ICC/Programmation — Semaine 9, Mini-projet Partie A CGC/SIE, avril 20 2026

Le mini-projet est à réaliser en binôme ou individuellement. Les groupes peuvent échanger des idées ou des approches générales, mais pas du code directement.

Contexte. Dans ce mini-projet, vous travaillerez avec deux cartes historiques. La première est *One Year After* (Sunday News, 5 mai 1946), une carte de presse représentant l'Europe d'après-guerre. La seconde est *Antiquae Urbis Romae*, une gravure en vue plongeante de la Rome antique publiée en 1588.

Le projet comporte deux parties indépendantes. La **Partie 1** utilise la carte d'Europe pour se familiariser avec les manipulations d'images de base : convertir une image couleur en niveaux de gris et appliquer un flou. La **Partie 2** utilise la carte de Rome pour un puzzle : certaines tuiles ont été découpées et mélangées, et vous devez écrire un algorithme qui détermine à quelle position appartient chacune d'elles en comparant les valeurs de pixels le long des bords.

La Partie B du mini-projet s'appuiera directement sur les fonctions que vous implémentez ici.

Préparation

Setup — create a fresh virtual environment

Note : *Passez directement à* ***Activate the virtual environment*** *si le virtual environment a déjà été créé.*

Ouvrez Visual Studio Code, ouvrez un terminal, naviguez jusqu'au dossier de votre projet, puis exécutez:

```
python3 -m venv .venv
```

Cela crée un dossier caché appelé `.venv` qui contient une copie privée de Python ainsi que les libraries que vous installerez. Rien n'est encore installé.

Activate the virtual environment

Vous devez activer le virtual environment à chaque fois que vous ouvrez un nouveau terminal. La commande dépend du système d'exploitation :

- **macOS / Linux :** `source .venv/bin/activate`
- **Windows (PowerShell) :** `.venv\Scripts\Activate.ps1`
- **Windows (Command Prompt) :** `.venv\Scripts\activate.bat`

Une fois activé, votre terminal affichera `(.venv)` au début de la ligne. **Si vous ne voyez pas ce préfixe, le virtual environment n'est pas actif et `import numpy` peut échouer. Contactez un TA ou un assistant si vous rencontrez des erreurs à cette étape.**

Install the required libraries

Faites cela une seule fois, juste après la première activation :

```
python3 -m pip install numpy Pillow
```

Téléchargez `miniproject-a-start.zip` depuis Moodle et extrayez-le dans votre espace de travail. Il contient :

- `miniprojectutils.py` — fonctions utilitaires (ne **pas** modifier ce fichier).
- `test_miniproject_a.py` — lancez ce fichier pour vérifier votre avancement (ne **pas** modifier ce fichier).
- `miniproject_a.py` — le fichier que vous compléterez. Implémentez chaque fonction là où vous voyez un commentaire `TODO`. Ne modifiez **pas** le nom des fonctions ni leurs paramètres.
- `imgs/europe.jpg` — la carte d'Europe de 1946.
- `imgs/rome.jpg` — la gravure de la Rome antique.
- `tiles/` — les tuiles manquantes de la carte de Rome.

Lancez **`miniproject_a.py`**. Il doit afficher une ligne et s'arrêter sans erreur. Si ce n'est pas le cas, appelez un assistant avant de continuer.

`miniproject_a.py` contient 7 fonctions à implémenter. Chaque fonction dispose d'une docstring expliquant ce qu'elle doit faire, de commentaires décrivant les étapes, et d'un ou plusieurs marqueurs **`TODO`** indiquant précisément où ajouter votre code. Lorsque vous implémentez un **`TODO`**, **remplacez** la ligne **`raise NotImplementedError(...)`** par votre code, sauf indication contraire dans les commentaires.

Pour vérifier votre avancement à tout moment, lancez **`test_miniproject_a.py`**. Ce fichier exécute tous les tests et affiche un ✓ en face de chaque fonction qui fonctionne correctement, et un **x** en face de celles qui ne sont pas encore implémentées ou qui contiennent un bogue. Un récapitulatif en fin d'exécution indique combien de sections sont validées. Lancez les tests fréquemment — après chaque fonction implémentée — afin de détecter les erreurs tôt, avant qu'elles n'affectent les fonctions qui en dépendent.

Représentation des images dans numpy. Une image couleur est un tableau numpy de forme $(\textit{hauteur}, \textit{largeur}, 3)$, où la dernière dimension contient les trois canaux de couleur rouge, vert et bleu (RVB), chacun étant un entier compris entre 0 et 255. Une image en niveaux de gris a la forme $(\textit{hauteur}, \textit{largeur})$: une seule valeur de luminosité par pixel. Toutes les fonctions nécessaires pour charger et enregistrer des images sont déjà fournies dans **`miniprojectutils.py`**.

Partie 1. Manipulation d'images

A. Conversion en niveaux de gris

(a) **Implémentez `rgb_to_gray(r, g, b)`.**

Cette fonction convertit un pixel couleur unique en une valeur de luminosité en niveaux de gris. Elle prend trois entiers **`r`**, **`g`**, **`b`** (chacun dans $[0, 255]$) et renvoie un **`int`** dans $[0, 255]$.

Faire la moyenne des trois canaux donne un mauvais résultat, car notre œil n'est pas également sensible à toutes les couleurs — le vert paraît environ 3,5 fois plus lumineux que le bleu. Utilisez plutôt cette formule perceptuelle :

$$\textit{gris} = 0{,}2126 \cdot r \;+\; 0{,}7152 \cdot g \;+\; 0{,}0722 \cdot b$$

Conseil : testez votre fonction sur quelques valeurs précises avant de continuer. Par exemple, **`rgb_to_gray(255, 0, 0)`** doit renvoyer 54, **`rgb_to_gray(0, 255, 0)`** doit renvoyer 182, et **`rgb_to_gray(255, 255, 255)`** doit renvoyer 255.

(b) **Implémentez `to_grayscale(img)`.**

Cette fonction convertit une image couleur complète en niveaux de gris. Elle reçoit une image couleur de forme $H \times L \times 3$ et doit renvoyer une nouvelle image en niveaux de gris de forme $H \times L$, obtenue en appelant **`rgb_to_gray`** sur chaque pixel.

Décommentez les lignes correspondantes dans **`grayscale_test`** et lancez le fichier. Il enregistre **`imgs/out/europe_gray.jpg`** ; ouvrez-le et vérifiez qu'il ressemble à l'image de droite ci-dessous.

Gauche : carte d'Europe originale en couleur. Droite : convertie en niveaux de gris.

Conseil de débogage. Si votre résultat semble incorrect, testez d'abord sur un petit tableau 3×3. Créez-en un avec `np.array([[[255, 0, 0], ...]])` et vérifiez les valeurs de pixels individuellement avec `print`. Il est bien plus facile de repérer les erreurs sur un petit tableau que sur une image complète.

B. Flou de boîte

Travailler avec de grandes images peut être lent. Une solution courante consiste à réduire la résolution de l'image avant de la traiter — mais supprimer un pixel sur deux perd de l'information. Une meilleure approche consiste à flouter d'abord l'image (en faisant la moyenne des pixels voisins), puis à sous-échantillonner. Dans cette section, vous implémentez l'étape de floutage.

Implémentez `simple_blur(img_gray, ksize)`.

Cette fonction applique un flou de boîte de taille `ksize` $\times$ `ksize` à une image en niveaux de gris. Un flou de boîte remplace chaque pixel par la moyenne du carré de `ksize` $\times$ `ksize` pixels qui l'entourent.

Vous l'implémenterez en deux passes 1D séparées — d'abord horizontale, puis verticale. Cela est équivalent au flou de boîte 2D et est plus efficace. Chaque passe utilise **`np.convolve`** (vu en cours) avec **`mode='valid'`**.

1. **Passe horizontale.** Appliquez **`np.convolve(ligne, noyau, mode='valid')`** à chaque ligne de l'image indépendamment. Chaque ligne passe de **`largeur`** à `largeur` $-$ `ksize` $+ 1$ valeurs ; la hauteur ne change pas.

2. **Passe verticale.** Appliquez la même convolution à chaque colonne du résultat de la passe 1. Chaque colonne passe de **`hauteur`** à `hauteur` $-$ `ksize` $+ 1$ valeurs.

Le noyau doit être un tableau numpy **`float64`** de longueur **`ksize`** avec toutes les valeurs égales à $1/$`ksize` (leur somme vaut donc 1). Stockez le résultat de la passe 1 en **`float64`** pour préserver la précision. Arrondissez à l'entier le plus proche à la fin de la passe 2 et stockez en **`int16`**.

Exemple : `simple_blur(img, 4)` sur une image en niveaux de gris 100×80 produit un résultat 97×77 ($100 - 3 = 97$ lignes, $80 - 3 = 77$ colonnes).

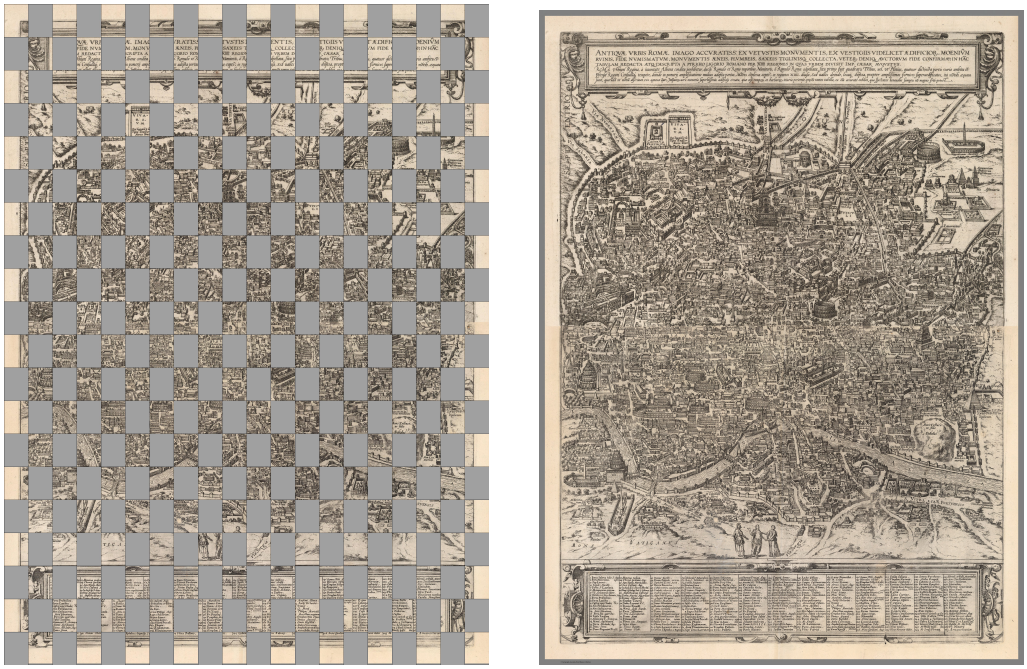
Décommentez les lignes correspondantes dans **`blur_test`** et lancez le fichier. Il enregistre **`imgs/out/europe_blurred.jpg`** ; le résultat doit ressembler à l'image ci-dessous.

Image en niveaux de gris après flou de boîte (**`ksize=4`**).

Partie 2. Reconstruction du puzzle cartographique

Mise en place

La carte de la Rome antique a été découpée en une grille de tuiles égales disposées selon un motif en damier. Les tuiles aux positions **paires** — où *ligne* + *col* est pair — sont les **tuiles données** : elles sont déjà placées et leur position dans la grille est connue. Les tuiles aux positions **impaires** — où *ligne* + *col* est impair — sont les **tuiles manquantes** : elles ont été retirées, mélangées dans un ordre aléatoire, et sauvegardées sous des noms de fichiers qui ne donnent aucun indice de position.



Gauche : le puzzle tel que donné (gris = tuiles manquantes). Droite : la carte complète correctement reconstruite.

Observation clé. Dans le motif en damier, chaque tuile manquante est entourée de tuiles données sur ses quatre côtés (ou par le bord de la grille). Cela signifie que chaque tuile manquante peut être associée indépendamment — il n'est jamais nécessaire d'attendre qu'une autre tuile manquante soit placée avant de commencer.

Ce qui vous est fourni. Le dossier **`tiles/`** contient :

- **`tile_given_r_c.jpg`** — un fichier par tuile donnée. La position $(r,\ c)$ dans la grille est encodée dans le nom du fichier.
- **`tile_missing_0.jpg`**, **`tile_missing_1.jpg`**, ...— les tuiles manquantes dans un ordre mélangé. Le numéro dans le nom de fichier n'est qu'un identifiant — ce n'est **pas** la position correcte dans la grille.
- **`partial_grid.jpg`** — un aperçu avec les tuiles données en place et des rectangles gris pour les emplacements manquants.
- **`config.json`** — lu automatiquement par **`miniprojectutils.py`** ; vous n'avez pas besoin de l'ouvrir.

Les fonctions utilitaires **`load_given_tiles`** et **`load_missing_tiles`** de **`miniprojectutils.py`** chargent les tuiles pour vous :

- **`load_given_tiles`** renvoie un dictionnaire associant **`(ligne, col)`** $\rightarrow$ image de la tuile.
- **`load_missing_tiles`** renvoie une liste d'images de tuiles indexées par numéro de fichier.

Fonctions à implémenter

(a) **Implémentez `get_edge(tile, side)`.**

Extrait une bande de pixels le long d'un bord d'une tuile sous forme de tableau numpy. **`side`** vaut l'une des chaînes **`'top'`**, **`'bottom'`**, **`'left'`**, **`'right'`**. Les tuiles sont des images couleur ; chaque pixel a trois valeurs (R, V, B) :

`'top'`	première ligne	$\rightarrow$ forme $(\textit{largeur},\ 3)$
`'bottom'`	dernière ligne	$\rightarrow$ forme $(\textit{largeur},\ 3)$
`'left'`	première colonne	$\rightarrow$ forme $(\textit{hauteur},\ 3)$
`'right'`	dernière colonne	$\rightarrow$ forme $(\textit{hauteur},\ 3)$

Pas de boucles — utilisez uniquement le découpage numpy. *Indice :* **`tile[0, :]`** donne la première ligne ; **`tile[:, -1]`** donne la dernière colonne.

Auto-test : **`get_edge(tile, 'top').shape`** doit valoir **`(tile.shape[1], 3)`**.

(b) **Implémentez `edge_difference(edge1, edge2)`.**

Renvoie un score mesurant à quel point deux bords sont différents. Les deux tableaux ont la même forme. Calculez la différence absolue pixel par pixel, faites la moyenne sur toutes les positions et tous les canaux de couleur, puis divisez par 255 pour ramener le résultat dans $[0,\ 1]$:

$$\textit{score} \ = \ \frac{\text{moyenne}(\ |\ \textit{bord1} - \textit{bord2}\ |\ )}{255} \ \in \ [0,\ 1]$$

Pas de boucles — utilisez **`np.abs`** et **`np.mean`**. Un score de 0 signifie que les bords sont pixel par pixel identiques ; un score de 1 signifie qu'ils sont aussi différents que possible.

Auto-test : **`edge_difference(e, e)`** doit renvoyer exactement **`0.0`** pour n'importe quel bord **`e`**.

(c) **Implémentez `find_best_tile(given_tiles, missing_tiles, r, c, grid, placed)`.**

Pour un emplacement vide à la position $(r,\ c)$, trouve la tuile manquante non encore placée qui s'y ajuste le mieux.

Paramètres :

- **`given_tiles`** — dictionnaire : **`(ligne, col)`** $\rightarrow$ image de la tuile, pour toutes les tuiles fixes.
- **`missing_tiles`** — liste de toutes les images de tuiles manquantes, indexées par numéro de fichier.
- **`r, c`** — coordonnées dans la grille de l'emplacement vide à remplir.
- **`grid`** — nombre de lignes (et de colonnes) de la grille.
- **`placed`** — dictionnaire associant les indices de fichiers déjà placés à leurs positions **`(ligne, col)`**. Ne **pas** réutiliser une tuile déjà présente dans **`placed`**.

Algorithme. Pour chaque tuile manquante non placée, calculez un score **`edge_difference`** moyen en utilisant tous les voisins directs de l'emplacement $(r,\, c)$ qui sont déjà connus (tuile donnée ou tuile manquante déjà placée) :

- Gauche en $(r,\, c-1)$: comparez son bord **`'right'`** avec le bord **`'left'`** du candidat.
- Haut en $(r-1,\, c)$: comparez son bord **`'bottom'`** avec le bord **`'top'`** du candidat.
- Droite en $(r,\, c+1)$: comparez le bord **`'right'`** du candidat avec son bord **`'left'`**.
- Bas en $(r+1,\, c)$: comparez le bord **`'bottom'`** du candidat avec son bord **`'top'`**.

Faites la moyenne des scores de tous les voisins disponibles (ignorez les directions correspondant à un bord de grille ou à un voisin non encore placé). Renvoyez l'indice de fichier du candidat ayant le score moyen le plus **faible**.

Indice : construisez une table de correspondance inverse **`pos_to_file = {v: k for k, v in placed.items()}`** pour retrouver rapidement l'indice de fichier occupant une position donnée.

(d) **Implémentez `reconstruct(given_tiles, missing_tiles, grid)`.**

Appelle **`find_best_tile`** pour chaque emplacement vide et assemble le placement complet.

Algorithme :

1. Construisez la liste des emplacements vides : tous les $(r,\, c)$ où $(r+c)$ est impair.
2. Initialisez **`placed = {}`**.
3. Pour chaque emplacement vide, appelez **`find_best_tile`** et ajoutez le résultat à **`placed`**.

Renvoyez **`placed`** : un dictionnaire associant chaque indice de fichier à sa position assignée $(r,\, c)$.

Décommentez les lignes dans **`reconstruct_test`** et lancez le fichier. La console affiche combien de tuiles ont été placées correctement et enregistre **`imgs/out/rome_reconstructed.jpg`**. Si toutes les tuiles sont correctes, elle affiche ✓ **Correct !**