Part II - Motion Compensation

Imports

In [ ]:
```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
```

Load and display images

Définissons quelques fonctions utiles pour charger et afficher des images.

In [ ]:
```
def load_image(path: str) -> np.ndarray:
    """
    Load image from path and convert it to grayscale

    Args:
        path (str): path to image
    Returns:
    ---- np.ndarray: grayscale image
    """
    img = cv2.imread(path,0)
    return cv.cvtColor(img, cv.COLOR_BGR2GRAY)

def display_images(images: np.ndarray, titles: list=None) -> None:
    """
    Display two images with matplotlib

    Args:
        images (np.ndarray): images to display
        titles (list, optional): titles for images. Defaults to None.
    """
    num_imgs = images.shape[0]
    fig, axis = plt.subplots(num_imgs // 2 + (num_imgs % 2), 2, figsize=(12, 8))
    if len(axis.shape) < 2:
        axis = np.expand_dims(axis, axis=0)
    for i in range(num_imgs // 2):
        for j in range(2):
            if i * 2 + j < num_imgs:
                axis[i, j].imshow(images[i * 2 + j], cmap='gray')
                if titles is not None:
                    axis[i // 2, j].set_title(titles[i * 2])

    plt.show()
```

Visualisons les données de référence de l'étude.

In [ ]:
```
pic1, pic2 = load_image('./images/1.png'), load_image('./images/2.png')
display_images(np.array([pic1, pic2]))
```

Out [ ]:

A - Backward estimation

Au TP1, vous avez réalisé de l'estimation de mouvement de type forward par blocs entre F1 et F2. Procédez maintenant à de l'estimation backward par blocs entre F1 et F2.

Note : c'est désormais F2 que l'on décompose en blocs et non pas F1.

A.1 - Algorithme de recherche exhaustive (Block Matching Algorithm)

Commencons par définir une fonction nous permettant de calculer localement la différence entre deux images, au voisinag d'un pixel donné, et sur une fenêtre donnée par les positions de ses coins. Nous allons utiliser l'erreur quadratique moyenne (MSE) pour le calcul de l'énergie E_{MSE}.

In [ ]:
```
def best_motion_vectors(frame2: np.ndarray, block_size: int, window_size: int, current_block: np.ndarray, x: int, y: int) -> tuple[int, int]:
    """
    Compute the ideal motion vectors between two frames
    Args:
        frame1 (np.ndarray): first frame
        frame2 (np.ndarray): second frame
        block_size (int): size of the block
        window_size (int): size of the window
        current_block (np.ndarray): current block
        x (int): x coordinate of the current block
        y (int): y coordinate of the current block
    Returns:
    ---- tuple[int, int]: ideal motion vectors
    """
    i, j = x, y
    h = np.inf
    dx = 0
    dy = 0
    norm = np.inf

    # Search in the window
    for k in range(1 - window_size, 1 + window_size):
        for l in range(1 - window_size, 1 + window_size):
            # Check if the block is in the window
            if i + k < 0 or i + k >= frame2.shape[0] or j + l < 0 or j + l >= frame2.shape[1]:
                continue
            # Get the new block for the current iteration
            new_block = frame2[i+k:block_size, j+l:block_size]

            # If the new block is not the same size as the current block, we skip it
            if new_block.shape != current_block.shape:
                continue

            # Compute the error between the two blocks
            E_new = np.sum(np.abs(new_block - current_block))

            # If a block is the same in both frames, we can stop the search
            if E_new == 0 and k == 0 and l == 0:
                return (0, 0)

            # Compute the distance between the two pixels at position (i, j) and (k, j)
            diff_x = i - k
            diff_y = j - l

            # If the new error minimizes the previous one, we update the error and the motion vectors
            if E_new < h:
                h = E_new
                dx = diff_x
                dy = diff_y
            elif E_new == h and np.linalg.norm((diff_x, diff_y)) < norm:
                norm = np.linalg.norm((diff_x, diff_y))
                dx = diff_x
                dy = diff_y

    return dx, dy
```

A présent, nous allons parcourir de manière naïve tous les pixels de l'image, et pour chacun d'entre eux, estimer le mouvement en calculant l'énergie E_{MSE} pour chaque vecteur de mouvement possible. Nous allons ensuite choisir le vecteur de mouvement minimisant E_{MSE}.

In [ ]:
```
def backward_BMA(frame1: np.ndarray , frame2: np.ndarray, block_size: int = 16, window_size: int = 7) -> tuple[np.ndarray]:
    """
    Compute the motion estimation between two frames using the block matching algorithm

    Args:
        frame1 (np.ndarray): first frame
        frame2 (np.ndarray): second frame
        block_size (int, optional): size of the block. Defaults to 16.
        window_size (int, optional): size of the window. Defaults to 7.
    """
    assert frame1.shape == frame2.shape, "Frames must have the same shape"

    frame_shape = frame1.shape

    # Normalize frames
    frame1 = frame1 / 255
    frame2 = frame2 / 255

    # Initialize motion vectors
    v_x = []
    v_y = []

    moving_blocks_x = []
    moving_blocks_y = []

    for i in tqdm(range(0, frame_shape[0], block_size)):
        for j in range(0, frame_shape[1], block_size):
            # Get the current block
            current_block = frame1[i:i+block_size, j:j+block_size]

            # Compute the motion vectors for the current block
            dx, dy = best_motion_vectors(frame2, block_size, window_size, current_block, i, j)

            v_x.append(dx)
            v_y.append(dy)
            moving_blocks_x.append(i)
            moving_blocks_y.append(j)

    return np.array(v_x), np.array(v_y), np.array(moving_blocks_x), np.array(moving_blocks_y)
```

A.2 - Résultats

Nous pouvons à présent chercher à visualiser les vecteurs de mouvement estimés. On se propose d'écrire une fonction calculant le champs de vecteurs de mouvments, puis qui les affichar sur l'image de référence.

In [ ]:
```
def BMA_display(frame1: np.ndarray, frame2: np.ndarray, block_size: int = 16, window_size: int = 16, scale: int = 500) -> tuple[np.ndarray]:
    """
    Compute the motion estimation between two frames using the block matching algorithm and display the results.
    A quiver plot is used to display the motion vectors.
    Args:
        frame1: first frame
        frame2: second frame
        block_size: size of the block
        window_size: size of the window
        scale: scale of the quiver plot
    Returns:
        v_x : component of the motion vectors
        v_y : component of the motion vectors
        moving_blocks_x : x coordinates of the moving blocks
        moving_blocks_y : y coordinates of the moving blocks
    """
    v_x, v_y, moving_blocks_x, moving_blocks_y = backward_BMA(frame1, frame2, block_size=block_size, window_size=window_size)
    fig, ax = plt.subplots(1, 1, figsize=(5, 5))
    ax.imshow(frame2, cmap='gray')
    ax.set_axis_off()
    ax.set_title("Backward Motion Estimation (block size: {block_size})")

    ax.quiver(moving_blocks_y + block_size // 2,
              moving_blocks_x + block_size // 2,
              v_y,
              -v_x,
              scale=scale,
              color='red',
              headwidth=2,
              headlength=2,
              headaxislength=2,
              width=0.002)

    plt.show()

    return v_x, v_y, moving_blocks_x, moving_blocks_y
```

On peut à présent visualiser les vecteurs de mouvement estimés, en définissant la taille de block et la taille de la fenêtre de recherche.

In [ ]:
```
# Define parameters (block size and window size)
BLOCK_SIZE = 40
WINDOW_SIZE = 20

# Use pic1 as first frame, that will decomposed into blocks
# Use pic2 as second frame, that will be used to find the best match for each block
v_x, v_y, moving_blocks_x, moving_blocks_y = BMA_display(pic2, pic1, block_size=BLOCK_SIZE, window_size=WINDOW_SIZE, scale=800)
```

Out [ ]:
```
  0%|          | 0/13 [00:00<?, ?it/s]
```

B - Sachant que l'on va vouloir reconstruire la frame F2 à partir de F1, pourquoi a-t-on donc fait de la ME backward et pas de la ME forward ? Que se serait-il passé sinon ? (pensez à quels artéfacts visuels on aurait obtenu)

En utilisant une fonction Block Motion Estimation, on peut reconstruire l'image **F2** à partir des blocks de **F1**. L'avantage est donc la prise en compte de potentielles nouvelles informations de **F2**, et pas seulement des blocks communs entre **F1** et **F2**.

En utilisant une forward Motion Estimation, on ne peut reconstruire l'image **F2** qu'à partir des blocks de **F1**. L'image reconstruite pourrait donc comporter des distorsions, des déformations, ou même des espaces vides entre les blocks.

C - Reconstruisez l'image F2 à partir de vos vecteurs de mouvements et de F1. Quelle est la MSE de reconstruction ?

Afin d'éviter une duplication de code, on définit une fonction qui reconstruit une image à partir d'une image de référence et d'un champ de vecteurs de mouvement.

Cependant, nous avons les coordonnées de chaque block de **F2**. Pour chaque block, on calculera donc les coordonnées du block correspondant de **F1** de la manière suivante :

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + \begin{pmatrix} dx \\ dy \end{pmatrix}$$

$$donc \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} - \begin{pmatrix} dx \\ dy \end{pmatrix}$$

In [ ]:
```
def reconstruct_frame(frame1: np.ndarray, block_size: int, v_x: np.ndarray, v_y: np.ndarray, moving_blocks_x: np.ndarray, moving_blocks_y: np.ndarray) -> np.ndarray:
    """
    Reconstruct a frame using the motion vectors and the moving blocks

    Args:
        frame1 (np.ndarray): first frame
        frame2 (np.ndarray): second frame
        block_size (int): size of the block
        v_x (np.ndarray): x component of the motion vectors
        v_y (np.ndarray): y component of the motion vectors
        moving_blocks_x (np.ndarray): x coordinates of the moving blocks
        moving_blocks_y (np.ndarray): y coordinates of the moving blocks

    Returns:
    ---- np.ndarray: reconstructed frame
    """
    frame_shape = frame1.shape
    frame2_reconstructed = np.zeros(frame_shape)

    for i in range(len(moving_blocks_x)):
        for x1, y1 in zip(moving_blocks_x[i], moving_blocks_y[i]):
            dx, dy = v_x[i], v_y[i]
            x1, y1 = x1 - dx, y1 - dy
            frame2_reconstructed[x1:x1+block_size, y1:y1+block_size] = frame1[x1:x1+block_size, y1:y1+block_size]

    return frame2_reconstructed
```

Nous pouvons maintenant l'utiliser pour reconstruire F2 à partir de F1 et des vecteurs de mouvement.

In [ ]:
```
# Reconstruct the frame
pic2_reconstructed = reconstruct_frame(pic1, BLOCK_SIZE, v_x, v_y, moving_blocks_x, moving_blocks_y)
# Compute the error image
error_image = pic2 - pic2_reconstructed

# Display the reconstructed frame
display_images(images=np.array([pic2_reconstructed, error_image]),
               titles=['First frame', 'Second frame', 'Second frame reconstructed', 'Error image'])

# Percentage of pixels that are the same in both frames
print(f"Reconstruction accuracy: {100 * (pic2.shape[0] * pic2.shape[1] - np.sum(pic2_reconstructed != pic2)) / (pic2.shape[0] * pic2.shape[1]):.2f}%")
print(f"Parameters -> block size: {BLOCK_SIZE}, window size: {WINDOW_SIZE}\n")
print(f"Reconstruction Mean Squared Error: {np.sum(np.square(pic2 - pic2_reconstructed)) / (pic2.shape[0] * pic2.shape[1]):.2f}")
```

Out [ ]:

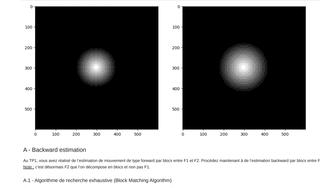

```
Reconstruction accuracy: 97.06%
Parameters -> block size: 40, window size: 20

Reconstruction Mean Squared Error: 19.19
```

Nous pouvons observer que la reconstruction de la deuxième image **F2** avec l'**estimation de mouvement** et l'image source **F1** entraîne une perte conséquente d'informations. Cela est également démontré par une **erreur quadratique moyenne élevée (MSE)**.

Cependant, il est important de souligner que la taille des blocs et la taille de la fenêtre de recherche peuvent affecter l'image résultante. Voici un exemple dans lequel nous sommes en mesure de reconstruire parfaitement la deuxième image.

In [ ]:
```
NEW_BLOCK_SIZE = 5
NEW_WINDOW_SIZE = 25

new_v_x, new_v_y, new_moving_blocks_x, new_moving_blocks_y = backward_BMA(pic2, pic1, block_size=NEW_BLOCK_SIZE, window_size=NEW_WINDOW_SIZE)

# Reconstruct the frame
new_pic2_reconstructed = reconstruct_frame(pic1, NEW_BLOCK_SIZE, new_v_x, new_v_y, new_moving_blocks_x, new_moving_blocks_y)
# Compute the error image
new_error_image = np.abs(pic2 - new_pic2_reconstructed)

# Display the reconstructed frame
display_images(images=np.array([new_pic2_reconstructed, new_error_image]),
               titles=['Second frame reconstructed', 'Error image'])

# Percentage of pixels that are the same in both frames
print(f"Reconstruction accuracy: {100 * (pic2.shape[0] * pic2.shape[1] - np.sum(pic2_reconstructed != new_pic2_reconstructed)) / (pic2.shape[0] * pic2.shape[1]):.2f}%")
print(f"Parameters -> block size: {NEW_BLOCK_SIZE}, window size: {NEW_WINDOW_SIZE}\n")
print(f"Reconstruction Mean Squared Error: {np.sum(np.square(pic2 - new_pic2_reconstructed)) / (pic2.shape[0] * pic2.shape[1]):.2f}")
```

Out [ ]:
```
  0%|          | 0/128 [00:00<?, ?it/s]
```



```
Reconstruction accuracy: 100.00%
Parameters -> block size: 5, window size: 25

Reconstruction Mean Squared Error: 0.00
```

A partir de ces deux exemples, nous pouvons observer une tendance de variation de l'exactitude de la reconstruction et de l'erreur quadratique moyenne (MSE). En effet, avec une taille de bloc assez petite et une taille de fenêtre beaucoup plus grande, l'exactitude de la reconstruction des images tend à être plus élevée.

D - Supposant que l'on n'aura plus accès à F2, quelles sont les 3 données "classiques" (en plus du paramètres de taille de bloc) dont on a besoin pour la reconstruire de façon parfaite en compensation de mouvement ?

Pour reconstruire parfaitement l'image **F2** en compensation de mouvement, nous avons besoin des données suivantes :

- La taille des blocs
- L'image de référence **F1**
- Le champ de vecteurs de mouvement (calculé en Backward estimation)
- L'image d'erreur $\varepsilon_{F2} = F2 - F1$

Ainsi, nous pouvons reconstruire l'image de référence **F2** en compensation de mouvement en utilisant la formule suivante :

$$F2 = F1 + \varepsilon_{F2}$$

où **F2'** est l'image reconstruite.

E - Reconstruisez F2 à partir de ces 3 données. Quelle est désormais la MSE de reconstruction ?

In [ ]:
```
perfect_reconstruction = pic2_reconstructed + error_image

# Display the reconstructed frame
display_images(images=np.array([perfect_reconstruction, error_image]), titles=['Perfect reconstruction', 'Error image'])

# Percentage of pixels that are the same in both frames
print(f"Reconstruction accuracy: {np.sum(pic2 - perfect_reconstruction) * 1e-8} / {(pic2.shape[0] ** 2) * 100:.2f}%")
print(f"Parameters -> block size: {BLOCK_SIZE}, window size: {WINDOW_SIZE}\n")
print(f"Reconstruction Mean Squared Error: {np.sum(np.square(pic2 - perfect_reconstruction)) / (pic2.shape[0] ** 2):.2f}")
```

Out [ ]:


```
Reconstruction accuracy: 100.00%
Parameters -> block size: 40, window size: 20

Reconstruction Mean Squared Error: 0.00
```

On peut observer de ces résultats que la reconstruction **F2'** correspond parfaitement à l'image de référence **F2**. En effet, l'**erreur quadratique moyenne (MSE)** est nulle, et la reconstruction accuracy est de 100%.

On vérifie alors la formule : $F2 = F1' + \varepsilon_{F2}$

F - Expliquez quelles données on peut coder avec perte pour garder une qualité de reconstruction de F1 et F2 "correcte". Justifiez vos explications par des images de rendus.

- On pourrait d'abord encoder l'image **F1**, puisque c'est l'image de référence. Pour cela, on peut utiliser de la compression avec quantification de la DCT. La DCT (Discrete Cosine Transform) est une transformée de Fourier discrète qui permet de passer d'une image spatiale à une image fréquentielle. La quantification peut être effectuée avec un algorithme de **run-length encoding**. Cela permet de réduire la taille de l'image avec perte, tout en gardant une qualité de reconstruction correcte.
- On pourrait ensuite encoder l'image d'erreur $\varepsilon_{F2} = F2 - F1$. On pourrait l'encoder de la même manière que l'image de référence **F1**.
- Enfin, on pourrait compresser le champ de vecteurs de mouvement. Pour cela, on pourrait un down-sampling du champ de vecteurs de mouvement pour réduire le nombre de vecteurs, puis faire de l'interpolation pour reconstruire le champ de vecteurs de mouvement.

N.B.: Je n'ai pas réussi à faire fonctionner la compression des différentes données. J'ai essayé de compresser l'image de référence **F1** avec de la DCT, puis de la quantifier avec un algorithme de **run-length encoding**. Cependant, je n'ai pas réussi à réduire la taille en bytes des données, car l'image **F1** est déjà de type `uint8`. De plus petit format supporté par numpy).