Reinders
Aymerick

Master 1 Physique et Ingénierie du vivant

# Rapport de stage :

## Programmation d'une imprimante 3D via Python



Faculté des Sciences
Université de Montpellier

Tuteurs : Marcelo Nöllmann, Jean-Bernard Fiche
Professeur responsable : Andrea Parmeggiani

# Sommaire

# I. Introduction

Les appareils de mesure sont en constante évolution dans la recherche en Biologie. Ainsi, un étudiant ayant effectué le Master physique et ingénierie du Vivant sera amené à programmer certains d'entre eux au cours de sa carrière.

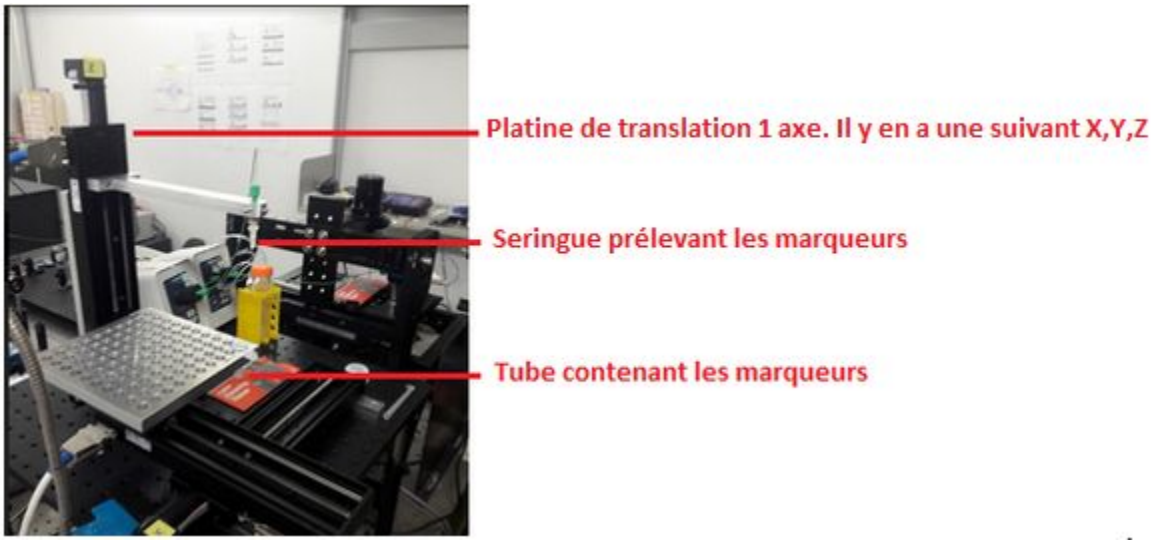
Mon stage s'est déroulé au Centre de Biochimie Structurale (CBS) de Montpellier , dans le groupe de Marcelo Nöllmann. L'activité de recherche de ce groupe se focalise sur l'étude de la ségrégation et du remodelage de l'ADN. Plusieurs organismes modèles sont étudiés au laboratoire : les bactéries (*Bacillus subtilis*, *Myxococcus xanthus,* etc.) et la mouche *Drosophila melanogaster*.

J'ai été encadré par Jean-Bernard Fiche (Ingénieur de recherche INSERM) et Marcelo Nöllmann (Directeur de recherche CNRS). Le groupe de M.Nöllman compte plusieurs microscopes utilisant la fluorescence pour imager les protéines ou l'ADN dans ces organismes.

Je vais me concentrer sur un montage particulier qui a pour but d'étudier les propriétés du vivant via la technique du FISH (Fluorescent In Situ Hybridization). Le principe de cette technique est assez simple, on va chercher à réaliser une hybridation entre une séquence d'ADN (ou d'ARN) cible et une sonde spécifique de cette séquence. Afin de pouvoir voir ses sondes par la suite, on va les marquer avec un marqueur fluorescent.

Dans notre cas, la partie ADN+sonde est réalisée au cours de la préparation de l'échantillon. En revanche, les marqueurs sont injectés par la suite par le manipulateur. Ma tâche va être de programmer une imprimante 3D afin que celle-ci puisse réaliser différentes actions (prélever/administrer un certain volume de marqueur) à des coordonnées précises. [1]

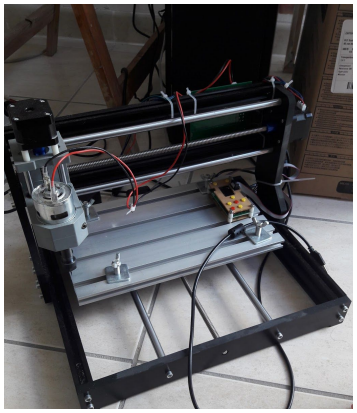
Ce rapport va donc présenter mon cheminement au cours du stage. Il présentera aussi les difficultés que j'ai rencontrées et les nouvelles connaissances que j'ai acquises.




*(Figure 1 : Photo du montage utilisant la technique du FISH programmé sous LabView)*

# II. Etat de l'art

## A. Présentation de l'imprimante 3D et choix du langage de programmation

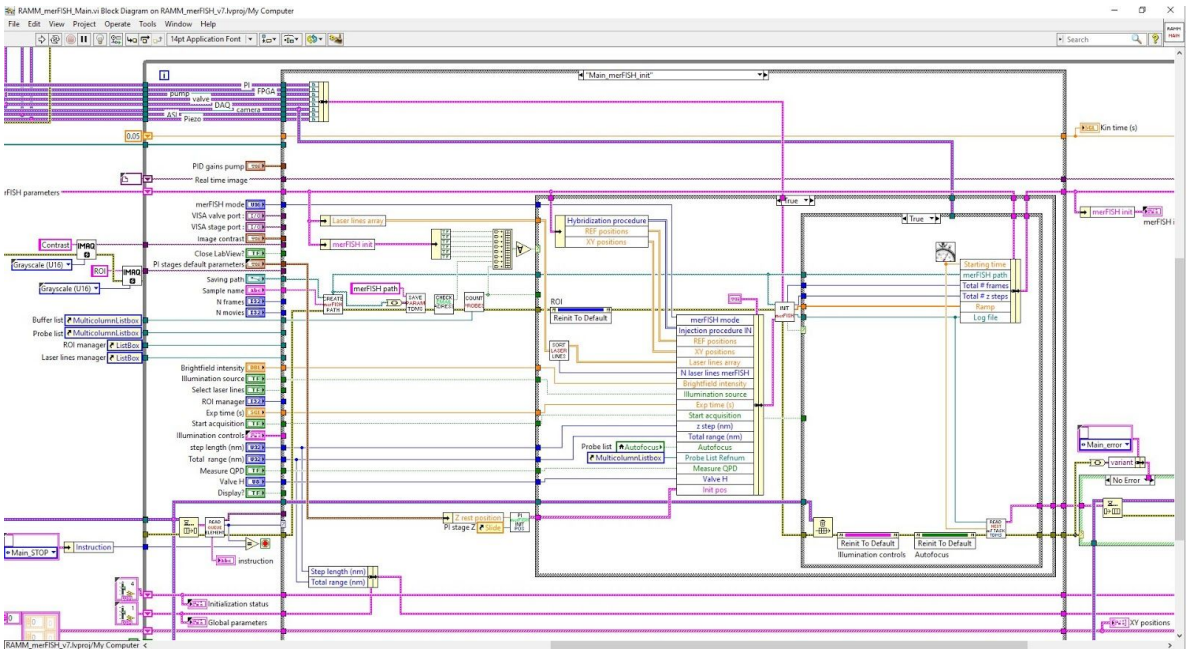


Mon stage c'est concentré exclusivement sur la troisième version du montage présenté en introduction.
Cette version va utiliser une imprimante 3D, elle permettra d'injecter des sondes fluorescentes dans une cellule fluidique en étant couplée avec une pompe et des valves.

Cette version diffère des 2 précédentes par le fait que le code assurant le contrôle de l'instrument sera exclusivement développé sous Python.

*(Figure 2 : Imprimante utilisée au cours de mon stage)*

Mais alors pourquoi changer de langage de programmation ?
En effet, les deux premiers montages utilisent LabView et sont parfaitement fonctionnels. Néanmoins, il est plus difficile de comprendre un programme LabView et de le modifier (de par l'absence de notations claires et de l'enchevêtrement de câbles) qu'un code Python.
Python a aussi l'avantage d'être open source et dispose d'un grand nombre de documentation sur internet. Enfin, l'analyse des images étant également codée sous Python, un unique langage de programmation sera utilisé à la fois pour l'acquisition des données et pour l'analyse de ces dernières.



*(Figure 3 : Programme LabView utilisé dans les 2 premières versions du dispositif)*

### B. Résultats des recherches précédentes

L'objectif était donc de réaliser un programme python afin de contrôler une imprimante 3D-mill. Cependant, je ne partais pas de rien. En effet, une première prise en main de l'instrument avait été réalisée et documentée par M.Nöllmann. Je disposais donc d'un document qui m'a permis de faire de la recherche bibliographique durant la première partie du stage. En particulier, j'avais à ma disposition :

1. Un programme Python [2 & Annexe 1] très simple permettant de contrôler l'imprimante.
2. Un document m'expliquant comment installer "bCNC" [3]. Il s'agit d'une interface graphique permettant de contrôler des machine-outils telles que l'imprimante utilisée pendant mon stage.

La difficulté de ce projet est le fait que l'imprimante fonctionne suivant un code qui lui est propre: Le G-code. Ce langage est bien documenté et est commun à la plupart des imprimantes 3D [4 & Annexe 2].
Il permet de réaliser de la programmation de commande numérique afin de définir des séquences d'instructions pour piloter l'imprimante.

Par exemple, la commande "Décale la tête de 5mm en X" s'écrira "G00 G21 G91 G17 X5" avec :

- G00 Déplacement rapide
- G21 Programmation en mm
- G91 Déplacement en coordonnées relatifs
- G17 Sélection du plan XY
- X5 Déplacement de 5mm suivant l'axe X

J'ai aussi utilisé des commandes particulières, connues sous le nom de "Real Time Commands" comme la commande "?" qui renvoie les coordonnées absolues, relatives ainsi que l'état (inactif ou non) de l'imprimante.

### C. Choix de la création d'un nouveau programme

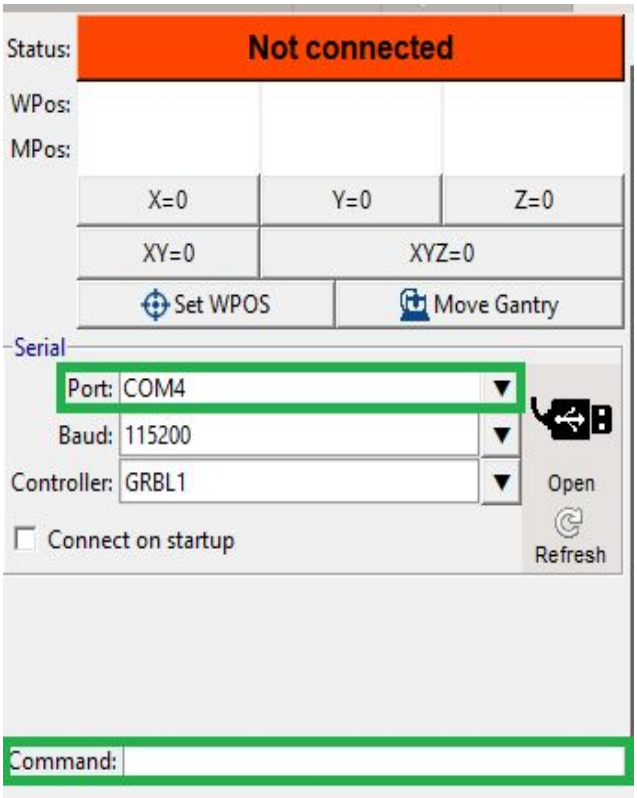
Une question que l'on peut se poser est pourquoi créer un nouveau programme alors qu'il en existe déjà un fonctionnel ? Le programme disponible sur internet n'est pas simple d'utilisation. En effet, il oblige l'utilisateur à modifier un document texte manuellement et surtout, il ne peut pas être intégré dans de plus gros programmes contrôlant plusieurs instruments en parallèle (caméra, platine de translation, pompe,...) .
L'objectif du stage était donc de créer un programme s'occupant lui même de la traduction : Action souhaitée par l'utilisateur → G-code. Pour ça, l'objectif était de créer une classe définissant toutes les opérations de base permettant de communiquer et contrôler l'imprimante. Elle devait être écrite sous Python et facilement intégrable dans un programme plus vaste qui contrôlerait d'autres instruments (caméra, pompe, etc…). Je devais enfin créer une documentation claire qui pourrait être utilisée par n'importe qui pour retrouver les informations concernant l'instrument et le programme.

# III. Méthodes

## A. Contrôle du bon fonctionnement de l'imprimante

Dans un premier temps, il était important de s'assurer du bon fonctionnement du matériel. Pour ça, j'ai installé Python version 3.7 et bCNC via le terminal anaconda prompt. bCNC est une application codée sous Python permettant de communiquer avec l'imprimante par l'intermédiaire de commandes G-codes.

L'interaction avec l'imprimante se fait par une communication série classique. Ainsi, les données sont transmises les unes après les autres. Elle s'oppose à la transmission parallèle, qui transmet simultanément les éléments d'information sur plusieurs voies. L'imprimante ne peut donc traiter qu'une seule information à la fois.




*(Figure 4 : Interface bCNC au démarrage)*

Une fois l'application lancée, on constate qu'il faut sélectionner le port de communication et ouvrir le lien entre l'imprimante et l'ordinateur. Ce paramètre est important car il faudra indiquer le port de l'imprimante dans le programme Python afin de pouvoir communiquer avec celle-ci.

GRBL quant à lui est un contrôleur permettant de contrôler les mouvements des imprimantes fonctionnant en port série. Le Baudrate (115200) indique la rapidité de modulation en communication de données. Il correspond en inverse à la durée en secondes du plus court élément du signal.

En tapant diverses commandes G-codes dans l'onglet "Command", j'ai pu m'assurer du bon fonctionnement de l'imprimante et des diverses instructions que celle-ci pouvait exécuter.

Par exemple, la commande "G00 G91 G21 X10" m'a permis de faire un déplacement relatif de 10 mm suivant l'axe X. Tandis que la commande "G28" réinitialise les axes à leurs origines.

### B. Test avec Python et le programme existant

Une fois que je me suis assuré du bon fonctionnement de l'imprimante et de la possibilité de communiquer avec elle, j'ai testé le programme Python fourni par mes tuteurs [2].
Celui-ci ne semblait pas fonctionner. En effet, les diverses commandes G-code mises dans le dossier "circle.nc" ne s'éxécutaient pas .

Je me suis alors demandé pourquoi le code ne fonctionnait pas alors que la totalité du G-code pouvait s'exécuter dans l'application bCNC. Je me suis finalement rendu compte que le problème venait du délais entre l'envoie des commandes G-codes et la fermeture de la communication avec le port série.

En effet, en fermant la communication directement après l'envoie des commandes, celles-ci n'avaient pas le temps d'être prises en compte par l'imprimante et donc exécutées par l'imprimante. J'ai donc pallié à ce problème en ajoutant l'instruction "time.sleep" avant la fermeture du port série et du fichier contenant le G-code.

```
29    # Close file and serial port
30    time.sleep(5)
31
32    f.close()
33
34    s.close()
35
```

*(Figure 5 : Programme d'origine modifié afin d'attendre 5 secondes avant la fermeture du port et du fichier contenant les G-codes)*

### C. Architecture souhaitée

Comme dit précédemment, le programme Python existant présentait plusieurs défauts (utilisateur devant coder lui même en G-code par exemple). Afin de palier à ces divers défauts et de rendre l'imprimante plus facile à utiliser, il fallait choisir une architecture particulière.

L'architecture retenue était l'architecture Modèle-vue-contrôleur (MVC)[4]. L'idée de cette architecture est de faciliter la lecture du programme et les échanges de matériel. Dans la partie contrôleur, on retrouve des programmes très simples qui seront spécifiques d'un instrument et très général (action de base). Ces programmes seront ensuite utilisés par les programmes stockés dans Modèle pour réaliser des expériences plus complexes (par exemple coupler l'imprimante et une pompe pour injecter des produits). Enfin, la partie Vue fait référence aux interface graphique (comme le bCNC).

De cette façon, si un jour on change de matériel, on n'aura pas besoin de modifier les méthodes de la classe.

Dans notre cas, l'utilisateur envoie une instruction que doit exécuter l'imprimante. Cette instruction est ensuite décodée via diverses méthodes afin que celle-ci puisse être envoyée à l'imprimante sous forme de G-code. Enfin, une fois l'instruction réalisée par l'appareil, on renvoie à l'utilisateur un message indiquant la bonne exécution de sa demande.

Afin de pouvoir réaliser cette architecture au mieux, il a donc fallu créer diverses méthodes (**OpenConncetion( )**,**CloseConncetion( )**,....) pour pouvoir dans un premier temps ouvrir et fermer la communication série avec l'imprimante. Puis dans un second temps des méthodes permettant de demander des informations à l'imprimante sur sa position et son statut (**Status ( )**,**Read_MPos ( )**,...) et de contrôler le déplacements des moteurs (**Move ( )**).

Pour qu'on puisse retrouver facilement les commandes envoyées à l'imprimante et leur bon déroulement ou non, on enregistre celles-ci dans un fichier texte ("log_file").

### D. Ecriture de la classe CNC ( )

L'objectif était de créer des méthodes basiques afin que celles-ci puissent être utilisées dans des programmes plus complexes pour pouvoir communiquer avec l'imprimante. Dans un premier temps, on commence par initialiser la classe **CNC ( )**

```
32  class CNC:
33
34      def __init__(self,port):
35          """
36          For the class instanciation, the input variable port is indicating which
37          COM port the 3D-Mill is connected to.
38
39          Parameters
40          ----------
41          port : Name of the port where the printer is connected
42
43          Returns
44          -------
45          None.
46
47          """
48          self.port = port
49          self.log_file=open("log_file.txt","w")
50          self.PrinterState={}
51          return
```

*(Figure 6 : Initialisation de la classe **CNC ( )** )*

On commence par définir la variable globale self.port qui gardera en mémoire le port série auquel l'imprimante est connecté (Ligne 48). L'utilisateur devra donc renseigner cette information lorsqu'il fera appel à la classe **CNC ( )**. L'instruction "self." permet que la variable obtenue soit soit accessible à toutes les méthodes. On ouvre ensuite le fichier texte (" log_file.txt") qui garde en mémoire les actions demandées et leur bon déroulement ou non (Ligne 49).Enfin, on définit un dictionnaire qui contiendra des informations concernant le statut de l'imprimante (Ligne 50).

### 1. Ouverture et Fermeture de la connexion série

```
53      def OpenConnection(self):
54          """
55          The OpenConnection method is opening the serial communication port and
56          keeps the output in self.
57          The time delay of 2s is important to keep, else the communication will
58          not be established and it won't be possible to control the 3D-mill.
59          When an error occurs, Python will normally stop and generate an error message.
60          We will change this message by using "try" and "except"
61
62          Returns
63          -------
64          None.
65
66          """
67          try:                                        # The try block lets you test a block of code for errors.
68              self.s  = serial.Serial(self.port,115200);  # Open the port and keeps the output in self.s
69              self.log_file.write("Port "+self.port+" ouvert\n")
70              time.sleep(2)
71              self.s.flushInput()                     # Remove data from input buffer
72          except serial.SerialException:              # The except block lets you handle the error.
73              print("The port "+self.port+" is already opened or is not connected ")
74              self.log_file.write("The port "+self.port+" is already opened or is not connected ")
75              pass
```

*(Figure 7 : Méthode **OpenConnection( )** )*

La méthode **OpenConnection( )** ouvre donc la communication avec l'imprimante. Ce qui était fait dans le programme de départ via la ligne de code :

```
s = serial.Serial('COM4',115200);
```

*(Figure 8 : Commande d'ouverture du port 'COM4' (Ligne 10 du code de départ, Annexe 1))*

On retrouve cette instruction à la ligne 68 avec l'utilisation de la variable *"self.port"* définie précédemment. Ici, le package Serial.py permet la communication avec les périphériques. On fixe le baud rate à 115200 car il est propre à notre imprimante. La variable ainsi obtenue se nomme alors "self.s".

Ensuite, on enregistre un message dans le fichier "log_file" comme quoi le port a bien été ouvert (Ligne 69), on attend 2 secondes pour l'initialisation (Ligne 70) et on vide le buffer afin de s'assurer que celui-ci soit prêt à recevoir des instructions.(Ligne 71).

Mais que se passe t il si l'utilisateur commet une erreur sur l'identification du port ou que celui-ci était déjà ouvert ou encore que l'imprimante n'est pas connectée ou fonctionnelle ?

Plutôt que de simplement utiliser le message d'erreur prédéfini par Python qui n'est pas forcément clair, on va remplacer celui-ci grâce aux fonctions "try" et "except" (Ligne 67/72).

Si les instructions dans le bloc try (Ligne 68 à 71) ne peuvent être exécutées et que l'erreur est une erreur de communication avec le port série (serial.SerialException, Ligne 72) alors on exécute les instructions dans le bloc "except".

C'est à dire écrire dans le fichier texte "log_file" et dans le terminal que le port est déjà ouvert ou qu'il n'est pas connecté (Ligne 73/74).

La méthode **CloseConnection( )** ferme la connection avec l'imprimante. Sa structure est basiquement la même que celle d'**OpenConnection( )** avec un temps avant la fermeture pour s'assurer que les commandes puissent bien être exécutées.

## 2. Etat de l'imprimante et informations relatives à sa position

```
94      def Status(self):
95          """
96          The Status method is checking if the 3D-mill is doing something or not
97
98          Returns
99          -------
100         None.
101
102         """
103         self.s.flushInput()                          # Remove data from input buffer
104         string2Send="?"                              # This G-code input ask to the 3D-mill what is his state
105         self.s.write(string2Send.encode())
106         grbl_out=self.s.readline()                   # The typical answer is: <Idle|MPos:0.000,0.000,0.000|FS:0,0|WCO:0.000,0.000,0.000>
107         A=grbl_out.decode().strip()                  # We just want the state so we will cut the message in 4
108         caractere = "|";                             # by using .split
109         B=A.split(caractere)
110         self.PrinterState["State"]=B[0]
111         self.PrinterState["MPos"]=B[1]
112         self.PrinterState["FS"]=B[2]
113         self.PrinterState["WPos"]=B[1]
114
```

*(Figure 9 : Méthode **Status ( )** )*

La méthode **Status( )** a pour but de de vérifier si l'imprimante est en train de réaliser une action ou si elle est prête à recevoir une nouvelle instruction. On commence par vider le buffer (Ligne 103). Il est préférable de supprimer les données du buffer si l'on souhaite pouvoir récupérer des informations provenant de l'imprimante. On définit la commande G-code que l'on désire envoyer : "?" (Ligne 104). Cette commande une fois envoyée et encodée en bits (Ligne 105) renvoie différentes informations en bits concernant l'état de l'imprimante que l'on récupère en Ligne 106 et que l'on décode en Ligne 107.

Les informations récupérées sont de la forme :

<Idle|MPos:0.000,0.000,0.000|FS:0,0|WCO:0.000,0.000,0.000>

On sépare ces 4 informations suivant le caractère "|" (Ligne 108 et 109). Enfin, on va stocker ces données dans le dictionnaire créé dans la partie initialisation (Ligne 110 à 113).
Cette méthode nous retourne donc le statut de l'imprimante (Ligne 110), ses coordonnées absolues (Ligne 111), sa vitesse de déplacement (Ligne 112) et ses coordonnées relatives à son point de départ (Ligne 113).

```
115     def WaitForIdle(self):
116         """
117         The WaitForIdle method wait 2 sec if the 3D-mill is already working
118
119         Returns
120         -------
121         None.
122
123         """
124         CNC.Status(self)                                    # Check the state of the device
125         WaitingTime=0
126         while self.PrinterState["State"].strip("<")=="Run":  # Wait until the status is no longer "run"
127             time.sleep(0.1)                                 # we wait 100 msec
128             CNC.Status(self)                                # Check if the status has changed
129             WaitingTime=WaitingTime+0.1
130             if WaitingTime >= 15.0:
131                 print("There is an error")
132                 self.log_file.write("There is an error, Timeout \n")
133                 break
```

*(Figure 10 : Méthode **WaitForIdle ( )** )*

La méthode **WaitForIdle( )** a pour but d'attendre si l'imprimante est en train de réaliser une action.

On appelle la méthode **Status ( )** à la ligne 124 qui va permettre de stocker dans le dictionnaire les 4 informations concernant notre imprimante. On compare celle qui nous intéresse (Statut). Tant que celui-ci est "Run" ( imprimante en train de fonctionner), on attend 100ms et on vérifie à nouveau l'état (Ligne 126 à 128).

Pour éviter que la boucle ne tourne à l'infini, j'ai ajouté un timeout d'une quinzaine de secondes (temps nécessaire pour que l'imprimante réalise son plus grand déplacement en X) permettant de stopper la boucle si celle-ci dure trop longtemps.
Pour ça je défini une variable WaitingTime (Ligne 125) égale à 0 et je lui ajoute 0.1 à chaque itération de la boucle while (Ligne 129). Si le temps d'attente dépasse 15 secondes, l'utilisateur reçoit un message d'erreur sur le terminal et le fichier "log_file" et la commande en cours est arrêtée (Ligne 130 à 133)

La méthode **WaitForIdle( )** est très importante car j'ai constaté au cours de mes divers tests que certaines instructions étaient partiellement exécutées par manque de temps. Au lieu de réaliser la totalité du déplacement demandé, l'imprimante n'en effectuait qu'une partie avant de passer à l'instruction suivante. Donc cette méthode permet de s'assurer que les moteurs de l'imprimante ne sont pas en mouvement avant d'envoyer l'instruction suivante.

```
205     def Read_MPos(self):
206         """
207         Read the machine position (Mpos - absolute position) of the 3D-mill
208 
209         Returns
210         -------
211         None.
212 
213         """
214         self.s.flushInput()
215         CNC.Status(self)
216         print(self.PrinterState["MPos"])
217 
```

*(Figure 11 : Méthode **Read_MPos ( )** )*

Enfin, les méthodes **Read_MPos( )**, **Read_WPos( )** et **FS( )** permettent respectivement de connaître la position absolue, la position relative et la vitesse de déplacement de la tête de l'imprimante.

Leur fonctionnement est très simple, on vide le buffer (Ligne 214), on appelle la méthode **Status ( )** (Ligne 215), on affiche la donnée qui nous intéresse (Ligne 216). Ici, on affichera les coordonnées absolues de l'imprimante

Elles sont très utiles pour s'assurer que les déplacements demandés ont bien été effectués et pouvoir renseigner l'utilisateur sur la position de l'imprimante.

### 3. Commandes de déplacement

```
77      def Homing(self):
78          """
79          Set the starting coordinates of the 3D-mill
80
81          Returns
82          -------
83          None.
84
85          """
86          string2Send="G90 X0 Y0 Z0\n"
87          self.log_file.write("Sending :{}".format(string2Send))
88          self.s.write(string2Send.encode())
89          grbl_out = self.s.readline()
90          self.log_file.write(" : " + grbl_out.decode().strip()+"\n")
91          CNC.WaitForIdle(self)
92
```

*(Figure 12 : Méthode **Homing ( )** )*

La méthode **Homing( )** est utilisée après l'initialisation et renvoie les trois moteurs à leur position de référence X=Y=Z=0.

L'instruction G-code indiquant de se déplacer aux coordonnées absolues X=0, Y=0 et Z=0 est écrite en ligne 86. Ensuite celle-ci est écrite dans le fichier "log_file" (Ligne 87) et envoyée à l'imprimante sous forme de bits (Ligne 88). On récupère l'information comme quoi la commande a bien été exécutée (Ligne 89) que l'on écrit dans le fichier "log_file" (Ligne 90).

Enfin, on s'assure que l'instruction soit terminée avant de continuer via la méthode **WaitForIdle ( )** (Ligne 91).

La méthode **Move( )** permet de réaliser un déplacement suivant X,Y et Z de l'imprimante.
L'utilisateur indique si le déplacement est absolu (il écrira alors 90 en référence au G-code G90) ou relatif aux coordonnées actuelles de l'imprimante (il écrira alors 91 en référence au G-code G91). Si le déplacement choisi est absolu, le code est :

```
160         if G==90 :
161             if 0.0<=X<=295.0 and 0.0<=Y<=145.0 and 0.0<=Z<=45.0:
162                 self.s.flushInput()
163                 string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
164                 self.log_file.write("Sending :{}".format(string2Send))
165                 self.s.write(string2Send.encode())
166                 grbl_out = self.s.readline()
167                 self.log_file.write(" : " + grbl_out.decode().strip()+"\n")
168                 CNC.WaitForIdle(self)
169                 return G,X,Y,Z
170             else :
171                 print("error in the coordinates entered")
172                 string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
173                 self.log_file.write("Sending :{}".format(string2Send))
174                 self.log_file.write(" : error in the coordinates entered\n")
175
```

*(Figure 13 : Code pour un déplacement absolu dans la méthode **Move ( )** )*

Si les valeurs renseignées ne dépassent pas les limites (Ligne 161), le programme est très similaire à celui de la méthode **Homing ( )**.La seule différence est dans le G-code envoyé qui s'adapte aux coordonnées voulues par l'utilisateur (Ligne 163).
Si les valeurs choisies pour XYZ ne sont pas les bonnes, on écrit dans le fichier "log_file" la commande souhaitée et que celle-ci n'a pas pu être réalisée (Ligne 172 à 174). On écrit aussi un message dans le terminal (Ligne 171).

Si le déplacement choisi est relatif, le G-code correspondant est :

```
176         if G==91:
177             CNC.Status(self)
178             A=self.PrinterState["MPos"].strip("MPos:")                          # Check the absolute cooridinates of the printer
179             caractere=","
180             B=A.split(caractere)
181             X1=float(B[0])+float(X)
182             Y1=float(B[1])+float(Y)
183             Z1=float(B[2])+float(Z)
184             if X1<0.0 or Y1<0.0 or Z1<0.0:
185                 string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
186                 self.log_file.write("Sending :{}".format(string2Send))
187                 self.log_file.write(" : error in the coordinates entered X:"+str(X1)+" Y:"+str(Y1)+" Z:"+str(Z1)+"\n")
188                 print("error in the coordinates entered")
189             elif X1>295.0 or Y1>145.0 or Z1>45.0:
190                 string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
191                 self.log_file.write("Sending :{}".format(string2Send))
192                 self.log_file.write(" : error in the coordinates enteredX:"+str(X1)+" Y:"+str(Y1)+" Z:"+str(Z1)+"\n")
193                 print("error in the coordinates entered")
194             else:
195                 self.s.flushInput()                                             # Remove data from input buffer
196                 string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"     # Translate X,Y,Z coordinates into G-code
197                 self.log_file.write("Sending :{}".format(string2Send))          # Write to the user what G-code is sent to the 3D-mill
198                 self.s.write(string2Send.encode())                              # Send g-code block to grbl (the 3D-mill)
199                 grbl_out = self.s.readline()                                    # Wait for grbl response with carriage return
200                 self.log_file.write(" : " + grbl_out.decode().strip()+"\n")
201                 CNC.WaitForIdle(self)
202                 return G,X,Y,Z
```

*(Figure 14 : Code pour un déplacement absolu dans la méthode **Move ( )** )*

Si le déplacement est en coordonnées relatives, le programme doit s'assurer que les coordonnées indiquées par l'utilisateur ne sont pas hors limite. Pour ça, on se sert de la méthode **Status( )** (Ligne 177). Grâce à celle-ci on connaît la position absolue de l'imprimante.

Il suffit ensuite d'ajouter les coordonnées actuelles avec les déplacements souhaités par l'utilisateur (Ligne 181 à 183) et vérifier que ceux-ci ne dépassent les limites des moteurs de l'imprimante (Ligne 184/189).

Si les valeurs choisies pour XYZ ne sont pas les bonnes, on écrit dans le fichier "log_file" la commande souhaitée et que celle-ci n'a pas pu être réalisée (Ligne 185 à 187 et Ligne 190 à 192). On écrit aussi un message dans le terminal (Ligne 188/193).

Si les valeurs renseignées ne dépassent pas les limites, le programme est très similaire à celui de la méthode **Homing ( )**.La seule différence est dans le G-code envoyé qui s'adapte aux coordonnées voulues par l'utilisateur (Ligne 196).

### E. Mise en forme

Dans un dernier temps, j'ai cherché à mettre en forme mon travail afin que celui-ci puisse être facilement compris par quelqu'un qui ne connaît pas l'appareil. J'ai donc rajouté de nombreuses annotations en anglais ainsi que renommé le nom de mes diverses commandes.

J'ai aussi essayé de transmettre les informations que j'ai acquises via l'ajout d'un dictionnaire en début de programme afin qu'un utilisateur puisse retrouver les limites en X,Y,Z en mm ou comment se déplacer en coordonnées absolus/relatifs. Il pourra par ailleurs manipuler ces valeurs s'il désire se déplacer au milieu de plateau par exemple.

```
22    MotorLimit={}
23    MotorLimit["XhighMM"]=295.0
24    MotorLimit["YhighMM"]=145.0
25    MotorLimit["ZhighMM"]=45.0
26    MotorLimit["Absolute"]=90
27    MotorLimit["Relative"]=91
```

*(Figure 15 : Dictionnaire "MotorLimit" regroupant différentes informations pratiques)*

J'ai aussi clarifié mes diverses recherches effectuées au cours du stage afin qu'un nouvel utilisateur puisse comprendre ma démarche et retrouver la documentation que j'ai utilisé.

# IV. Résultats et test de la classe CNC ( )

Afin de s'assurer du bon fonctionnement des classes, je me suis servis d'une commande python en dehors de la classe CNC :

```
if __name__ == "__main__":
```

*(Figure 16 : Ligne 269 Programme Final, Annexe 3)*

Cette commande permet aux instructions qui la suivent de n'être exécutées que dans le programme de départ (le "main") mais ces instructions ne seront pas exécutées si la classe est appelée dans un autre programme.

Une fois ces diverses tâches effectuées, j'ai donc obtenu au terme de mon stage un code assez volumineux ([6] & Annexe 3) bien que relativement simple. Je vais montrer ici le bon fonctionnement de celui-ci ainsi que les résultats obtenus. Imaginons que l'utilisateur désire faire une suite de déplacements en coordonnées absolues puis relatives comme suit :

```
269 if __name__ == "__main__":
270
271     cnc = CNC('COM4')
272     cnc.OpenConnection()
273     cnc.Homing()
274     cnc.Move(90,18,5,3)
275     cnc.Move(90,-18,5,3)
276     cnc.Read_MPos()
277     cnc.Move(91,-18,5,3)
278     cnc.Read_MPos()
279     cnc.Homing()
280     cnc.CloseConnection()
```

*(Figure 17 : Exemple de programme possible (Pour rappel 90:Déplacement absolue, 91:Déplacement relatif))*

On remarque tout de suite que l'instruction en ligne 275 est impossible, les valeurs absolues de X allant de 0 à 295mm. Ce qui va se traduire dans le fichier log_file par :

```
Port COM4 ouvert
Sending :G90 X0 Y0 Z0
 : ok
Sending :G00 G90 X18 Y5 Z3
 : ok
Sending :G00 G90 X-18 Y5 Z3
 : error in the coordinates entered
Sending :G00 G91 X-18 Y5 Z3
 : ok
Sending :G90 X0 Y0 Z0
 : ok
Port COM4 closed
```

*(Figure 18 : Fichier "log_file" obtenu après exécution des instructions de la Figure 14)*

L’utilisateur se rend donc compte que les actions sont bien réalisées sauf celle pour laquelle il a commis une erreur de saisie.

Le terminal obtenue après exécution est le suivant :

```
In [7]: runfile('C:/Users/NoelFlantier/Desktop/Driver.py', wdir='C:/Users/NoelFlantier/Desktop')
error in the coordinates entered
MPos:18.000,5.000,3.000
MPos:0.000,10.000,6.000

In [8]:
```

*(Figure 19 : Terminal obtenue après exécution des instructions en Figure 14)*

Il se contente d’afficher la présence de l’erreur et les coordonnées absolues demandées en ligne 276 et 278 par l’utilisateur.

# V. Discussion

Au cours de mes multiples recherches, j'ai pu constater qu'il y a différents paramètres auxquels il faut faire attention lorsqu'on souhaite contrôler l'imprimante.

Premièrement, les délais entre les nouvelles commandes mais aussi ceux durant l'ouverture et la fermeture. En effet, sans ceux-ci l'appareil n'effectue pas les commandes demandées ou saute certaines d'entre elles. Néanmoins, je pense avoir réussi à régler ce problème et avoir mis en lumière l'importance des timers mais aussi de la méthode **WaitForIdle( )**.

Un autre point important, est le fait de ne pas confondre coordonnées absolues ou relatives et par conséquent de s'assurer que l'imprimante a bien effectué un **Homing ( )** en début de programme.

Je conseille aussi fortement d'effectuer un **Homing ( )** en fin de programme car si à la fin de la manipulation l'utilisateur constate que l'imprimante n'est pas à sa position d'origine, ca signifie qu'il y a eu une erreur différente de celles rencontrées dans mes recherches ( débranchement soudain de l'imprimante,....).

Maintenant que les commandes de base de l'imprimante ont été définies, il faudra intégrer celles-ci dans des programmes plus important notamment lors de la mise en série de l'imprimante avec une pompe par exemple. Si l'on désire demander à la pompe de prélever un volume de liquide, on se servira de la méthode **WaitForIdle ( )** afin que l'imprimante ne se déplace pas en même temps.

La création d'une interface sera aussi nécessaire afin de simplifier au maximum l'utilisation du dispositif final pour que celui-ci puisse être utilisé par quelqu'un qui ne maîtrise pas particulièrement Python. Même si je pense avoir réussi à simplifier l'utilisation de l'imprimante pour des déplacements de base.

# VI. Conclusion

L'objectif du stage était de programmer une imprimante 3D afin que celle-ci puisse réaliser différentes actions (prélever des marqueurs par exemple) à des coordonnées précises. C'était la première fois que je me servais d'une interface (bCNC) permettant de contrôler une imprimante 3D et tout simplement la première fois que j'utilisais d'une imprimante 3D en tant que telle.

A cette fin, j'ai réalisé un travail de fond qui permettra à d'autres après moi de contrôler l'imprimante et de l'utiliser pour des expériences de Hi-M. En effet, j'ai créé une classe permettant de réaliser les fonctionnalités de base pour contrôler et communiquer .

J'ai par ailleurs réussi à réaliser les actions élémentaires nécessaires avec l'imprimante (se déplacer, attendre, transmettre des informations) selon l'architecture souhaitée par mes encadrants et détailler mes méthodes afin que celles-ci puissent être utilisées par d'autres après moi.

Enfin, j'ai appris à utiliser de nouveaux outils , en particulier pour communiquer et transmettre des codes pythons à distance via l'utilisation d'outils de collaboration pour la programmation comme GitKraken et GitHub. En effet, je pouvais directement transmettre mon programme et en discuter avec mes responsables mais aussi garder en mémoire les modifications apportées à celui-cl

Enfin j'ai appris une méthode de travail pour écrire des programmes plus complexes via l'utilisation de classes sous python et mis en pratique les diverses connaissances acquises au cours de ma formation.

# VII. Bibliographie :

[1]Microscopy-Based Chromosome Conformation Capture Enables Simultaneous Visualization of Genome Organization and Transcription in Intact Organisms (Andrés M.Cardozo Gizzi,Diego I.Cattoni,Jean-BernardFiche,Sergio M.Espinola,JulianGurgo,OlivierMessina,ChristopheHoubron,YukiOgiyama,Giorgio L.Papadopoulos,GiacomoCavalli,MouniaLagha,MarceloNollmann)
https://www.sciencedirect.com/science/article/pii/S1097276519300115

[2] Programme Python disponible sur GitHub (Sungeun K. Jeon, 2012)
Répertoire GitHub : grbl/doc/script/simple_stream.py
https://github.com/grbl/grbl/blob/master/doc/script/simple_stream.py

[3]Répertoire GitHub : bCNC/README.md
https://github.com/vlachoudis/bCNC/blob/master/README.md

[4]Site Internet : Wikipédia, Article : Programmation de commande numérique
https://fr.wikipedia.org/wiki/Programmation_de_commande_num%C3%A9rique

[5]Livre : Python For the Lab (Aquiles Carattino,2020)

[6]Répertoire GitHub : Stage_Aymerick/cnc-python/driver.py

# VIII. Annexes

```python
#!/usr/bin/env python
"""\
Simple g-code streaming script for grbl
"""

import serial
import time

# Open grbl serial port
s = serial.Serial('/dev/ttyUSB0',115200)

# Open g-code file
f = open('circle.nc','r');

# Wake up grbl
s.write(b"\r\n\r\n")
time.sleep(2)   # Wait for grbl to initialize
s.flushInput()  # Flush startup text in serial input

# Stream g-code to grbl
for line in f:
    #l = line.strip() # Strip all EOL characters for streaming
    string2Send=line.strip()+"\r\n" #"".join([l,'\r'])
    print("Sending :{}".format(string2Send))
    #s.write(str.encode("G17 G20 G90 G94 G54")) # Send g-code block to grbl
    s.write(string2Send.encode()) # Send g-code block to grbl
    grbl_out = s.readline() # Wait for grbl response with carriage return
    print(" : " + grbl_out.decode().strip())

# Wait here until grbl is finished to close serial port and file.
#raw_input("  Press <Enter> to exit and disable grbl.")

# Close file and serial port

f.close()
```

Annexe 1 : Code Python disponible sur internet fourni par mes tuteurs

**Common G Codes (avec certaines extensions non standardisées)**

| | |
|---|---|
| G00 | Déplacement rapide |
| G01 | Interpolation linéaire |
| G02 | Interpolation circulaire (sens horaire, anti-trigo) |
| G03 | Interpolation circulaire (sens anti-horaire, trigo) |
| G04 | Arrêt programme et ouverture carter (pour nettoyer) (temporisation - suivi de l'argument F ou X en secondes) |
| G10/G11 | Écriture de données / Effacement de données (suivi de l'argument L suivant le type de données à écrire) |
| G17 | Sélection du plan X-Y |
| G18 | Sélection du plan X-Z |
| G19 | Sélection du plan Y-Z |
| G20 | Programmation en pouces |
| G21 | Programmation en mm |
| G28 | Retour à la position d'origine |
| G31 | Saute la fonction (mode *Interrupt* utilisé pour les capteurs et les mesures pièces et de longueur d'outil) |
| G33 | Filetage à pas constant |
| G34 | Filetage à pas variable |
| G40 | Pas de compensation de rayon d'outil |
| G41 | Compensation de rayon d'outil à gauche |
| G42 | Compensation de rayon d'outil à droite |
| G54 à G59 | Activation du décalage d'origine pièce (*Offset*) |
| G68 / G68.1 | Activation du mode "Plan incliné" (*Tilted plane working*) pour les centres d'usinage 5 axes |
| G70 | Cycle de finition |
| G71 / G71.7 | Cycle d'ébauche suivant l'axe Z (appel de profil balisé entre les arguments P et Q) |
| G75 | Cycle de gorge |
| G76 / G76.7 | Cycle de filetage |
| G83 | Cycle de perçage débourrage |
| G69 | Annulation du mode *Tilted plane working* (Plan incliné) |
| G84 | Cycle de taraudage rigide |
| G90 | Déplacements en coordonnées absolues |
| G91 | Déplacements en coordonnées relatives |
| G94/G95 | Déplacement en pouces par minute/pouce par tour |
| G96 ; G97 | Vitesse de coupe constante (vitesse de surface constante) ; Vitesse de rotation constante ou annulation de G96 |

Annexe 2 : Commandes Gcode de base disponibles sur wikipédia

Annexe 3 : Code final

```
"""
Created on Wed May 27 14:24:47 2020

@author: Aymerick Reinders

The CNC class is used to control a 3D-mill instrument using G-codes. The
following methods are the most basic functions that are neede to interact with
the device.


"""

MotorLimit={}
MotorLimit["XhighMM"]=295.0
MotorLimit["YhighMM"]=145.0
MotorLimit["ZhighMM"]=45.0
MotorLimit["Absolute"]=90
MotorLimit["Relative"]=91

import time
import serial

class CNC:

    def __init__(self,port):
        """
        For the class instanciation, the input variable port is indicating which
        COM port the 3D-Mill is connected to.

        Parameters
        ----------
        port : Name of the port where the printer is connected

        Returns
        -------
        None.

        """
        self.port = port
        self.log_file=open("log_file.txt","w")
        self.PrinterState={}
        return

    def OpenConnection(self):
        """
        The OpenConnection method is opening the serial communication port and
        keeps the output in self.
        The time delay of 2s is important to keep, else the communication will
        not be established and it won't be possible to control the 3D-mill.
        When an error occurs, Python will normally stop and generate an error message.
        We will change this message by using "try" and "except"

        Returns
        -------
```

```
        None.

        """
        try:                                        # The try block lets you test a block of code for errors.
            self.s  = serial.Serial(self.port,115200);  # Open the port and keeps the output in self.s
            self.log_file.write("Port "+self.port+" ouvert\n")
            time.sleep(2)
            self.s.flushInput()                     # Remove data from input buffer
        except serial.SerialException:              # The except block lets you handle the error.
            print("The port "+self.port+" is already opened or is not connected ")
            self.log_file.write("The port "+self.port+" is already opened or is not connected ")
            pass

    def Homing(self):
        """
        Set the starting coordinates of the 3D-mill

        Returns
        -------
        None.

        """
        string2Send="G90 X0 Y0 Z0\n"
        self.log_file.write("Sending :{}".format(string2Send))
        self.s.write(string2Send.encode())
        grbl_out = self.s.readline()
        self.log_file.write(" : " + grbl_out.decode().strip()+"\n")
        CNC.WaitForIdle(self)
```

```
    def Status(self):
        """
        The Status method is checking if the 3D-mill is doing something or not

        Returns
        -------
        None.

        """
        self.s.flushInput()                             # Remove data from input buffer
        string2Send="?"                                 # This G-code input ask to the 3D-mill what is his state
        self.s.write(string2Send.encode())
        grbl_out=self.s.readline()                      # The typical answer is: <Idle|MPos:0.000,0.000,0.000|FS:0,0|WCO:0.000,0.000,0.000>
        A=grbl_out.decode().strip()                     # We just want the state so we will cut the message in 4
        caractere = "|";                                # by using .split
        B=A.split(caractere)
        self.PrinterState["State"]=B[0]
        self.PrinterState["MPos"]=B[1]
        self.PrinterState["FS"]=B[2]
        self.PrinterState["WPos"]=B[3]

    def WaitForIdle(self):
        """
        The WaitForIdle method wait 2 sec if the 3D-mill is already working

        Returns
        -------
        None.

        """
        CNC.Status(self)                                                # Check the state of the device
        WaitingTime=0
        while self.PrinterState["State"].strip("<")=="Run":             # Wait until the status is no longer "run"
            time.sleep(0.1)                                             # we wait 100 msec
            CNC.Status(self)                                            # Check if the status has changed
            WaitingTime=WaitingTime+0.1
            if WaitingTime >= 15.0:
                print("There is an error")
                self.log_file.write("There is an error, Timeout \n")
                break

```

```
    def Move(self,G,X,Y,Z):
        """
        The Move method is sending instructions to the 3D-mill in order to
        change the positions of the motors.The user must indicate the
        X,Y,Z coordinates as input.

        Parameters
        ----------
        G : 90 : Absolute coordinates 91 : Relative coordinates
        X : X coordinates
        Y : Y coordinates
        Z : Z coordinates

        Returns
        -------
        G : 90 : Absolute coordinates 91 : Relative coordinates
        X : X coordinates
        Y : Y coordinates
        Z : Z coordinates

        """
        if G != 90 and G!=91:
            print("Wrong value for G. Muste be 90 or 91")
            self.log_file.write("Wrong value for G. Muste be 90 or 91\n")
        if G==90 :
            if 0.0<=X<=295.0 and 0.0<=Y<=145.0 and 0.0<=Z<=45.0:
                self.s.flushInput()                                                 # Remove data from input buffer
                string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"  # Translate X,Y,Z coordinates into G-code
                self.log_file.write("Sending :{}".format(string2Send))              # Write to the user what G-code is sent to the 3D-mill
                self.s.write(string2Send.encode())                                  # Send g-code block to grbl (the 3D-mill)
                grbl_out = self.s.readline()                                        # Wait for grbl response with carriage return
                self.log_file.write(" : " + grbl_out.decode().strip()+"\n")
                CNC.WaitForIdle(self)
                return G,X,Y,Z
            else :
                print("error in the coordinates entered")
                string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
                self.log_file.write("Sending :{}".format(string2Send))
                self.log_file.write(" : error in the coordinates entered\n")
```

```
        if G==91:
            CNC.Status(self)
            A=self.PrinterState["MPos"].strip("MPos:")                    # Check the absolute cooridinates of the printer
            caractere=","
            B=A.split(caractere)
            X1=float(B[0])+float(X)
            Y1=float(B[1])+float(Y)
            Z1=float(B[2])+float(Z)
            if X1<0.0 or Y1<0.0 or Z1<0.0:
                string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
                self.log_file.write("Sending :{}".format(string2Send))
                self.log_file.write(" : error in the coordinates entered X:"+str(X1)+" Y:"+str(Y1)+" Z:"+str(Z1)+"\n")
                print("error in the coordinates entered")
            elif X1>295.0 or Y1>145.0 or Z1>45.0:
                string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"
                self.log_file.write("Sending :{}".format(string2Send))
                self.log_file.write(" : error in the coordinates enteredX:"+str(X1)+" Y:"+str(Y1)+" Z:"+str(Z1)+"\n")
                print("error in the coordinates entered")
            else:
                self.s.flushInput()                                      # Remove data from input buffer
                string2Send="G00 G"+str(G)+" X"+str(X)+" Y"+str(Y)+" Z"+str(Z)+"\n"      # Translate X,Y,Z coordinates into G-code
                self.log_file.write("Sending :{}".format(string2Send))   # Write to the user what G-code is sent to the 3D-mill
                self.s.write(string2Send.encode())                       # Send g-code block to grbl (the 3D-mill)
                grbl_out = self.s.readline()                             # Wait for grbl response with carriage return
                self.log_file.write(" : " + grbl_out.decode().strip()+"\n")
                CNC.WaitForIdle(self)
                return G,X,Y,Z
```

```
    def Read_MPos(self):
        """
        Read the machine position (Mpos - absolute position) of the 3D-mill

        Returns
        -------
        None.

        """
        self.s.flushInput()
        CNC.Status(self)
        print(self.PrinterState["MPos"])


    def Read_WPos(self):
        """
        Read the working position (Wpos - relative position) of the 3D-mill

        Returns
        -------
        None.

        """
        self.s.flushInput()
        CNC.Status(self)
        print(self.PrinterState["WPos"])


    def FS(self):
        """
        Read the FS of the 3D-mill

        Returns
        -------
        None.

        """
        self.s.flushInput()
        CNC.Status(self)
        print(self.PrinterState["FS"])

```

```
247     def CloseConnection(self):
248         """
249         The CloseConnection method is closing the serial communication port.
250         The time delay of 5s is important tobe sure that all the instruiction
251         has been executed.
252 
253         Returns
254         -------
255         None.
256 
257         """
258         try:
259             time.sleep(2)
260             self.log_file.write("Port "+self.port+" closed")
261             self.log_file.close()
262             self.s.close()
263         except AttributeError:
264             print("The port "+self.port+" can't be closed")
265             self.log_file.write("The port "+self.port+" can't be closed")
266             self.log_file.close()
267 
268 
269 if __name__ == "__main__":
270 
271     cnc = CNC('COM4')
272     cnc.OpenConnection()
273     cnc.Homing()
274     cnc.Move(90,18,5,3)
275     cnc.Move(90,-18,5,3)
276     cnc.Read_MPos()
277     cnc.Move(91,-18,5,3)
278     cnc.Read_MPos()
279     cnc.Homing()
280     cnc.CloseConnection()
281 
```