PROJET POO

Gaëlle ALLEON | L2 Biologie (BBCM)
Bryan BEKONO | L2 Mathématiques (MB)
TDE groupe 10

Tables des matières

Présentation du sujet **2**

Tâches à réaliser **2**
- Planning 2
- Répartition des tâches 2
  - Command Line Interface (CLI) 2
  - Graphical User Interface (GUI) 3

Réalisation des tâches **3**
- PARTIE 1 : “CLI” Command Line Interface 3
  - Extraction des métadonnées 4
  - Exploration des répertoires 4
  - Stéganographie 5
    - Dissimuler le message 5
      - Principe 5
      - Code 6
    - Extraire le message 6
      - Principe 6
      - Code 6
  - Gestion des arguments et des options 6
- PARTIE 2 : “GUI” Graphical User Interface 7
  - Page d’accueil 7
  - Métadonnées 8
  - Stéganographie 9
    - Encoder 9
    - Décoder 9

Critiques du projet **10**
- CLI : 10
- GUI : 10

Annexes **10**

Présentation du sujet

L'objectif principal de ce projet est de réaliser un logiciel de traitement d'images. Ici, nous nous limiterons aux formats PNG et JPEG.

Notre logiciel devra pouvoir réaliser plusieurs actions telles que:

- ➔ l'extraction de métadonnées des fichiers images
- ➔ l'exploration complète d'un répertoire contenant entre autres des fichiers images
- ➔ l'utilisation de la stéganographie pour dissimuler un message dans une image
- ➔ l'extraction de ce message

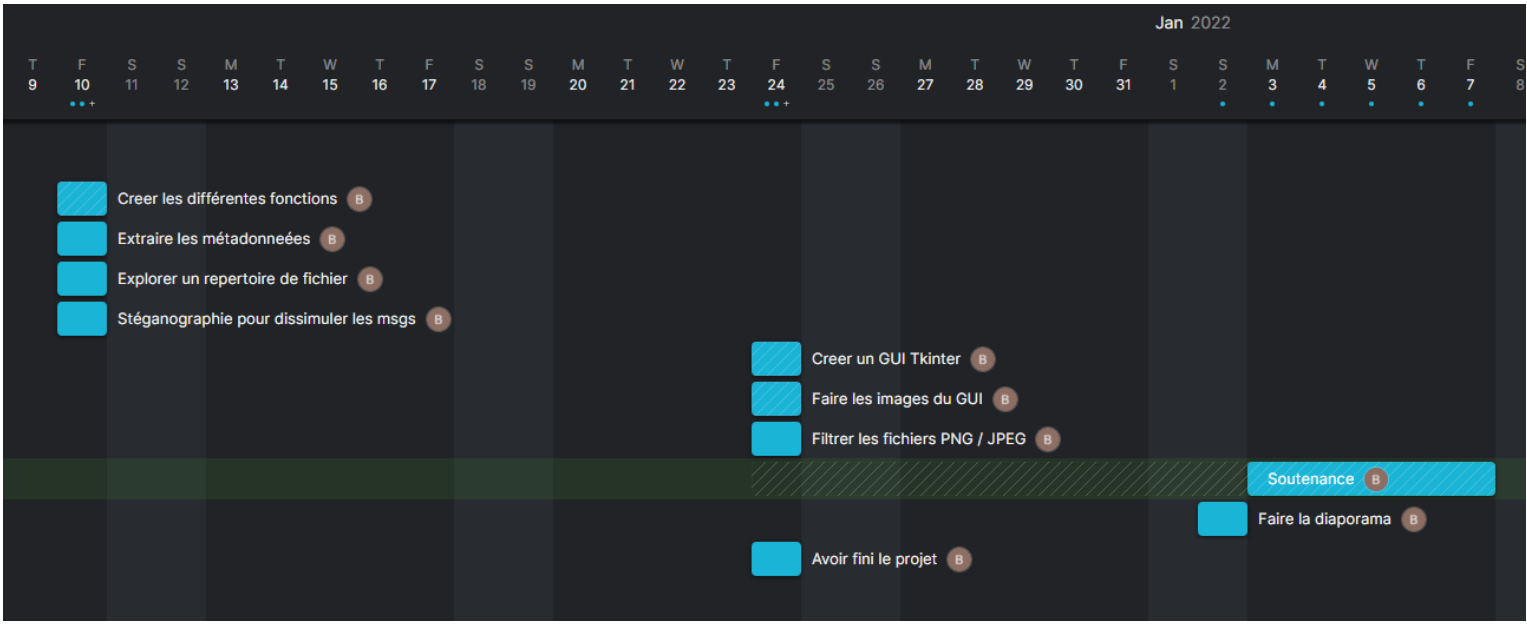
Nous devons écrire deux programmes:

- ➔ un en mode console (CLI):
  Il faudra ici créer différentes options (à entrer en argument) permettant de réaliser les différentes actions.
- ➔ un en mode graphique (GUI)

Tâches à réaliser

Planning

Nous avons pour réaliser ce projet identifier les principales sous-tâches à réaliser. Et nous avons pu planifier nos recherches de la manière suivante:



Répartition des tâches

Command Line Interface (CLI)

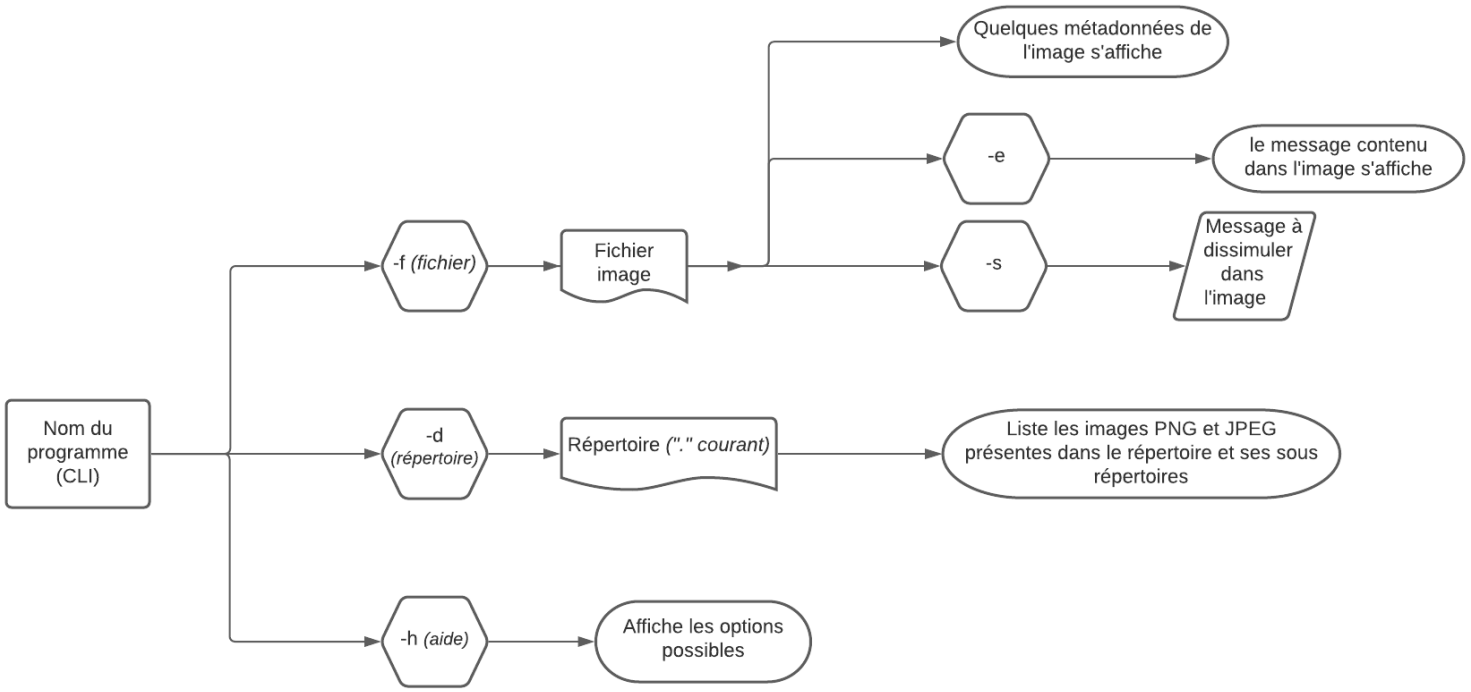
1. Extraction des métadonnées: **Gaëlle**
2. Exploration d'un répertoire: **Bryan**
3. Dissimuler un message dans l'image: **Gaëlle et Bryan**
4. Extraire le message de l'image: **Gaëlle et Bryan**
5. Créer les options pour l'utilisateur (-h / -e /…): **Bryan**
6. Docstrings & Commentaires: **Gaëlle**

Graphical User Interface (GUI)

1. Création des différentes fenêtre et boutons: **Bryan**
2. Esthétique de l'interface (Photoshop): **Bryan**
3. Adaptation des fonctions principales en GUI **: Bryan**
4. Docstrings & Commentaires: **Gaëlle et Bryan**

Réalisation des tâches

PARTIE 1 : "CLI" Command Line Interface



Extraction des métadonnées

Nous avons créé une fonction **metadonnées(*image):**

Pour savoir comment extraire les métadonnées des images PNG et JPEG, nous avons dû mener des recherches sur la façon dont ces données étaient stockées. Or ces deux formats ne stockent pas les métadonnées de la même manière.

Une image PNG (Portable Network Graphics) est formée de plusieurs fragments d'information appelés chunk. Chaque chunk est un entité ayant un rôle précis dans la définition de l'image. Le deuxième chunk d'une image PNG est le "Image header" (IHDR). Il contient les informations des métadonnées.

Pour les images JPEG (Joint Photographic Experts Group) la structure de l'image est différente. Il s'agit ici d'un format compressé. Les données sont découpées en morceaux balisés à l'aide d'un marqueur spécifique. Le plus souvent les métadonnées d'une image JPEG sont enregistrées selon un format EXIF ce qui correspond à un marquage avec le marqueur APP1.

Les résultats de nos recherches nous ont amenés à rechercher une méthode tolérante de ces deux formats.

Nous avons choisi d'utiliser l'outil **hachoir-metadata** proposée par la bibliothèque hachoir de Python. Nous avions également besoin d'un **parser** (analyseur) pour convertir les données de l'image en données exploitable par Python.
Nous avons ajoutez une gestion d'erreur pour deux possibilités:

- Si l'image n'a pas de métadonnées enregistrées
- Si le format du fichier n'est pas pris en charge par la bibliothèque hachoir.

L'avantage de cette méthode est qu'elle est simple, rapide et qu'elle prend en charge la plupart des formats de fichiers.
L'inconvénient est le le fait qu'elle n'affiche pas toutes les métadonnées et que le fonctionnement de notre programme nécessite l'installation de cette bibliothèque.

Exploration des répertoires

Pour l'exploration du répertoire nous avons utiliser la fonction **listdir** de la bibliothèque **os**

Stéganographie

Dissimuler le message

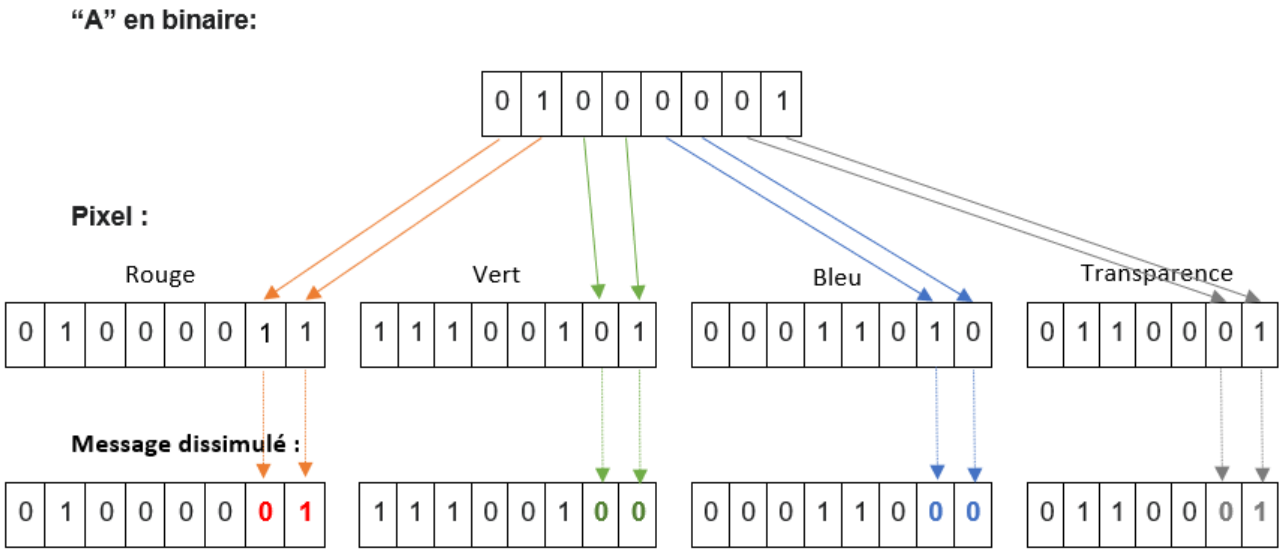
Principe

Nous cherchons à dissimuler un texte ASCII dans une image.

Le code ASCII permet de coder 128 caractères en binaire *(de 0000000 à 1111111)*. Le message que nous allons alors dissimuler est une succession de partition binaire. (1 caractère = 1 octet)

Une image matricielle comme celles enregistrées sous le format PNG ou JPEG (compressée) est une grille de pixels. Un pixel peut posséder jusqu'à quatre dimensions: **rouge (R)** ; **vert (G)** ; **bleu (B)** et **transparence (A)**
Chacune de ces dimensions est codée en binaire. Une dimension est codée par un octet soit 8 bits. Autrement dit un pixel à quatre dimensions est codé par quatre octets.
Il existe pour un octet, 256 valeurs possibles (2^8) soit pour les trois couleurs il existe plus de 16 millions de nuances possibles (256^3).

Pour dissimuler le message nous allons modifier légèrement les pixels, la nuance changera mais l'écran et notre œil ne pourra pas faire la différence entre le pixel d'avant et après l'action du programme. Voici un schéma de la manière dont nous allons nous y prendre:



Code

Nous avons créé une fonction **Encode():**

Nous avons trouvé le moyen de traduire le message à dissimuler en binaire. Cela a été rendu possible grâce à la fonction **format()** et **ord()** qui peuvent prendre en charge du texte Unicode et le transformer en binaire avec l'option **"08b".** Nous avons ajouté un délimiteur **"$top"** qui termine le message. Cela va avoir pour objectif, lors de l'extraction du message de savoir où il se termine précisément.
Pour inclure ce message dans l'image il a fallu récupérer les données des pixels sous formes d'octet de bits. Nous avons dû transformer la liste de données en tableau grâce à la fonction **array()**. Nous avons pu modifier les bits grâce à la fonction **int()** et **bin()**
Puis nous avons laissé à l'utilisateur la possibilité d'enregistrer l'image modifiée sous un nouveau nom.

Extraire le message

Principe

Pour extraire le message nous devons réaliser l'opération inverse.
Lors du décodage nous arrêterons ce dernier lorsque nous aurons obtenu le délimiteur "$top" que nous prendrons soin d'enlever avant l'affichage du message.

Code

Nous avons créé une fonction **Decode():**
Bien sûr nous devons tout d'abord recomposer l'image sous forme de tableau grâce à **array()**
Puis nous devons extraire les deux derniers bits de chaque dimensions (les bits les moins significatifs). On utilise la même méthode que lorsqu'on voulait les modifier c'est-à-dire avec **bin().** Chaque bits récoltés est stockés dans une variable puis on les a regroupés en octet formant ainsi un caractère du message. Lorsqu'on a décodé notre délimiteur **"$top"** cela indique au programme que le décodage est terminé et on peut alors afficher le message complet à l'utilisateur grâce à **print()**.

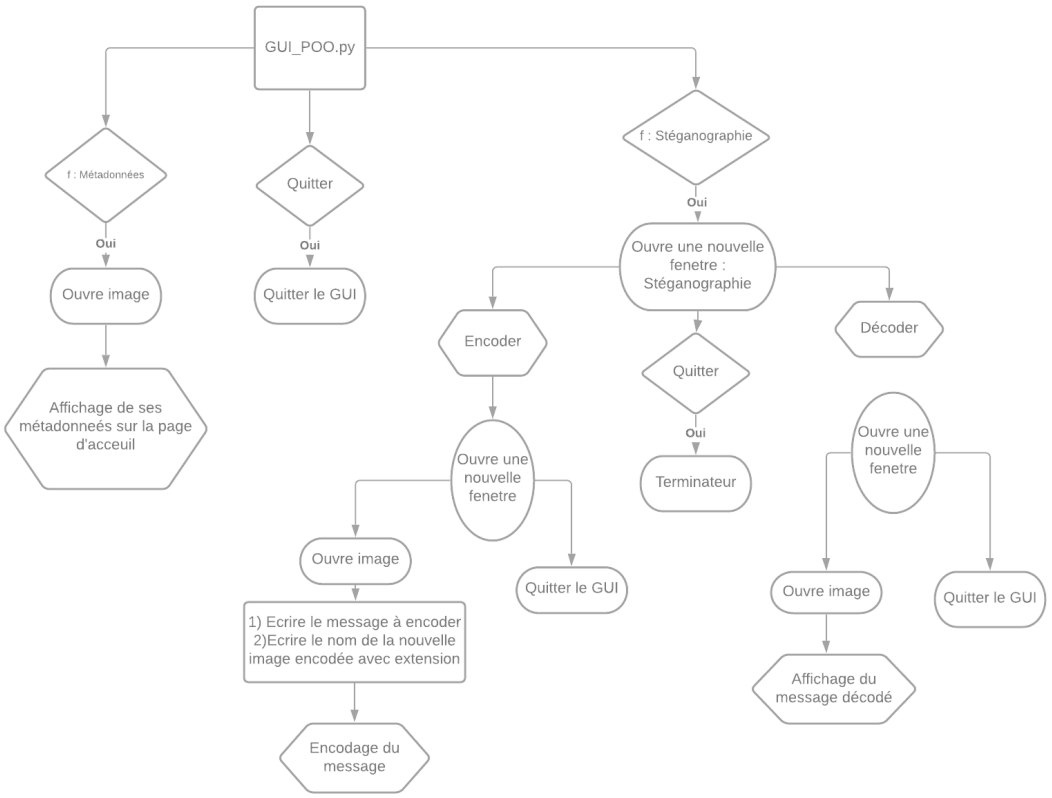
Gestion des arguments et des options

Pour accomplir les actions selon la demande de l'utilisateur nous avons créé une fonction **main()**.

Celle-ci fait appel à l'outil **sys.argv** (de la bibliothèque **sys**) qui permet de connaître l'argument renseigné dans l'invite de commande. Ainsi nous avons à l'aide

de **boucle if_else** testé les différentes possibilités et selon l'argument entré nous avons appelé les fonctions décrites précédemment afin de réaliser les actions du logiciel.

PARTIE 2 : "GUI" Graphical User Interface



Page d'accueil

Pour coder l'interface graphique nous avons utilisé la bibliothèque graphique **tkinter, numpy** et **hachoir**.
À l'aide du logiciel **Photoshop**, nous avons réalisé **l'IHM** ou encore "**UX Design**" du GUI, nous avons fait en sorte que ce soit cohérent et esthétique.
La mise en forme de la page a été possible grâce aux options disponibles du module **tkinter** : nous avons donc utilisé les options; **Label() Button() Place()** ou encore **Pack()** pour le faire.Sur cette page d'accueil on peut retrouver les deux fonctions principales **Métadonnées** et **Stéganographie**, ainsi qu'un bouton **Quitter** qui permet de fermer le programme. Pour chaque fenêtre nous avons créer des boutons quitter en utilisant la commande **"fenetre".destroy() .**

Métadonnées

Nous avons modifié la fonction métadonnée pour qu'elle puisse être exécuté grâce à la fonction **OpenImage,** au lieu de prendre le nom de l'image dont nous voulons extraire les métadonnées, nous avons fait en sorte qu'une fois le bouton appuyé nous pouvons sélectionner une image dans l'ordinateur et cette image qui à été choisie soit utilisée comme paramètre pour la fonction.
L'affichage en suite de ces métadonnées à été un peu plus fastidieux car nous avons du créer un Frame en sur la page d'accueil du GUI qui va donc elle pouvoir afficher les lignes des métadonnées dessus

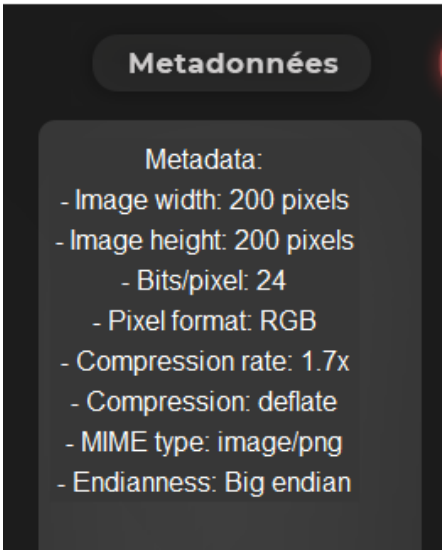
```
typesfichier = (('Images PNG', '*.png'),('Images JPG', '*.jpg'))
nomfichier = fd.askopenfilename(title='Choisissez une image',initialdir='/Desktop',filetypes=typesfichier)
global fenetre
if not nomfichier:
    messagebox.showerror("Erreur", "Vous n'avez pas choisi d'image !")
else:
    metadonnees(nomfichier)
```

```
if not metadata:
    # Si il n'ya pas de métadonnées dans l'image on affichera qu'il est impossible de les extraire
    metas = Label(cadre , text = "Impossible d'extraire les métadonnées", bg ="#373737", fg = "#FFFFFF",font = ("Helvetica",10))
    metas.pack()

for line in metadata.exportPlaintext():  # Pour chaque ligne des métadonnées on affichera la ligne

    metas = Label(cadre , text = line, bg ="#373737", fg = "#FFFFFF",font = ("Helvetica",12))
    metas.pack()
```

```
"""Creation d'un cadre pour l'affichage des métadonnées"""
cadre = Frame(fenetre, width =210, height= 390, bg ="#373737" )
cadre.place(x=28, y=80)
```



Stéganographie

Ce bouton stéganographie permet d'ouvrir une nouvelle fenêtre en **TopLevel** qui va nous demander de choisir entre décoder et encoder une image, mais comme vu sur la vidéo de démonstration ces boutons ne fonctionne pas à cause d'une erreur d'affichage sur la nouvelle fenêtre qui est censée s'ouvrir.
Le problème suivant "**Tclerror : "pyimage" doesn't exist**" malgré de nombreuses recherches afin de régler cette erreur aucune ne fût concluante, nous avons donc décidé de laisser les 3 programmes individuellement sinon nous ne pouvions pas finir le projet. Ces 2 fonctions **Encoder** et **Décoder** fonctionnent individuellement.

Encoder

La fonction encoder dans le GUI sous le nom du script "**Sel_Encoderr.py**" comme vu dans le diagramme **UML** et la vidéo de démonstration nous demander de sélectionner une image, cette image va donc ensuite être affichée sur une nouvelle fenêtre qui contient **2 Entry Text** dont le contenu va être utilisé comme variable afin de pouvoir encoder le message à cacher.
Nous avons donc pris le **StringVar()** de chaque entrée et les avons placé dans leur variable pour que le bouton encoder puisse faire le reste.

Décoder

Dans cette fonction décoder donnée sous le nom du script "**Sel_Decoderr.py**" permet donc comme aussi vu dans le programme UML d'afficher l'image et d'ensuite afficher le texte caché dans un EntryText qui sert de frame pour ce dernier. Pour réaliser cela nous avons tout simplement **return(message[:-5])**, le texte encodée dans l'image ou bien **return("Il n'y a pas de message caché.")** si il n'y avait rien dedans.

Critiques du projet

CLI

Le code pour les options aurait pu être écrit plus simplement qu'avec des nested if else.

L'exploration d'un répertoire n'est pas si poussée que ça, il aurait été possible de peut-être pouvoir cliquer sur les dossiers pour ensuite voir ce qu'il y contenait.

L'impossibilité de pouvoir créer une image .jpg/.jpeg avec un message caché.

GUI

Nous aurions pu effectuer le GUI en utilisant la méthode **class** de python qui aurait permis la cohésion de toutes les pages et donc sans doute éviter l'erreur qui nous à obligé à utiliser trois scripts au lieu d'un seul.

L'affichage des métadonnées en GUI à ses défaut, il risque d'avoir trop d'informations et donc le cadre ne sert plus à grand chose.

L'enregistrement de la nouvelle image encodée se fait dans le répertoire courant du script et est uniquement en .png ce qui n'est pas pratique, de plus nous devons donner un nom à cette nouvelle image encodée directement dans le GUI au lieu que le boutons encodés.

Annexes

Pour plus d'informations concernant les fonctions que nous avons programmées vous pouvez consulter la documentation Doxyfile dans le fichier du projet et les commentaires laissés sur les scripts python.
GitHub du projet : https://github.com/oftenbryan/ProjetPOO