

FreeCAD

La dernière version de ce document est téléchargeable à [ce lien](#) !

Table des matières

I. Prise en mains.....	4
I.1. Préliminaires.....	4
I.1.a Installation.....	4
I.1.b Premier contact incontournable.....	4
I.1.c FreeCAD et les autres logiciels de CAO.....	4
I.2. Généralités.....	5
I.2.a Pour commencer, quelques vidéos avec la version 0.19.....	5
I.2.b Une bonne série de vidéos en français.....	5
I.2.c Quelques exemples avec des explications écrites.....	5
I.2.d Pour aller plus loin.....	6
I.2.e Autre vidéos.....	6
I.3. Manuels en ligne.....	7
I.3.a Les principaux ateliers (une bonne référence).....	7
I.3.b Les ateliers externes.....	7
I.3.c Pour aller plus loin.....	7
I.3.d Autres références.....	7
II. Premières précisions personnelles.....	8
II.1. Les icônes de l'arborescence.....	8
II.2. Si plus rien ne marche.....	8
II.3. Attention aux pièges dans les esquisses.....	8
II.4. Duplication, récupération d'esquisses.....	10
II.4.a Les fonctions de clonage.....	10
II.4.b L'outil géométrie externe du Sketcher.....	10
II.4.c L'outil Copie Carbone.....	11
II.4.d L'outil ShapeBinder (ou Forme liée).....	11
II.5. Problème très important (limite bug de conception).....	12
II.5.a Un exemple simple de problème rencontré.....	12
II.5.b Sites consultés et bonnes pratiques.....	14
II.6. Différence entre Part, Body et Feature.....	15
II.7. Résultat d'une opération booléenne.....	15
II.8. Paramétrage d'un projet.....	16
II.8.a Utilisation d'un fichier de paramètres.....	16
II.8.b Utilisation d'un sketch master.....	16
II.9. Coupes et sections.....	17

III. Import-Export.....	18
III.1. Utiliser FreeCAD pour la découpeuse laser.....	18
III.1.a Sites et vidéos.....	18
III.1.b Deux méthodes que j'ai utilisées.....	18
III.2. Impression 3D (???).....	19
III.3. FreeCAD et la CNC (???).....	19
IV. FreeCAD et Python.....	21
IV.1. Introduction.....	21
IV.1.a Pour commencer.....	21
IV.1.b Utiliser la complétion dans le console <i>Python</i>	21
IV.1.c Pour aller vraiment plus loin.....	22
IV.1.d Et le Debug (quand vous écrirez de gros programmes) ?.....	22
IV.1.e Python et C++.....	23
IV.2. Création d'objets dans FreeCAD.....	24
IV.2.a Manipulations géométriques.....	24
IV.2.b Utilisation de classes.....	25
IV.2.c Le Proxy.....	26
IV.2.d Premières descriptions de classes.....	26
IV.2.e Propriétés d'un objet de type <i>FeaturePython</i>	27
IV.2.f Installation, création de commandes.....	29
IV.3. Exemple de construction d'objet : épi(hypo)cycloïde.....	30
IV.3.a Extrusion à partir d'un script.....	30
IV.3.b Construction directe du corps.....	31
IV.3.c Première utilisation de classe.....	32
IV.3.d Installation.....	33
IV.3.e En faire une commande ??.....	34
IV.4. Approfondissements.....	35
IV.4.a Classes ViewProvider.....	35
IV.4.b Scenograph, Coin et Open Inventor.....	35
IV.5. Autres références peut-être intéressantes.....	36
IV.6. Sous-objets topologiques et géométriques.....	38
IV.6.a Sous-objets topologiques.....	39
IV.6.b Support d'un Vertex (Point).....	40
IV.6.c Support d'un Edge (Curve).....	40
IV.6.d Support d'une Face (Surface).....	42
IV.6.e Courbes coordonnées d'une surface.....	43
IV.6.f Exemple avec des B-Spline.....	46
IV.6.g Et l'inverse : du simple au complexe?.....	46
IV.6.h La fonction toShape(...).....	47
IV.7. VRAC.....	48
IV.7.a Selection.....	48
IV.7.b B- Spline.....	48
IV.7.c Open-Cascade.....	48
IV.7.d vrac.....	48

V. Animation.....	50
V.1. Avant Assembly 3.....	50
V.1.a L'atelier Animation.....	50
V.1.b L'atelier A2plus.....	50
V.2. L'atelier Assembly 3.....	50
V.3. L'atelier Assembly 4.....	50
V.3.a Introduction.....	50
V.3.b Premiers exemples.....	51
V.3.c Extraits (plus techniques) du forum.....	51
V.3.d Hexapod.....	52
V.3.e Sous-Assemblages.....	52
V.4. Animation avec des macros Python.....	53
V.5. Engrenages.....	53
V.6. Collision et contact.....	55
V.6.a Approximation numérique d'un minimum.....	55
V.6.b La méthode distToShape().....	55
V.6.c Surveiller les modifications.....	57
V.7. Vrac.....	58
VI. Utilisation de fenêtres dans FreeCAD.....	60
VI.1. Environnement graphique QT.....	60
VI.2. Bonnes pratiques.....	60
VI.3. Création d'une fenêtre de dialogue.....	63
VI.3.a Fenêtres simples (????).....	63
VI.4. Widgets, QMainWindow et QDialog.....	65
VI.5. À trier.....	66
VII. L'atelier FEM.....	68
VIII. En vrac.....	69
VIII.1. L'atelier LaserCutInterlocking(bof ?).....	71

Remarque liminaire Ce document ne veut pas être un tutoriel complet d'initiation à FreeCAD. Il faut plutôt y voir l'ensemble des notes que j'ai prises lors de mon apprentissage de ce logiciel. J'y ai regroupé de nombreuses références existantes qui m'ont permis de progresser dans son utilisation , références que j'ai eu parfois du mal à trouver (d'où l'intérêt de ces notes). J'y ai aussi glissé quelques notes et remarques personnelles en espérant qu'elles puissent être utiles à d'autres personnes .

Introductory Note This document is not intended to be a complete tutorial for getting started with FreeCAD. Rather, you should see all the notes I took while learning this software. I have grouped together many existing references that have allowed me to progress in its use, references that I have sometimes had trouble finding (hence the interest of these notes). I also put in some personal notes and remarks in the hope that they may be of use to other people.

I. Prise en mains

I.1. Préliminaires

I.1.a Installation

- ✓ Installation (version 0.19 en octobre 2020) : voir [cette page](#).
 - *Attention*, il y a eu un gros changement entre les version 0.16 et la version 0.17 : les vidéos ou les explications concernant la 0.16 peuvent ne plus être valables à partir de la version 0.17.
 - Sous *Linux Debian*, la version des dépôts officielles de *Freecad* ne fonctionne pas correctement et je l'ai remplacée par la version du site *FreeCad*, qui est en fait une *app-Image* et ne nécessite aucune installation (sauf à rendre le fichier exécutable).
Il faut la lancer avec quelque chose du genre :
[App-Images/FreeCAD_0.18_xxxxxxx.AppImage](#)
- ✓ Au hasard des vidéos concernant *FreeCad*, j'ai rencontré l'utilisation de deux logiciels qui peuvent être intéressants dans d'autres domaines :
 - *Simple screen recorder* : Logiciel de capture d'écran pour faire des vidéos
Sous Debian, il y a aussi *RecordMyDesktop*, *Vokoscreen*
 - *Key-mon* : pour montrer à l'écran comment sont utilisés la souris et le clavier

I.1.b Premier contact incontournable

- ✓ Une [très bonne page](#) (en français) qui explique bien l'interface, et qui branche en particulier sur [cette page](#) expliquant bien les divers modes de navigation (voir en particulier le *touchpad*).
- ✓ Ensuite, vous pouvez regarder quelques vidéos et alterner avec la partie « références ».
- ✓ Une [page intéressante](#) expliquant la philosophie de construction des objets, surtout la partie intitulée « Méthodologie d'édition de fonctions ».
- ✓ Éventuellement, voir [comment personnaliser l'interface](#) ainsi que les [barres d'outils](#).

I.1.c FreeCAD et les autres logiciels de CAO

- ✓ Pour récupérer des projets Fusion 360
 - Il est dit sur [cette page](#) de passer par un export/import STEP.
 - J'ai aussi rencontré [cadexchanger](#) qui a une version d'évaluation gratuite mais je ne suis pas allé voir plus loin....
 - Dans [cette vidéo](#), on voit comment transformer un objet mesh en body !
- ✓ Dans [cette page](#), il y a un post qui parle des autres logiciels semblables et en particulier de [Solvespace](#), qui paraît avoir la préférence de l'auteur

I.2. Généralités

I.2.a Pour commencer, quelques vidéos avec la version 0.19

- ✓ Un [exercice débutant](#) (avec 0.19) utilisant l'atelier *PartDesign* ainsi que des symétries et des répétitions circulaires.
- ✓ Un [exercice simple](#) (avec 0.19) avec *PartDesign* autour de la révolution et de la création de second corps.
- ✓ Un [exemple utilisant un tableur](#) (atelier *Spreadsheet*) pour paramétrer un objet. : *un bon réflexe !*

I.2.b Une bonne série de vidéos en français

La [série de vidéos suivantes](#) est issue du [site CAD Printer](#).

- ✓ Elle est très intéressante mais date de 2016/2017 et dans la [vidéo d'intro](#), on voit qu'elle utilise la version 0.16 ; donc faire attention car il y a eu de grosses modifications entre la 0.16 et la 0.17.
- ✓ Mais personnellement je n'ai pas vu d'incompatibilité.

Modulo la remarque précédente, c'est une très bonne introduction à *FreeCAD*, et on peut regarder ces vidéos les unes à la suite à des autres. J'en ai extrait ci-dessous quelques unes, dont je mets en évidence les points forts, mais c'est un choix tout personnel.

- ✓ Une [vidéo intéressante](#) concernant le paramétrage de *FreeCAD* ainsi que la description des fonctions et des icônes permettant de travailler sur les esquisses.
- ✓ Un [bel exemple de surface de révolution](#) (icône 🗑️) montrant aussi comment on peut utiliser un axe de révolution différent des axes du repère.
- ✓ Une vidéo (intéressante même si le début est un peu long) sur [comment utiliser une géométrie externe](#) (icône 📦) et expliquant quelques pièges associés.
- ✓ Une vidéo sur l'[utilisation des fonctions de symétries et de répétition](#) : icônes 📐, 📐, 📐, etc.
- ✓ Une vidéo sur [arrondis, champfrein et dépouilles](#)
- ✓ Une très bonne vidéo [introduction à l'atelier Part](#).
 - Mais elle parle aussi de la fenêtre *Propriétés* qui apparaît en dessous de la hiérarchie et qui permet de modifier couleur et placement de tous les objets (qu'ils aient été produits avec l'atelier *Part* ou avec l'atelier *PartDesign*) .
 - Et elle montre aussi comment on peut utiliser successivement différents ateliers.
- ✓ Une superbe vidéo montrant [comment faire un filetage](#) (et [une autre](#))
- ✓ Une vidéo [lissage et blayage](#) (loft et sweep)
- ✓ [Protusion et cavité](#) sur une surface non plane
- ✓ Vidéo de [transition entre les versions 0.16 et 0.17](#)

I.2.c Quelques exemples avec des explications écrites

Pour ceux qui préfèrent lire des textes plutôt que de regarder des vidéos : une [autre entrée possible](#) à partir d'exemples commentés dont :

- ✓ la célèbre [pièce de Lego](#) ;
- ✓ un [cube joliment creusé](#) (atelier *Part* et placements) ;
- ✓ un [beau filetage](#) et un [autre filletage](#).

I.2.d Pour aller plus loin

- ✓ Pour comprendre les corps, [regarder cette vidéo](#) et/ou aussi [cette vidéo de jp-willm](#)
- ✓ Une [boite avec pas de vis](#) : il fait une soustraction avec une hélice et des répétitions avec l'atelier *draft* ; il navigue aussi entre les ateliers *Part* et *PartDesign*
- ✓ Réalisation d'une [filière](#)
- ✓ Construire [un hand spinner](#)
- ✓ Un exemple de [modélisation d'une maison](#) (pas encore regardé)
- ✓ Création d'un [filetage avec FreeCAD](#) (pas encore regardé)
- ✓ [hélice de bateau](#) (pas encore regardé)
- ✓ La [création d'un pentagramme](#) avec utilisation de clones (et [quelques critiques](#))
- ✓ Réalisation d'un [porte atos](#) (avec multiple et draft)
- ✓ Un [exercice de visserie](#)

I.2.e Autre vidéos

- ✓ Introduction à l'animation (mais avec 0.17) et le module Animation : [partie I](#), [partie II](#), [partie III](#)
Toutefois ce module ne fonctionne plus avec la version 0.19.
 Idem pour cette animation d'un [moteur diesel](#)
- ✓ Un [très long projet](#) de quelqu'un qui hésite : ça peut être intéressant de regarder (1h45mn)
- ✓ Comment [importer une image](#) (mais en passant par Gimp et SVG)

I.3. Manuels en ligne

I.3.a Les principaux ateliers (une bonne référence)

Chacune des pages suivantes est disponible en français aussi bien qu'en anglais : il suffit de cliquer sur la langue en haut (dans le cadre **Autres langues**).

La [table des matières](#) de l'aide en ligne (où l'on retrouve comment installer FreeCAD). On y retrouve en particulier des descriptions des premiers ateliers utilisés :

- ✓ l'[atelier Start](#), qui est juste la page présentée lorsqu'aucun projet n'est ouvert ;
- ✓ l'[atelier Part](#) qui permet de manipuler les volumes (cubes, sphères, cones, etc.) surtout à l'aide d'opération booléennes ;
- ✓ l'[atelier Part Design](#) qui permet de faire des pièces plus complexes à base d'esquisses, de protusion, de perçage, etc. ;
- ✓ l'[atelier Sketcher](#) qui permet de faire des esquisses (base de l'atelier *Part Design*) ;
- ✓ l'[atelier Draft](#) qui contient aussi des outils de dessin 2D, et en particulier l'outil clone (tête de mouton) permettant de cloner tout une esquisse comme dans l'animation du piston.
- ✓ l'[atelier SpreadSheet](#) qui permet de paramétrer les objets que l'on construit ;
- ✓ l'[atelier TechDraw](#) qui permet de réaliser des dessins cotés (voir [un tuto d'introduction](#)) ;
- ✓ l'[atelier Path](#) qui permet de générer les sorties pour CNC et découpeuses laser.
- ✓ L'[atelier draft](#) qui permet de faire des dessins en 2D mais plutôt avec des objets qui s'accrochent à une grille (voir par exemple [cette vidéo](#)). Mais cet atelier permet aussi de faire des réseaux orthogonaux ou circulaires de corps existants (??? Doc ???)

I.3.b Les ateliers externes

Aux ateliers précédents, on peut ajouter des ateliers externes en fonction de ses besoins. On en trouve la liste [sur cette page](#) (la [même en anglais](#)). On en particulier peut citer :

- ✓ l'[atelier Fasteners](#) qui permet d'ajouter facilement vis, boulons, écrous, etc. ;
- ✓ l'[atelier FCGear](#) concernant les engrenages.

Comme il est expliqué sur [cette page dédiée](#), ces ateliers externes s'installent à l'aide de `<outils><Gestionnaire d'add-on>`. Il est indispensable de relancer FreeCAD avant leur utilisation.

- ✓ L'[atelier Assembly 4](#) étudié en détails dans la suite de ce papier.

I.3.c Pour aller plus loin

- ✓ L'outil [Fragments booléens](#) (qui donne toutes les combinaisons possibles) et où l'on parle aussi de programmation.
- ✓ Un [bel exemple costaud](#) pour faire un pied de fauteuil à 5 branches
- ✓ Une [boite en plusieurs parties](#) avec le couvercle et les vis de fixation (assemblage)

I.3.d Autres références

- ✓ Création d'[engrenages](#) mais avec une vieille version de FreeCAD
- ✓ Un site intéressant avec plein de références [sur cette page](#) ou [sur celle-là](#).
- ✓ Une [page intéressante](#) sur les nouvelles fonctionnalités.

II. Premières précisions personnelles

II.1. Les icônes de l'arborescence

Sur [cette page](#), on peut voir la signification des icônes apparaissant dans la vue arborescente.



indique seulement que l'objet doit être [recalculé](#) ;



indique le [Tip](#) du corps : fonction résultante, qui est exportée vers l'utilisateur ;



indique que l'objet n'est pas attaché à quelque chose, il ne dépend que sa propriété [Placement](#) ;



indique une esquisse non entièrement contrainte ;



indique que l'objet présente une erreur qui doit être corrigée.

II.2. Si plus rien ne marche



- ✓ Quand on n'arrive pas à faire ce que l'on veut dans l'onglet *Modèle (Model)* ou que toutes les icônes sont grisées et donc inutilisables, ou que le programme couine sans arrêt, *ce n'est pas forcément parce que le programme s'est planté* comme je l'ai souvent cru au début !
- ✓ Dans un tel cas, il faut toujours commencer par vérifier s'il n'y a pas quelque chose en cours dans l'onglet *Tâches (Task)*, que ce soit une esquisse, une protusion ou tout autre chose. Il suffit alors de fermer cette tâche pour que généralement tout rentre dans l'ordre.
- ✓ Pour les quelque (rares) cas où j'ai vraiment eu un plantage, *FreeCAD* propose une récupération du dernier état sauvegardé. En fait, la plupart des plantages que j'ai subis sont arrivés après que j'ai abusé des <Ctrl+Z> pour faire de nombreux retours en arrière.

II.3. Attention aux pièges dans les esquisses


Parfois (et même souvent au début) il est impossible de pouvoir faire une protusion ou une esquisse après avoir refermé un sketch ; cela se traduit de deux façons différentes :

- ✓ lorsque l'on ouvre la fenêtre protusion (ou autre) l'objet s'efface de l'écran ;
- ✓ on reçoit le message « *Failed to validate broken face* » ou « *Linked shape object is empty* ».

Il faut alors penser à vérifier les points suivants.

- ✓ S'il ne possède qu'une courbe, le contour servant de base à la protusion doit être bien fermé ; commencer par vérifier que les points, que l'on croit identiques, le sont bien, en essayant par exemple de les déplacer ; si ce n'est pas le cas, les sélectionner utiliser la  contrainte de coïncidence (raccourci « C » sur le clavier).
- ✓ Une fois l'esquisse fermée, on peut aussi, après avoir sélectionné l'icône sans l'ouvrir, utiliser l'outil <Sketch><Valider l'esquisse>, d'icône , qui permet entre autres de chercher les sommets ouverts : ils apparaissent alors en orange, ce qui permet de les repérer.
- ✓ Le contour peut contenir plusieurs courbes fermées, mais dans ce cas :
 - il doit y avoir une courbe externe déterminant le bord extérieur de l'objet,
 - une ou plusieurs courbes internes à la précédente, qui ne doivent pas se couper franchement : ces courbes délimitent alors des trous dans le contour externe.

- ✓ Le contour qui doit servir de base à la protusion (extrusion) ne doit contenir :
 - que des *lignes vertes* s'il est complètement contraint,
 - que des *lignes blanches ouvertes* s'il n'est pas complètement contraint.

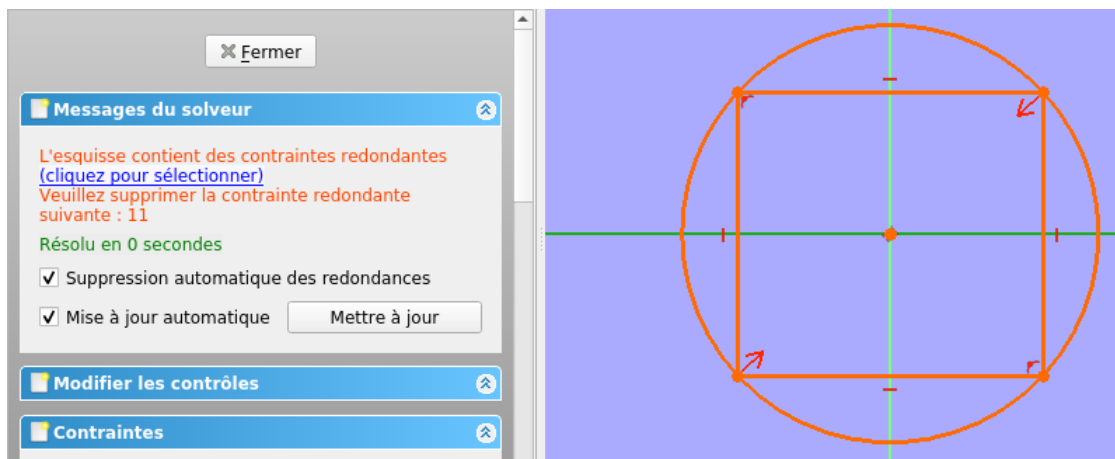
S'il contient des *lignes bleues*, alors ce sont des lignes de construction, et elles ne peuvent pas servir à une protusion : si l'on veut les récupérer dans la construction, il faut les transformer en lignes « réelles » avec l'icône .

Dans le cas du message «*Pad : Result has multiple solids. This is not supported at this time* », bien vérifier que le corps activé (en gras) lors de la création du sketch est bien celui que l'on pense : il m'est arrivé de sélectionner un corps sans l'activer et de penser y avoir créé un sketch, alors que le sketch était créé dans le corps activé !

En revanche, il est tout à fait possible de créer, le plus souvent à la racine du projet, un « master sketch » qui ne respecte pas les consignes précédentes, mais qui ne sera utilisé que pour y récupérer des dimensions,

- ✓ soit à l'aide de **Sketch.Constraints**....
- ✓ soit à l'aide de l'outil **Copie Carbone**, d'icône .

Quand on travaille dans un sketch, il est préférable de laisser cochée la case « *Suppression automatique des redondances* », sinon on se fait souvent insulter avec le message « *L'esquisse contient des contraintes redondantes ...* » : par exemple si l'on essaie d'inscrire mettre dans un cercle centré en O, un rectangle que l'on veut aussi centrer en O.





Remarque Lorsque l'on rencontre, ce message de contraintes redondantes, cliquer sur le lien proposé puis aller parcourir la fenêtre de Contraintes, car les contraintes en cause n'apparaissent pas en général spontanément à l'écran.

Attention Les lignes de construction sont visibles uniquement en mode édition : on ne les voit plus quand on affiche le sketch en dehors de ce mode !

II.4. Duplication, récupération d'esquisses

- ✓ *Bien que ce ne soit pas une très bonne pratique* pour des projets compliqués (voir section suivante), il est possible de construire des esquisses en récupérant des éléments géométriques du même *body*, voire d'un *body* différent.
- ✓ Il est possible dans FreeCAD de faire du copier/coller, que ce soit par menu ou en utilisant les raccourcis clavier, mais ce n'est pas toujours évident. Il me paraît préférable d'utiliser les fonctions suivantes.


II.4.a Les fonctions de clonage


- ✓ **La fonction clone de l'atelier *Part Design***, d'icône  : appliquée à une esquisse, elle crée un *Body* et y place un clone de cette esquisse.
- ✓ **La fonction clone de l'atelier *Draft***, d'icône  : appliquée à une esquisse, elle en crée simplement un clone au niveau supérieur de l'arborescence. On peut ensuite déplacer ce clone où l'on veut.

Dans les deux cas cette copie est dynamiquement liée à l'élément d'origine : c'est-à-dire qu'elle en suit les modifications et les évolutions.

II.4.b L'outil géométrie externe du Sketcher

C'est l'[outil de l'atelier *Sketcher*](#) d'icône .

- ✓ Cet outil permet de copier des arêtes et des sommets du corps dans lequel se trouve le sketch. Ces copies sont des lignes de constructions, apparaissant en magenta pour les distinguer des autres lignes de construction, de couleur bleue.
- ✓ Attention la « matière » peut empêcher de voir les lignes que l'on veut copier :
 - il est donc, la plupart du temps, nécessaire de cacher cette matière en « éteignant » le(s) *pad(s)* correspondant(s) à l'aide de la barre d'espace ;
 - mais on peut aussi modifier le style de représentation des objets, en le passant à *Filaire* (*Wireframe*) avec l'icône , ou plutôt la liste déroulante qu'elle permet d'ouvrir.
- ✓ La copie ainsi réalisée est alors liée dynamiquement à l'élément d'origine : c'est-à-dire qu'elle en suit les modifications et les évolutions.

Attention Comme les lignes de construction, ces lignes magenta ne sont visibles que lorsque le sketch est en mode édition ; il n'est donc pas possible de les copier d'un sketch à un autre avec .



II.4.c L'outil Copie Carbone

C'est l'[outil de l'atelier Sketcher](#) d'icône .

- ✓ Cet outil permet de recopier toute la géométrie (même les lignes de construction) et les contraintes d'une esquisse quelconque ; lorsque cette esquisse n'est pas située dans le même corps, il faut utiliser la touche <Ctrl>.
- En l'absence de contraintes sur la géométrie initiale, cette copie n'est pas dynamique : elle ne suit pas les évolution des lignes originelles.
- En revanche, comme il y a copie des contraintes, la copie est dynamique sur les éléments contraints dans le sketch d'origine ; et l'on peut donc utiliser cet outil pour faire un clone d'un sketch contraint : par exemple pour animer un mécanisme à l'ide d'un sketch master situé à l'origine de l'arborescence et que l'on copie ainsi dans les différents corps utilisés.
- ✓ On peut récupérer des esquisses qui ne sont pas dans le même plan avec <Ctrl> + <Alt> ; toutefois, personnellement, je n'ai réussi qu'à copier la première esquisse d'un corps.

II.4.d L'outil ShapeBinder (ou Forme liée)

C'est l'[outil de l'atelier Part Design](#) d'icône .

- ✓ Cet outil crée, dans le corps actif, une copie dynamique d'un élément quelconque, qui peut être un corps entier, un sketch, une face, une arête, etc.
- ✓ Ma façon préférée de l'utiliser est la suivante :
 - sélectionner et activer un corps ;
 - sélectionner dans l'arborescence (ou sur le schéma, surtout pour les faces ou les arrêtes) l'élément que l'on veut copier ;
 - en cliquant sur l'icône , un élément *ShapeBinderxxx* est alors créé dans l'arborescence du corps actif (on peut éteindre tous les autres éléments pour le voir mieux sur le schéma) ; il est alors possible de récupérer des éléments de ce *Shapebinderxxx* à l'aide de .
 - Si l'on n'a pas pensé assez tôt à créer ce *ShapeBinderxxx*, il peut se retrouver plus bas dans l'arborescence que l'esquisse où l'on voudrait l'utiliser ; comme j'ai peur que cela ne puisse créer des problèmes de références circulaires, je préfère le remonter avec le menu contextuel <clic droit> <Déplacer après un autre objet>
- ✓ Si l'on a modifié le placement de certains objets, par exemple à l'aide de la fenêtre *Propriété/Valeur*, il se peut que la forme que l'on vient de créer ne soit pas où on l'attend : dans un tel cas, mettre l'option *Trace Support Shape*, à *true* résout en général le problème.

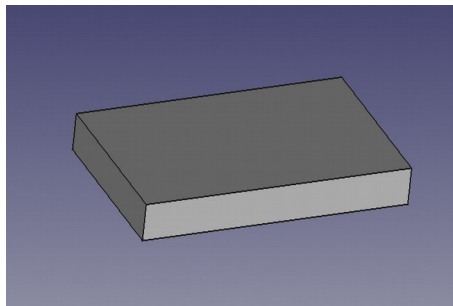
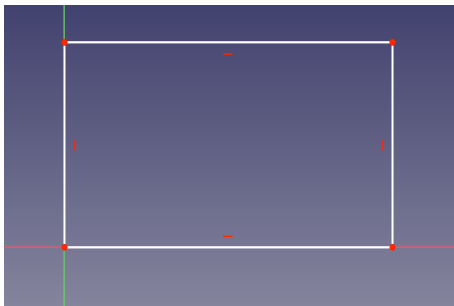
II.5. Problème très important (limite bug de conception)

Attention au [problème](#) concernant la renumérotation des noms de côtés (problème de dénomination topologique ou *topological naming issue*) lorsque l'on modifie une esquisse.

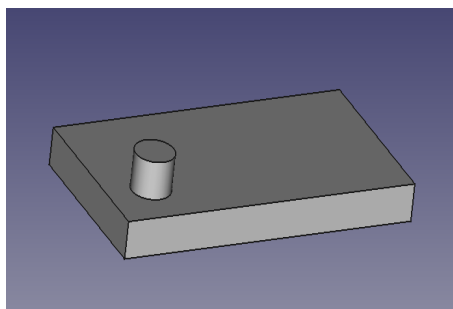
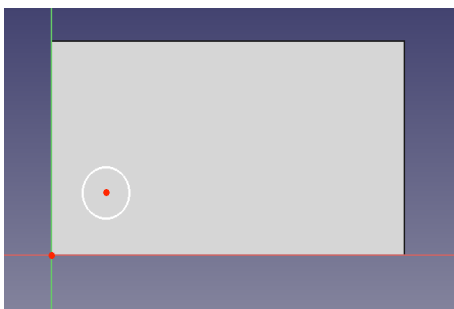
II.5.a Un exemple simple de problème rencontré

Le Problème

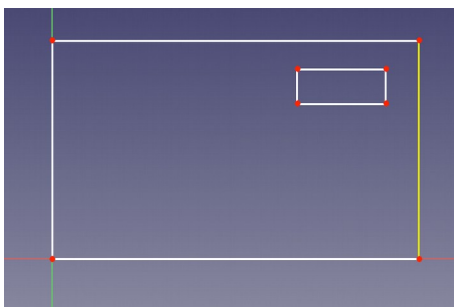
- ✓ Partant d'un projet vide, créer un sketch sur le plan XOY, puis faire une protusion standard (10mm) ; on obtient le *pad* :



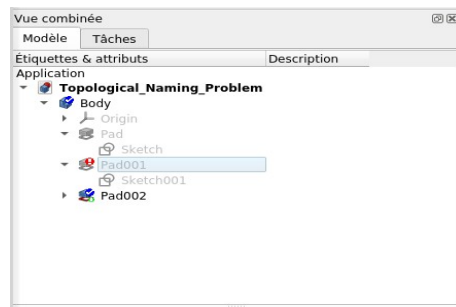
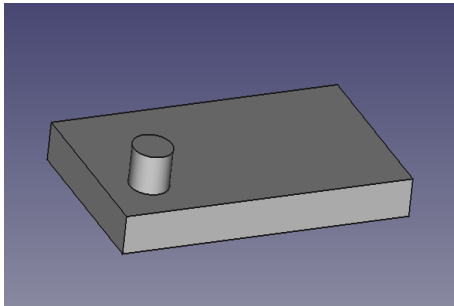
- ✓ Sélectionner sa face supérieure et y accrocher un sketch pour faire un tenon cylindrique ; après protusion, on obtient :



- ✓ Si l'on s'aperçoit alors que l'on a oublié un trou rectangulaire dans *pad*, le plus naturel est de revenir sur *sketch* et d'y ajouter un rectangle comme suit :

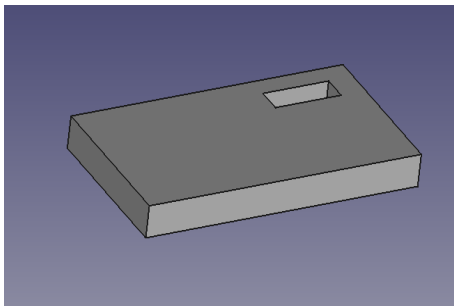


- ✓ Mais quand on ferme cette esquisse, on revient au même corps que précédemment sans voir le trou rectangulaire que l'on pensait bien avoir créé, et on voit dans l'arborescence qu'il y a un problème avec *Pad001* :



Explication


- ✓ Si à l'aide de la barre d'espace, on « allume » *Pad*, on voit bien le trou rectangulaire :



Remarque : Il est tout à fait normal de ne pas voir le tenon cylindrique puisque toute la suite des opérations a été éteinte.

- ✓ Mais si l'on « allume » à nouveau *Pad001*, rien n'a changé.
- ✓ En fait tout cela est dû à la façon dont on a créé *Sketch001*.
 - On l'a attaché à la face supérieure de *Pad* (le bloc initial).
 - Comme on peut le voir dans la fenêtre *Propriété* lorsque *sketch001* est sélectionné, il est attaché en mode *FlatFace* à *Pad[Face6]*.
 - Initialement, avant d'y avoir fait le trou rectangulaire, *Pad* avait 6 faces numérotées de 1 à 6, et *sketch001* a été défini attaché à la 6.
 - Mais la création du trou rectangulaire a créé 4 nouvelles faces et la face supérieure où doit être positionné *sketch001* est maintenant la face 10.
 - On peut voir le numéro d'une face en la survolant avec la souris : lorsque la face devient jaune, son numéro est affiché en bas à gauche de la barre d'état de la fenêtre *FreeCAD* dans un format du type <Nom_Fichier>.Body.Padxxx.Facexx.
- ✓ Pour remédier au problème, il suffit d'attacher *sketch001* à la face 10. Pour cela :
 - Sélectionner et afficher *Pad*, puis sélectionner (simple clic) *Sketch001*.
 - Aller dans la fenêtre *Propriété*, partie *Attachement* ligne *Map Mode* et cliquer sur les « ... »
 - Dans la fenêtre qui s'ouvre remplacer *Face6* par *Face10*, soit en le tapant soit en cliquant sur la face correspondante dans la fenêtre graphique ; puis cliquer sur <OK>.
 - Tout revient alors dans l'ordre, éventuellement après avoir forcé le re-calcul de *Pad001* et l'avoir allumé avec la barre d'espace.

Qu'aurait-on pu faire dans ce cas pour éviter un tel problème ?




- ✓ Le plus simple eut été de créer un *Sketch002* dans lequel on aurait mis le rectangle, puis de faire une cavité (*pocket*) à l'aide de ce sketch. D'où une première règle que j'essaie de m'imposer : *ne jamais revenir dans l'arbre effectuer une opération qui peut modifier la numérotation des faces si plus loin dans l'arbre j'ai accroché une esquisse à une face.*
- ✓ L'idéal eut été de ne pas créer *sketch001* sur une face du premier corps.
 - On aurait pu le construire sur un [plan de référence](#) (*datum plane*, icône ) que placé à la bonne côte ($z=10$ ici), l'idéal étant que cette côte soit définie dans un fichier de paramètres.
 - On aurait pu aussi placer ce *sketch001* dans un des plans de coordonnées de l'objet (*OXY* dans notre cas) et translater l'esquisse avec la propriété *Placement* (paramétrée évidemment) pour l'amener sur la bonne face.

II.5.b Sites consultés et bonnes pratiques

- ✓ Voici [ce qu'on en dit dans la doc officielle](#).
 - Sur cette page, on trouve un bel exemple de problème posé par l'introduction d'une nouvelle face lorsque l'étage intermédiaire déborde du niveau 0.
 - Mais la solution proposée me paraît bien fragile car, par exemple, le premier plan de référence introduit n'est pas lié dynamiquement à la face supérieure du niveau 0 : tout se passe bien tant que l'on n'en change pas la hauteur ! Pour moi, il manque un fichier de paramètres.
 - ✓ On trouve [sur cette page](#), et [plus particulièrement ici](#), les bonnes pratiques à respecter pour avoir un modèle stable !
 - On y trouve en particulier l'éloge de l'utilisation de fichiers de paramètres.
 - Il est donc préférable de baser toute esquisse dans un des plans *OXY*, *OYZ*, ou *OZX* de l'objet, voire dans un plan de référence dont la position est calculée en fonction des données du fichier de paramètres. Voir éventuellement [la fonction Attachment de l'atelier Part](#).
 - Si l'on veut aller vraiment plus loin, s'intéresser à la propriété *Map Mode* des sketches.
 - ✓ Comme il est expliqué dans [ce post](#), on peut avoir intérêt à changer le solver, mais c'est quelque chose que je n'ai pas encore essayé.
 - ✓ Un post intéressant sur [l'intérêt des plans de référence](#) (*datum planes*) et leur bon usage :
 - ça ne sert à rien de les lier à une face, ça ne fait que repousser le problème ;
 - il faut de préférence les positionner par rapport aux plans de base de l'objet en utilisant leur propriété *Placement* ;
 - mais on peut aussi, de la même façon placer, le plan d'une esquisse par rapport aux plans de base de l'objet en utilisant leur propriété *Placement* ; le seul intérêt d'utiliser un plan de référence est qu'on le voit (en jaune par défaut) sur l'écran ;
 - ✓ Un post du forum sur [comprendre les "attachement"](#) et une [autre page sur le sujet](#)
 - ✓ Une autre sur [comment déboguer un modèle](#) : penser entre autres au graphe de dépendance, que l'on obtient avec <Outils><Graphique de dépendance>. Mais il faut au préalable avoir installé *graphwiz*.
 - ✓ Une [macro intéressante](#) (pas vérifié) pour contourner le problème de dénomination topologique
- Mais ce problème [serait en passe d'être réglé ???](#)

II.6. Différence entre Part, Body et Feature

Au début j'ai beaucoup hésité avec les objets que je manipulais et qui apparaissaient dans la hiérarchie de l'onglet *Modèle* (Model).



- ✓ Le [Body ou Corps](#) () est l'objet de base que l'on construit à l'aide de l'atelier *Part Design*.
Un objet construit à l'aide de l'atelier *Part* n'est pas un *Body* :
 - il n'a d'ailleurs pas la même icône à gauche de son nom ;
 - je ne peux pas directement créer une esquisse sur l'une des ses faces à l'aide de l'atelier *Part Design* ; il faut au préalable l'avoir inclus dans un *Body* où il devient une *BaseFeature* ;
Pour une explication sur cette histoire de *BaseFeature*, voir [cette page](#) ou [cette autre page](#).
- ✓ Le [Part ou Std Part](#) () est un container qui permet de regrouper des objets, par exemple pour toujours les déplacer ensemble. À ne pas confondre avec l'atelier *Part* !
- ✓ Le [Group ou Std Group](#) () est un conteneur permettant simplement de regrouper des objets dans la vue arborescence pour la rendre mieux organisée.
- ✓ La [Feature ou Fonctionnalité](#) est une étape dans le processus de fabrication d'un corps (*body*).

Il y a une [très bonne explication ici](#), ??? revoir ce paragraphe ??? en particulier avec des schémas intéressants ([schéma seul](#)) . Et tout compte fait, c'est très bien expliqué dans les pages [Body](#), [Std_Part](#) et [Feature](#), à condition des les lire jusqu'au bout, ce que je n'avais pas fait au début.

Remarque : il est maintenant possible de déplacer les fonctionnalités à l'intérieur d'un objet en utilisant le clic droit de la souris, mais utiliser avec modération car on risque de casser des liens.

II.7. Résultat d'une opération booléenne

Le résultat d'une opération booléenne (*cut*, *union*, *intersection*) n'est pas un *body* !

- ✓ Si je veux mettre une esquisse sur l'une des ses faces en sélectionnant la face et en cliquant sur l'icône , on peut rencontrer l'une des éventualités suivantes.
 - Si je n'ai aucun *body* dans mon projet, j'ai le message : « *No ctive Body* »
 - Si j'ai un *body* actif dans mon projet alors FreeCAD va inclure cette esquisse dans la *body* en question en me demandant donc si je vue une copie dépendante, indépendante de la géométrie de cette face, voire en créer une référence croisée (*cross-ref*)
 - Si aucun *body* existant n'est actif, FreeCAD rend actif le premier qui lui tombe sous la main et on est ramené au problème précédent.
- ✓ Pour pouvoir mettre une esquisse sur une face d'un objet résultat d'une opération booléenne, il faut d'abord « l'inclure » dans un *body* en cliquant sur l'icône  de l'atelier *PartDesign*.
 - L'objet résultat booléen devient alors invisible ;
 - le *body* créé contient une *BaseFeature* visible sur laquelle on peut alors travailler.
Pour une explication sur cette histoire de *BaseFeature*, voir [cette page](#) ou [cette autre page](#).

II.8. Paramétrage d'un projet

II.8.a Utilisation d'un fichier de paramètres

Je commence la plupart du temps par créer un fichier tableur (*Spreadsheet*) dans lequel je mets les valeurs numériques des longueurs (et angles) permettant de construire mon modèle, pour le paramétrer facilement. Ça se fait avec l'[atelier Spreadsheet](#), très bien expliqué sur [cette page](#).

Pour pouvoir, utiliser ces dimensions dans le projet, il faut leur avoir attribué un *alias*.

- ✓ Le tuto précédent montre comment faire à l'aide de la fenêtre Propriétés du menu contextuel.
- ✓ Mais je me suis aperçu qu'il y a (dans ma version linux) un champ *alias* en haut à droite dans lequel il suffit de taper la nom de l'alias que l'on veut donner à la cellule sélectionnée. On peut utiliser un copié-collé mais *ne pas oublier de terminer par un retour chariot* ! Toute cellule possédant un alias passe alors en fond jaune.

Dans le cours du projet :

- ✓ on peut utiliser les paramètres de ce tableur en cliquant sur la petite pastille bleue de la case dans laquelle on veut entrer une dimension ;
- ✓ on peut utiliser la seule valeur ou paramètre ou toute formule correcte ;
- ✓ comme le paramètre *<alias>*, s'utilise sous la forme *<nom_spreadsheet>.<alias>*, je donne toujours le nom *p* à mon fichier de paramètres, même si la complétion automatique facilite ensuite bien la saisie.
- ✓ Il est souvent préférable voire indispensable de mettre les dimensions (*mm*, °) après les valeurs numériques, sinon cela peut poser problème lors de certaines opérations.

Attention


- ✓ On peut aussi utiliser un *Spreadsheet* pour y recueillir certaines données d'un projet (cf. toujours sur [cette page](#)).
- ✓ Mais on ne peut pas avoir dans le même *SpreadSheet* des valeurs qui définissent le modèle, et des valeurs extraites du le modèle. Même si elles sont cloisonnées et ne créent pas réellement de dépendances cycliques, *FreeCAD* retournera ce genre d'erreurs.
- ✓






II.8.b Utilisation d'un sketch master

Il y a une technique analogue utilisant un *master sketch* comme [dans cette vidéo](#).

- ✓ Cette esquisse n'a pas besoin d'être correcte comme si elle servait à une extrusion.
- ✓ En donnant des noms aux contraintes utilisées dans cette esquisse, on peut les utiliser dans d'autres sketches ou dans des protusions, exactement comme les valeurs d'une feuille de calcul, dont c'est un peu une version graphique.
- ✓ En cliquant la case *référence*, on peut obtenir la valeur d'une dimension que l'on ne peut plus définir sous peine de sur-contraindre l'esquisse et l'utiliser ailleurs.
- ✓ La syntaxe pour utiliser ces valeurs est *<nom_sketch>.Constraints.<nom_contrainte>*. La complétion auto fonctionne pour les deux premiers niveaux mais pas pour le troisième, ce qui impose de bien connaître les noms de contraintes de l'esquisse à laquelle on se réfère.

II.9. Coupes et sections

- ✓ **Méthode 1** : On peut utiliser l'[atelier TechDraw](#) qui permet de réaliser des dessins cotés
 - Voir [tuto d'introduction](#) et [la page dédiée à TechDraw](#)
 - Pour insérer une section, sélectionner dans la page *TechDraw* la projection concernée et utiliser l'outil [insérer une vue en coupe](#), d'icône . Voir éventuellement [cette vidéo](#) (à 3mn).

Les sections ainsi créées sont liées dynamiquement à l'objet, c'est-à-dire qu'elles suivent les modifications de l'objet.
- ✓ **Méthode 2** : En utilisant <Affichage><Plan de Coupe>, une fenêtre s'ouvre sous la hiérarchie des objets où l'on peut choisir et déplacer un plan qui coupe l'objet.
 Mais les volumes apparaissent alors comme creux ce qui n'est pas toujours l'effet attendu. On ne doit pas pouvoir avoir autre chose comme il est confirmé [dans ce post](#) (chercher « creux »)
- ✓ **Méthode 3** : On peut utiliser la [fonction Coupes \(Cross Section\)](#), d'icône , de l'atelier Part.
 - On peut voir comment dans [ce post](#)
 - Mais en fait, on ne peut appliquer cette fonction qu'à un body (et un seul à la fois) ! Pour les projets contenant plusieurs *bodies* il faut faire autant de coupes de corps. On ne peut même pas l'appliquer à un *Part* qui ne contient qu'un seul *body*.
 - Attention, comme il est dit sur la page de description de fonction, l'objet n'est pas paramétrique : c'est-à-dire
 - > > qu'il n'est pas modifier si on modifie l'objet coupé
 - > > et surtout qu'il n'est pas modifié si on modifie la position du plan de coupe, d'où l'intérêt de pouvoir faire des multi-coupes !
- ✓ **Méthode 4** : On peut utiliser les [outils de division de formes](#). Voici une [super vidéo](#) montrant comment utiliser essentiellement la fonction *Trancher (Slice Apart)*  de l'atelier Part.
 - À partir de 1 mn 10, on voit comment récupérer un objet en format STL et le convertir par plusieurs opérations en un vrai solide 3D (un peu hors sujet pour ce qui nous intéresse ici)
 - À partir de 5 mn 50, on voit comment couper un objet avec une plaque (ayant une épaisseur) en utilisant la fonction [<Slice apart>](#)  de l'atelier Part.
 - À partir de 9 mn 20, on voit comment faire de même mais avec un plan (sans épaisseur) : ce plan étant obtenu avec la fonction [Part Primitives](#), d'icône .
 - À partir de 13 mn 40, on voit comment plus généralement faire la même chose, mais avec une feuille sans épaisseur, qui est obtenue à partir d'un fil par l'[extrusion d'icône](#) .
 - À partir de 18mn, on parle des [autres outils de division de formes](#).

De telles coupes sont dynamiques et évoluent lors l'on modifie l'objet définissant la coupe.
 En revanche, j'ai l'impression qu'elles ne s'appliquent qu'à un *Body* et pas à un *Part* !


Voir éventuellement [cet exemple](#) et [cet autre](#)

III. Import-Export

III.1. Utiliser FreeCAD pour la découpeuse laser


III.1.a Sites et vidéos

Une [vidéo intéressante](#) parlant de *.dxf* aussi bien en import qu'en export.



- ✓ On y voit aussi l'utilisation de libreCAD pour ouvrir des *.dxf* que FREECAD ne sait pas ouvrir.
- ✓ Il y a aussi l'utilisation de la fonction [Draft vers Esquisse](#) (icône ) de l'[atelier Draft](#) : cette fonction permet de convertir un dessin au trait en esquisse (et réciproquement).

III.1.b Deux méthodes que j'ai utilisées


Méthode 1 : c'est [sur cette page](#) que j'ai trouvé une première méthode qui marche pour générer un *.dxf* à l'aide de *FreeCAD* (la méthode 2, de la page car je n'ai pas compris la méthode 1).

- ✓ Je me suis mis dans l'atelier *Draft*.
- ✓ Pour chaque morceau à découper :
 - je l'ai affiché frontalement et j'ai rendu tous les autres invisibles ;
 - avec l'icône , j'en ai effectué une projection sur le plan XY ;
 - il a été alors créé un objet de nom *Shape2DView...* que l'on peut voir dans le plan XY.
- ✓ Dans le plan XY, j'ai pu arranger entre eux ces *Shape2DView* en jouant avec leur *placement*.
- ✓ Après avoir sélectionné tous ces objets *Shape2DView...*, j'ai fait <fichier><export> au format <Autodesk DXF 2D (*.dxf)>.
- ✓ Après avoir ouvert ce fichier avec *Inkscape*, j'ai pu mettre différentes couleurs pour préciser l'ordre des découpes.
- ✓ Il reste à vérifier si ce fichier est correct pour la découpeuse ??????



Méthode 2 : J'ai aussi trouvé un autre méthode avec *TechDraw*


- ✓ Je me suis mis dans l'atelier *TechDraw*
- ✓ Puis j'ai inséré une page (de dessin technique)
 - soit avec  <TechDraw> <Insérer une nouvelle page par défaut> ,
 - soit avec  <TechDraw> <Insérer une nouvelle page à partir d'un modèle>, ce qui permet de choisir la taille de la page ainsi que l'existence ou non d'un cartouche.

Il apparaît un objet <Page> dans la hiérarchie ; c'est là que ça se passe ensuite.

- ✓ Ensuite, pour chaque morceau à découper, je l'ai affiché en vue frontale et j'ai pu alors insérer le dessin correspondant dans la page précédente en utilisant  <TechDraw><Nouvelle vue>.

Remarque : il ne paraît pas indispensable de rendre les autres éléments invisibles.

- ✓ On peut alors :
 - déplacer les différents dessins en utilisant la poignée qui est en dessous <View....>
 - Remarque : 'il n'y pas de poignée <clic-droit><Activer/désactiver les cadres> ;
 - insérer des côtes avec les icônes jaunes :  , etc.

- ✓ Pour finir j'ai utilisé  <TechDraw> <Exporter une Page au format DXF>.

Mais on peut aussi utiliser <Fichier><Exporter> qui permet un choix ???

??? Voir quelle est la meilleure méthode et si les fichiers obtenus sont OK ???

III.2. Impression 3D (???)

<https://www.sculpteo.com/fr/tutoriel/preparer-votre-fichier-pour-limpression-3D-avec-freeCAD/>

reprendre pour MESH et STL ?????

J'ai déjà signalé le [vidéo suivante](#) pour son utilisation d'un *master sketch*, mais à la fin (???) elle raconte des choses intéressantes sur la façon d'imprimer son objet.

III.3. FreeCAD et la CNC (???)

- ✓ Pour commencer, la [documentation officielle de l'atelier Path](#) et un [exemple expliqué](#).
- ✓ Quelques [pages parlant de CAM et de G-code](#) (mais en anglais) et [ça commence ici](#).
- ✓

<https://www.youtube.com/watch?v=vyHwJ19PJp4> (CNC)

Une [vidéo intéressante](#) (en anglais) travaillant sur [la pièce faite sur cette vidéo](#)

Un [post intéressant en français](#)

Voici quelques lien vers des infos intéressantes pour l'utilisation de l'atelier Path

Wiki de l'atelier Path d'où est tiré l'image expliquant les différentes hauteurs et profondeurs a modifié dans les parcours d'outil https://wiki.freecadweb.org/Path_Workbench/fr

L'excellent site d'un adhérent du Fablab qui explique l'usinage et en particulier le sens d'usinage en fonction du sens de rotation de l'outil et du déplacement de l'outil. <https://www.makerslide-machines.xyz/fr/2018/08/13/fr-usinage-en-opposition-en-avalant-ou-mixte/>

Autre site expliquant le sens d'usinage
<https://www.sandvik.coromant.com/fr-fr/knowledge/milling/pages/up-milling-vs-down-milling.aspx>
<http://outillage.metiers-et-passions.com/documents/usinage-de-aluminium/168.html>

Vois les règles générales qu'on m'a indiqué concernant le sens d'usinage :

Pour le bois : Opposition ou climb en anglais (Fortement conseillé a la défonceuse)

Pour les métaux : Avalant ou conventionnel

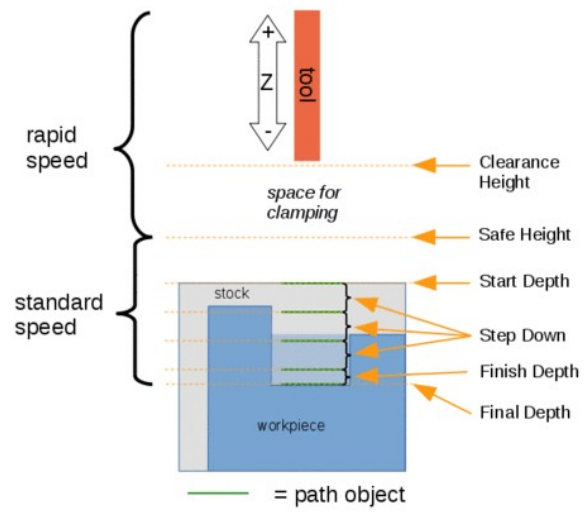
Application pour le réglage des différents paramétrage d'usinage :

Sur Android et Ios (en anglais) gratuite avec pub : FSWizard

Sur Windows (Angalis Français) gratuit 1 mois : HsmAdvisor

Commandes Path

De nombreuses commandes ont différentes hauteurs et profondeurs :



Référence visuelle pour les propriétés de profondeur (paramètres)

Comment [convertir stl en step](#)

IV. FreeCAD et Python

IV.1. Introduction

Python permet de faire beaucoup de choses dans *FreeCAD* :

- ✓ faire de l'animation (voir la partie correspondante???) ,
- ✓ créer des objets ou modifier des objets existants (leur forme ou leur représentation),
- ✓ modifier l'interface de *FreeCAD*, etc.

IV.1.a Pour commencer

- ✓ Si vous débutez en *Python* ou pour toute question, voici un [cours sur Python](#) avec un bon sommaire, et [un autre](#) qui paraît pas mal fait. Il y a aussi le [tutoriel du site Python.org](#).
- ✓ Ce que propose *FreeCAD* pour écrire du *Python* n'est vraiment pas à la hauteur et il vaut mieux utiliser un bon IDE (Enregistrement de Développement Intégré). Sur [cette page](#), on en trouve une bonne revue. [Thonny](#) est certainement l'un des plus simples pour commencer.
- ✓ La page [Débuter avec les scripts FreeCAD](#) me paraît essentielle pour bien démarrer :
 - elle explique comment utiliser le code *Python* dans *FreeCAD* ;
 - si vous ne voyez pas la console *Python*, c'est qu'elle n'est peut-être pas ouverte ; dans ce cas, vous pouvez l'ouvrir en cochant *Console Python* dans le menu <Affichage><Panneaux>.
 - la partie *Modules intégrés* contient une description précise des fonctionnalités des modules *App* et *Gui* que l'on utilise quasiment toujours quand on programme un script ;
 - mais la partie *Utilisation des modules supplémentaires* n'est indispensable immédiatement.

IV.1.b Utiliser la complétion dans le console *Python*

La documentation *Python* concernant *FreeCAD* n'est pas toujours évidente à trouver et j'en ai beaucoup appris par auto-complétion dans la console *Python*. Pour comprendre cela, imaginons avoir ouvert un fichier *FreeCAD* contenant un cube *Box* créé avec l'atelier *Part* :

- ✓ si on tape **App.** (avec le point, et en respectant la casse) dans la console *Python*, alors une fenêtre s'ouvre et montre tout ce que l'on peut entrer ;
- ✓ en prenant le premier choix (la touche <Entrée> suffit), on obtient la commande **App.ActiveDocument**
- ✓ qui retourne une référence sur le document courant, quelque chose du genre :
<Document object at 0x560e73a5eb20>
- ✓ ce n'est pas son nom, qui est retourné par :
App.ActiveDocument.Name
que l'on peut obtenir par complétion de **App.ActiveDocument.** (avec le point) ;
- ✓ Si l'on veut accéder à la *Box* créée, précédemment il suffit de retaper **App.ActiveDocument.**
et de la compléter en **App.ActiveDocument.Box**
ce qui retourne une référence à cette boîte du type :
<Part::PartFeature>
- ✓ On peut alors afficher la hauteur de cette boîte en entrant (toujours par complétion) :
App.ActiveDocument.Box.Height

- ✓ On peut même modifier cette hauteur en tapant par exemple :

`App.ActiveDocument.Box.Height=50`

C'est ainsi, en partant

- ✓ soit de la racine App pour tout ce qui concerne les données de construction des objets
 - ✓ soit de la racine Gui pour tout ce qui concerne la partie visualisation de ces objets
- que l'on peut explorer les fonctions offertes par FreeCAD.

IV.1.c Pour aller vraiment plus loin

- ✓ Dans ce [post de forum](#) , il est dit que :
 - le module *Part* est essentiel car c'est lui qui construit les objets *OpenCascade* pour *FreeCAD* ;
 - comme un objet *FreeCAD* (*Python*) est construit en C++ puis passé en *Python*, on peut en trouver sa description sous deux formes différentes :
 - > le fichier `xxxPyImp.cpp` qui a permis de le définir,
 - > un fichier `xxxPy.xml` certainement obtenu par documentation automatique ;
 Mais ce n'est pas forcément très évident à lire.
- ✓ La [page de références](#) de la programmation de *FreeCAD* en *Python*, et [plein d'exemples](#) dont celui expliquant [comment écrire une macro](#) traçant un rectangle et en faire une commande.
- ✓ Plein [d'autres exemples de macros](#)
- ✓ Pour aller encore plus loin, la [page de référence](#) de la programmation Python pour *FreeCAD*.
- ✓ [Exemples de snippets \(extraits de code\)](#) , dont la liste des objets, de leurs propriétés, etc.
- ✓ Un [article très intéressant](#) montrant comment créer une nouvelle commande et l'intégrer au menu via un atelier : cette commande, qui réagit à la souris, permet de construire un segment.
- ✓ On peut aussi penser au [FreeCAD_Mod_Dev_Guide](#) qui se veut une référence.

IV.1.d Et le Debug (quand vous écrirez de gros programmes) ?

Comme il est expliqué dans [la vidéo de cette page](#) il y a deux moyens d'utiliser un éditeur externe pour éditer et corriger les macros *FreeCAD*

- ✓ Au début de la vidéo, l'auteur montre comment on peut ouvrir *FreeCAD* à partir d'un programme *Python*, mais cela suppose de disposer des bibliothèques *FreeCAD*, ce qui n'est pas le cas avec la version App-Image que j'utilise.
- ✓ À partir de 8mn, il montre comment « intercepter » le fonctionnement d'une macro lancée à partir de *FreeCAD* pour obtenir des infos de debug.
 - J'ai commencé par installer *ptsvd* en tapant dans un terminal :


```
pip3 install ptvsd
```

 car chez moi, **pip** correspond à *Python* 2
 - J'ai alors eu un problème car *FreeCAD* ne trouvait pas le package *ptvsd* ; il bloquait sur :


```
import site
```
 - je pense que c'est parce qu'il n'allait pas le chercher au bon endroit. En m'inspirant de ce que j'ai trouvé sur [ce site dédié à Python](#), j'ai donc ajouté :


```
import site  
site.addsitedir("/home/jm/.local/lib/python3.7/site-packages/")  
import ptvsd
```

 et tout est rentré dans l'ordre.

Voir dans le menu <Macro><Attacher au débogueur distant> ????

IV.1.e Python et C++

<https://sametmax.com/appeler-du-code-c-depuis-python-avec-ctypes/>

<https://koor.fr/Python/CodeSamples/NativeSample.wp>

<https://cpp.developpez.com/tutoriels/interfacer-cpp-python/>

Il y a aussi un cours sur Qt/PySide2 <https://koor.fr/Python/SupportPythonQt/slide1.wp>

Un [exemple intéressant](#)

La fonction ***PyArg_ParseTuple***

De [cette page](#) (concernant malheureusement *Python 2*) on trouve une bonne explication du principe permettant d'étendre Python à C++. En ce particulier pour la fonction ***PyArg_ParseTuple***.

- ✓ La récupération des arguments est réalisée par la fonction ***PyArg_ParseTuple()*** :
 - son premier argument est le tuple contenant les arguments ;
 - son deuxième est une chaîne de format ;
 - suivent un certain nombre (dépendant de la chaîne de format) de pointeurs vers les données à remplir avec ce qui est dans le premier argument.
- ✓ Le retour de cette fonction est différent de 0 si tout s'est bien passé.
- ✓ Quelques [exemples avec Python3](#)
- ✓ Une [bonne description des options de format](#) dont « d » et « O » ou « O ! » (la lettre O)
- ✓ Sur [cette page](#), une bonne comparaison, avec ***scanf***

IV.2. Création d'objets dans FreeCAD

Il sera certainement plus profitable de lire cette partie (plutôt théorique) en parallèle avec la suivante (plutôt pratique) qui expose les différentes étapes qui m'ont permis de créer un objet de type *FeaturePython*, et qui est certainement plus aboutie.

Remarque Avant de penser à utiliser des objets scriptés, on peut aussi regarder du côté des [DynamicData objects](#) comme le prescrit [suzanne.soy dans ce post de forum](#).

IV.2.a Manipulations géométriques

Avec *FreeCad*, on peut construire des objets scriptés, c'est-à-dire définis par programme, et donc à l'aide de macros. Comme ces macros ne sont pas, pour des raisons de sécurité, incluses dans les fichiers *FreeCAD*, il faut penser à envoyer leurs fichiers *.FCMacro* ou *.py* en même temps que les fichiers *.FCStd* des projets contenant de tels objets.

Dans [le livre de Yorik](#), on trouve une [page de manipulation de géométrie](#).

- ✓ On y parle de la différence entre un objet *FreeCAD* et sa représentation.
- ✓ On y explique comment construire une *Shape* (ou *TopoShape*) en partant des points.
 - Au début est le Sommet (*Vertex* plur. *Vertices*)
 - Avec deux Sommets, on construit un Côté (*Edge*)
 - Avec au moins un Côté, on peut former un Fil (*Wire*)
 - Avec un Fil fermé, on peut créer une Face (*Face*)
(un fil contenu dans un plan??? je ne sais pas)
Pour obtenir une face trouée, il suffit de disposer de plusieurs Fils : les « Fils » internes deviennent des trous.
 - Avec une ou plusieurs Faces, on peut créer une Coquille (*Shell*)
 - À partir d'une coquille étanche (*watertight*), on peut créer un Solide (*Solid*)
 - En joignant plusieurs faces (de types quelconques) on obtient un Composé (*Compound*)
- ✓ Et enfin on y montre comment utiliser cette *Shape* pour introduire un nouvel objet, de type *Part::Feature*, dans le projet en cours, soit avec *addObject* soit avec *show*.

La [page précédente](#) branche [aussi sur cette page](#) qui reprend tout en détail et qui contient beaucoup d'informations intéressantes. On y trouve en particulier ces lignes :

```
cylinder = Part.makeCylinder(3,10,Base.Vector(0,0,0), Base.Vector(1,0,0))
sphere = Part.makeSphere(5,Base.Vector(5,0,0))
diff = cylinder.cut(sphere)
```

permettant de faire une soustraction booléenne. Remarquer que cette méthode *cut* marche avec les objets *Part* mais pas avec les objets *PartDesign*. Pour lesquels il faut passer par leur *Shape*.

Pour finir, une très bonne [description des objets PartFeature](#) qu'il est bon de lire et de relire !

Remarque On peut ajouter des courbes à un sketch en utilisant un script Python comme [dans cet exemple](#) qui ajoute un arc d'ellipse.

IV.2.b Utilisation de classes

Mais comme *Python* est orienté objet, il est alors quasi indispensable de regarder du côté des classes.

- ✓ Pour commencer directement dans *FreeCAD*, voir [cette page intéressante](#) (ou la [retrouve ici](#)).
 - On y explique simplement comment créer une classe fournissant un rectangle paramétré par ses deux longueurs, que l'on peut modifier dans la fenêtre « Propriété ».
 - Pour pouvoir ainsi définir par une classe un objet destiné à avoir une représentation 3D, il est faut qu'il possède une propriété *Proxy* et qu'il soit donc de type *Part::FeaturePython*, et non pas de type *Part::Feature* comme les objets manipulés sur la [page de manipulation de géométrie](#) ; voir la partie « *fixing code* » [de cette page](#), et éventuellement [cette page](#).
 - On y voit aussi comment créer une icône pour accéder à cette fonction de création d'objet.
- ✓ Un autre exemple, issu du wiki ([Part. 1](#) et [Part. 2](#)), d'utilisation des classes pour créer des objets scriptés auxquels on associe des propriétés et des méthodes.

Pour plus d'information sur la programmation orientée objet en *Python*, on peut regarder :

- ✓ cette [explication rapide](#) ou [cette page](#) qui me paraît très bien expliquée ;
- ✓ un [cours un peu plus fouillé](#), et qui ressemble à [celui-là](#).
- ✓ une [bonne explication de la méthode super\(\)](#) et [encore là](#) ; voir en particulier la différence entre *Python 2* et *Python 3*)

Mais il y en a aussi [une autre en anglais](#) sur le site précédent (extraite de [ce tuto](#)).

Bonne documentation des classes utilisées par *FreeCAD* et de leur héritage sur le site <http://free-cad.sourceforge.net> avec plusieurs possibilités d'accès :

- ✓ par [modules](#) avec en particulier [App](#) et [Gui](#)
- ✓ par [namespaces](#) ou par [classes](#)

Attention Lorsque j'ai programmé en utilisant des classes, il y avait une différence entre faire du copier/coller depuis un éditeur de texte quelconque et utiliser *import*.

- ✓ Lorsque lors de la mise au point, je faisais du copier/coller de programmes depuis mon éditeur *Python* (*Geany*) vers la console *Python*, tout se passait bien mais il y avait des erreurs lorsque je rechargeais un fichier contenant un objet de l'une de mes classes (même avec un fichier *Python* bien installé dans le répertoire *Mod*). **??? REVOIR CELA ????**
- ✓ Cela ne se produisait pas lorsque j'utilisais les fonctions et/ou définitions de classes en les définissant avec *import*. Mais prendre garde dans ce cas que les fonctions ne sont pas mises à jour après modification et sauvegarde dans l'éditeur externe.
 - La première solution que j'avais trouvée pour être sûr d'avoir les dernières modifications était de quitter *FreeCAD* et le relancer.
 - Mais comme expliqué dans [cette page](#) ou [dans celle-là](#) (où l'on parle aussi du répertoire `__pycache__`), le plus simple est d'utiliser :


```
import importlib
importlib.reload(<module>)
```

Pour voir une fonction importée, penser à la fonction *inspect.getsource*, sauf bien évidemment pour les fonction built-in

IV.2.c Le Proxy

Lorsque j'ai commencé à utiliser les classes, je suis tombé sur des histoires de *Proxy*, ce qui est longtemps resté obscur pour moi !

- ✓ C'est dans [cette page de Yorik](#) que j'en ai trouvé la première définition « *store our class ...* » : j'en ai compris que les classe servant à définir l'objet était stocké dans la propriété **Proxy** de l'objet que l'on a inséré dans le document *FreeCAD*.
- ✓ Mais c'est dans [ce post \(2011\)](#), et [cet autre post \(2010\)](#) de [wmayer](#) (*Just a few words about how our document framework is working...*) que j'ai trouvé l'explication la plus précise.
 - Un document *FreeCAD* a accès à la liste des objets qu'il contient, mais ils ne savent pas de quel type d'objet il s'agit, ni ce qu'ils font en réalité.
 - Une fois que le document *FreeCAD* est chargé, ou à chaque fois qu'il est mis à jour, ou recalculé, le système appelle la méthode `execute()` de chaque objets qui a été modifié, « *touched* », et des objets qui en dépendent.
 - Comme *FreeCAD* et donc tout le système a été écrit en C++, sans la moindre intervention de *Python*, pour réaliser un lien entre C++ et le *Python* qui permet à l'utilisateur de créer de nouveaux objets, il y a la classe *FeaturePython*, qui est dérivée de la classe *DocumentObject*.
 - Chaque objet de cette classe *FeaturePython* contient via son *Proxy* un lien vers un objet en pur *Python*, lien que l'on définit dans la fonction `__init__()` de la classe ; ce lien permet de déléguer tous les évènements/notifications qui viennent du *framework* C++ aux bonnes méthodes *Python* définies dans l'objet. C'est toute la magie de la chose.

IV.2.d Premières descriptions de classes

C'est sur [cette page de Yorik](#) que j'ai trouvé la meilleure introduction sur la façon de créer des classes pour construire des objets *FreeCAD*.

- ✓ Comme dans toute classe *Python*, il doit y avoir une fonction (méthode) `__init__`, qui est appelée à chaque création d'un objet de cette classe.
 - C'est dans cette fonction `__init__` que l'on trouve une instruction du type
`obj.Proxy = self`
qui relie le *Proxy* de l'objet *FreeCAD* (**obj**) à l'objet (**self**) *Python* que l'on crée .
 - C'est aussi dans cette fonction `__init__` que l'on peut ajouter à l'objet *FreeCAD* créé des propriétés qui seront (ou non) affichées dans la fenêtre *Propriété* de l'objet (cf. ci-dessous).
- ✓ Dans la définition de la classe, on doit aussi trouver une fonction (méthode) `execute`,
 - Cette fonction est appelée à chaque mise à jour du document, `doc.recompute()`, lorsqu'un objet *FreeCAD* de cette classe est marqué (*touched*) comme devant être recalculé, c'est-à-dire lorsque l'un de ses propriétés a été modifiée ou qu'il dépend d'un objet qui doit être recalculé.
 - C'est dans cette fonction que se trouve en général le code *Python* permettant de construire l'objet *FreeCAD*, la plupart du temps en construisant une *Shape*, que l'on affecte ensuite à l'objet par une instruction du type :
`obj.Shape = myNewShape`.
C'est pourquoi cette méthode `execute` possède un argument **obj** qui est l'objet *FreeCAD* que l'on veut modifier dans le code *Python*.
- ✓ On peut voir en voir une mise en œuvre dans l'[exemple de cette page](#).

- ✓ Ce qui précède permet de définir une classe avec laquelle on peut, comme dans tout langage orienté objet, créer un objet de cette classe puis l'insérer dans le fichier *FreeCAD* actif. Mais comme, avec *FreeCAD*, il y a une séparation stricte entre,
 - d'une part, la partie objets qui est essentiellement gérée par le module **App**
 - d'autre part, la partie représentation 3D qui est gérée par le module **Gui**,
 on ne verra en général aucune représentation de l'objet créé sur la fenêtre 3D (*main view area*). Pour avoir une vue 3D d'un objet de type *FeaturePython*, le plus simple est d'affecter :


```
obj.ViewObject.Proxy = 0
```

 - Pour avoir une représentation personnalisée de l'objet, il est aussi possible de créer son propre *ViewProvider*, mais c'est bien moins facile à gérer (voir ci-dessous).
 - J'imagine que l'affectation précédente (où l'on peut remplacer **0** par n'importe quoi sauf **None**) oblige *FreeCAD* à utiliser le *ViewProvider* commun à tous les objets de type *Part*.
 - Même si ce n'est pas très propre et que cela rompt la séparation voulue par *FreeCAD* entre la partie *App* et la partie *Gui*, cette affectation peut très bien se faire dans la fonction `__init__`.
- ✓ Enfin, ne pas oublier de faire un **App.ActiveDocument.recompute()**, pour forcer le rafraîchissement de la vue.

Principales méthodes que l'on peut surcharger pour des objets de type *FeaturePython* :

- ✓ elles sont décrites sur [cette page du wiki](#) ou aussi [sur cette page](#), avec en particulier la méthode **onChanged**, appelée dès qu'une des propriétés de l'objet est modifiée (cf. ci-dessous).
- ✓ Dans [cet échange](#), on parle des méthodes utilisables avec un objet de type *FeaturePython* qui sont documentées : **execute**, **onBeforeChange**, **onChanged**, **onDocumentRestored**.

IV.2.e Propriétés d'un objet de type *FeaturePython*

Pour un objet de type *FeaturePython*, on peut définir des propriétés, stockées avec l'objet, et dont chacune apparaît alors dans la fenêtre « Propriété » de *FreeCAD* où l'on peut la modifier.

- ✓ Sur [cette page](#) se trouve la liste de toutes les propriétés d'un objet *FeaturePython*, avec en plus une description intéressante de l'utilisation de la méthode **addProperty()**.
- ✓ Sur [cette page](#) et plus précisément [à cet endroit](#), on trouve un exemple d'utilisation
 - de propriétés du type longueur **App::PropertyLength**,
 - d'une propriété de type chaîne **App::PropertyString**.
 On peut aussi citer **App::PropertyFloat** et **App::PropertyInteger**.
- ✓ Voir éventuellement [cette page](#) qui décrit l'éditeur de propriétés.

Sur [cette page de wiki](#), on parle des types des propriétés :

- ✓ Sont-elle modifiables, en lecture seule ou cachées dans l'éditeur ?
- ✓ On peut les définir comme « transient » et donc non sauveées dans le fichier enregistré ?

C'est la méthode **onChanged** qui est appelée à chaque fois que l'une de ces propriétés de l'objet *FreeCAD* est modifiée. Voir éventuellement cette discussion sur [onChanged vs. execute](#)

Dans [cet échange du forum](#) j'ai rencontré des propriétés du type *PropertyPythonObject* :

- ✓ d'après [la documentation](#), une telle propriété est destinée à stocker un objet Python (variable, structure, fonction) ; il faudrait que je relise la [la page précédente](#) en détail (???) ;
- ✓ pour une telle propriété, il ne paraît pas possible de la voir dans la fenêtre « Propriété » ;
- ✓ d'après [cet échange du forum](#), cela a à voir avec `__getstate__()` et `__setstate__()`.

Ce n'est pas encore très clair pour moi ???

Comment faire pour que la représentation 3D réagisse immédiatement à la modification par les flèches haut/bas d'une propriété de type dimension ?

IV.2.f Installation, création de commandes

- ✓ Une fois que l'on a créé des classes, il faut *installer* les fichiers correspondants. Sinon quand, après avoir lancé *FreeCAD*, on ouvre un fichier contenant un objet de type *FeaturePython* non standard, on obtient un message d'erreur du genre :

```
<class 'AttributeError': Module __main__ has no class xxxxxx
```

- ✓ Je me suis inspiré de ce que j'ai compris dans [cet échange du forum](#), et je place en général le fichier définissant les classes dans le sous-répertoire *Mod/fpo* du répertoire que *FreeCAD* explore au démarrage (sous linux, c'est *.FreeCAD*, voir [cette page](#) pour les autres systèmes).

Attention Si le fichier contenant le code provient de l'éditeur de macros de *FreeCAD*, il faut en modifier l'extension, de *.FCMacro* à *.py*, sinon Python ne pourra pas le trouver.

- ✓ Il faut aussi placer un fichier de nom *__init__.py* dans le répertoire *Mod/fpo* pour indiquer à Python que ce répertoire contient des modules qu'il pourra éventuellement utiliser.

Pour plus de précision, voir ci-dessous dans l'exemple des cycloïdes.

Remarque Dans [cette page](#) il est dit que cet fichier *__init__.py* ne serait plus nécessaire à partir de la version 3.3 de Python ?

Or celle que l'on utilise avec *FreeCAD* est 3.8.6 comme on peut le vérifier avec :

```
import sys
sys.version
```

Améliorations possibles

- ✓ On peut alors en faire facilement une commande comme il est expliqué dans la dernière partie de [cette page](#) (à partir de «*Bien sûr, il serait fastidieux* »).
- ✓ On peut aussi lui associer une fenêtre pour définir dès la construction les caractéristiques de la courbe attendue : voir dans la partie concernant *Qt*.
- ✓ On peut créer un nouvelle atelier contenant cette fonction comme [à la fin de cette page](#).

IV.3. Exemple de construction d'objet : épi(hypo)cycloïde

À la suite d'une question d'un membre du FABLAB, j'ai essayé de construire des plaques dont le contour externe est une [épicycloïde](#) ou une [hypocycloïde](#).

Dans tous les scripts qui suivront, j'utiliserai la fonction suivante :

```
def cyclo_p(R, k, t):
    x = R*(1+1/k)*math.cos(t)-R/k*math.cos((k+1)*t)
    y = R*(1+1/k)*math.sin(t)-R/k*math.sin((k+1)*t)
    return Base.Vector([x, y, 0])
```

dont les paramètres d'entrée sont :

- ✓ **R** qui est le rayon du cercle de base (sur lequel roule le petit cercle)
- ✓ **k** dont la valeur absolue est le rapport des rayons des cercles ;
si $k > 0$ (resp. $k < 0$), c'est une épicycloïde (resp. hypocycloïde)
- ✓ **t** qui mesure en radians l'angle entre Ox et Ol , avec l le point de contact des deux cercles.

Elle retourne les trois coordonnées du point courant de la cycloïde. Bien qu'il suffise de deux coordonnées pour un point d'un plan, il est préférable pour la suite que le résultat soit sous la forme d'un **Base.Vector**, qui est un type souvent utilisé par FreeCAD.

Remarque Il faut évidemment la faire précéder cette définition des deux instructions :

```
import math
from FreeCAD import Base
```

IV.3.a Extrusion à partir d'un script

Dans un premier temps, j'ai construit le profil formé par la courbe dans un sketch en utilisant la macro **Rosace_Epi_Hypo_Cycloïde_A01.FCMacro**. Avant de lancer la fonction **courbe()** qu'elle définit, ouvrir un nouveau fichier FreeCAD et y créer une esquisse, de nom *Sketch*.

Explications

- ✓ On commence par échantillonner le segment $[0, 2\pi]$

```
lst_t = [2*math.pi*t/Nb_Pnt for t in range(Nb_Pnt)]
```
- ✓ À l'aide de la liste précédente, on crée la liste des **Base.Vector** définissant les sommets

```
lst_pnt_geo = [cyclo_p(Rayon, Rapport, t) for t in lst_t]
```
- ✓ Il faut alors la fermer avec le premier point (sinon, problème pour extruder ensuite)

```
lst_pnt_geo.append(lst_pnt_geo[0])
```
- ✓ La liste précédente permet d'ajouter des segments à la géométrie de l'esquisse, et il est nécessaire de mettre des contraintes de coïncidence pour assurer la fermeture du polygone.

```
while(i<Nb_Pnt):
    sk.addGeometry(Part.LineSegment(lst_pnt_geo[i], lst_pnt_geo[i+1]), False)
    if(i>0):
        sk.addConstraint(Sketcher.Constraint('Coincident', i-1, 2, i, 1))
    i=i+1
```
- ✓ Enfin on termine avec un **doc.recompute()**.

Après avoir exécuté **courbe()**, on peut extruder le sketch, après l'avoir fermé s'il était ouvert.

Remarques

- ✓ Dans un premier temps, j'ai découvert les fonctions `addGeometry` et `addConstraint` en regardant ce qui s'écrit dans la console Python lorsque je travaille avec l'interface graphique de FreeCAD : c'est souvent pour moi la première source de documentation.
- ✓ Mais ensuite j'ai trouvé cette [page sur les contraintes de sketch en Python](#) et [ce post du forum](#).
- ✓ Le schéma précédent fonctionne très bien si l'esquisse est vide lorsqu'on l'utilise. Mais si l'esquisse contient déjà des éléments, il faut utiliser `sktch.GeometryCount` qui donne le nombre d'éléments déjà présent (et que j'ai aussi trouvé en utilisant la complétion)

IV.3.b Construction directe du corps

Ensuite, j'ai avec la macro `Rosace_Epi_Hypo_Cycloide_B01.FCMacro` construit directement le corps sans passer par un sketch. L'essentiel se trouve dans le bloc :

```
def rosace_epi_hypo(Rayon=10,Rapport=3,Epaiss=3,Nb_Pnt=50):
```

- ✓ Après avoir commencé comme dans la macro précédente construit la liste `lst_pnt_geo`, on la transforme d'abord en objet `Part.wire`, qui est un contour fermé filiforme.

```
lst_pnt_wire = Part.makePolygon(lst_pnt_geo)
```

- ✓ On crée alors la face pleine correspondant à l'intérieur de ce contour.

```
lst_pnt_face = Part.Face(lst_pnt_wire)
```

- ✓ Et une extrusion permet d'en obtenir un objet de type `Shape`

```
lst_pnt_shape = lst_pnt_face.extrude(Base.Vector(0,0,Epaiss))
```

- ✓ On peut alors ajouter au document un objet de type `Part::Feature`, et mettre dans sa propriété `Shape` ce que l'on vient de calculer.

```
feat = doc.addObject("Part::Feature", "Rosace_Feature»)
```

```
feat.Shape = lst_pnt_shape
```

- ✓ Mais cet objet `feat` est de type `Part` et pas de type `PartDesign`, ce qui empêche de travailler dessus, ne serait-ce que pour y faire un trou au milieu. Pour avoir un `Body`, il faut :

- en ajouter un au document :

```
ros = doc.addObject("PartDesign::Body", "Rosace_Body")
```

- puis faire pointer sa propriété `BaseFeature` sur l'objet la `feature` précédente :

```
ros.BaseFeature = feat
```

- On peut aussi en profiter pour éteindre cette dernière.

```
feat.ViewObject.Visibility = False
```

Comme toujours en période de test, j'ai commencé par copier cette macro, dans la console *Python*.

- ✓ On peut alors l'utiliser avec les valeurs par défaut en tapant :

```
rosace_epi_hypo()
```

- ✓ Mais si on veut par exemple une hypocycloïde à 5 rebroussements, il suffit de taper :

```
rosace_epi_hypo(Rapport=-5)
```

La démarche précédente repose essentiellement sur une programmation procédurale même si elle s'applique sur des objets FreeCAD. Mais il est plus intéressant (et un peu moins évident) de créer des classes et d'utiliser une programmation objet. C'est le but de ce qui suit.

IV.3.c Première utilisation de classe

En m'inspirant essentiellement de [cette page](#) (et de [sa suite](#)) et en utilisant ce qui précède, j'ai dans la macro `Rosace_Epi_Hypo_Cycloïde_C01.FCMacro`, créé la classe `Cycloid` ; elle contient essentiellement deux fonctions :

- ✓ le constructeur appelé à chaque création d'objet de cette classe :

```
__init__(self, obj)
```

- Le paramètre `obj` désigne un objet du fichier *FreeCAD* préalablement ajouté au fichier par une instruction du type :

```
obj = App.ActiveDocument.addObject('Part::FeaturePython', "Cycloid")
```

- La première instruction fait pointer le `Proxy` de l'objet *FreeCAD* vers l'objet *Python* :

```
obj.Proxy = self
```

- Ensuite, sont ajoutées des propriétés qui seront affichées dans la fenêtre *Propriété/Données* et que l'utilisateur pourra modifier comme il modifie les dimension d'une *Box* de l'atelier *Part*.

Par exemple en ce qui concerne le rayon du cercle de base :

```
obj.addProperty('App::PropertyFloat', 'Rayon', 'Dimensions',
                'Rayon du cercle de base').Rayon = 10.0
```

Pour plus d'information sur les propriétés, voir dans la section précédente (??? ref).

- ✓ la fonction spéciale `execute(self, obj)`

- C'est cette fonction qui fournit l'ensemble des instructions de réalisation effective de notre objet : on y retrouve les instructions vues dans la partie « Construction directe du corps ».
- Elle construit d'abord un *Shape* puis se termine par l'instruction qui l'affecte à `obj.Shape` :
`obj.Shape = lst_pnt_shape`
- Cette fonction est exécutée à chaque `doc.recompute()` ; c'est elle qui permet la mise à jour de la forme à chaque modification des données.
- On pourrait ajouter une méthode `onChanged(self.obj, prop)`, qui serait alors appelée à chaque modification d'une propriété, même avec les flèches haut/bas ; le nom de la propriété modifiée étant contenu dans la variable `prop`, on peut tester ce qui a été modifié.

Ce qui précède permet de créer un objet et de l'insérer dans le fichier *FreeCAD*. Mais cela n'est pas encore suffisant pour en avoir une représentation écran. Pour cela, l'idéal serait de définir une classe `ViewProviderCycloid` définissant comment l'objet est affiché.

- ✓ Toutefois pour un objet de type *FeaturePython*, cette classe `ViewProviderCycloid` n'est pas indispensable dans un premier temps, et l'on peut se contenter de l'affectation :

```
obj.ViewObject.Proxy = 0
```

- ✓ Dans une seconde réalisation de cette macro, j'ai utilisé un `ViewProvider` qui m'a uniquement servi à définir une icône particulière pour les objets de contour cycloïdal que je venais de définir. C'est la méthode `getIcon` qui s'en charge.
 - J'ai l'ai définie en *XPM*, mais ça a l'aire d'être un vieux truc !
 - On peut aussi utiliser un fichier `.svg` ou `.png` en faisant retourner son nom par `getIcon`.

À chaque exécution de la fonction `create()`, qui permet d'empaqueter tout cela, on construit facilement une forme attendue, et on peut en modifier les caractéristiques avec la fenêtre *Propriétés/Données*. Tout va bien jusqu'au moment où l'on sauvegarde un fichier contenant un tel objet et que l'on veut ensuite le recharger dans une nouvelle session de *FreeCAD*..

IV.3.d Installation

La macro précédente fonctionne très bien et fournit un objet modifiable à la volée à l'aide de la fenêtre *Propriétés/Données* mais il y a deux inconvénients.

- ✓ À chaque redémarrage de *FreeCAD*, il faut charger/exécuter cette macro.
- ✓ Si l'on ouvre un fichier *FreeCAD* contenant une cycloïde avant d'avoir exécuté la définition des classes, alors il y a génération d'erreurs et les objets ne sont plus éditables.

Pour remédier à cela, il faut installer ces fichiers de définitions de classes dans un endroit où *FreeCAD* va pouvoir les trouver.

- ✓ Pour permettre à *FreeCAD* de charger le code Python définissant la classe **Cycloid**, j'ai commencé par renommer le fichier en *cycloid.py*. Car par défaut les fichiers de macros de *FreeCAD* ont pour extension *.FCMacro*.
- ✓ Ensuite, j'ai placé ce fichier dans le sous-répertoire *Mod/fpo* du répertoire que *FreeCAD* explore au démarrage pour y trouver les Macros et les *add-on* ; sous linux, c'est *.FreeCAD*, voir [cette page](#) pour les autres systèmes. Toutefois, il faut respecter la structure suivante :

<i>.FreeCAD</i>	Répertoire contenant les macros et les <i>add-on</i>
<i>Mod</i>	Répertoire contenant les modules
<i>fpo</i>	Répertoire <i>feature python objects</i>
<i>__init__.py</i>	Fichier vide
<i>cycloid</i>	Répertoire dédié aux cycloïdes
<i>__pycache__</i>	Répertoire créé par Python
<i>__init__.py</i>	Fichier vide
<i>cycloid.py</i>	Fichier contenant les définitions de classes

Les fichiers *__init__.py* peuvent être laissés vides : leur rôle est d'indiquer à *Python* que les répertoires correspondants contiennent des modules (cf. [cette page](#) ou [cette autre](#)).

Avec une telle installation,

- ✓ la classe **Cycloid** est chargée lorsque *FreeCAD* en a besoin, et il n'y a plus d'erreur à l'ouverture de fichiers *FreeCAD* contenant des objets de cette classe ;
- ✓ plutôt que de faire un *copier/coller* des lignes de programme ou de les exécuter via le menu *<Macro>* de *FreeCAD*, on pourra utiliser **import** dans la console Python : avec la disposition ci-dessus, on peut par exemple écrire :
from cycloid.cycloid import *
- ✓ on peut alors remarquer qu'il y a eu création d'un répertoire *__pycache__* où Python stocke une version compilée des modules ; c'est d'ailleurs un moyen de vérifier que le module est bien chargé : il suffit de l'effacer pour vérifier qu'il est à nouveau généré ?

Remarque Comme indiqué [sur cette page \(Setting things up\)](#), j'avais dans un premier temps installé le répertoire *fpo* dans le répertoire *Macro* de mon répertoire *.FreeCAD* (sous linux). mais j'ai eu des problèmes car j'ai l'habitude de balader mon répertoire des macros utilisateur en fonction des projets sur lesquels je travaille : lorsque le répertoire *Macro* que j'ai choisi à la dernière utilisation de *FreeCAD* n'est celui par défaut, *FreeCAD* ne trouve pas les classes lors du lancement, ce qui provoque des erreurs.

IV.3.e En faire une commande ??

On peut alors en faire facilement une commande comme il est expliqué dans la dernière partie de [cette page](#) (à partir de «*Bien sûr, il serait fastidieux* »).

Pour aller plus loin openBrain propose : *Pour le dialogue de création (et si tu veux pousser plus loin), ce qui serait bien est de le convertir en widget et d'utiliser les méthodes setEdit()/unsetEdit() pour l'afficher dans la panneau des tâches. Comme ça même après création, on peut encore utiliser le GUI pour éditer.*

<https://forum.freecadweb.org/viewtopic.php?t=55169>

<https://forum.freecadweb.org/viewtopic.php?t=5977>

✓ [Macro : Task Panel Usage and Documentation](#)

voir cet exemple https://github.com/mnesarco/FreeCAD_Uti ... pov.py#L37

IV.4. Approfondissements

??? Cette partie est encore en travaux ???

IV.4.a Classes ViewProvider

Cette partie contient des références sur la construction de *ViewProvider* de compétition, donnant parfois (cf. [l'histoire de la molécule](#)) une représentation bien différente de ce qu'est vraiment l'objet. Cela m'a obligé à aller voir jusqu'à la *Scenegraph* et au module *Coin* : ce n'est pas simple !

C'est en [lisant cette discussion](#) (surtout à partir [du 7eme post](#)) que j'ai été amené à regarder cette page très intéressante [sur les objets scriptés \(qui existe aussi en français\)](#). D'abord il m'a fallu un certain temps pour m'apercevoir que :

- ✓ certains objets y sont créés en tant que **Part::Feature**, pour lesquels d'après la remarque précédente on peut se passer de décrire un **ViewProvider** ;
- ✓ alors que d'autres y sont créés en tant que **App::Feature**, pour lesquels il est indispensable de créer un **ViewProvider**.

Voir éventuellement ce [diagramme des relations entre les classes](#)

Concernant [la discussion précédente](#) :

- ✓ voir [ce bug report](#) (signalé dans la discussion) ;
- ✓ voir aussi [cette modif concernant la molécule](#) ;
- ✓ toute la suite parle des **So...** et il faut que je la comprenne.

La [page principale sur Viewprovider](#) (mais peu de choses sauf *geticon*)

Concernant l'exemple de l'Octaèdre https://wiki.freecadweb.org/index.php?title=Scripted_objects

- ✓ J'ai commencé par mettre un `FreeCAD.ActiveDocument.recompute()`
- ✓ Contrairement à ce qui se passe pour un objet de type `Box`, la partie « `Coordinate3` » du `ViewObject` n'est pas mise à jour lorsque l'on change une dimension ! Et effectivement, il n'y a rien concernant cela dans la fonction `updatedata` !

Voir [Ref Api ViewProvider](#) ??? ou [cette page](#) ou encore [celle-là](#)

Voir cette page dédiée <https://wiki.freecadweb.org/Viewprovider> ????

Tout [une partie sur le](#) [une partie sur le ViewProvider](#) d'un `Part::Feature`

<https://wiki.freecadweb.org/Viewprovider>

IV.4.b Scenegraph, Coin et Open Inventor

Quelques explications concernant les termes *Coin*, *scenegraph* et autres *Open Inventor langage* que l'on rencontre souvent dans ce contexte.

- ✓ Une [scenegraph](#) est une structure de donnée souvent utilisée par un programme graphique,

comme par exemple FreeCAD

- ✓ La [page de la doc FreeCAD sur scenegraph](#) où l'on trouve : *One of the big jobs FreeCAD does for you is translating OpenCASCADE geometry information into OpenInventor language.*

On y trouve aussi trois lignes de macros permettant de visualiser le scenegraph d'un objet ou plutôt sa représentation en code *Open Inventor* :

```
obj = FreeCAD.ActiveDocument.ActiveObject
viewprovider = obj.ViewObject
print viewprovider.toString()
```

- ✓ Pour manipuler tout cela on utilise l'interface [Pivy](#) :
 - Sur [cette page](#) : on trouve l'affirmation suivante : *Therefore, in FreeCAD, the terms "Pivy", "Coin" or "Open Inventor" refer to the same thing essentially.*
 - On y trouve aussi un code montrant comment *coin* permet de traquer par *callback* les événements liés par exemple à la souris.
 - Pour comprendre un peu mieux toutes ces histoires d'*Open Inventor Language* et des fonctions [Soxxx\(\)](#), j'ai trouvé le livre [The Inventor Mentor](#) dont j'ai un pdf sur disque.

Voir éventuellement [comment insérer un objet dans le scenegraph](#)

IV.5. Autres références peut-être intéressantes

- ✓ Quand on cherche des infos pour écrire en *Python*, penser à utiliser le Menu <Aide><Doc. Automatique des modules Python> où l'on trouve parfois des choses intéressantes. Mais ce n'est pas toujours évident.
- ✓
- ✓ Des [histoires de souris et de modification de curseur](#)
- ✓ [Ideas needed how to pause a macro until some inputs/selections are done](#)
- ✓
- ✓ Un [vieux post \(2012\) sur la récupération des objets scriptés](#) où l'on insiste sur l'importance de la méthode `attach()` et où l'on parle de `__getstate__()` et `__setstate__()`.
 - <https://forum.freecadweb.org/viewtopic.php?t=40812>
 - https://wiki.freecadweb.org/Scripted_objects_saving_attributes
 - Dans https://wiki.freecadweb.org/Scripted_objects/fr on trouve : *Il est possible d'empaqueter du code Python dans un fichier FreeCAD en utilisant la sérialisation json avec un `App::PropertyPythonObject`, mais ce code ne peut jamais être exécuté directement et a donc peu d'utilité pour notre propos ici.*
- ✓
- ✓ [QT: comment detecter un clic dans la vue 3D?](#) Et [ce post](#)
- ✓ Pour faire une fusion voir <https://wiki.freecadweb.org/Scripts>
- ✓ Mettre en lien avec [Pyramids-and-Polyhedrons/tree](#) ????
- ✓ [Au sujet des objets de type compound](#)

Autre [échange intéressant](#) au sujet du passage de cpickle à Json : ainsi on ne peut plus stocker une classe python dans le Proxy

voire (pour les plus avancés) une [page de création d'interfaces](#)

[Peut-être un truc intéressant à regarder](#)

[Accéder par une variable à une propriété d'un FeaturePython et la modifier](#)

[Loop on Part::FeaturePython onChanged](#)

[App::FeaturePython equivalent to Part:: Multi Union/Fuse](#)

[FeaturePython / Scripted Object question](#)

[FeaturePython restoring on document load](#)

<https://raw.githubusercontent.com/FreeCAD/FreeCAD/master/src/Mod/TemplatePyMod/FeaturePython.py>

<https://forum.freecadweb.org/viewtopic.php?t=47132>

<https://forum.freecadweb.org/viewtopic.php?f=10&t=24734&p=194209&hilit=proxy#p194209>

Une autre page intéressante sur [les répertoires par défaut de FreeCAD](#)

Un truc à regarder <https://forum.freecadweb.org/viewtopic.php?f=22&t=45754&start=20>

IV.6. Sous-objets topologiques et géométriques

Ce texte reflète mon expérience de découverte de ces notions à la suite en particulier de la lecture sur le forum de ce [post de Chris_B](#) ou [cet autre](#). Il ne faut pas chercher dans ce qui suit une liste exhaustive des propriétés concernées (ce qui serait impossible) mais plutôt une introduction permettant au lecteur de devenir un peu plus autonome dans la découverte de leur manipulation.

Un objet *FreeCAD*, expose naturellement certaines de ses caractéristiques.

- ✓ Une *Box* par exemple expose ses trois dimensions dans la fenêtre *Propriété/Données*, et l'utilisateur peut les y modifier facilement avec visualisation immédiate du résultat.
- ✓ Il en est de même pour un (tronc de) cône, qui expose angle, hauteur et ses deux rayons.
- ✓ Pour un objet *PartDesign*, c'est dans les fonctions (*features*) qui ont servi à le construire que l'on trouve certaines propriétés comme par exemple dimension d'une protusion ; et on peut aussi les modifier avec effet immédiat sur la représentation 3D.
- ✓ Pour un objet de type *FeaturePython*, le concepteur peut aussi exposer certaines propriétés dans la fenêtre *Propriété/Données* (cf. *addProperty*)

Mais il y a bien d'autres entités que l'on peut aller dénicher dans un objet et qui sont liés au format qu'utilise *FreeCAD* pour représenter ses objets 3D en interne : le **B-Rep** (**B**oundary **R**epresentation en anglais) traduit par *Représentation Frontière* ou *Représentation par les Bords*.

Dans *FreeCAD*, à l'exception des [objets de type Mesh](#), quasiment tous les objets ayant une représentation 3D possède une [Shape \(de classe Part::TopoShape\)](#).

- ✓ Par exemple : un [Body](#) est construit à partir d'un [Part Feature](#) créé avec [l'atelier PartDesign](#)
- ✓ Une *Shape* est un objet interne, sous-objet d'un [Body](#) ([Bodyxxx.Shape](#))
- ✓ Un objet de type [Part](#) qui regroupe plusieurs [Bodies](#) possède donc une collection de *Shapes*, mais n'a pas de forme qui lui est propre.

Contrairement à la [CSG](#) qui travaille à partir de solides simples et d'opérations booléennes, avec la *B-Rep*, un objet 3D est entièrement représenté par son bord. C'est une technique qui sépare :

- ✓ d'une part, les éléments topologiques, de type '*Part::TopoShape*', comme par exemple faces, arêtes, sommets) et les relations (adjacence, incidence) qu'ils ont entre eux ;
- ✓ d'autre part, les éléments géométriques de type *Part::Geom...*, comme *Part::GeomLine* (pour une ligne) ou '*Part::GeomCircle*' (pour un cercle) ; on peut les voir comme les supports des précédents, qui définissent leur position dans l'espace.

Sur [cette page](#), on en trouve une présentation succincte et un organigramme intéressant.

Pour un objet donné, toutes ces entités se trouvent dans son sous-objet **Shape** ; par exemple, pour une **Box** créée avec l'atelier *Part*, et que l'on peut manipuler en *Python* à l'aide de :

```
>>> obj = App.ActiveDocument.Box
```

ces entités topologiques et géométriques se trouvent dans :

```
>>> obj_shape = obj.Shape
```

Il en est de même pour tout objet 3D *FreeCAD*, quelle que soit la façon dont on l'a construit.

IV.6.a Sous-objets topologiques

Par ordre de complexité décroissante, les principales entités topologiques d'un objet **obj** 3D de *FreeCAD* sont retournées par les propriétés suivantes.

- ✓ **obj.Shape.Faces** retourne la liste des faces de **obj**.
 - Ainsi pour tout entier **i** strictement inférieur à **len(obj_shape.Faces)**, **obj.Shape.Faces[i]** donne une référence vers la face d'index **i** de **obj**.
 - Prendre garde qu'en programmation *Python*, l'index **i** commence à 0, alors que dans le *Gui* de *FreeCAD*, les indices des faces qui s'affichent dans la barre d'état commencent à 1.
- ✓ **obj.Shape.Wires** retourne la liste des contours fermés plans de **obj**.
 - Ainsi pour tout entier **j** strictement inférieur à **len(obj_shape.Wires)**, **obj.Shape.Wires[j]** donne une référence vers le contour fermé d'index **j** de **obj**.
 - Même mise en garde que précédemment concernant le domaine où varie **j**.
 - Toutefois **Wires** est aussi une propriété d'un objet face et l'instruction : **obj.Shape.Faces[i].Wires** retourne la liste de tous les contours fermés de la face d'index **i** de **obj**.
 - En revanche, la complétion automatique ne fonctionne pas si l'on tape **obj.Shape.Faces[i]**. Lorsque le point « . » est ainsi précédé d'un sélecteur, la fenêtre de complétion n'a pas l'air de fonctionner correctement ; pour la retrouver j'ai dû faire l'affectation : **f = obj.Shape.Faces[i]** avant de taper **f.**, et de pouvoir la retrouver.
- ✓ **obj.Shape.Edges** retourne la liste des arêtes de **obj**.
 - On peut faire exactement les mêmes remarques que précédemment concernant l'index **k** permettant d'accéder à un élément d'une telle liste.
 - Cette propriété **Edges** peut s'appliquer aussi bien à un objet 3D, qu'à l'une des ses **Faces**, voire à l'un des **Wires** ; elle retourne alors les arêtes correspondantes.
- ✓ **obj.Shape.Vertexes** donne la liste des sommets de **obj** ; mais comme précédemment, cette propriété peut aussi s'appliquer à l'une de ses **Faces**, l'un de ses **Wires** ou l'un de ses **Edges**.

Comme on peut le vérifier à l'aide de la console *Python*, chacun des sous-objets précédents possède le même type, qui est *Part::TopoShape*.

```
>>> obj.Shape.Faces[0].TypeId , obj.Shape.Vertexes[0].TypeId
'Part::TopoShape' , Part::TopoShape'
```

En revanche, chaque catégorie a un **ShapeType** différent :

```
>>> obj.Shape.Faces[0].ShapeType , obj.Shape.Vertexes[0].ShapeType
'Part::TopoShape' , 'Vertex'
```

À chacune des entités topologiques précédente est associée une entité géométrique, support sur laquelle elle est construite : un point, une courbe ou une surface. Nous les étudions dans la suite.

IV.6.b Support d'un Vertex (Point)

- ✓ Si **Box** est un cube par défaut construit avec *Part*, on obtient la liste de ses sommets avec :

```
>>> App.ActiveDocument.Box.Shape.Vertexes
```

```
[<Vertex object at 0x55a4e9d85bb0>, <Vertex object at 0x55a4e9dce3e0>, ....]
```

- ✓ Pour obtenir le sommet qui s'affiche comme *Vertex6* dans la barre d'état, taper :

```
>>> v = App.ActiveDocument.Box.Shape.Vertexes[5]      # Décalage de 1
```

- ✓ La récupération des coordonnées du sommet se fait alors avec :

```
>>> v.Point
```

```
Vector (10.0, 0.0, 0.0)
```

- ✓ Si l'on ne veut que la liste des points de *Face3*, on tape :

```
>>> App.ActiveDocument.Box.Shape.Faces[2].Vertexes
```

```
[<Vertex object at 0x55a4e98f1150>, <Vertex object at 0x55a4e8f62980>, ...]
```

IV.6.c Support d'un Edge (Curve)

Si **e** est une arête, avec donc **e.ShapeType == 'Edge'**, alors **e.Curve** retourne une référence vers une courbe la contenant et que *FreeCAD* traite le plus souvent comme courbe paramétrée.

Prenons l'exemple du (tronc de) cône défini avec les valeurs par défaut de l'atelier *Part*.

- ✓ Commençons par récupérer la *Shape* de l'objet :

```
>>> obj_sh = App.ActiveDocument.Cone.Shape
```

- ✓ Dans mon cas, le cercle supérieur s'affiche dans la barre d'état comme *Edge1*, et je le récupère donc avec :

```
>>> edg_1 = obj_sh.Edges[0]
```

Je peux en afficher le **ShapeType**

```
>>> edg_1.ShapeType
```

```
'Edge'
```

- ✓ Je fais alors pointer **curv_1** vers sa **Curve**

```
>>> curv_1 = edg_1.Curve
```

- ✓ Cette courbe est de type '*Part::GeomCircle*' et, comme on peut le vérifier, *FreeCAD* en utilise une représentation paramétrique avec un angle variant de 0 à $6.28... = 2\pi$ radians.

```
>>> curv_1.TypeId, curv_1.FirstParameter, curv_1.LastParameter
```

```
'Part::GeomCircle', 0.0, 6.283185307179586
```

- ✓ J'ai découvert les trois propriétés précédentes en utilisant la complétion usuelle dans la console *Python*. Elle permet aussi de trouver plein d'autres propriétés comme par exemple :

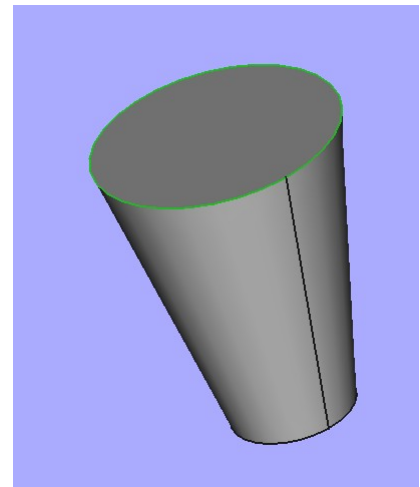
```
>>> curv_1.isPeriodic(), curv_1.__doc__      # qui retourne la docstring
```

```
True, 'Describes a circle in 3D space ....'
```

- ✓ Remarquer que l'on peut aussi trouver le domaine de variation du paramètre au niveau du **Edge**.

```
>>> edg_1.ParameterRange
```

```
(0.0, 6.283185307179586)
```



Intéressons-nous maintenant à la génératrice colorée en vert.

- ✓ Dans mon cas, je la vois s'afficher dans la barre d'état comme *Edge2*, et je le récupère donc avec :

```
>>> edg_2 = obj_sh.Edges[1]
```

- ✓ Je peux en afficher le *ShapeType*

```
>>> edg_2.ShapeType
```

```
'Edge'
```

- ✓ Je fais alors pointer *curv_2* vers sa *Curve*

```
>>> curv_2 = edg_2.Curve
```

- ✓ Cette courbe est de type '*Part::GeomLine*'

```
>>> curv_2.TypeId
```

```
'Part::GeomLine'
```

- ✓ Si comme précédemment, on essaie d'en trouver le domaine du paramètre, avec *curv_2* :

```
>>> curv_2.FirstParameter , curv_2.LastParameter
```

```
-2e+100 , 2e+100
```

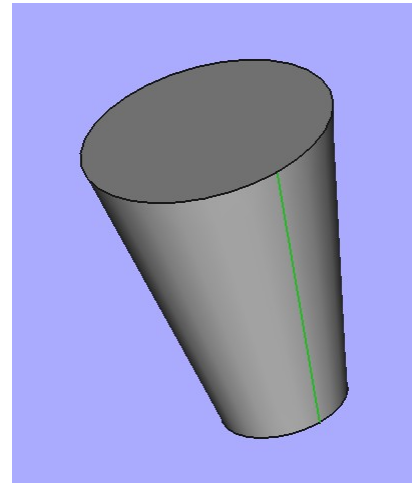
et avec l'objet topologique dont c'est le support :

```
>>> edg_2.ParameterRange
```

```
(0.0, 10.198039027185569)
```

on voit que *FreeCAD* traite ici différemment

- l'arrêt (*Edge*) qui est un segment donc borné,
- son support qui est une droite quasi non bornée ($2e+100$ jouant le rôle de l'infini).



Si l'on coupe le cône par un plan qui n'est pas orthogonal à son axe de symétrie, on obtient une ellipse ou une hyperbole voire exceptionnellement une parabole suivant l'orientation du plan.

- ✓ Si, comme ci-contre l'ellipse s'affiche en tant que *Edge1*, on la récupère l'objet topologique par :

```
edg = App.ActiveDocument.Cut.Shape.Edges[0]
```

puis son support par :

```
curv = edg.Curve
```

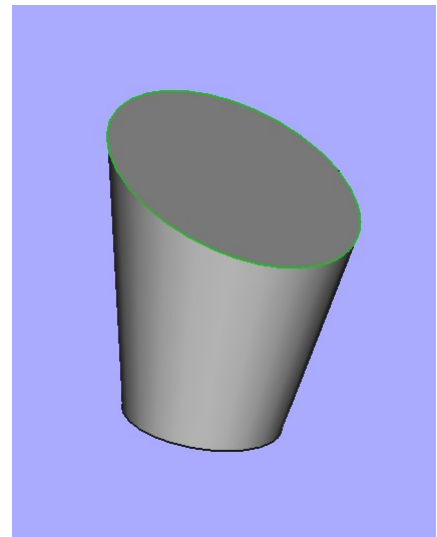
- ✓ Avec la complétion automatique, on voit que l'on peut avoir :

```
curv.Eccentricity
```

```
curv.Focus1
```

```
curv.MajorRadius
```

dont les noms sont suffisamment éloquentes.



Le lecteur curieux pourra vérifier :

- ✓ que dans le cas d'un ellipse, les bornes du paramètre sont les mêmes avec *edg* et avec *curv*,
- ✓ alors que dans le cas d'une hyperbole, l'objet topologique reste borné alors que son support géométrique (comme pour les droite) est quasi infini !

IV.6.d Support d'une Face (Surface)

Si **f** est une face, avec donc **f.ShapeType == 'Face'**, alors **f.Surface** retourne une référence à une surface la contenant, que FreeCAD traite comme une surface paramétrée, les paramètres étant appelés **U** et **V**.

Prenons encore l'exemple du (tronc de) cône défini avec les valeurs par défaut de l'atelier *Part*.

- ✓ Commençons par récupérer la Shape de l'objet :

```
>>> obj_sh = App.ActiveDocument.Cone.Shape
```

- ✓ Dans mon cas, la surface latérale (en vert) s'affiche dans la barre d'état comme *Face1*, et je le récupère donc avec :

```
>>> fac_1 = obj_sh.Faces[0]
```

Rappelons que pour l'index il y a un décalage de 1 entre ce qui est vu dans le *Gui* et celui de la liste *Python*.

- ✓ Je peux en afficher le **ShapeType** :

```
>>> fac_1.ShapeType
```

```
'Face'
```

- ✓ Je fais alors pointer **surf_1** vers sa **Surface** qui est de type *'Part::GeomCone'*

```
>>> surf_1 = fac_1.Surface
```

```
>>> surf_1.TypeId
```

```
'Part::GeomCone'
```

- ✓ En utilisant la complétion, on voit que l'on peut alors en trouver son sommet, son axe (point & vecteur), son demi-angle au sommet, ...

```
>>> surf_1.Apex
```

```
Vector (0.0, 0.0, -10.0)
```

```
>>> surf_1.Center , surf_1.Axis
```

```
Vector (0.0, 0.0, 0.0) , Vector (0.0, 0.0, 1.0)
```

```
>>> surf_1.Radius
```

```
2.0
```

```
>>> surf_1.SemiAngle
```

```
0.19739555984988078
```

- ✓ Cette surface, de type *'Part::GeomCone'*, est paramétrée par l'angle de rotation autour de l'axe du cône et la distance sur l'arête qui vaut au maximum $10/\cos(\text{surf_1.SemiAngle})$ car le cône est de hauteur 10mm. On trouve bien cette limite supérieure avec le **ParameterRange** de la face, mais pas avec la surface dont les génératrices sont des droites, donc « infinies ».

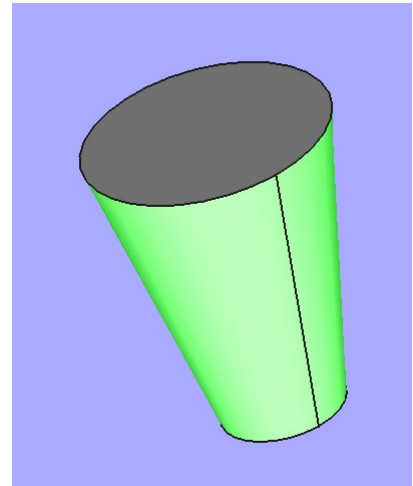
```
>>> fac_1.ParameterRange
```

```
(0.0, 6.283185307179586, 0.0, 10.198039027185569)
```

```
>>> surf_1.bounds() # ne pas oublier les ( ) car c'est une méthode
```

```
(0.0, 6.283185307179586, -2e+100, 2e+100)
```

Remarque L'ensemble des couples (**U,V**) permettant de décrire toute la surface est donc le rectangle défini par : $0 \leq U \leq 6.28\dots$ et $0 \leq V \leq 10.19\dots$



Faisons la même chose avec la face supérieure de ce cône, en vert sur le dessin ci-contre. Dans mon cas, elle s'affiche dans la barre d'état comme *Face2*.

- ✓ Je le récupère donc avec :

```
>>> fac_2 = obj_sh.Faces[1]
```

- ✓ Je peux en afficher le *ShapeType* :

```
>>> fac_2.ShapeType
```

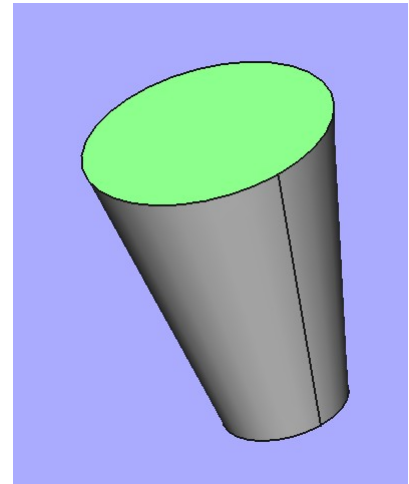
```
'Face'
```

- ✓ Je fais alors pointer *surf_2* vers sa *Surface* qui est de type *'Part::GeomPlane'*

```
>>> surf_2 = fac_2.Surface
```

```
>>> surf_2.TypeId
```

```
'Part::GeomPlane'
```



La surface, de type *'Part::GeomPlane'*, est un plan paramétré par ses deux coordonnées ;

- ✓ comme un tel plan est non borné, la méthode *bounds()* appliquée à cette surface retourne des valeurs « infinies » de l'ordre de $2e+100$,

```
>>> surf_2.bounds()
```

```
(-2e+100, 2e+100, -2e+100, 2e+100)
```

- ✓ alors que les paramètres décrivant la face sont les mêmes coordonnées mais bornées par 4 :

```
>>> fac_2.ParameterRange
```

```
(-4.0, 4.0, -4.0, 4.0)
```

IV.6.e Courbes coordonnées d'une surface

Pour visualiser la façon dont une surface est paramétrée avec les valeurs *U* et *V*, on peut tracer les deux familles de courbes coordonnées, à *U* constant pour la première et à *V* constant pour la seconde. Pour cela, en plus de ce que l'on a déjà vu, on utilise :

- ✓ d'abord *surface.uIso(u)* qui retourne la courbe coordonnée de *surface* à *U* constant ; c'est un objet géométrique ;
- ✓ puis la méthode *toShape(v_min, v_max)* qui transforme l'objet géométrique précédent en un objet topologique que l'on peut alors ajouter au document courant.

Remarque On peut aussi utiliser la méthode *toShape* sans paramètre lorsque les courbes que l'on manipule sont bornées ; mais dans le cas d'un cône par exemple, les génératrices (droites géométriques) sont non bornées et il est indispensable de mettre des bornes.

La fonction suivante permet de visualiser ces deux familles de lignes.

```
def trace_isoline(body_name, face_number, nb_curves=10):

    # get the face (here face_number is what we see with the GUI)
    face = App.ActiveDocument.getObject(body_name).Shape.Faces[face_number-1]

    # get the geometric surface of the face
    surface = face.Surface

    # get the parametric bounds (u_min, u_max, v_min, v_max) of the (parametric) surface
    # One can also use limits = surface.bounds(),
    # but it is "inifnite" for non bound objects such lines
    limits = face.ParameterRange
    u_min, u_max, v_min, v_max = limits

    # Drawing of the u-isoparametric curve
    u = u_min
    h = (u_max-u_min)/ nb_curves
    while u <= u_max:
        # get the u-isoparametric curve
        lu = surface.uIso(u)
        # Add it to the document with blue color
        obj = App.ActiveDocument.addObject('Part::Feature', "ligne_Iso_u")
        obj.ViewObject.LineColor = (0.0,0.0,1.0)
        # Put the Shape of the curve into the object
        #(the use of v_min and v_max is madatory in the case of line, to clip it)
        obj.Shape = lu.toShape(v_min,v_max)
        # Next u value
        u += h

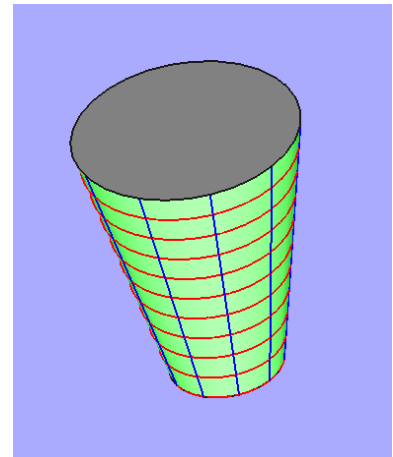
    # Likewise drawing of the v-isoparametric curve
    v = v_min
    h = (v_max-v_min)/ nb_curves
    while v <= v_max: Face1
        lv = surface.vIso(v)
        obj = App.ActiveDocument.addObject('Part::Feature', "ligne_Iso_v")
        obj.Shape = lv.toShape(u_min,u_max)
        obj.ViewObject.LineColor = (1.0,0.0,0.0)
        v += h
```

On peut utiliser cette fonction pour tracer ces courbes coordonnées sur la surface verte d'un cône de nom **Cone** obtenu avec l'atelier *Part*. Comme elle s'affiche en tant que *Face1*, on tape :

```
>>> trace_isoline('Cone',1)
```

On voit alors :

- ✓ les courbes bleues : sur chacune d'elles, l'angle **U** est constant, alors que **V** varie de 0 à la longueur maximum de l'arête.
- ✓ les courbes rouges : sur chacune d'elles, la distance **V** est constante, alors que l'angle **U** varie de 0 à 2π .



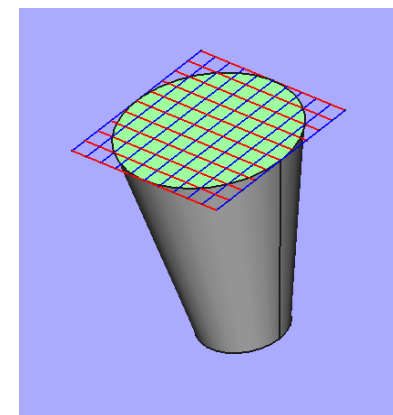
Si on utilise alors cette fonction **trace_isoline** pour obtenir les courbes coordonnées à **U** et à **V** constant de la face supérieure du même cône :

```
>>> trace_isoline('Cone',2)
```

on obtient le quadrillage ci-contre, qui correspond bien au domaine donné par le **ParameterRange** :

$$-4 \leq U \leq 4 \text{ et } -4 \leq V \leq 4$$

mais qui n'est pas tout à fait ce que l'on attend.



Pour récupérer le cercle, on peut utiliser **OuterWire**, qui retourne un **Wire** dont on peut alors extraire le cercle.

```
>>> e = fac_2.OuterWire.Edges[0]
```

```
>>> e.Curve
```

Circle (Radius : 4, Position : (0, 0, 10), Direction : (0, 0, 1))

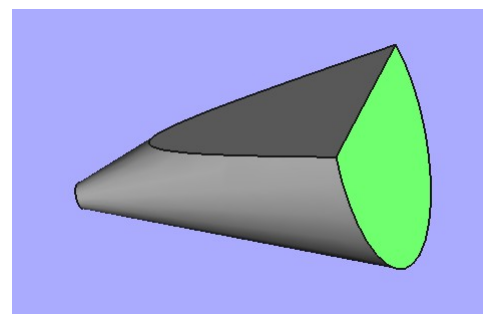
Remarque Dans un cas comme celui de droite, où

- ✓ l'intersection du plan avec le cône est une hyperbole,
 - ✓ la génératrice initiale du cône est en dessous,
- le **OuterWire** possède trois composantes, ce qui n'est pas évident à première vue !

```
>>> f = App.ActiveDocument.Cut.Shape.Faces[2]
```

```
>>> f.OuterWire.Edges
```

```
[<Edge object at 0x558e4df9c660>,
 <Edge object at 0x558e4de7eed0>,
 <Edge object at 0x558e4de99350>]
```



IV.6.f Exemple avec des B-Spline

Regardons le cas de la face verte du loft ci-contre réalisé entre un rectangle du plan $z=0$ et un cercle du plan $z=50$.

Comme elle s'affiche en tant que *Face6*, je pose :

```
>>> f = App.ActiveDocument.Body.Shape.Faces[5]
>>> c=f.Surface
>>> c.TypeId
```

```
'Part::GeomBSplineSurface'
```

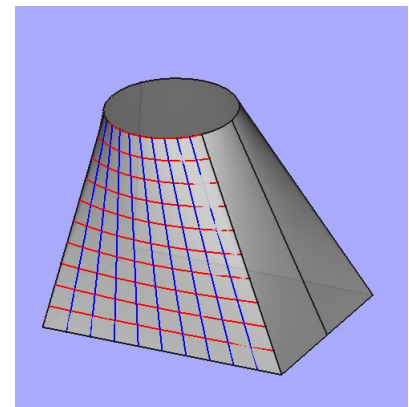
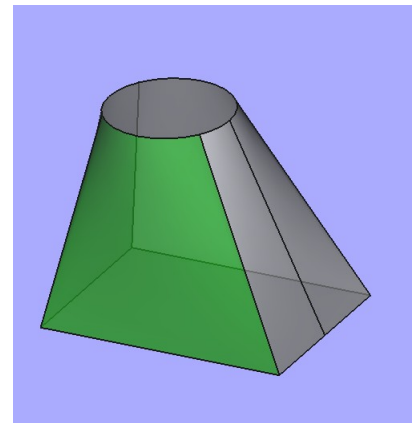
Pour cette surface de type *B-Spline* les bornes des paramètres *U* et *V* sont les mêmes, qu'on les prenne sur l'objet topologique *f* ou sur l'objet géométrique *c*.

```
>>> c.bounds() , f.ParameterRange
(0.0, 62.340035, 0.0, 1.0) , 0.0, 62.340035, 0.0, 1.0)
```

Pour visualiser ce paramétrage, on peut lancer :

```
>>> trace_isoline('Body',6)
```

- ✓ chaque courbe bleue correspondant à une valeur donnée de *U* choisie entre 0 et 62.34..., alors que *V* varie de 0 à 1.
- ✓ chaque courbe rouge correspondant à une valeur donnée de *V* choisie entre 0 et 1, alors que *U* varie de 0 à 62.34...



IV.6.g Et l'inverse : du simple au complexe?

Inversement, [sur cette page](#), on peut trouver comment construire des objets géométriques ainsi que des objets topologiques.

- ✓ Certaines fonctions comme *Part.makeLine()* ou *Part.makePolygon()* donnent directement un objet de type *'Part::TopoShape'*, et il faut passer par *.Curve* pour en obtenir l'objet géométrique sous-jacent
- ✓ D'autres comme *Part.LineSegment()* ou *Part.BezierCurve()* retournent un objet géométrique, et il faut ensuite utiliser une méthode *.toShape()* pour le transformer en un objet topologique, de type *'Part::TopoShape'*.
- ✓ On peut aussi regarder par exemple [ce post](#) qui en parle.

IV.6.h La fonction toShape(...)

J'ai décortiqué la fonction dans Curve2dPyImp.cpp

toShape(...) method of Part.Line instance

Return the shape for the geometry.

toShape(...) method of Part.Plane instance

Return the shape for the geometry.

J'ai décortiqué la fonction dans Curve2dPyImp.cpp

PyArg_ParseTuple()

<https://matthieu-brucher.developpez.com/tutoriels/python/api-c-numpy/> (pour O!)

<http://sdz.tdct.org/sdz/introduction-au-scripting-avec-python.html>

https://python.jpvweb.com/python/mesrecettespython/doku.php?id=exemple_python_cpp

<https://www.oreilly.com/library/view/python-in-a/0596001886/re1107.html>

<https://docs.python.org/3/c-api/arg.html> (pour O!)

<https://docs.python.org/fr/3.7/c-api/arg.html> (pour O!)

https://askcodez.com/lexension-de-python-avec-c-passer-dune-liste-a-pyarg_parsetuple.html

int PyArg_ParseTuple(PyObject *args, const char *format, ...)

Parse the parameters of a function that takes only positional parameters into local variables.

Returns true on success; on failure, it returns false and raises the appropriate exception.

IV.7. VRAC

IV.7.a Selection

Voir le script de [cette page](#) sur la sélection

Voir aussi https://wiki.freecadweb.org/Selection_API pour distinguer entre `getSelection` et `getSelectionEx`

IV.7.b B- Spline

[Un peu de théorie](#) sur les courbes de Bézier et le B-spline

<https://wiki.freecadweb.org/B-Splines>

<https://forum.freecadweb.org/viewtopic.php?t=13124>

<https://fr.wikipedia.org/wiki/Spline>

<https://fr.wikipedia.org/wiki/B-spline>

<https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/notes.html>

<https://www.ljk.imag.fr/membres/Nicolas.Szafran/ENSEIGNEMENT/MASTER2/CS/courbes-polynomiales.pdf>

<http://info.usherbrooke.ca/ogodin/enseignement/imn428/Chapitres/imn428-chap05.pdf>

https://www.fil.univ-lille1.fr/~aubert/m3ds/m3ds_courbe.pdf

https://team.inria.fr/virtualplants/files/2014/09/cours_NURBS.pdf

<https://forum.freecadweb.org/viewtopic.php?f=22&t=55630>

les [NURBS = Non Uniform Rational Basis Spline](#)

[surface flattening](#)

IV.7.c Open-Cascade

<https://dev.opencascade.org/doc/refman/html/index.html>

https://dev.opencascade.org/doc/refman/html/_b_rep___tool_8hxx.html

✓ Un index de [tout ce que l'on peut faire avec openCascade](#)

✓ Une [page intéressante sur les handle](#)

https://dev.opencascade.org/doc/overview/html/occt__tutorial.html

IV.7.d vrac

Dans [ce post de Chris_G](#), on trouve : *Second important thing : get a clear understanding of BREP (Boundary Representation) used by OpenCascade to create geometric objects. It is separated into 2 levels:*

- Geometry (points, curves, surfaces) that are used as support entities to define topology
- Topology (vertexes, edges, faces, wires, shells, solids) that are "visible" (and bounded) objects

built upon a geometry

http://free-cad.sourceforge.net/SrcDocu/d9/d35/classPart_1_1TopoShapeFacePy.html

voir [toujours de chris_G](#) (revoir toute la discussion)

<https://forum.freecadweb.org/viewtopic.php?p=232159>

<https://forum.freecadweb.org/viewtopic.php?t=7316>

<https://forum.freecadweb.org/viewtopic.php?t=53782>

<https://forum.freecadweb.org/viewtopic.php?t=15313>

<https://forum.freecadweb.org/viewtopic.php?t=15313>

On dirait que Surface et Curve sont définis dans <BRep_Tool.hxx>

voir https://dev.opencascade.org/doc/refman/html/class_geom___surface.html

V. Animation

V.1. Avant Assembly 3

V.1.a L'atelier Animation

Pour faire de l'animation, on voit beaucoup de vidéos utilisant l'atelier *Animation* que l'on peut théoriquement installer avec le gestionnaire d'*add-on*. Mais il ne fonctionne plus avec les dernières versions, qui utilisent Python 3

- ✓ Un [exemple d'animation d'engrenages](#) avec les ateliers d'engrenages *Fcgear*
- ✓ Un [bel exemple \(compliqué\)](#) d'animation avec un circuit de billes
- ✓ On trouve aussi [cette jolie vidéo](#). Mais c'est assez vieux et je n'ai pas pu la faire tourner.

V.1.b L'atelier A2plus

- ✓ Un [assemblage d'engrenages](#) en français (la Pascaline) avec le module *A2plus* (qu'il faut télécharger avec le gestionnaire d'*add-on*).
 - Il utilise des fichiers indépendants aussi bien pour le support que pour les engrenages.
 - On y voit l'utilisation des contraintes des coplanarité et de coaxialité
 Mais rien ne bouge à la fin ! Ce n'est pas de l'animation.
- ✓ Assemblage d'[une bibliothèque](#) avec le module *A2plus* (*pas encore regardé*)
- ✓ Un autre [exemple d'assemblage](#) avec le module *A2plus* ; il y a aussi une [autre vidéo](#) (plus compliquée) montrant comment réaliser le découpage des pièces.

V.2. L'atelier Assembly 3

Un [site très intéressant sur Assembly 3](#) (à revoir en détail et en priorité)

<https://bassmatifreecad.github.io/FreecadUserBook/en/asm3/link.htm>

<https://www.youtube.com/watch?v=UYHIX1IEfTE&feature=youtu.be> (vidéo privée ???)

La [page wiki](#) à laquelle beaucoup de références amènent

Un [tuto sur cette page](#)

<https://bassmatifreecad.github.io/FreecadUserBook/en/asm3/asm.htm> (broken)

https://wiki.freecadweb.org/Assembly3_Workbench (page globale)

<https://github.com/ceremcem/freecad-notes>

<https://www.youtube.com/watch?v=KJ3Dm6Rt4Nc>

<https://youtu.be/oFXIDKFzync?t=263>

V.3. L'atelier Assembly 4

V.3.a Introduction

L'atelier *Assembly4* permet de faire des assemblages et aussi des animations. Mais il faut avouer qu'il n'est pas très simple à prendre en mains. Il ne fait pas partie des ateliers installés à l'origine et il faut utiliser le gestionnaire d'*Add-on* : installation assez longue pour moi, ne pas s' impatienter.

- ✓ La [page officielle d'Assembly4](#) (très pauvre pour le moment 02/01/20)

- ✓ Une [présentation générale](#) (anglais) par son concepteur principal et [les tutos qu'il a faits](#). En particulier [ce tutoriel](#) et [sa version écrite](#) pour positionner sur un axe trois roulements à billes (importés en .step), mais tout cela paraît bien compliqué pour le résultat obtenu !
- ✓ Voir des explications plus techniques (mais aussi redondantes) sur [cette page](#) ou [celle-là](#).

V.3.b Premiers exemples

On trouve beaucoup d'exemple simples, voir simplistes et dans lesquels l'utilisation d'Assembly4 ne paraît pas indispensable (on peut d'en sortir avec un simple placement)

- ✓ Un [assemblage d'engrenages](#) (en anglais)
 - C'est un assemblage assez simple de deux petits engrenages qui tournent à la fin ! En fait il utilise le « Placement » de chaque engrenage et Assembly4 lui permet seulement d'entrer les bornes et l'incrément d'angle dans des fenêtres au lieu de le faire en programmation.
 - Il dit aussi vers 10mn30 que «l'atelier Assembly4 est difficile à utiliser et qu'il est préférable de commencer par essayer A2plus ».
- ✓ Une [animation intéressante avec Assembly4](#) d'un disque et d'un bras (type bielle/manivelle).

Il y a sur le forum, un sujet [Assembly 4 workbench](#) qui contient des exemples intéressants.

- ✓ D'abord des exemples simples
 - Comment [voir se visser un boulon](#)
 - Comment [faire glisser un écrou](#) (et [le fichier FreeCAD](#))
 - Une [chaîne à 5 côtés qui s'enroule](#)
- ✓ Ensuite, viennent des animations plus intéressantes (et plus difficiles)
 - D'abord un [moteur à deux pistons](#)
 - Une tout autre idée : [ciseaux/ascenseur](#)
 - [Tout une armoire \(avec portes\)](#)
 - Comment [mettre des boulons avec Asm4](#) (boulons fixes)
 - Un [monstre intéressant](#) sur cette page. Voir aussi l'échange associé
- ✓ L'un des participants [ppemawm](#) a montré quelques exemples vraiment très fouillés : les fichiers FreeCAD ne sont pas disponibles sur le forum mais il les envoie sur demande. On peut citer :
 - un exemple de carburateur (c.f. [this carburator](#) [this carburator subassembly](#))
 - un ["adjustable clamp"](#)
 - Un [mini hélicoptère](#) dont on peut trouver les [plans sur ce site](#)
 - Une [tondeuse à gazon](#) assez complexe avec plusieurs fichiers et sous-assemblages

Il explique aussi [Méthode top/down](#) et y donne de très bon conseils.
- ✓ Plus généralement on peut jeter un œil sur les [User ShowCases](#)

V.3.c Extraits (plus techniques) du forum

C'est en décortiquant le sujet [Assembly 4 Workbench](#) du forum que j'ai trouvé ce qui suit.

- ✓ Avec les [premières versions](#), il était indispensable de mettre chaque Body dans un Part avant de pouvoir l'utiliser avec Assembly 4, mais [depuis la version 0.9](#) ce n'est plus indispensable, le Part restant indispensable pour les objets importés (comme les .step).


- ✓ Le [principe d'Assembly4](#) repose sur l'utilisation d'objets `App::Link` qui permettent (même en dehors de *Assembly 4*) de créer des « hologrammes » et de multiplier facilement un objet.
- ✓ Avec *Assembly 4*, on peut utiliser [Fasteners Workbench](#) (vis et boulons).
- ✓ Comme [expliqué dans ce post](#), bien comprendre que *Assembly4* n'est pas destiné à résoudre des contraintes, il ne peut qu'aligner des *LCS (Local Coordinates Systems)*. Si l'on a besoin de contraintes, il faut créer un Sketch (qui sait résoudre les contraintes) et aligner les *LCS* dessus.
- ✓ Pour [avoir des variables dépendantes](#), il suffit d'en créer dans la fenêtre « *Add-variables* » de *Assembly 4* et de définir leurs expressions dans la fenêtre *Propriété/vue* après avoir sélectionné `<Model><Variable>` dans l'arborescence.
- ✓ Un [exemple de mécanisme](#) utilisant rotation et glissement (variable utilisant un test ternaire)
- ✓ À quoi correspond l'erreur [Error : Links go out of the allowed scope](#) ?
- ✓ Attention, [si on édite un sketch](#) il y a un risque pour que les *LCS* attachés ne fonctionnent plus car il peut y avoir renumérotation des sommets, côtés sur lesquels ils sont accrochés.
- ✓ Au sujet du [partage des variables](#) entre plusieurs niveaux d'assemblage
- ✓ Au sujet [des modèles explosés](#)

V.3.d Hexapod

C'est cette [belle vidéo d'hexapod](#) qui m'a incité à en faire un !

- ✓ La [vidéo qui m'a permis de démarrer](#) en FreeCAD
 - ✓ Un [projet éducol](#) et [un autre](#) permettant de faire un hexapod et de le piloter avec un Arduino
 - ✓ Un [problème de CentraleSupélec/2008/TSI](#) sur le sujet
- Hervé m'a envoyé quelques références intéressantes
- ✓ Une [plateforme de Stewart](#) pilotées par ESP-32
 - ✓ Une [bille qui roule sur une surface limitée](#) avec des steppers
 - ✓ Une [réalisation bien plus ambitieuse](#)
 - ✓ Une [autre avec des legos techniques](#)
 - ✓ Une [vraie plateforme \(qui en jette\)](#) et [une dernière](#)

V.3.e Sous-Assemblages

- ✓ Une [vidéo intéressante](#) : mettre sur une maison une fenêtre de deux carreaux qui s'ouvrent.
- ✓ Une autre idée de [suspension de voiture](#)
- ✓ Comment utiliser des [LCS dans des sous-assemblage](#) (il faut utiliser `<Assembly><Import Datum Object>` ou l'icône )
- ✓ Un [tuto avec subassembly](#) pas exceptionnel mais où l'on voit l'utilisation de `<Import Datum>`

V.4. Animation avec des macros Python

J'ai trouvé deux façons faire de l'animation sans l'atelier précédent.

- ✓ En utilisant, comme pour [ce Piston](#), un script master que l'on clone. Il suffit ensuite de modifier ce master pour faire de l'animation de toute la construction. C'est alors *FreeCAD* qui recalcule à chaque instant les différentes positions du corps et ce n'est donc pas très rapide.
- ✓ En utilisant directement le placement des objets (cas des engrenages) et c'est bien plus rapide car *FreeCAD* n'a alors quasiment rien à calculer.
 - Un [premier exemple](#) d'un cylindre troué qui tourne : il n'y a pas d'explication orale mais c'est intéressant car on y voit expliqué la dépendance entre ce que l'on peut créer comme objets et les lignes de script Python correspondantes.
 - Une [vidéo simple d'animation](#) de deux engrenages très bien expliquée (en anglais)
- ✓ Enfin j'ai eu envie de faire un mix des deux :
 - Je commence par faire un tour en laissant *FreeCAD* calculer les positions.
 - C'est assez lent, mais on peut stocker ces positions dans un tableau (voire plusieurs). On peut aussi les stocker dans un fichier.
 - On peut ensuite rejouer l'animation avec ces positions et c'est bien plus rapide

Je pense que c'est ce que fait [Ric Lefrog](#) sur [cette vidéo](#) (bien que je me demande s'il ne le fait pas à la main en ajustant chaque position?)

Attention Quand on lance une macro, une animation qui tourne avec un timer, il est très difficile de la stopper car elle tourne [dans un thread à elle](#) et il est impossible de d'y accéder à l'aide de la console. Lorsque l'on teste une telle macro, il est donc préférable d'en faire un copier-collé dans la console python et de l'y exécuter ; on peut alors facilement l'arrêter avec `timer.stop()`.

V.5. Engrenages

La [page officielle de FCGear](#) qui mène à :

- ✓ la page pour les [engrenages à développantes](#) (*InvoluteGear*)
- ✓ la page pour les [engrenages à crémaillère](#) (*InvoluteRack*)

Mais on trouve aussi :

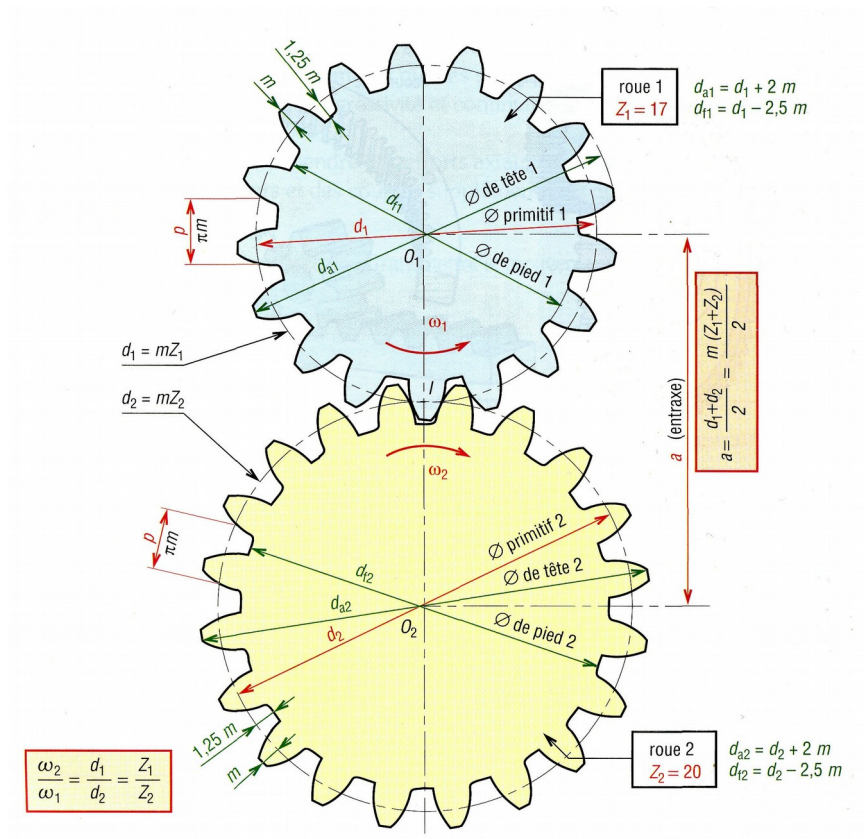
- ✓ une page sur les [vis sans fin](#) (*WormGear*)
- ✓ une page sur [engrenages coniques](#) (*BevelGear*)
- ✓ une page sur les [engrenages couronnes](#) (*CrownGear*)

Une [courte présentation intéressante](#) des différents termes utilisés

Le module m est défini par $m = \text{<Diamètre Primitif>} / \text{<Nombre de dents>}$

Et donc : $m * \pi = \text{<longueur de l'arc du cercle primitif pour passer d'une dent à l'autre>}$

Sur [cette page](#) (ou [encore ici](#)) j'ai trouvé un dessin résumant bien les choses



Un [poly intéressant de PTSI](#) sur les types de transmissions (dont boîtes de vitesses)

Ne [pas oublier le wiki](#)

Une [vidéo élémentaire](#) sur le système pignon-crémaillère

V.6. Collision et contact

- ✓ Un [code à regarder](#) et surtout [le module collision de cette page](#) qui utilise surtout `common()`. Je n'y ai pas trouvé de `proximity()`.
- ✓ Un [post de forum](#) très intéressant concernant l'utilisation de la méthode `proximity()` avec deux arguments, dont le second est une valeur de tolérance. À relire en détail !
Revoir *But I think there is a bug because if I set the tolerance to 0 it seems to go into an infinite loop (unless the translation is also changed to 10.0).*
- ✓ D'autres [exemples d'utilisation de proximity\(\)](#) (mais avec un seul paramètre)
et `f.tessellate(.1)` ??? c'est quoi ???

voir la [méthode distToShape\(\)](#) et [ce post qui en parle](#)

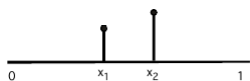
revoir aussi [cette recherche](#)

https://dev.opencascade.org/doc/refman/html/class_b_rep_extrema___dist_shape_shape.html#details

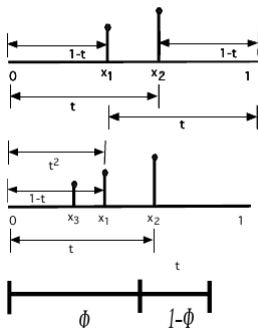
[class_b_rep_extrema___dist_shape_shape.html#details](#)

Un [article théorique](#) sur les collision en 3D

V.6.a Approximation numérique d'un minimum



Dans [ce fichier](#) (que j'ai sur disque *Minimisation_numerique.pdf*), on explique une méthode efficace de recherche de minimum qui repose sur le schéma ci-contre utilisant le nombre d'or.



Attention, il faut prendre $k = 0.6180339$ (inverse du nombre d'or) et non pas $k = 0.616$ comme indiqué !

V.6.b La méthode distToShape()

C'est avec `<Aide><Documentation automatique><Part>` que j'ai trouvé la meilleure description de la fonction (méthode) `distToShape`. C'est en fait une méthode de la classe `Shape` (`TopoShape`). Si `sh1` et `sh2` sont deux `Shapes` (`Toposhapes`) l'évaluation de `sh1.distToShape(sh2)` retourne le tuple `(dist, vectors, infos)`.

- ✓ `dist` est la distance des deux corps en mm
- ✓ `vectors` est une liste de paires de `App.Vector` réalisant le minimum de distance, le premier sur `sh1` et le second sur `sh2`.

- ✓ Infos contient des information additionnelles : c'est une liste de t-uples du type :
(topo1, index1, params1, topo2, index2, params2)
- **topo1, topo2** : identifient le type d'élément : '[Vertex](#)', '[Edge](#)', '[Face](#)'.
 - **index1, index2** sont les index des éléments (à partir de zéro).
 - **params1, params2** : paramètres de description interne de l'élément.
Pour un sommet : aucun paramètre
Pour un côté : un paramètre réel
Pour une face : un t-uple formé de deux paramètres réels

V.6.c Surveiller les modifications

Pour résoudre mes problèmes de contact, j'avais besoin de faire évoluer mon objet contact à chaque fois que je faisais bouger l'un des éléments, et j'ai donc envoyé une [question sur le forum](#).

- ✓ On peut d'abord ajouter un `App::PropertyLink` à l'objet que l'on mettra automatiquement à jour ; en créant par exemple dans le `__init__()` d'une classe `BodyObs`

```
def __init__(self, obj):
    ...
    obj.addProperty("App::PropertyLink", "BodyLink")
```

puis en définissant ensuite cette propriété `BodyLink` pour la faire pointer vers l'objet que l'on veut espionner, avec par exemple :

```
fp = App.ActiveDocument.addObject("Part::FeaturePython", "BodyObs")
BodyObserver(fp)
fp.ViewObject.Proxy = 0
fp.BodyLink = App.ActiveDocument.getObject('Box')
```

Mais cela a l'inconvénient de ne pas se mettre à jour lorsque l'on utilise les flèches haut et bas pour positionner l'objet espionné : il faut attendre de changer de ligne.

- ✓ Pour corriger le problème précédent, on peut utiliser

```
class DocObs:
    def slotChangedObject(self, obj, prop):
        print("slotChangedObject : {}-{} = {}".format(obj, prop, getattr(obj, prop)))
    def slotBeforeChangeObject(self, obj, prop):
        print("slotBeforeChangeObject : {}-{} = {}".format(obj, prop, getattr(obj, prop)))

do = DocObs()
App.addDocumentObserver(do)
# App.removeDocumentObserver(do)
```

Attention ce n'est pas pour rien qu'il a mis un `remove` car on dirait que les fonctions s'accumulent !

- ✓ Il m'a d'ailleurs donné deux références [App document observer](#) et [Gui document observer](#)

<https://forum.freecadweb.org/viewtopic.php?f=22&t=50756&p=435898&hilit=contact#p435898>

<https://forum.freecadweb.org/viewtopic.php?f=18&t=45330&p=388218&hilit=contact#p388218>

<https://forum.freecadweb.org/viewtopic.php?f=18&t=45330&p=388218&hilit=contact#p388218>

Voir aussi [ce qui concerne la BoundingBox](#)

On changed :

<https://forum.freecadweb.org/viewtopic.php?f=22&t=39106>

<https://forum.freecadweb.org/viewtopic.php?t=48182&p=413136>

https://wiki.freecadweb.org/FeaturePython_methods

https://wiki.freecadweb.org/Scripted_objects

https://wiki.freecadweb.org/Property_editor

https://wiki.freecadweb.org/Swept-path_Analysis_GSoC_Project

<https://forum.freecadweb.org/viewtopic.php?f=22&t=12426>

<https://forum.freecadweb.org/viewtopic.php?f=8&t=9029>

<https://www.youtube.com/watch?v=RrhAD585Wlw>

OpenCascade avec Python <https://www.progmodcon.com/quick-tutorial-for-getting-started-with-pythonocc/>

<https://wiki.freecadweb.org/OpenCASCADE>

Des lignes de programme intéressantes dans

<https://github.com/FreeCAD/FreeCAD-macros/blob/master/Utility/HighlightCommon.FCMacro>

<https://forum.freecadweb.org/viewtopic.php?f=18&p=340874#p340494>

V.7. Vrac

Une [super présentation de Git et GitHub](#)

et [une autre intéressante](#) (en ce qui concerne la collaboration à plusieurs)

Une [page à relire](#) (moteur d'avion Clerget) si j'ai le temps ... et [aussi celle-là](#)

Écrire correctement du code Python (Trouvé sur le forum Voir où mettre cela???)

- ✓ Voir <https://github.com/apps/lgtm-com> (qui permet de vérifier du code python ??)
The previous documents are a bit too technical, so I prefer nicer documents that explain them
- ✓ [How to Write Beautiful Python Code With PEP](#)
- ✓ [Documenting Python Code: A Complete Guide](#)

Voir éventuellement [python profiler](#) qui devrait permettre d'analyser le code Python

- ✓
- ✓ Pour [analyser objets et sous-objets](#) en Python
- ✓ [getSubObject\(\) by default return a Shape of the object.](#) *To return the object, add an extra argument retType=1. Checkout the docstring of getSubObject() to find out more details. To obtain the linked object, you can use obj.getLinkedObject(). This function is available in all types of document object, it will recursively resolve to a non-link type object if it is a link. Or, if the object is not a link, it will return itself. Checkout docstring of getLinkedObject() to find out more options.*
- ✓ [Macro : TreeToAscii](#) Pour développer toute l'arborescence

VI. Utilisation de fenêtres dans FreeCAD

Avec *FreeCAD*, toute utilisation de bouton et autres graphiques passe par *PySide*, ou mieux *PySide2*, qui fournit un interface à la bibliothèque graphique *Qt*.

- ✓ Pour une introduction générale à *QT* voir la section suivante « Environnement graphique Qt »
- ✓ Mais avec quelques connaissances sur la programmation orientée objet, on peut aussi directement commencer avec la section ???

VI.1. Environnement graphique QT

- ✓ Avec *PySide*, on utilise essentiellement une programmation orientée objet (*POO*) : pour obtenir une fenêtre, on commence en général par définir une classe et l'on crée ensuite un objet de cette classe. Si nécessaire, voir des références dans la partie « Utilisation de classes ».
- ✓ La page [exemples de niveau débutant](#) donne deux exemples que l'on peut copier coller dans la console Python pour avoir une première idée. Mais attention au [dernier exemple](#) :
 - il a été écrit du temps de *Python2*, et il faut donc ajouter des () au **print** ;
 - il faut créer **routine2**, ..., **routine5** sur le modèle de **routine1**, sinon erreurs.
- ✓ Sur la page [exemples de niveau intermédiaire](#), on peut trouver d'autres exemples à regarder, mais surtout voir la parties « Bonnes pratiques » avant de se lancer dans leur étude.
- ✓ Pour une référence précise des Widgets utilisés dans une fenêtre, on peut se référer à :
 - [cette page](#) pour les différents types utilisables,
 - [cette autre page](#) qui parle de la disposition en grille (la plus simple selon moi) ;
 - enfin la page [PyQt Quick Guide](#), qui est intéressante pour chercher une classe donnée.

Attention comme ces pages (et plus généralement tout ce [tutoriel PyQt](#)) concerne l'utilisation de **Qt** dans un cadre qui n'est pas celui de *FreeCAD*, chaque programme débute par :

```
app = QtGui.Qapplication(sys.argv)
```

Supprimer cette ligne qu'il ne faut surtout pas utiliser avec *FreeCAD* !

- ✓ Dans [cet article](#) (cf. [aussi ici](#)) on voit comment créer une grille avec [Qt_Qt_Creator \(Qt-Design\)](#) ; on peut alors l'utiliser directement avec un fichier **.ui** sans le convertir en *Python*.

VI.2. Bonnes pratiques

En m'inspirant de [ce post du forum](#) (pour les numéros de lignes dont il parle [voir le fichier](#)) ou de cet [échange](#), plein d'enseignements, j'en ai conclu ce qui suit.

- ✓ Les [PySide Intermediate Examples](#) ne sont pas vraiment bien programmés, et il faut éviter de suivre comme eux la méthode utilisant **QDialog** brut + **StayOnTopHint** + instanciation sans parent + appel avec **exec**, car cela présente les inconvénients suivants :
 - la fenêtre de dialogue se positionne alors devant toutes les fenêtres ouvertes, même celles ne dépendant pas de *FreeCAD* ;
 - le dialogue apparaît comme une fenêtre supplémentaire dans la barre des tâches de l'OS ;
 - le dialogue n'est pas lié à la fenêtre *FreeCAD*, et ne sera donc pas détruit si *FreeCAD* est fermé ; et si plusieurs instances *FreeCAD* sont ouvertes, on ne sait pas laquelle l'a ouvert ;
 - Le **StayOnTopHint** ne marche pas sur tous les OS.
- ✓ Son auteur préconise plutôt de faire comme [dans cet exemple](#) ou [dans cette macro](#).

- Faire hériter la classe du dialogue de `QtGui.QDialog` et l'instancier avec `FreeCADGui.getMainWindow()` comme parent `QtCore.Qt.Tool` comme `WindowFlag`

en écrivant :

```
class GDMLColourMap(QtGui.QDialog) :
    def __init__(self) :
        super(GDMLColourMap, self).__init__(Gui.getMainWindow(),
                                             QtCore.Qt.Tool)

        self.initUI()
```

- Faire apparaître le dialogue en appelant `show()` plutôt que `exec()`
Pour comprendre la différence, voir [cette page](#), [celle-là](#) ou [cette dernière](#).
De préférence mettre ne pas mettre ce `show()`, ou cet `exec()`, dans la fonction `__init__`, mais plutôt après avoir créé le widget.
 - Si le dialogue ne présente pas d'intérêt à rester chargé après fermeture, on peut libérer la mémoire en lui mettant l'attribut `WA_DeleteOnClose` ([voir cette page](#))
 - On peut trouver [à cet endroit](#) ainsi que [sur la page correspondante](#), des renseignements plus précis sur les paramètres précédents, `Qt::Dialog`, `Qt::Tool` et `WA_DeleteOnClose` ; voir en particulier pourquoi leur donner un parent est important dans ce cas.
 - Au sujet des [Windows Flags](#)
- ✓ Pour la disposition de la fenêtre,
- il est préférable d'utiliser le *layouts*, par exemple un `QGridLayout` comme dans [cet exemple](#) où il est facile de comprendre comment ça marche.
 - L'utilisation de `addWidget(QWidget, int r, int c, int rowspan, int colspan)` permet d'utiliser plusieurs cellules pour y mettre un champ plus long que les autres.
 - Pour la police utilisée, on peut créer une seule fois `newFont` puis l'utiliser pour tous les *widgets*. Mais il y a plus malin : changer une seule fois la font du dialogue `self.setFont(blabla)` et elle sera propagée à tous les widgets enfants.

Autres remarques de programmation vues dans ce post

- ✓ Éviter les *try except*
- privilégier les autres possibilités de test, comme existence de propriétés, de sélections, etc. ;
 - si on doit les utiliser, il faut toujours spécifier le type d'erreur que l'on s'attend à trouver, comme par exemple `except KeyError: blabla'`.
- ✓ Quand la fenêtre n'est pas modale, prendre garde à ce que vaut `ActiveDocument` lorsqu'on l'utilise : l'utilisateur du fichier peut avoir changé le document entre temps. Par exemple, lors d'une sélection, il est préférable d'utiliser `doc = selobj.Document`.
- ✓ Pour faire un "toggle" sur un booléen, par exemple `self.MakeFillet`
- le plus évident est de faire `self.MakeFillet = not self.MakeFillet` ;
 - Mais il y a plus malin : le signal '`clicked`' utilisé pour appeler la fonction émet l'état de la `checkbox`. Il suffit donc d'écrire : (??? à tester???)
`def onCheckbox1(self, state):`
`self.MakeFillet = state`

Pour mettre une fenêtre en mode modal

self.setWindowModality(QtCore.Qt.ApplicationModal)

Une [remarque intéressante](#) concernant le garbage collection, qui jette ce qui ne paraît plus nécessaire !

- ✓ Une guitare avec [boite de dialogue à plusieurs onglets](#)
- ✓ <https://forum.freecadweb.org/viewtopic.php?t=11801> pour getStandardButtons(self):
- ✓ [modifier boutons OK et Cancel](#)

VOIR CETTE ASTUCE ??????????????

Pour recharger un module

```
# test.FCMacro
# -*- coding: utf-8 -*-

if sys.modules.__contains__("MyModule"):
    del sys.modules["MyModule"]
import MyModule

# Utilisation du module MyModule
```

VI.3. Création d'une fenêtre de dialogue

En fait, il y a

les fenêtres qui sont de simples widget et qui vont aller s'afficher n'importe où l'on veut

les fenêtres qui vont s'afficher dans le panneau de tâches

VI.3.a Fenêtres simples (????)

Je ne sais pas s'il faut garder ce qui suit ????????????????,

Dans le [post de vocx sur cette page](#), on trouve un exemple très simple qui permet de construire des fenêtres simples de dialogue (qui évite de passer par *QtDesigner* et un fichier *ui*). Mais il faut garder ce genre de choses pour des cas simples.

```
class CustomTaskPanel(QtGui.QWidget):
    def __init__(self):
        self.base = QtGui.QWidget() # ?? Ça a l'air de marcher en mettant seulement
        self.form = self.base      # self.form=QtGui.QWidget()
        label = QtGui.QLabel(self.form)
        label.setText("something")
    def getStandardButtons(self):
        return int(QtGui.QDialogButtonBox.Cancel)
            | int(QtGui.QDialogButtonBox.Ok)
            | int(QtGui.QDialogButtonBox.Apply)
    def clicked(self, bt):
        if bt == QtGui.QDialogButtonBox.Apply:
            print("Apply")
    def accept(self):
        print("Accept")
        self.finish()
    def reject(self):
        print("Reject")
        self.finish()
    def finish(self):
        Gui.Control.closeDialog()
        # Gui.ActiveDocument.resetEdit()

Gui.Control.showDialog(CustomTaskPanel())
```

dans https://wiki.freecadweb.org/Manual:Creating_interface_tools on trouve

Once we have our BoxTaskPanel that has 1- a widget called "self.form" and 2- if needed, accept and reject functions, we can open the task panel with it, which is done with these two last lines:

```
panel = BoxTaskPanel()
FreeCADGui.Control.showDialog(panel)
```

C'est dans ce fichier [TaskDialogPython.cpp](#) (que j'ai trouvé en cherchant SowDialog dans le GitHub <https://github.com/FreeCAD/FreeCAD>) que j'ai vaguement compris pourquoi le widget que l'on

envoi à ShowDialog doit contenir un widget appelé « form ». En effet on y voit que ShowDialog appelle TaskDialogPython() et c'est cette dernière méthode qui teste s'il y a un attribut « form » dans l'objet.

VI.4. Widgets, QMainWindow et QDialog

De cette [présentation de Widgets](#) et de [cette page](#) j'ai tiré ce qui suit.

- ✓ Les *widgets* sont les blocs de base pour les interfaces graphiques utilisateur (*GUI*)
- ✓ Chaque composant du *GUI* (bouton, étiquette, input) est un widget qui est placé à l'intérieur d'une fenêtre ou qui est lui-même une fenêtre indépendante. Chaque type de widget est dérivé de la classe *QWidget*, qui hérite elle-même de la classe *QObject*.
- ✓ *QWidget* n'est pas une classe abstraite et peut être utilisée comme container pour d'autres widgets. De cette classe, on peut dériver facilement d'autres classes personnalisées.
- ✓ *QWidget* permet de créer une fenêtre à l'intérieur de laquelle on place d'autres widgets.
- ✓ En tant que *QObject*, chaque *QWidget* peut être créé avec un parent qui en est alors le propriétaire ; cela assure que la fenêtre est affichée à l'intérieur de la zone de son parent et qu'elle sera détruite quand le parent sera fermé.
- ✓ Un widget qui n'est pas embarqué dans un autre parent widget est appelé fenêtre.
 - Il est en général de type *QMainWindow* ou d'une classe héritée de *QDialog*.
 - Une telle fenêtre es en général munie d'un cadre et d'une barre de titre.
 - Il lui est associée une icône dans la barre de tâches.
- ✓ Il y normalement une seule application de type *QMainWindow* qui produit le cadre général avec éventuellement barre d'outils, barre de menu, barre d'état, etc.
- ✓ Dans cette fenêtre principale, on trouve des fenêtres secondaires de type (hérité de) *QDialog*, qui comme leur nom l'indique permettent de dialoguer avec l'utilisateur.

VI.5. À trier

Voir [quelques macros](#)

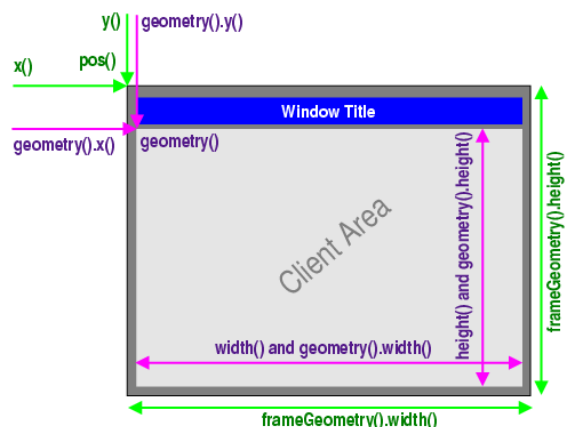
Comment [stopper une animation](#) lancée avec un timer

C'est [sur cette page](#) (même si elle traite de l'utilisation de *FreeCAD* dans d'autres applications) que j'ai trouvé la fonction `getMainWindow()`, ou plus précisément `Gui.getMainWindow()`, qui retourne la fenêtre principale de *FreeCAD*.

- ✓ J'ai pu ensuite en récupérer sa `geometry()`, qui donne sa position ainsi que sa taille. C'est très utile quand on utilise deux écrans pour être sûr que la fenêtre que l'on ouvre va se situer dans la même zone que la fenêtre *FreeCAD*.
- ✓ Au début j'avais essayé `QtGui.getMainWindow()`, comme indiqué sur la page, mais ça ne marchait pas : pour l'instant, je ne vois pas encore bien la différence entre `Gui` et `QtGui`.
- ✓ J'y ai aussi trouvé l'utilisation de `metaObject().className()` : j'ai l'impression que tester son égalité avec `"Gui::View3DInventor"` permet de savoir si l'on a affaire à une fenêtre de représentation d'un objet *FreeCAD*.
- ✓ On y trouve aussi l'utilisation de `childs` et de `findChildren(QtGui.QMainWindow)` (pour trouver tous les enfants d'une fenêtre ???)

Sur [cette page](#), quelques précisions concernant la géométrie d'une fenêtre (dimensions et position) : il y a dans *Qt* deux types de fonctions :

- ✓ celles qui incluent le cadre de la fenêtre :
`x()`, `y()`, `frameGeometry()`,
`pos()` et `move()` .
- ✓ celles qui excluent le cadre de la fenêtre :
`geometry()`, `width()`, `height()`,
`rect()` et `size()` .



Contraintes de sketch https://wiki.freecadweb.org/Sketcher_ConstrainCoincident/fr

Sur [la position souris](#)

<https://stackoverflow.com/questions/52429399/pyside2-how-to-get-mouse-position>

Pour la fenêtre globale

```
ww=Gui.getMainWindow()
```

```
ww.geometry()
```

Pour la fenetre active

```
ww=doc.ActiveView
```

```
ww.getSize() : donne la taille
```

```
getCursorPos() retourne la position de la souris
```

VII. L'atelier FEM

https://wiki.freecadweb.org/FEM_ConstraintContact

Présentation générale

<https://forum.freecadweb.org/viewtopic.php?t=37162>

La [page officielle](#)

Un [Exemple](#) de maison dont on charge le toit

Un [Exemple de torision](#) dans une vidéo en allemand

<https://www.youtube.com/watch?v=Y9I1zYnMj4g> (allemand)

<https://www.youtube.com/watch?v=dhrynRdBOIg>

Plein de références [sur cette page](#)

Revoir create Task Panel [sur cette page](#)

VIII. En vrac

Comment [construire un fil de fer](#)

Une [histoire de poulies et de longueur de la courroie](#) (intéressant, à voir)

Voir https://wiki.freecadweb.org/OpenSCAD_Module/fr

Une [longue discussion sur les Link](#) dans le forums

Au sujet de [la documentation des scripts/projets Pyhton](#) (en particulier des *docstrings*)

Dans [ce post du forum](#) on parle de fonctions permettant de détecter des points manquants ou non coïncidants

```
skt.detectMissingPointOnPointConstraints()  
skt.makeMissingPointOnPointCoincident()
```

Voir [description générale des objets Part_Feature](#) qui branche en particulier sur [cette page du forum](#) et [celle-là en français](#) où l'on parle des problème d'*Angular déflexion*.

Voir [fonction permettant de transformer une Face en Sketch](#)

Un exemple de [Croix de Malte](#) avec Fusion

Dans FreeCAD il y a un [atelier pyramides et polyhères](#)

Link et clone <https://forum.freecadweb.org/viewtopic.php?t=6662> ????

Une [discussion très intéressante](#) sur le fait que l'on ne peut pas créer un PartDesign Body à partir d'un link de sketch

<https://www.youtube.com/watch?v=3VSIDv2x8tY> (vis et écrou B.A. BA)

Sur le site <https://wayofwood.com> on trouve qq vidéos intéressantes avec *FreeCAD* 0.19

✓ D'abord pour faire [un benchdog](#) (utilisation de l'atelier Part)

✓ Puis pour faire un modèle pour réaliser des [mortaises](#) (utilisation de l'atelier PartDesign)

Revoir tout ce qui est en dessous de cette vidéo <https://www.youtube.com/watch?v=0Mz6o9sqde4>

<https://www.dailymotion.com/video/x4q9ffg> (jolie pièce)

Comment [arranger les formes à découper](#)

Comment [faire du texte à partir d'une image](#)

Faire un [sweep le long d'une courbe](#)

Des [tutos peut-être intéressants](#) en français sur vectorisation, découpe et gravure.

On trouve [sur cette page](#) quelques remarques intéressantes sur le gravage et le mode découpe avec un Z-set volontairement déréglé.

[Faire une « mise en plan » de son objet 3D avec Freecad 17](#) (coupe???)

[Ecrire sur un objet cylindrique \(gobelet, bague...\)](#)

Voir Pyflow ?????

Sur la page concernant les [objets paramétrés](#), on trouve des références vers :

- ✓ une page intéressante sur [l'éditeur de propriétés](#) ; on y voit comme exemple l'explication des propriétés de *vue* et de *données* d'un objet *Part_Design* ;
- ✓ une page expliquant [comment créer des objets par script \(python\)](#) et surtout donnant la liste des propriétés possibles ;
- ✓ une page expliquant [le positionnement des objets](#) dans FreeCAD ;
- ✓ enfin une page sur [l'activation du graphe de dépendance](#).

Le [DAG view](#) est un nouvel outil permettant de mieux voir les relations entre les objets.

- ✓ Voir aussi éventuellement [cette page](#), mais c'est peut-être plus vieux ???
- ✓ Une [page de forum](#) expliquant le DAG.
- ✓ Comparer avec [graphique de dépendance](#) ???
- ✓ https://wiki.freecadweb.org/Feature_editing

<https://forum.freecadweb.org/viewtopic.php?t=14409>

<https://forum.freecadweb.org/viewtopic.php?t=40731>

Une [discussion intéressante](#) sur Placement et Attachement d'un sketch,

Voir aussi https://wiki.freecadweb.org/Std_TreeSyncPlacement ???

La [doc officielle sur Attachment](#) et sur [Placement](#) https://wiki.freecadweb.org/Tasks_Placement

Comment [repositionner un sketch](#)

https://wiki.freecadweb.org/PartDesign_CoordinateSystem

<https://www.freecad.info/index.php/2020/07/01/part-design-map-a-sketch-to-a-face/>

Post de forum intéressant [sur le MapMode](#) et la [liste des vampeurs de MapMode](#)

Une [vidéo pour contruire une 3eme main](#) avec Assembly (muette?)

Revoir <https://www.youtube.com/watch?v=qoutY6HQyBY> ????

Voir cette recherche [cette recherche](#)

https://duckduckgo.com/?q=freecad+datum+plane+vs+shapebinder&atb=v204-1__&ia=web

https://www.youtube.com/channel/UCoe3BcVuLC9I2_yFud5vp8Q/videos

VIII.1. L'atelier LaserCutInterlocking(bof ?)

- ✓ Grâce au menu <Outil><Addon Manager>, j'ai pu installer l'atelier externe *Interlocking laser cut workbench* qui permet de gérer facilement la création de boîtes pour découpeuse laser.
- ✓ Ensuite, j'ai suivi [cette vidéo](#) dont on trouve le texte correspondant [sur cette page](#). Mais je n'ai pas vu la possibilité de paramétrer avec un spreadsheet.
- ✓ On peut aussi regarder [cette page](#)
- ✓ Une [autre utilisation](#) de Laser Cut Interlocking et d'Assembly 2 et l'atelier Path

Mes réglages Vokoscreen

Micro : Audio internet stéréo analogique

Mixer Pusle : Périphérique Entrée / micro (45% environ)