Programmation C avancée
TP 6 : le meilleur de deux mondes

L'objectif de ce TP est de profiter du meilleur de deux mondes : celui de Python et celui du langage C. En Python, on peut écrire rapidement et facilement du code produisant des effets complexes. Malheureusement le langage Python est globalement lent car, notamment, il n'est pas compilé, l'interpréteur doit déterminer les types, les constructions de base sont en fait déjà très évoluées (une liste Python est un tableau (mémoire contiguë) de références d'objets : `void**`). Le langage C, lui, est très proche de la machine. Le langage est très simple et ne possède pas de constructions primitives évoluées. Les exécutables produits sont très efficaces mais écrire du code C qui fonctionne, qui est robuste et qui est réutilisable est difficile. Le langage C est simple, c'est programmer dans ce langage qui est difficile. Produire du code haut-niveau en C est toujours possible (le langage est généraliste pour sûr) mais c'est long et très pénible.

Pour une boucle `for` Python, le langage doit souvent itérer sur des créations d'objets

```
for object in iterable:
```

En interne, `iterable` est un objet qui doit posséder un méthode `__iter__`. Ne cherchez pas à connaître le `sizeof` de cet objet, mais l'introspection impose à Python de stocker beaucoup d'informations (dans l'instance ou encore sa classe...). À chaque tour de boucle, Python doit instancier (des `mallocs` et des `__init__`) un nouvel objet pour l'identificateur `objet`. L'ancien objet écrasé par le nouveau doit être libéré par le garbage collector (`free`)... Bref, toute fonctionnalité évoluée et flexible possède un coût, un coût qui peut être élevé.

La boucle `for` de C est très primitive. On itère de manière générale sur des entiers (un mot machine suffit qui pourra facilement occuper un registre) et quelque cycles CPU suffisent pour faire un tour. Par contre, pour itérer sur un ensemble, et bien c'est dur ! Il faut connaître le nombre d'éléments, il faut créer une variable entière, il faut initialiser cette variable à 0 et l'incrémenter à chaque tour jusqu'à atteindre le cardinal de l'ensemble. Dans le corps de chaque boucle, il faut faire de l'arithmétique sur les adresses pour obtenir l'adresse de l'objet courant... Bref, les opérations sont simples et rapides mais le programmeur a plus de travail en C.

Au final, nous vivons dans un monde où les deux langages sont nécessaires mais pas aux même endroits et aux même moments. Dans ce TP, nous allons tenter de profiter du meilleur de ces deux mondes : la flexibilité de Python avec l'efficacité du langage C.

L'objectif de ce TP est de programmer une bibliothèque Python de traitement d'images écrite en C

QUOI ???? Une bibliothèque Python écrite en C ? C'est impossible monsieur, relisez-vous !

Alors la bibliothèque sera Python et basée sur `PIL.Image`, son API sera Python, mais en interne, elle va appeler du C pour aller plus vite et la plupart du code sera ainsi en C.

Pour des questions de simplicité, nous allons nous restreindre à des images carrées de taille 512 par 512 pixels.

Du coté du langage C :

Du côté du C, il faut programmer les fonctionnalités primitives mais lourdes sous la forme d'une bibliothèque partagée. Il faut ainsi produire un catalogue de fonctions manipulant des images ici sous forme bitmap (tableau de tableaux de pixels). Le C est particulièrement efficace sur les traitements pixel par pixel (ce sont les boucles for surtout qui sont légères).

Une fois les fonctionnalités en place dans ce module C, il faut le compiler avec les trois objectifs suivants :

- On veut du code efficace
- On veut du code rechargeable partout (ne contenant aucune adresse absolue)
- On veut du code objet pouvant être capturé puis appelé par d'autres programmes

Ces trois objectifs s'obtiennent par deux lignes de compilation avec `gcc`. Si votre module s'appelle `lib_img.c`, vous devrez procéder comme il suit :

```
gcc -c -fPIC lib_img.c -o lib_img.o -Wall -ansi -O2
gcc -shared lib_img.o -o lib_img.so
```

La première ligne utilise `-O2` pour la performance et `-fPIC` pour le code relogeable partout (Position Independant Code). La seconde ligne utilise `-shared` pour produire un `shared object`. Les `shared object` sont les bibliothèques dynamiques partagées sous `Unix` (analogues des `dll` de windobe).

Le `shared object` contient maintenant des pointeurs de fonction que Python est capable de charger dynamiquement à l'exécution via la librairie standard `ctypes` de Python.

Du coté du langage Python :

Côté Python, on programme une bibliothèque creuse mais utilisant des interfaces pour déléguer le travail au code C.

Pour charger les fonctions C dans un programme Python, il faut utiliser `cdll.LoadLibrary` de la bibliothèque `ctypes` de Python. C'est l'analogue de `dlopen` du langage C, cela permet d'ouvrir une bibliothèque dynamiquement.

Vous devrez définir des types Python compatibles avec le langage C. C'est le coeur de `ctypes`. Il faudra en particulier définir un type Python compatible, d'un coté, avec `PIL.Image` et de l'autre côté avec des `unsigned char[512][512][4]` (4 pour r,g,b,a ; red, green, blue et canal alpha).

Enfin, il faudra des fonctions de conversion entre les types utilisateurs, et les types internes que l'on ne veut pas publier car ils sont juste techniques et présents pour des raisons d'interfaces.

Comme il y a beaucoup de notions/fonctionnalités à découvrir/comprendre, prenez le temps de lire, relire, modifier et exécuter le prototype téléchargeable à propos de l'inversion des couleurs d'une image.

Bibliothèque à écrire :

Téléchargez le code Python de Pierre-Yves Angrand (major 2017 national (5248 inscrits) au Capès de math option informatique). Il a produit du code Python faisant des seuillages, des diminutions/augmentations sur des canaux de couleurs, des transformations en niveaux de gris, de l'interpolation d'images, de la segmentation d'images, de la stéganographie (marquage et signature d'une image dans une autre...), des dilatations/réductions.

Téléchargez l'exemple fonctionnel d'une capacité graphique portée de Python au niveau du C. En utilisant la bibliothèque `PIL` de Python, l'exemple montre comment compiler l'inversion des couleurs d'une image. Durant l'exécution, la bibliothèque `timeit` teste les deux stratégies : le C va 300 fois plus vite que Python pour inverser les couleurs.

Pour ce TP, portez vers le langage C, en utilisant `PIL`, les fonctionnalités graphiques implémentées en Python par le sieur Angrand. Fournissez aussi un fichier Python s'assurant des tâches techniques (conversion de types dans Python...). Tentez de fournir une bibliothèque simple d'utilisation (avec des classes si vous savez faire...) cachant les détails techniques à l'utilisateur.

De manière idéale, vous devez implanter une classe wrapper de `PIL.image` de Python qui stocke de manière feignante dans Python les deux représentations possibles de l'image. Si le code est correctement feignant, il ne sert à rien de recharger l'image tant que l'utilisateur ne souhaite pas l'afficher. Dans le même esprit, tant le programmeur n'a pas fini de balancer des éléments graphiques sur son canevas, la libMLV ne détermine pas la couleur des pixels. Les fonctionnalités dites 'LAZY' sont aussi une bonne manière de mettre en place les fonctionnalités lourdes. Si vous n'avez rien compris à ce dernier paragraphe, ignorez-le ou demandez des conseils.

Pensez à documenter votre bibliothèque côté Python et à surveiller les performances de vos approches.

Pour aller plus loin :

La grosse amélioration à apporter sur une telle bibliothèque est la suivante : savoir gérer toutes les tailles d'images (rectangulaires). Si vous souhaitez relever ce challenge, il vous faudra lire pas mal de documentation sur la bibliothèque Python `ctypes`. Vous devrez peut-être rechercher des exemples, notamment avec les construction `pointer()` et `POINTER()` de `ctypes`.