs_hullpan.view_symbolic_recreate(viewHandle)
```

When identifying panel components on the symbolic view, it is possible to create the detail view of this component

```
res = kcs_draft.model_identify(point, model) #identify the component
componentHandle = res[2] #handle to the component subpicture
if model.PartType == "stiffener":
⇒ viewHandle = kcs_hullpan.view_detail_new(3, componentHandle)
```

The value of 3 means, that the stiffener detail view should be created. The list below shows all possible values:

- 2 – flange
- 3 – stiffener
- 4 – bracket
- 5 – seam

2.8 Curve functions

For historical reasons, these functions are still in the **kcs_hull** module, not **kcs_hullpan**. Please remember to import an appropriate module. The first function allows to create the CUR statement from the 2D contour drawn in the drawing. This statement can be then executed using **kcs_hullpan.stmt_exec()**, as usual.

```
... #create and draw the 2D contour (Contour2D class instance)
curveName = 'C1' #chosen curve name
planeCode = KcsUtilPan.getPlane(panelName)[0]
```



```

⇒   stCUR = kcs_hull.pan_curve_create(viewHandle, panelName, curveName, \
        planeCode, contour)
    ... #activate the panel
    kcs_hullpan.stmt_exec(0, stCUR) #create the curve

```

The function **KcsUtilPan.getPlane()** returns a list of interesting data concerning the location of the given panel. The first element is the so-called panel's plane code. It can take the following values

- 1 – the panel is located in the X plane
- 2 – the panel is located in the Y plane
- 3 – the panel is located in the Z plane
- -1, -2, -3 – the panel is NOT located in one of the principal planes, given above, but its plane is closest to the X plane (-1), Y plane (-2), or Z plane (-3)

The contour, given as the last argument (Contour2D class instance) should be created and drawn on the view identified by **viewHandle**, displaying also the panel being modified.

 See the file **kcs_ex_hullcurve1.py** in the **Vitesse\Examples\Hull** folder for the complete example of the proper usage of this function.

If you prefer to create your curve on CGDB databank for later reference, use the following pattern

```

    ... #create and draw the 2D contour (Contour2D class instance)
    curveName = 'C1'
⇒   res = kcs_hull.pan_curve_store(viewHandle, curveName, contour)
    if res == kcs_util.success():
        ... #use the curve

```

As before, the contour should be created as the Contour2D class instance, and drawn on the view identified by **viewHandle**. The result is the status code. The value of ZERO means, that the curve has been successfully created and stored on CGDB.

2.9 Document references

Document references are represented in Vitesse by instances of the **DocumentReference** class, and may be attached to many model objects, including plane panels.

```

    docRef = KcsDocumentReference.DocumentReference()
    docRef.SetType('drawing')
    docRef.SetDocument('PANDWG01')
    docRef.SetPurpose(kcs_draft.kcsDWGTYPE_GEN) #Databank (General drawing)
⇒   kcs_hullpan.document_reference_add(docRef, panelName)
⇒   docRefList = kcs_hullpan.document_reference_get(panelName)
    for docRef in docRefList[:]:
        if docRef.GetType() == 'vitesse': #Remove 'vitesse' references
⇒       kcs_hullpan.document_reference_remove(docRef, panelName)

```

Tribon system is able to manage document references attached to various objects, not only to plane panels. They 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.

3 Curved Hull Modelling

Tribon M3 is the first release offering the ability to model curved hull objects in Vitesse. In order to use this new functionality, the program must import the **kcs_chm** module.

```
import kcs_chm
```

For defining curved hull objects a descriptive language, based on XML standard, has been developed.



*It is beyond the scope of this training guide to describe this language. Here you can find only some introductory material, describing this functionality from the Vitesse programmer's point of view. Details can be found in the Curved Hull Modelling User's Guide (see the documentation at **Tribon M3 Hull → Curved Modelling → User's Guide Batch → Input Language of Curved Hull Modelling**).*

3.1 Introduction to XML

The name XML stands for **eXtensible Mark-up Language**. It is a data format originating from SGML (**S**tandard **G**eneralized **M**ark-up **L**anguage), and is used extensively in connection with the web technology and HTML. It's main goal is to describe structured data.



The official description of the XML standard can be found at the URL <http://www.w3.org/XML/>

Example:

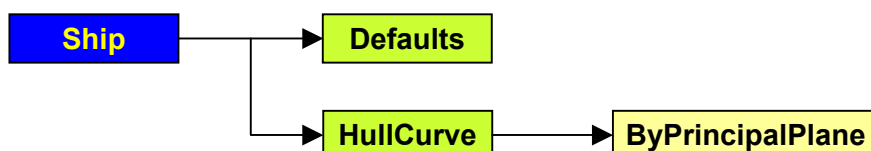
```
<?xml version="1.0" encoding="UTF-8"?>
<Ship>
  <!-- a comment -->
  <Defaults Surface="SPHULL" XMin="FR40" YMin="0" XMax="FR80"/>
  <HullCurve ObjId="SPX951">
    <ByPrincipalPlane X="FR30"/>
  </HullCurve>
</Ship>
```

The definition comes as a tree of XML **elements**, defining the structured data. The element definitions are given by XML **tags** enclosed in triangle brackets (e.g. <Ship>). In simplest case, the element definition has the following form:

```
<Tag_name>
... contents (other elements)
</Tag_name>
```

The first line is the so-called "opening tag", declaring the element name, and the last line is the "closing tag", terminating the element definition. You can recognise the closing tag by the slash character (/) coming right after the opening triangle bracket, followed by the same element name, that has been used in the opening tag. The opening and closing tags must be properly paired, otherwise the syntax error occurs. All lines coming between the opening and closing tags define the element contents – the subordinate elements, which must be defined using the similar pattern. The indentation of the source lines is syntactically not important, although it improves the readability of the XML document.

Following this rules, we can find that the above example defines the following tree of XML elements:



The lowest levels of the XML element tree (the leaves) do not have any subordinate elements (e.g. "ByPrincipalPlane" and "Defaults"). Then, their definition can be simplified and enclosed in a single tag, as shown below

```
<Tag_name ... />
```

where the ellipsis stands for the **attributes** defined for the given element, and the slash character '/' found just before the closing triangle bracket denotes the end of the element definition – the closing tag is then not used.

The **attributes** can also appear in the full form of the element definition (with opening and closing tags). They are always given as definitions of the form

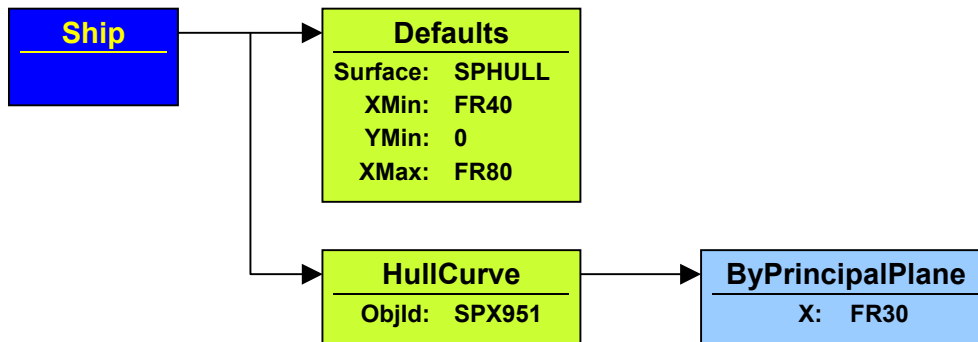
`attribute_name="attribute_value"`

placed in the opening tag and separated by spaces, tabs or newlines. The attribute value is always given as a string, even if it represents some numerical property of the element.

❗ *XML language is case-sensitive, when interpreting the element and attribute names! "Ship" and "SHIP" are two different element names.*

Attribute order is irrelevant.

Looking again at our example, we can see the following definition of elements and their attributes



The "Ship" element does not have any attributes – all its information is contained in subordinate elements. Since it is the top-level element, we will often refer to it as the **root element**. It is assumed, that each XML document defines a SINGLE root element. If we need to provide more than a single definition in an XML document, we have to put them as subordinate elements to some other – single root element.

The elements in the XML document have mutual relationships. We can say, for example, that the element "HullCurve" is the child (subordinate element) of the element "Ship", or in other words, the element "Ship" is the parent of the element "HullCurve". Each element (except from the root element) has exactly one parent, and each element can have zero or more children. Each element can have also zero or more attributes.

Our example shows also two more type of tags: processing instructions

`<?xml version="1.0" encoding="UTF-8"?>`

... and comments

`<!-- a comment -->`

Processing instructions provide information not on the data described by the document, but on the properties of the document itself. In this case it specifies the version of the XML standard, and the character encoding, that has been used. They can be recognised as the tags having the question marks (?) as the first and last character within the triangle brackets. These tags do not have any "closing", and appear mainly at the top of the XML document.

The comments start with '<!--' and terminate with '-->'. The comment text may be placed within these tokens.

The XML standard does not restrict the choice of element names, attribute names, or allowed attribute values. Usually, however, the application will expect the XML document to define specific elements, with specific attributes. Such restrictions can be given as the XML schema in the format developed by the World Wide Web Consortium (W3C).

❗ *Tribon M3 provides two such XML schema files: **HullBasic.xsd**, and **CurvedHull.xsd**. They can be found in the Bin/xml folder.*

In order to specify the restrictions in the XML file, we have to add two special attributes to the root element.

`<Ship xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="C:\Tribon\M3\Bin\xml\CurvedHull.xsd">`


The first attribute "**xmlns:xsi**" is a reference to the XML Schema language definition, and the second attribute "**xsi:noNamespaceSchemaLocation**" points to the location to the actual XML schema, that the current document applies to. With this declaration added, it is possible to validate the document against the provided XML Schema.

3.2 Handling XML files in Python

Since the XML documents are standard text files, it is virtually possible to handle them using standard Python abilities of handling files. This becomes, however, extremely difficult with the growing complexity of the document, because then the program must take care of handling the tree structure of the XML elements and a variety of their attributes.

Therefore interfaces have been developed to facilitate the handling of the XML documents in accordance to the standards published by W3C. One of the popular methods of managing XML documents is DOM (Document Object Model), which is a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents. The W3C web pages (<http://www.w3.org/DOM/>) provide more details about this standard.

Tribon M3 provides the PyXML package, giving access to various classes and functions for XML document manipulation.

 Please study the code of "Example 1.py", which creates the example document discussed in section 3.1 with added references to the curved hull XML Schema, and outputs it to the file.

As a rule of thumb, we can take the following approach, when creating XML documents:

1. Import at least the **Document** class from **xml.dom.minidom** module
2. Create an instance of the **Document** class
3. Use its **createElement()** method for creating elements, **createComment()** method for creating comments, etc.
4. Set the values of attributes using the element's **setAttribute()** method
5. Append the elements, comments, and other nodes under the parent node using its **appendChild()** method

❗ After creating a **Document** class instance, the document does not have yet any root element. Remember to add it to the **Document** class instance using the **appendChild()** method!

For storing the created document on the disk:

1. Import the **PrettyPrint()** function from **xml.dom.ext** module,
2. Open the file for writing
3. Use the **PrettyPrint()** method, passing references to the created XML document and to the open file.
4. Close the file

The PyXML package takes care of adding the proper processing instructions to the XML document. If you need to modify an existing XML document, first you need to have its DOM representation, which can be obtained using the following code

```
from xml.dom.minidom import parse
try:
    => doc = parse("D:\\Example.xml") #Build XML document from file
    try:
        ... #work on the Document
    finally:
    => doc.unlink() #Prepare for cleanup
except:
    kcs_ui.message_confirm("Error reading XML file!")
```

When you obtain the XML document using the above approach, you will find soon, that among the original elements of the document, the **parse()** function has inserted many additional Text nodes containing spaces, tab characters or newlines. This is caused by the fact, that the **PrettyPrint()** function inserts spaces, tabs and line breaks (newline characters) to improve the readability of the document, and the **parse()** function interprets them faithfully. The **unlink()** method helps the Python garbage collection mechanism to clean up the multitude of objects, that are created during the building of the DOM tree.

Although these additional Text elements do not harm much (with small to medium size of the XML files), you may want to get rid of them, by executing the following code:

```

def collectTextNodes(parent, tab):
    for node in parent.childNodes:
        if node.nodeType == node.TEXT_NODE and node.nodeValue.strip() == "":
            tab.append(node)
            collectTextNodes(node, tab)

nodesToRemove = []
⇒ collectTextNodes(doc.documentElement, nodesToRemove)
for node in nodesToRemove:
⇒     node.parentNode.removeChild(node)

```

 The `documentElement` property returns the reference to the **root element** of the XML document.

The `collectTextNodes()` function traverses recursively the XML tree and locates all Text nodes, that have only whitespaces as their values. After collecting, they are removed from the tree structure. The above code is also a simple example of how we can traverse the DOM tree and remove existing elements.

An alternative approach, as opposed to modifying the parsed DOM tree, is to combine all lines of the XML document, stripping all leading and trailing whitespaces, and parse the resulting string

```

from xml.dom.minidom import parseString
try:
    xmlFile = file("D:\\Example.xml", "r") #read ALL lines of the file
    try:
        lines = xmlFile.readlines()
    finally:
        xmlFile.close()
    #join the stripped lines together
⇒ xmlString = "".join([s.strip() for s in lines])
⇒ doc = parseString(xmlString) #Build XML document from a string
    try:
        ... #work on the Document
    finally:
        doc.unlink()
except:
    kcs_ui.message_confirm("Error reading XML file!")

```

Every other node in the DOM tree is a direct or indirect child node of the root element. For managing the DOM tree we can use the following interface of the Node class:

- `node.parentNode` – reference to the parent element of **node**
- `node.nextSibling` – reference to the next element at the same level (under the same parent), or None, if there is no next element
- `node.previousSibling` – reference to the previous element at the same level (under the same parent), or None, if there is no previous element
- `node.localName` – local name of **node** (element name)
- `node.nodeType` – type of **node** (Node.ELEMENT_NODE, Node.COMMENT_NODE, etc.)
- `node.getAttribute(attrName)` – the value of the attribute **attrName** (an empty string for non-existing attributes)
- `node.setAttribute(attrName, value)` – sets the value of the attribute **attrName** to **value**
- `node.removeAttribute(attrName)` – removes the attribute **attrName** from the element **node**
- `parent.firstChild` – reference to the first child element under **parent**
- `parent.lastChild` – reference to the last child element under **parent**
- `parent.childNodes` – list of references to the child elements under **parent**
- `parent.appendChild(node)` – adds **node** as the last child element under **parent**
- `parent.replaceChild(new, old)` – replaces the child element **old** with the element **new**
- `parent.removeChild(node)` – removes **node** from the list of child elements of **parent**
- `parent.insertBefore(node, other)` – places **node** in the list of child elements before the element **other**
- `node.getElementsByTagName(tagName)` – list of elements located somewhere below **node** (direct or indirect child elements), whose element name is **tagName**



Further details of the `minidom` API can be found in the documentation of the `xml.dom.minidom` package

Exercise 6: Modifying an existing XML file

Write the program that updates the example XML document from page 17 by:

- changing the location of the hull curve SPX951 to X=FR40,
- adding the hull curve SPZ25 at position Z=LP25 before hull curve SPX951
- adding (appending) the hull curve SPEXT to be defined by three points:
X = FR40, Y = 0, Z = LP25
X = FR40, Y = 3000, Z = LP27
X = FR42, Y = 0, Z = LP25

After modifying the updated XML document should be stored as the file **Example2.xml** in the SB_SHIPDATA folder.

Hint: See the documentation of the **Input Language of Curved Hull Modelling** in the section **Seam by Plane** (Hull curves by three points are generated in a similar way as seams)

3.3 Modelling curved hull objects using XML

All curved hull objects can be modelled using the special descriptive language based on XML standard. Once the input XML file is prepared, using methods explained in the preceding section, it can be interpreted using the statement

```
objectList = kcs_chm.run_XML(inputXMLFileName, logFileName)
```

which generates curved hull objects, defined in the input file, whose full path is given as the **inputXMLFileName** argument, and returns a list of Model class instances describing curved hull model objects, that have been built. Any error messages together with all information about the generation process are written to the log file, whose full path is given as the **logFileName** argument. If an empty string is provided, the log messages will be written to the application's log file. If an error is encountered an exception is raised, but the error details can be found in the log file.

If you create curved hull objects from scratch, using the above function is sufficient to achieve your goal, but if you are modifying existing objects, you have first to obtain their current definition

```
kcs_chm.output_XML(objectList, outputXMLFileName)
```

The XML representation of the curved hull model objects, defined as Model class instances in the list **objectList** is written to the file, whose full path is given as **outputXMLFileName** argument.

 *Any existing file will be overwritten!*

After getting the XML file using **kcs_chm.output_XML()** function, you can parse it into the internal DOM representation, as shown in the previous section, make appropriate modifications to the DOM tree, store the modified XML document in the disk file, and finally – interpret the modified XML file using the **kcs_chm.run_XML()** function.

Exercise 7: Modifying the shell profile

Write the program that updates the shell profile type and parameters for the **whole** shell profile indicated by the user (type "longitudinal" or "transversal"). These data are stored in the "Material" element. There can be one such element under "ShellProfile" element, which defines the default material for shell stiffeners, and the "ShellStiffener" elements can have their own "Material" elements, overriding the default values. Example:

```
<Ship xsi:noNamespaceSchemaLocation="D:\Tribon\M3\bin\xml\CurvedHull.xsd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ShellProfile ObjId="TTPL220">
    <Material Type="20" Parameters="300 12" Grade="A" />
    ...
  <Branch>
    ...
    <ShellStiffener Symmetry="Symmetric">
      <Material Type="20" Parameters="300 12" Grade="A" />
      ...
    </ShellStiffener>
  </Branch>
</ShellProfile>
</Ship>
```

Update the "Material" elements under "ShellProfile", and remove any other existing "Material" elements under "ShellStiffener" elements.

3.4 Direct modelling of curved hull objects

The **kcs_chm** module provides also functions manipulating curved hull objects directly. For example, the function shown below will split the shell profile or the shell stiffener at a given position defined by some other model object

```
objectToSplit = KcsModel.Model("transversal", "SPT9")
splittingObject = KcsModel.Model("plane panel", "JUMBO-PLF9100")
⇒ kcs_chm.stiffener_split(objectToSplit, splittingObject)
```

or by the 3D plane

```
objectToSplit = KcsModel.Model("transversal", "SPT9")
point = KcsPoint3D.Point3D(0, 0, 5000)
normalVector = KcsVector3D.Vector3D(0, 0, 1)
splittingObject = KcsPlane3D.Plane3D(point, normalVector)
⇒ kcs_chm.stiffener_split(objectToSplit, splittingObject)
```

objectToSplit is always a Model class instance describing either the shell profile or the shell stiffener, that is going to be split. **splittingObject** can be either the Model class instance or the Plane3D class instance describing the object, that defines the location of the split. After successful split, the whole shell profile is locked. The lock is released by calling

```
kcs_chm.store(shellProfileObject)
```

which also stores the updated shell profile on the databank, or ...

```
kcs_chm.skip(shellProfileObject)
```

which releases the lock only, without saving the changes.

- ❗ The function **kcs_chm.store()** updates also all objects related to the object being stored (e.g. shell stiffeners, trace curves, limit tables, etc.). The function **kcs_chm.skip()** also releases the lock from all related objects.

Two neighbouring stiffeners can be combined into one

```
stiffener1 = KcsModel.Model("curved stiffener", "SPT9-S1")
stiffener2 = KcsModel.Model("curved stiffener", "SPT9-S2")
⇒ combinedStiffener = kcs_chm.stiffener_combine(stiffener1, stiffener2)
profile = KcsModel.Model("transversal", "SPT9")
⇒ kcs_chm.store(profile)
```

- ❗ Please note, that the splitting or combining of stiffeners locks the entire shell profile, to which the stiffeners belong. In order to release the lock, the functions **kcs_chm.store()** or **kcs_chm.skip()** must be called for the **PROFILE**, not for the stiffeners.

The properties of the shell stiffeners can be obtained and updated by Vitesse, which uses the classes in the **KcsShStiffProp** module for storing the properties of the stiffener, its profile and the ends.

```
stiffener = KcsModel.Model("transversal", "SPT9")
stiffener.SetPartType("stiffener")
stiffener.SetPartId(6001) #Part ID of the stiffener
⇒ prop = kcs_chm.stiffener_prop_get(stiffener) #ShStiffProp class instance
profType = prop.GetProfileType() #profile type
profParam1 = prop.GetProfileParameter(0) #first parameter
profParam2 = prop.GetProfileParameter(1) #second parameter
```

The above code retrieves the properties of the given stiffener on a transversal, where we are building the Model class instance ourselves, providing appropriate settings for model's type, name, part type and part ID. The Model class instance could be also obtained by identifying the indicated stiffener (see **kcs_draft.model_identify()**). Updating of the shell stiffener's properties can be done as shown below

```
stiffener = KcsModel.Model("transversal", "SPT9")
stiffener.SetPartType("stiffener")
stiffener.SetPartId(6001) #Part ID of the stiffener
prop = KcsShStiffProp.ShStiffProp()
prop.SetFilletWeldDepth(3.5)
⇒ kcs_chm.stiffener_prop_set(stiffener, prop)
```

❗ The **ShStiffProp** class initialises all properties to **None**, and the **kcs_chm.stiffener_prop_set()** function updates only the properties, whose value is not **None**. Therefore, it is NOT necessary to obtain the stiffener properties first, in order to update only a few of them – just set the new properties.

📖 See the source code of the **KcsShStiffProp** module for the available settings and methods.

The function below creates a shell curve or a shell seam as an intersection between the surface and a principal plane

```
curveName = "MYCURVE"
res = kcs_util.pos_to_coord(1, 40) #translate FR40
if res[0] == 0:
    xPos = res[1]
    minPoint = KcsPoint3D.Point3D(xPos - 1.0, 0.0, 2000.0)
    maxPoint = KcsPoint3D.Point3D(xPos + 1.0, 100000.0, 100000.0)
    surfaceName = "SPHULL"
⇒ model = kcs_chm.curve_principal_create(curveName, "X=FR40", \
    minPoint, maxPoint, surfaceName)
    kcs_chm.store(model)
```

❗ If the first argument (**curveName**) is a valid name for shell seams, as defined by the **Hull Reference Object**, a hull seam is created. Otherwise, a shell curve is created.

If the plane, that intersects the surface is not principal, we have to define it by specifying the point lying on the plane, and the normal vector, defining the plane's orientation. Then, another function must be used:

```
curveName = "MYCURVE"
point = KcsPoint3D.Point3D(30000, 0, 10000)
normalVector = KcsVector3D.Vector3D(1.0, 1.0, 0.0)
plane = KcsPlane3D.Plane3D(point, normalVector)
minPoint = KcsPoint3D.Point3D(25000, 0.0, 2000.0)
maxPoint = KcsPoint3D.Point3D(35000, 100000.0, 100000.0)
surfaceName = "SPHULL"
⇒ model = kcs_chm.curve_planar_create(curveName, plane, \
    minPoint, maxPoint, surfaceName)
    kcs_chm.store(model)
```

Both functions return a Model class instance of the shell curve (seam), that has been created, suitable for passing to **kcs_chm.store()** function.

Holes can be created in curved panels. In order to provide the hole settings, you need to define properly an instance of the **PanHoleOptions** class.

```
options = KcsPanHoleOptions.PanHoleOptions()
... #indicate the curved panel - model, and the origin - point
#origin set along the Y axis (X and Z co-ordinates taken from point)
options.SetOriginAlongAxis(2, point, 1)
options.SetDirection(anotherPoint) #direction from point to anotherPoint
... #set remaining options (e.g. designation, burn, mark, develop options)
⇒ kcs_chm.cpan_hole_create(model.Name, options)
    kcs_chm.store(model)
```

You can set not only the origin and direction of the hole, but also: the designation, marking length, and options regarding the development, burning and marking of the hole. If the hole is successfully created, the curved panel is locked – the function **kcs_chm.store()** must be used to store the changes and release the lock.

It is possible to delete a curved hull model object by calling

```
kcs_chm.delete(model)
```

where **model** is a Model class instance describing the curved hull model object to delete. All related objects are automatically updated or deleted. In a similar fashion, we can recreated the curved hull model object

```
kcs_chm.recreate(model)
```


3.5 View handling

In Curved Hull Modelling application we can create a number of special views displaying the curved hull model objects. Tribon Vitesse provides functions similar to the function `kcs_draft.view_new()`, that create such views and return their handles. The properties of these views are handled by instances of special classes, provided as an argument to these functions

```
handle = kcs_chm.view_shellexp_new(viewOptions)
handle = kcs_chm.view_bodyplan_new(viewOptions)
handle = kcs_chm.view_curvedpanel_new(panel, viewOptions)
```

Function	View properties class	View properties module
<code>kcs_chm.view_shellexp_new()</code>	ShellXViewOptions	KcsShellXViewOptions
<code>kcs_chm.view_bodyplan_new()</code>	BodyPlanViewOptions	KcsBodyPlanViewOptions
<code>kcs_chm.view_curvedpanel_new()</code>	CurvedPanelView	KcsInterpretationObject

After the view is created, it can be manipulated using standard abilities of the `kcs_draft` module. The panel argument in the `kcs_chm.view_curvedpanel_new()` function is the Model class instance describing the curved panel. The general pattern of creating these views consists of three stages:

1. Initialise an instance of the appropriate view properties class.
2. Set properties using methods specific to the given view properties class.
3. Create the view, passing the prepared instance of the view properties class.

i Since the methods and attributes of the view properties classes are specific to the given type of view, consult the source file of these classes to find the proper methods, that should be used.

If you want to update some options of the view, first get its settings by calling

```
options = kcs_chm.view_modify(viewHandle)
```

The result is an instance of one of the "view properties classes" mentioned above, containing the settings, that were used, when the given view was created. You can modify the settings, remove the old view, and create it again with the new settings.

i All curved hull modelling views are created in scale 1:1, and at the origin of the drawing form. They must be scaled and moved appropriately after creation, using `kcs_draft` module functions (like for 'standard' model views)

There are two other kinds of views, that do not require any special "view properties class":

```
stiffener = KcsModel.Model("curved stiffener", "SPT9-S1")
handle = kcs_chm.view_shprof_new(stiffener)
```

which creates the shell profile view for the given curved stiffener defined as the Model class instance, and

```
plate = KcsModel.Model("curved plate", "ES233-CPAN1-103")
handle = kcs_chm.view_devpla_new(plate)
```

or

```
plate = KcsModel.Model("developed plate", "ES233-CPAN1-4P")
handle = kcs_chm.view_devpla_new(plate)
```

which create the view of the developed shell plate defined as the Model class instance. There can be a little problem with getting proper names of the curved stiffeners and plates indicated by the user, since the function `kcs_draft.model_identify()` does not always return Model class instances objects with type of "curved stiffener" or "curved plate" ("developed plate"), as required by these functions. On a view displaying the curved panels, you can get the model type of "curved panel", and the part type of "stiffener" or "plate" and a proper part ID. On a view displaying shell profiles and plates, you can get the model type of "developed plate" (with the plate name), or "transversal" or "longitudinal" with the part type of "stiffener".

Summing up, usually the identified Model class instance cannot be used directly as an argument to the functions `kcs_chm.view_shprof_new()` or `kcs_chm.view_devpla_new()`. Fortunately, in such cases the name can be derived from the proper subpicture, as shown below:

```
res = kcs_draft.model_identify(point, model)
name = kcs_draft.subpicture_name_get(res[1]) #derive from SUBVIEW
```


for identification on views displaying plates, where each plate is a separate SUBVIEW, and

```
res = kcs_draft.model_identify(point, model)
name = kcs_draft.subpicture_name_get(res[2]) #derive from COMPONENT
```

for identification on views displaying shell profiles or curved panels, where the plates and stiffeners are the COMPONENTS. Then the name obtained using this method can be used for defining a new Model class instance having the "curved stiffener", "curved plate", or "developed plate" type, as required.

 *In order to create the shell profile view, the given stiffener must be on the profile databank.*

It is possible to send the curved stiffener to the profile databank by using the following Vitesse code:

```
model = KcsModel.Model("curved stiffener", "SPT9-S1")
kcs_chm.stiffener_to_profdb(model)
```


4 Weld Planning

Tribon Vitesse is now able to access the functionality of the Weld Planning system. The interface has the following features:

- function to perform a weld calculation and create a weld table,
- function to get the calculation result,
- functions to change some data in an existing weld table,
- function to delete an existing weld table,
- function to perform a weld check.

The functions are made available in the Python program by the insertion of the statement

```
import kcs_weld
```



Details of the Weld Planning system can be found in the Tribon documentation at [Tribon M3 Weld Planning](#) → [Weld Planning](#) → [Weld Planning – User's Guide](#)

Following the convention used in the whole Tribon Vitesse API, the module contains the variable `kcs_weld.error`, that is set to the string describing the error, when an exception is raised.

4.1 Python classes for storing the welding information

The main object handled by the Weld Planning Vitesse API is the weld table, represented in Python as the `WeldTable` class. It stores such attributes as:

- the name of the weld table, its comment, and status,
- calculated values of the total weld length, total suspension length, and total connection length,
- welded joints information.

Welded joints are represented by instances of the `WeldedJoint` class, and each `WeldTable` contains a list of such instances. Each `WeldedJoint` class instance stores the information about two parts, that are welded, and some characteristics of the welded joint itself:

- the joint name, and comment,
- the weld type,
- calculated values of the joint length, suspension length, and connection length,
- assembly name, part name, part type, and extended part name of the two welded parts,
- the manual flag,
- information about the welds forming the welded joint.

The welds are represented by instances of the `Weld` class, and each `WeldedJoint` contains a list of such instances. Each `Weld` class instance stores the following information:

- the weld name and comment,
- the weld length, and weld leg length,
- the weld position,
- number of weld layers,
- connection, rotation, and inclination angles,
- torch vector,
- start and end of suspension,

- bevel code, thickness, and quality of both parts being welded,
- test procedure, process, and standard process,
- the weld geometry (as a **GeoContour3D** class instance)



The source files **KcsWeldTable.py**, **KcsWeldedJoint.py**, and **KcsWeld.py** contain the complete description of the available attributes and methods.

The classes described above contain the appropriate methods for getting and setting the values of their attributes. It is rather unlikely, that you will be setting the numerical attributes manually. Instead you would rather use the functions in the **kcs_weld** module, that automate the process of weld detection and calculation.

4.2 Weld Planning functions

The system is able to detect welds within the assembly. The detection process can be launched by calling the function

```
status = kcs_weld.weld_calculation(assemblyName, 0)
```

if the calculation should be made for the given assembly only, or

```
status = kcs_weld.weld_calculation(assemblyName, 1)
```

if the calculation should take into account the given assembly and all subordinate assemblies. The **assemblyName** argument should be the path name of the assembly, for which the calculation should be performed. The result is **1**, if the calculation have been done successfully, or **0**, if an error has been encountered.

The result of the weld detection process is automatically stored in the databank, and can be retrieved by executing the following code:

```
weldTable = KcsWeldTable.WeldTable()
⇒ if kcs_weld.weld_properties_get(assemblyName, weldTable) == 1:
    totalWeldLen = weldTable.GetTotalWeldLength() #calculated properties
```

The function **kcs_weld.weld_properties_get()** returns **1**, if successful in getting the properties. Other values indicate an error. Of course, we can investigate not only the attributes resulting from the calculation process, but all other information stored in the **WeldTable** class instance, like the comments, weld types, weld process specification, etc.

After getting current information about the weld properties of the assembly, we can update some attributes, and set the properties back to the assembly.

```
... #obtain the weld table of the assembly
nJoints = weldTable.GetNumberWeldedJoints()
for n in range(nJoints):
    weldedJoint = weldTable.GetWeldedJoint(n)
    weldedJoint.SetJointComment("Joint no. %d" % n)
⇒ kcs_weld.weld_properties_set(assemblyName, weldTable)
```



This function currently does not set properly the properties to the assembly. No exception is raised.

It is possible to delete from the databank an existing weld table object for the given assembly, using the function

```
status = kcs_weld.weld_delete(assemblyName)
```

The **status** of **1** indicates success, **0** – failure.

Exercise 8: Calculating total weld length of an assembly

Write the program that produces the report containing the total weld length, and joint lengths for all welded joints. These data should be calculated and retrieved for an assembly selected by the user.

If the trainees are familiar with the **kcs_assembly** module (advanced), the assembly should be selected from the list collected by the program. Otherwise, the user should be prompted to key in the assembly path name.