Automatiser les tâches de conception de circuit imprimé : greffons pour KiCAD et FreeCAD

J.-M Friedt, 8 mars 2020
FEMTO-ST/département temps-fréquence, Besançon

KiCAD et FreeCAD convergent pour fournir un environnement cohérent de conception électronique et mécanique assistée par ordinateur. Ces deux outils rendent leurs fonctions accessibles depuis Python, langage permettant d'automatiser un certain nombre de tâches répétitives et donc fastidieuses. Nous proposons de rédiger quelques greffons (*plugins*) pour distribuer des vias le long de lignes de transmsmissions radiofréquences (KiCAD) puis automatiser la réalisation du boîtier contenant un circuit imprimé avec les ouvertures pour ses connecteurs (FreeCAD).

1 Contexte

Convié par le réseau des électroniciens du Centre National de la Recherche Scientifique (CNRS) à présenter les solutions libres de conception électronique et mécanique de circuits imprimés, l'intervention se conclut par une série de questions portant sur des comparaisons avec certaines fonctionnalités des alternatives propriétaires, et en particulier dans le cas de la conception de circuits imprimés radiofréquences. Dans ce contexte, il est classique de distribuer une forte densité de vias entre un plan de masse en face inférieure pour entourer les pistes en face supérieure transportant des signaux radiofréquences et minimiser les risques de couplages par rayonnement en imposant des liaisons de faible impédance vers le potentiel de référence qu'est la masse. Si une fonctionnalité n'existe pas, KiCAD fournit la capacité à combler cette lacune par l'ajout de fonctionnalités au travers de greffon (*plugin*). Comme à peu près tous les logiciels libres actuels (KiCAD, FreeCAD, GNU Radio, QGis ...), l'API est exportée vers Python pour permettre de compléter les fonctionnalités manquantes par des *plugins*.

Nous allons donc explorer l'interaction avec le circuit imprimé (*printed circuit board* – PCB) de KiCAD pour automatiser les tâches répétitives, ce que tout développeur abhorre, et exposer cette fonctionnalité comme greffon afin de partager avec autrui les algorithmes de modification de circuits routés. Dans la continuité de ces efforts, nous aborderons l'automatisation des tâches dans FreeCAD pour la conception automatique du boîtier autour du circuit imprimé : l'ensemble des fonctions accessibles par les icônes est ici aussi exporté sous forme de fonctions Python qui permettront de rationnaliser les étapes de conception de la boîte à partir des contours du circuit imprimé.

Avant de nous lancer dans la programmation, mentionnons en introduction que les fichiers stockés par KiCAD sont en ASCII et limpides à lire :

```
(pad 29 smd rect (at 5.7 -0.8) (size 1.5 0.55) (layers F.Cu F.Paste F.Mask))
...
(fp_line (start 1 0.6) (end -1 0.6) (layer F.Fab) (width 0.1))
...
(via (at 37.1 32.2) (size 0.6) (drill 0.4) (layers F.Cu B.Cu) (net 2))
(segment (start 37.1 32.2) (end 37.2 32.2) (width 0.25) (layer B.Cu) (net 2) (tstamp 59845870))
...
```

et donc à éditer avec son outil favori (`sed`) pour par exemple modifier les largeurs de toutes les pistes ou la couche sur laquelle elles se trouvent.

2 Interaction avec Python : prototypage d'un greffon

KiCAD propose une console Python pour se familiariser avec son interface. La plus grande difficulté rencontrée pour prendre en main cette API a été de trier les informations obsolètes obtenues sur le web [1] pour en extraire le principe sous jacent et le mettre en œuvre avec la version actuelle de KiCAD 5.

Ainsi si nous désirons insérer un texte dans un circuit routé après avoir `apt-get install python3-wxgtk4.0` pour permettre d'accéder à la console Python de KiCAD (`PCBNew` → Tools → Scripting Console), rien n'est plus simple puisque, depuis PCB New qui permet d'éditer son circuit routé, nous fournissons les commandes :

```
import pcbnew
board=pcbnew.GetBoard()
newvia=pcbnew.VIA(board)
newvia.SetLayerPair(pcbnew.PCBNEW_LAYER_ID_START, pcbnew.PCBNEW_LAYER_ID_START+31)
newvia.SetPosition(pcbnew.wxPoint(15000000,15000000))
newvia.SetViaType(pcbnew.VIA_THROUGH)
newvia.SetWidth(1000000)
board.Add(newvia)
pcbnew.Refresh()

txt = pcbnew.TEXTE_PCB(board)
txt.SetText("Hello")
txt.SetPosition(pcbnew.wxPoint(22000000, 15000000))
txt.SetHorizJustify(pcbnew.GR_TEXT_HJUSTIFY_CENTER)
txt.SetTextSize(pcbnew.wxSize(1000000, 1000000))
txt.SetThickness(200000)
board.Add(txt)
pcbnew.Refresh()
```

pour atteindre le résultat (Fig. 1), à savoir ajouter un via ou du texte aux coordonnées fournies en nanomètres (10^{-6} mm) sur le circuit imprimé. Nous avons choisi d'adresser les couches par leur numéro puisque `http://docs.kicad-pcb.org/doxygen/layers__id__colors__and__visibility_8h.html` indique que les identifiants des couches supérieure et inférieure sont `F_Cu = PCBNEW_LAYER_ID_START`, `B_Cu = PCBNEW_LAYER_ID_START+31`

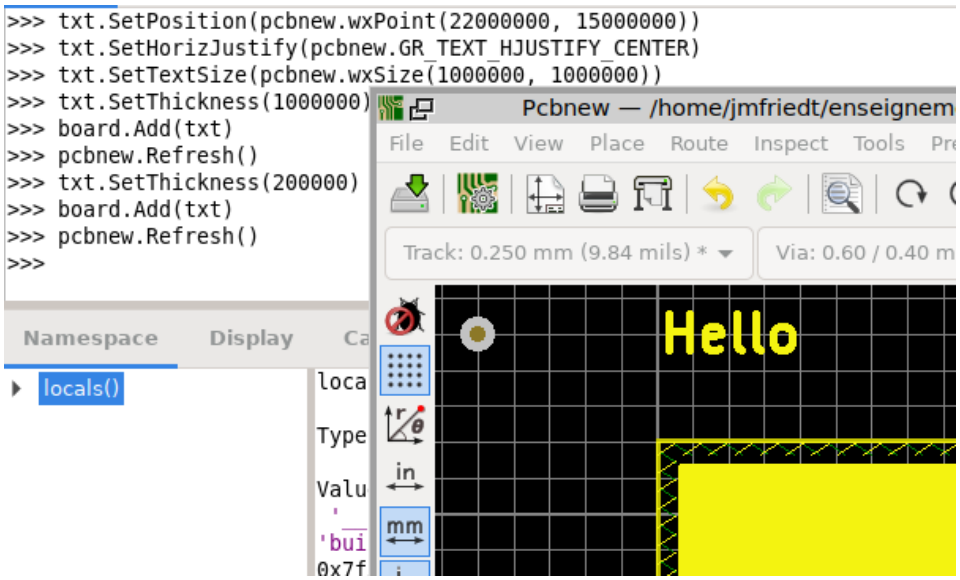

FIGURE 1 – Manipulation d'un circuit routé sous KiCAD au moyen d'un script Python.

Si au lieu d'entrer directement les commandes dans la console nous désirons les éditer dans notre éditeur de texte favori pour rédiger un script `demo_jmf.py` chargé dans la console, alors la première instanciation s'obtiendra par `import demo_jmf` pour ensuite être mise à jour, après avoir chargé les méthodes adéquates par `import importlib`, par `importlib.reload(demo_jmf)`.

La fonctionnalité proposée ci-dessus ne présente évidemment strictement aucun intérêt, sinon de permettre d'appréhender la co-simulation et routage d'antennes planaires [2] au travers de fichiers communs de configuration.

Router des lignes de transmissions radiofréquences en les décorant par des vias à la masse pour les isoler les unes des autres se nomme *via fence*. Cette fonctionnalité, qui distribue des vias connectés à la masse le long de pistes propageant des signaux radiofréquences, est classiquement disponible dans les logiciels propriétaires d'analyse de systèmes radiofréquences tels qu'Altium (`https://techdocs.altium.com//display/ADOH/Via+Shielding` et `https://www.altium.com/documentation/altium-designer/via-stitching-and-via-shielding-ad`) ou Cadence (`https://resources.pcb.cadence.com/blog/the-`

`case-for-stitching-vias-on-your-pcb-2`). Nous n'avons pas identifié de telle fonctionnalité dans KiCAD, et avons du admettre lors des questions en fin de présentation une déficience du logiciel libre face à l'offre propriétaire. Qu'à cela ne tienne, si une fonctionnalité manque dans un logiciel libre, notre devoir est de l'ajouter.

3 Rédaction du greffon

Soit un utilisateur qui a routé une trace et qui désire l'identifier comme une ligne de transmission radiofréquence en l'entourant de vias à la masse. Que faut-il faire pour atteindre ce résultat, et le fournir comme greffon KiCAD utilisable depuis l'interface utilisateur ? Il va nous falloir manipuler des traces, insérer des vias (nous venons de voir comment faire) après avoir calculé les coordonnées le long de la trace, et encapsuler ce programme dans le formalisme d'un greffon. Pour ce faire, nous nous inspirerons de [3] mais surtout [4] qui explicite les interfaces à déclarer. La principale difficulté rencontrée pour passer d'un script fonctionnel dans la console Python de KiCAD à un greffon tient aux indentations : exécuter le greffon sous Python 3 par `python3 -m mon_script.py` permet au moins de déverminer les erreurs de syntaxe et d'indentation.

Nous commençons par mettre en place l'environnement de travail en chargeant les classes et le circuit en cours d'édition, ainsi que quelques constantes que sont la distance entre vias adjacents `dL` et la distance `dr` des vias à la piste que nous désirons décorer (Fig. 2) :

```
import pcbnew
import os
import math
dL=2000000  # spacing between vias
R=2000000 # distance to track
pcb = pcbnew.GetBoard()
```

Nous désirons connecter les vias à la masse : nous recherchons donc le numéro dans la liste des couches de la masse supposée être nommée `GND` :

```
# see https://kicad.mmccoo.com/2017/02/01/the-basics-of-scripting-in-pcbnew/
nets = pcb.GetNetsByName()
gndnet = nets.find("GND").value()[1]
gndclass = gndnet.GetNetClass()
print(str(gndnet.GetNetname())+" "+str(gndnet.GetNet()))
```

Munis de cette information, nous commençons la rédaction de la classe afin de la rendre compatible avec les méthodes attendues d'un greffon :

```
class TransmissionLine(pcbnew.ActionPlugin):
 def defaults(self):
  self.name = "TransmissionLine"
  self.category = "A descriptive category name"
  self.description = "A description of the plugin and what it does"
  self.show_toolbar_button = True # Optional, defaults to False
  self.icon_file_name = os.path.join(os.path.dirname(__file__), 'transmission_line.png'→
      ↪)
```

et commençons à appréhender le sujet à résoudre en parcourant la liste des pistes (`for track in pcb.GetTracks():`) et traiter celles qui sont sélectionnées (`track.IsSelected()` :

```
 def Run(self):
  pcb = pcbnew.GetBoard()
  print("Transmission: "+str(dL)+" "+str(R))
  for track in pcb.GetTracks():
   start = track.GetStart()
   end = track.GetEnd()
   if track.IsSelected():
```

Pour chaque trace sélectionnée, nous calculons le nombre `n` de vias qu'il faudra placer, et distribuons ces vias uniformément le long du segment en suivant l'équation de droite correspondante, en prenant soin de traiter séparément le cas des lignes verticales pour lesquelles le coefficient directeur `a` de la droite est infini :

```
    n=math.floor(track.GetLength()/dL)                    # nombre de vias
    if ((track.GetEnd().x-track.GetStart().x) != 0):  # si la trace n'est PAS verticale→
        ↪ ...
     a=(track.GetEnd().y-track.GetStart().y)/(track.GetEnd().x-track.GetStart().x)
     b=track.GetStart().y-a*track.GetStart().x          # y=ax+b
     theta=math.atan(a)                                 # cartesien -> polaire
     x=min(track.GetStart().x,track.GetEnd().x)         # point de depart
```

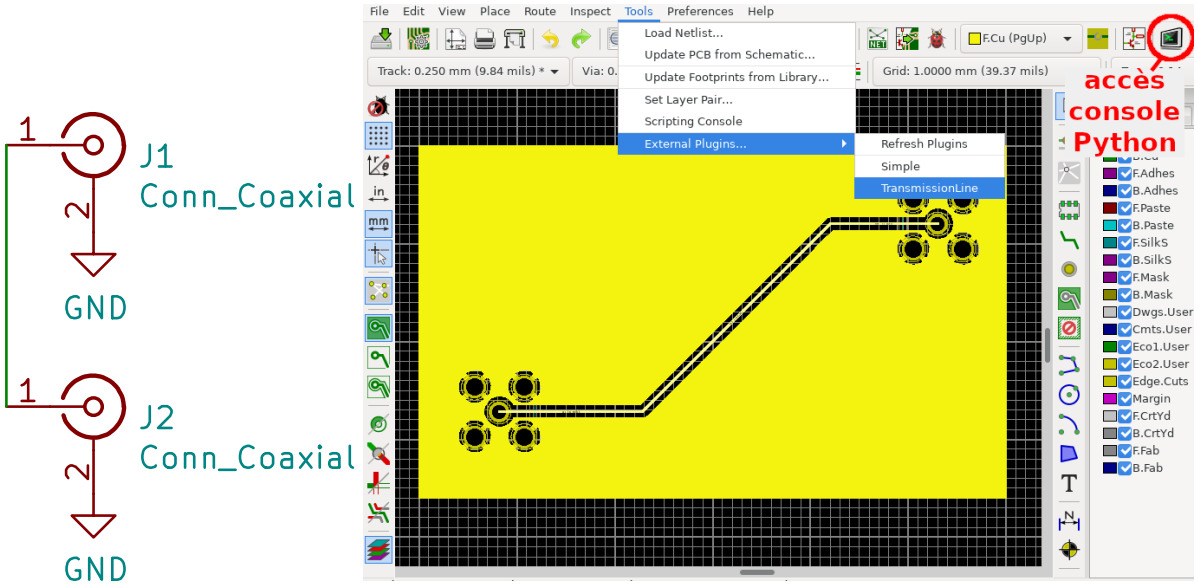

FIGURE 2 – Schéma de deux connecteurs coaxiaux reliés par une ligne de transmission (gauche), et routage avant décoration de la ligne de transmission par les vias. Noter que la piste a été sélectionnée et apparaît plus claire que le plan de masse qui l'entoure.

```
    while x<max(track.GetStart().x,track.GetEnd().x):
     yp=a*x+b+R*math.sin(theta+3.14159/2)
     xp=x+R*math.cos(theta+3.14159/2)
```

Une fois les coordonnées du via identifiées, celui-ci est ajouté comme nous l'avons vu auparavant entre les couches supérieure et inférieure et en assignant le via à la masse par `newvia.SetNetCode(gndcode)` :

```
     newvia=pcbnew.VIA(pcb)
     newvia.SetLayerPair(pcbnew.PCBNEW_LAYER_ID_START, pcbnew.PCBNEW_LAYER_ID_START→
         ↪+31)
     newvia.SetPosition(pcbnew.wxPoint(xp,yp))
     newvia.SetViaType(pcbnew.VIA_THROUGH)
     newvia.SetWidth(1000000)
     newvia.SetNetCode(gndcode)
     pcb.Add(newvia)
     x=x+dL*math.cos(theta)
```

Nous laissons le lecteur reproduire ce mécanisme pour le via qui se trouve de l'autre côté de la piste ou traiter le cas des pistes verticales qui réutilisent les mêmes commandes avec des coordonnées (x_p,y_p) calculées un peu différemment (la solution se trouve sur le dépôt github cité en fin de conclusion). Après ajout de chaque via, nous incrémentons la position `x=x+dL*math.cos(theta)`.

Une fois les vias ajoutés le long de la ligne de transmission, nous mettons à jour le dessin du circuit imprimé `pcbnew.Refresh()`.

Le greffon est initialisé par `TransmissionLine().register()` qui conclut le fichier que nous constatons être finalement de structure fort simple. Les fonctionnalités basiques du greffon sont fonctionnelles (Fig. 3) malgré quelques défauts qui mériteraient à être corrigés pour le rendre réllement utilisable, en particulier le recouvrement de via dans les angles. On pourra aussi noter dès maintenant que les traces arrondies sont promises pour la prochaine mouture de KiCAD de la fin d'année, et qu'il faudra trouver un algorithme plus intelligent que la sub-division des segments en sous-segments équidistants. Nous verrons en conclusion que la relève est déjà prise.

En prenant soin de placer le script Python dans le répertoire `/usr/share/kicad/scripting/plugins`, nous éviterons même de devoir chercher le script dans la liste des outils mais pourrons nous contenter de cliquer sur une icône ajoutée au menu des actions sur la barre en haut de l'interface graphique (Fig. 3, indications rouges).

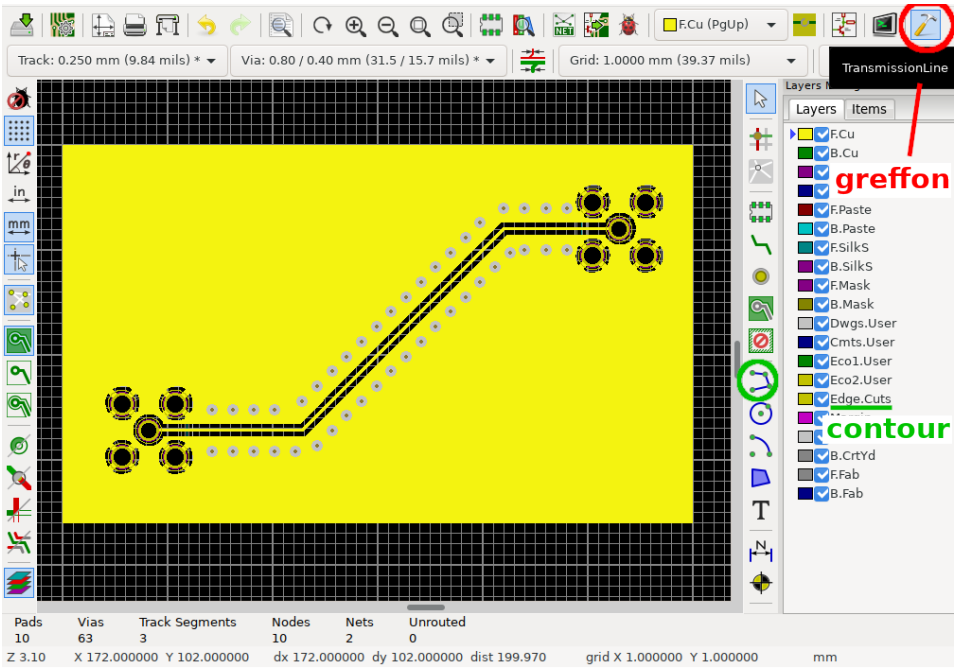

FIGURE 3 – Résultat de l'exécution du greffon pour décorer la ligne de transmission de vias. Quelques vias qui se recouvraient dans les angles ont été éliminés manuellement. En rouge l'emplacement de l'icoône du greffon. En vert les éléments graphiques nécessaires à définir manuellement le contour du circuit imprimé qui sera nécessaire dans FreeCAD.

4 Greffon pour FreeCAD

Nous venons de finir de concevoir un circuit, et maintenant il faut le mettre en boîte pour le protéger de son environnement. La symbiose entre KiCAD et FreeCAD permet de facilement appréhender la conception du boîtier à partir du circuit imprimé, et automatiser la procédure par des programmes Python. FreeCAD, et notamment la version 0.18.4 que nous utilisons, exposant son API (et celle d'OpenCASCADE sur lequel il s'appuie [5]), ce que nous venons d'apprendre sur les greffons KiCAD est presque directement applicable à FreeCAD. Voyons comment importer le circuit (et ses connecteurs) de KiCAD dans FreeCAD, et automatiser la conception de la boîte quelque-soit la forme du circuit imprimé. Seule contrainte, nous devons penser dans KiCAD (Fig. 3, indications vertes) à inclure une couche de détourage *Edge.Cuts* qui est de toute façon nécessaire si nous exportons un fichier Gerber pour réaliser le circuit imprimé en gravure anglaise (fraisage des pistes).

Un greffon FreeCAD externe a été rédigé (en Python) pour importer directement un fichier KiCAD (`https://github.com/easyw/kicadStepUpMod`) qui s'installe par "Tools"→"Addon Manager"→ "Workbenches"→ "kicadStepUpMod". Alors que jusqu'alors il fallait exporter un fichier VRML ou Step de KiCAD pour l'importer dans FreeCAD, il est désormais possible de directement importer le fichier PCB en cliquant sur "Load KiCAD PCB Board and Parts" (troisième icône) qui en pratique fait appel à la fonction `onLoadBoard`. Nous chargeons le circuit imprimé et, si les modèles de connecteurs sont disponibles, les fichiers Step associés. Notre objectif est de nous appuyer sur ce greffon pour automatiser la création d'une boîte permettant d'encapsuler le circuit imprimé tout en laissant traverser les connecteurs.

Nous constatons au chargement du circuit imprimé qu'un *Sketch* est associé au circuit imprimé : c'est la forme du circuit tel que déterminé par le couche *Edge.Cuts*. Nous en profitons pour utiliser le "2D Offset" du *Part Workbench* qui effectue l'homothétie d'une courbe. Nous allons procéder deux fois à cette opération, une première fois pour laisser un peu de jeu entre le circuit imprimé et la face interne du mur, puis une seconde fois pour définir l'épaisseur de la boîte. Lors du passage de la courbe 2D à 3D par extrusion ("Part Workbench"→"Extrude a Selected Sketch"), nous voulons que la surface externe soit un peu plus basse que la surface interne pour définir l'épaisseur du fond du boîtier : la courbe extérieure obtenue par la double homothétie de la forme du circuit imprimé est abaissée de 1 mm. Une fois les deux

courbes extrudées, il reste à soustraire le volume intérieur du volume extérieur pour obtenir une boîte. Cette séquence d'opérations s'automatise par le script Python suivant qui suppose que le circuit imprimé a été chargé et a donné lieu à la création de la variable *PCB_Sketch_387*.

```
import FreeCAD
import Part
import kicadStepUptools
# kicadStepUptools.onLoadBoard('demo.kicad_pcb',None,False)
contour=FreeCAD.ActiveDocument.PCB_Sketch_387
finner =FreeCAD.ActiveDocument.addObject("Part::Offset2D", "Offset2D")
finner.Source = contour     # mur interieur
finner.Value = 1.0          # position relative au PCB (ajustement)
finner.Placement.Base.z=-1
fouter = FreeCAD.ActiveDocument.addObject("Part::Offset2D", "Offset2D")
fouter.Source = finner      # mur exterieur
fouter.Value = 1.0          # epaisseur du mur
fouter.Placement.Base.z=-1 # relatif a finner = epaisseur du fond
murint = FreeCAD.ActiveDocument.addObject('Part::Extrusion', 'Extrude')
murint.Base=finner
murint.LengthFwd=10.0       # hauteur de la boite
murint.Solid=True
murext = FreeCAD.ActiveDocument.addObject('Part::Extrusion', 'Extrude')
murext.Base=fouter
murext.LengthFwd=10.0       # hauteur de la boite
murext.Solid=True
boite=FreeCAD.ActiveDocument.addObject("Part::Cut","Cut")
boite.Base = murext         # soustrait l'interieur de la boite (creuse)
boite.Tool = murint         # ... de l'exterieur
```

Tout comme pour KiCAD, ce script peut soit s'exécuter dans la console Python ("View" → "Panels" → "Python Console") ou se charger depuis cette même console en sauvant le script dans un fichier (`import boite` si le nom du fichier est `boite.py`). La principale difficulté dans cet exercice est de pallier l'absence quasi totale de documentation par la lecture des sources sur le dépôt github du greffon, mais il s'agit sinon de programmation FreeCAD on ne peut plus standard avec la manipulation des divers champs de position des objets et leur assemblage par opérations booléennes (Fig. 4) tel que discuté en détail dans [6].

Plus intéressant, la gestion des connecteurs. Nous ne savons pas à priori combien il y a de connecteurs ni où ils se trouvent. Nous proposons la démarche suivante : sachant que les connecteurs sont les seuls éléments dépassant du capot, nous plaçons exprès ce dernier au-dessus de la boîte où il n'intersecte que les connecteurs et aucun composant ou circuit imprimé. Nous itérons sur tous les connecteurs supposés être des objets FreeCAD importés en même temps que le circuit imprimé (attribut `Part::Feature`) : pour chaque connecteur, nous cherchons l'intersection avec le capot, effectuons une homothétie de cette courbe d'intersection pour laisser un peu de jeu entre le capot et le connecteur, et finalement effectuons la soustraction de l'ouverture du connecteur avec le capot.

Ces opérations se résument tout d'abord par la création du capot, ici à une altitude arbitraire de 10 mm

```
chapeau=FreeCAD.ActiveDocument.copyObject(contour,True)
chapeau.Placement.Base.z=10  # hauteur du capot
chapeau.Label="chapeau"
chapeaularge = FreeCAD.ActiveDocument.addObject("Part::Offset2D", "Offset2D")
chapeaularge.Source = chapeau #
chapeaularge.Value = 2.0          # epaisseur du mur
chapeauplein= FreeCAD.ActiveDocument.addObject('Part::Extrusion', 'Extrude')
chapeauplein.Base=chapeaularge
chapeauplein.Solid=True
chapeauplein.LengthFwd=1.0        # epaisseur du capot
```

Ensuite, nous itérons sur les connecteurs pour trouer le capot aux bons endroits. Étant donné que seul un motif peut être fourni comme argument d'une opération booléenne, nous mémorisons l'état du capot dans une liste `t` qui s'incrémente à chaque nouvelle opération :

```
t=list()
t.append(chapeauplein)
objects = FreeCAD.ActiveDocument.findObjects("Part::Feature")
for object in objects:
 wires=object.Shape.slice(FreeCAD.Base.Vector(0,0,1),10.0)
 comp=Part.Compound(wires)
 if (len(wires)>0):
```

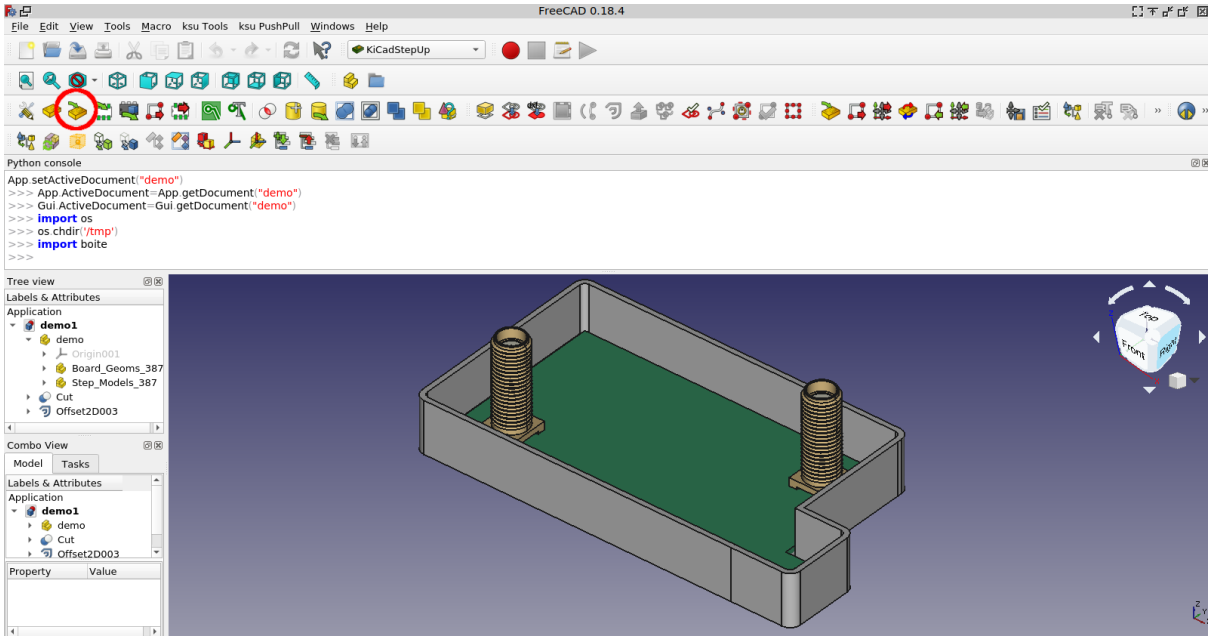

FIGURE 4 – La boîte automatiquement créée à partir du circuit imprimé chargé par l'icône encadrée en rouge. En haut, la console Python pour charger le script supposé ici être localisé dans `/tmp`. Une excroissance a volontairement été ajoutée en bas à droite du circuit imprimé pour démontrer la flexibilité de l'approche.

```
  slice=FreeCAD.ActiveDocument.addObject("Part::Feature","nouveau")
  slice.Shape=comp
  trou = FreeCAD.ActiveDocument.addObject("Part::Offset2D", "Offset2D")
  trou.Source = slice
  trou.Value = 1.0                    # agrandit le trou
  trouplein= FreeCAD.ActiveDocument.addObject('Part::Extrusion', 'Extrude')
  trouplein.Base=trou              # donne une epaisseur
  trouplein.Solid=True               # DOIT etre un volume pour etre soustrait
  trouplein.LengthRev=2.0            # epaisseur du capot
  trouplein.LengthFwd=2.0            # epaisseur du capot
  tmpchapeau=FreeCAD.ActiveDocument.addObject("Part::Cut","Cut")
  tmpchapeau.Base = t[-1]
  tmpchapeau.Tool = trouplein
  t.append(tmpchapeau)

FreeCAD.ActiveDocument.recompute()
```

Pour que le cylindre représentant le connecteur puisse être soustrait du capot, il faut absolument qu'il s'agisse d'un volume (et non d'une surface comme nous obtenons si nous oublions l'attribut `trouplein.Solid=True`).

Le résultat est acceptable (Fig. 5), même si un petit épaulement permettrait de maintenir le capot en place : nous laissons cette optimisation comme exercice au lecteur.

Il ne reste plus qu'à réaliser la boîte – ici en impression additive sur une imprimante UltiMaker 2+ – et constater le parfait alignement des ouvertures avec les connecteurs (Fig. 6).

5 Conclusion

La beauté du logiciel libre est de pouvoir contribuer aux lacunes des outils proposés pour combler des points de détails qui n'auraient pas été abordés par les développeurs principaux du logiciel. Au travers des greffons (*plugin*), il est possible d'ajouter des fonctionnalités périphériques sans être obligé de rentrer dans le cœur du code principal du logiciel. Nous avons illustré ce concept pour KiCAD, logiciel de conception de circuits imprimés, dont les développeurs n'ont pas nécessairement abordé toutes les attentes de la conception de circuit radiofréquences, ainsi que pour FreeCAD, logiciel de conception mécanique. Alors que nous étions en marche de combler cette lacune, une alternative plus stable et complète est apparue

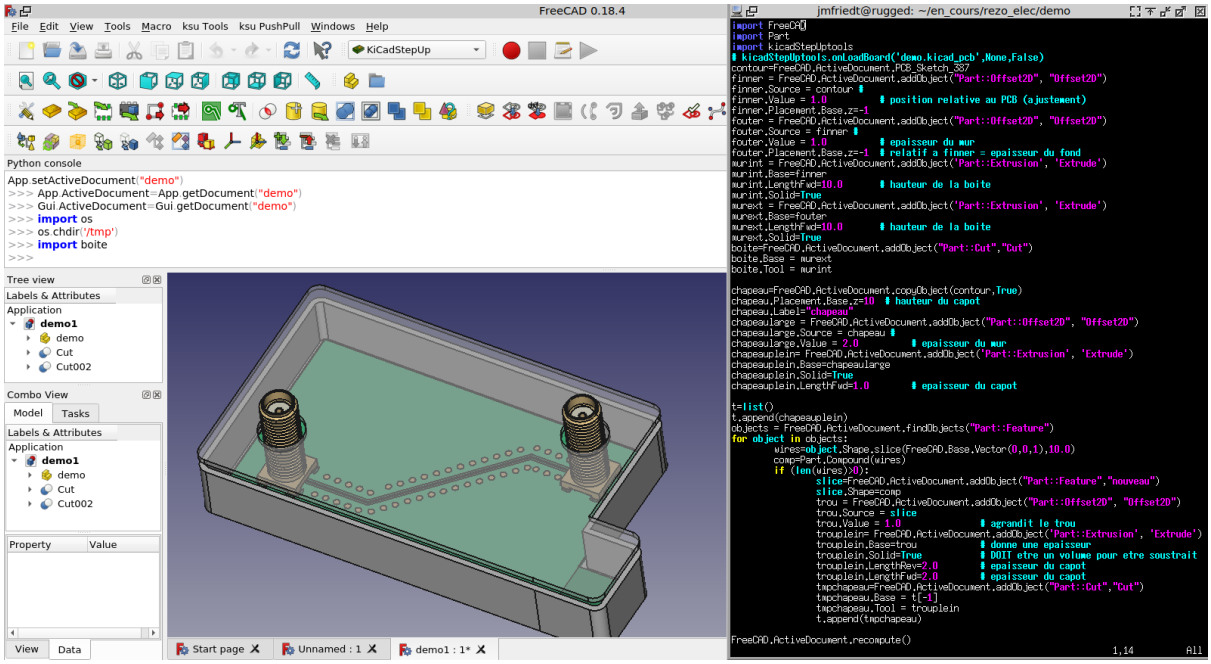

FIGURE 5 – Le capot percé des passages de connecteurs SMA. Ce rendu en transparence du capot permet de visualiser la piste décorée de ses vias qui a servi de prétexte à ces exercices. À droite, la cinquantaine de lignes de code Python qui ont permis d'automatiser cette séquence.

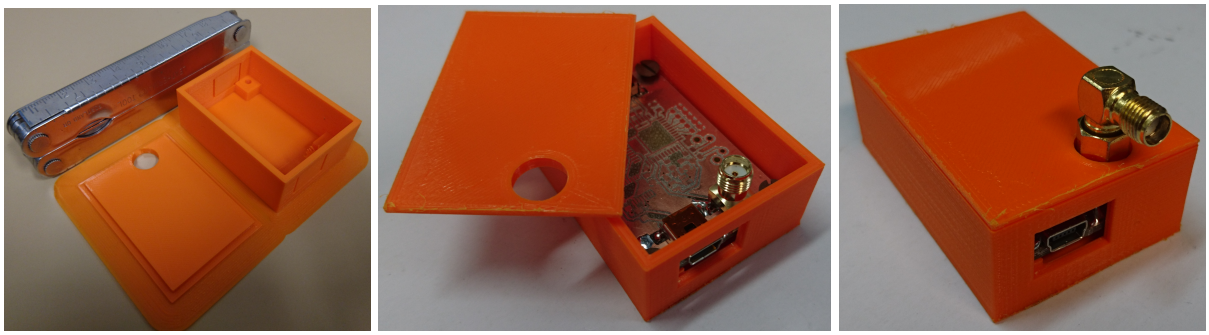

FIGURE 6 – Exemple de boîte réalisée pour laisser les ouvertures vers les connecteurs en face supérieure (SMA) et en face avant (USB) en cohérence avec le circuit imprimé.

sur le web sous la forme des greffons RF-tools-KiCAD disponibles à `https://github.com/easyw/RF-tools-KiCAD` et annoncés à `https://hackaday.com/2019/11/19/kicad-action-plugins/`.

Tous les codes associés au greffon KiCAD décrits dans cet article sont disponibles à `https://github.com/jmfriedt/kicad_transmissionline`.

Remerciements

É. Carry (FEMTO-ST, Besançon) a réalisé l'impression de la boîte et relu le manuscrit. Cette étude a été motivée par le séminaire organisé par le réseau des électroniciens du CNRS à Strasbourg fin 2019.

Références

[1] `https://kicad.mmccoo.com/kicad-scripting-table-of-contents/` (pour KiCAD 4)

[2] J.-M Friedt, É. Carry, O. Testault, *Petites antennes réalisées par impression additive : de la conception à la visualisation des diagrammes de rayonnement (en vrai et en virtuel)*, Hackable **31**, pp.80-96 (Oct-Nov. 2019)

[3] `https://github.com/svofski/kicad-teardrops`

[4] `http://docs.kicad-pcb.org/doxygen/md_Documentation_development_pcbnew-plugins.html`

[5] Y. van Havre & B. Collette, *Open-source design ecosystems around FreeCAD*, FOSDEM 2020 à `https://fosdem.org/2020/schedule/event/freecad/`

[6] J.-M Friedt, *Dessiner des carrés avec des ronds : simulation d'un ordinateur mécanique en scriptant sous FreeCAD*, Hackable (à paraître)