



Semestre 9 en Intelligence Artificielle

Analyse de Vidéos

Projet 1 : Comparaison de méthodes, avec ou sans Deep Learning, de calcul de flot optique dense

Auteurs

Bonhomme Romain
Pierre Romain

Professeur

Jenny Benois-Pineau

Assistant de Projet

Boris Mansencal

Année Académique 2024-2025

1 Description du Code

Dans cette première section, deux méthodes de calcul de flot optique dense seront étudiées, l'une utilisant des méthodes de Deep Learning et l'autre pas.

1.1 Méthode Classique (sans Deep Learning) : Lukas-Kanade

1.1.1 Explication de la méthode

Le code implémente un algorithme de flot optique basé sur la méthode de Farneback, qui constitue une extension de la méthode de Lucas-Kanade. Cette méthode repose sur l'approximation locale des mouvements dans une séquence d'images. Elle part du principe que l'intensité d'un pixel reste constante entre deux images successives, ce qui est exprimé par l'équation d'égalité de l'intensité lumineuse :

$$I(x, y, t) = I(x + u, y + v, t + 1),$$

où (u, v) représentent les composantes du déplacement (flot optique) et $I(x, y, t)$ l'intensité à la position (x, y) et au temps t . En développant cette équation par approximation de Taylor et en négligeant les termes non linéaires, on obtient une équation linéaire :

$$I_x u + I_y v + I_t = 0,$$

où I_x , I_y , et I_t désignent respectivement les dérivées spatiales et temporelles de l'intensité.

Pour résoudre ce problème, la méthode de Lucas-Kanade suppose que le flot optique est constant dans une fenêtre locale autour de chaque pixel. Dans cette fenêtre, un système surdéterminé est construit à partir des équations linéaires des pixels voisins. Ce système peut être exprimé sous forme matricielle :

$$A\mathbf{v} = \mathbf{b},$$

où A est une matrice contenant I_x et I_y pour chaque pixel de la fenêtre, $\mathbf{v} = (u, v)^T$ est le vecteur de déplacement recherché, et $\mathbf{b} = -I_t$. Ce système est résolu par la méthode des moindres carrés, qui donne :

$$\mathbf{v} = (A^T A)^{-1} A^T \mathbf{b}.$$

Le résultat est un champ de flot optique approximatif pour chaque pixel, obtenu en itérant cette procédure sur toute l'image.

La méthode de Farneback étend cette approche en traitant le problème à une échelle plus globale et en modélisant le flot optique à l'aide d'un modèle quadratique des variations locales de l'intensité lumineuse. Au lieu de supposer un flot optique constant dans une petite fenêtre locale, Farneback suppose une approximation polynomiale pour décrire le mouvement dans un voisinage plus large. Les coefficients de ce polynôme sont déterminés à partir de l'intensité et des dérivées d'ordre supérieur dans l'image, permettant ainsi une estimation du flot optique plus robuste, notamment pour des mouvements plus rapides et dans des zones à faible texture. Ce modèle est particulièrement adapté aux scènes avec des déformations complexes ou des mouvements non linéaires.

1.1.2 Explication du code

Le script Python *main_lukas_kanade.py* implémente le calcul du flot optique à l'aide de la méthode de Lucas-Kanade (Farneback), basé sur l'implémentation fournie par OpenCV¹. Il traite une séquence d'images et les fichiers de flot optique correspondants, en évaluant les erreurs entre le flot optique estimé et le flot de référence. Les outils utilisés incluent OpenCV pour le calcul de flot, PyTorch pour le traitement tensoriel, et Matplotlib pour la visualisation.

La fonction principale, *compute_optical_flow_lk*, lit une série d'images et leurs fichiers de flot *flo*, calcule le flot optique à l'aide de la fonction *cv2.calcOpticalFlowFarneback* d'OpenCV, et évalue plusieurs métriques. Ces métriques incluent l'erreur de point final moyenne (aEPE), l'erreur angulaire moyenne (aAE) et l'erreur quadratique moyenne (MSE). Les fonctions utilitaires comme *read_flo_file* et *compute_metrics* permettent respectivement de lire les fichiers de flot et d'effectuer des comparaisons précises entre les images compensées et d'origine.

Enfin, une visualisation du flot optique est réalisée par la fonction *viz*, qui mappe le flot estimé à une image RGB. Cette visualisation peut être visualisée aux FIG.5 et 6.

1.2 Méthode utilisant du Deep Learning : RAFT

1.2.1 Explication de la méthode

La méthode RAFT (Recurrent All-Pairs Field Transforms) repose sur une approche dense et itérative pour le calcul du flot optique. Contrairement aux techniques traditionnelles qui estiment localement le flot, RAFT génère des correspondances globales en utilisant une représentation d'all-pairs (toutes les paires) entre les pixels des deux images. Ces correspondances sont raffinées de manière itérative à l'aide d'un réseau neuronal récurrent, ce qui permet de modéliser des relations complexes tout en garantissant une convergence précise et cohérente.

Le pipeline RAFT commence par l'extraction d'une représentation de caractéristiques des deux images à l'aide d'un réseau convolutionnel. Ensuite, un volume de corrélation 4D est construit pour chaque paire de pixels entre les images. À chaque itération, une couche de mise à jour exploite ce volume pour ajuster le champ de flot optique initialisé. L'actualisation du flot est basée sur un module de réseau récurrent, comme le GRU (Gated Recurrent Unit), qui intègre des informations globales sur les correspondances tout en conservant une mémoire des itérations précédentes.

L'objectif d'optimisation de RAFT est souvent formulé par une fonction de perte entre le flot estimé \mathbf{u} et le flot de vérité terrain \mathbf{u}_{gt} , utilisant des métriques comme l'erreur de point final (EPE). Grâce à son implémentation, RAFT est capable de traiter efficacement des déformations complexes et des petits détails tout en maintenant une grande précision, ce qui le rend particulièrement adapté aux tâches exigeantes de flot optique.

1. https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

1.2.2 Explication du code

Le script `main_raft.py` implémente le calcul du flot optique basé sur le modèle RAFT².

Plusieurs fonctions utilitaires facilitent les prétraitements et l'analyse. La fonction `load_image` charge une image et la transforme en tenseur PyTorch, tandis que `read_flo_file` lit les fichiers de flot optique au format `.flo`. Ces fichiers contiennent les informations de déplacement pixel-par-pixel et sont redimensionnés avec une interpolation bilinéaire à une taille standard (440×1024).

Une autre fonction, `viz`, sert à visualiser les données. Les images d'entrée, les flux estimés et les flux au sol (ground truth) sont combinés et affichés sous forme RGB. Le flot optique est transformé en représentation visuelle avec des couleurs distinctes via une fonction de mappage, ce qui permet une comparaison qualitative entre les flux estimés et les flux réels. Cette visualisation peut être visualisée aux FIG.7 et 8.

2. <https://github.com/princeton-vl/RAFT>

2 Analyse des courbes aEPE, aAE et MSE

2.1 Description des métriques

Cette première partie détaille la définition des 3 erreurs utilisées pour mesurer les performances des méthodes implémentées.

2.1.1 average End Point Error (aEPE)

L'erreur "Average End Point Error" (aEPE) est une métrique qui mesure la distance euclidienne moyenne entre les points finaux des vecteurs de flot estimés et ceux des vecteurs de flot réels (ground truth). Si $\mathbf{u}_{est} = (u_{est}, v_{est})$ représente le flot estimé et $\mathbf{u}_{gt} = (u_{gt}, v_{gt})$ représente le flot réel, l'aEPE est définie par la formule suivante :

$$aEPE = \frac{1}{N} \sum_{i=1}^N \sqrt{(u_{est,i} - u_{gt,i})^2 + (v_{est,i} - v_{gt,i})^2},$$

où N est le nombre total de pixels dans l'image, et $(u_{est,i}, v_{est,i})$ et $(u_{gt,i}, v_{gt,i})$ sont les composants horizontal et vertical du flot estimé et réel au pixel i .

Cette erreur est une mesure de la précision globale du flot optique estimé en termes de la distance moyenne entre les vecteurs de déplacement estimés et réels pour tous les pixels. Une faible valeur d'aEPE indique que les vecteurs estimés sont proches de ceux réels, tandis qu'une valeur élevée suggère une erreur importante dans l'estimation du mouvement à travers l'image. L'aEPE est souvent utilisée pour comparer différentes méthodes de calcul du flot optique, car elle offre une mesure simple et directe de la qualité de l'estimation du flot sur l'ensemble de l'image.

2.1.2 average Angular Error (aAE)

L'erreur "Average Angular Error" (aAE) est une métrique utilisée pour évaluer la précision des directions des champs de flot optique estimés. Elle mesure l'angle moyen entre les vecteurs de flot estimés et les vecteurs de flot réels (ground truth). Si $\mathbf{u}_{est} = (u_{est}, v_{est})$ représente le flot estimé et $\mathbf{u}_{gt} = (u_{gt}, v_{gt})$ représente le flot réel, l'aAE est définie par la formule suivante :

$$aAE = \frac{1}{N} \sum_{i=1}^N \cos^{-1} \left(\frac{\mathbf{u}_{est,i} \cdot \mathbf{u}_{gt,i}}{|\mathbf{u}_{est,i}| |\mathbf{u}_{gt,i}|} \right),$$

où $\mathbf{u}_{est,i} \cdot \mathbf{u}_{gt,i}$ est le produit scalaire des vecteurs de flot estimé et réel au pixel i , et $|\mathbf{u}_{est,i}|$ et $|\mathbf{u}_{gt,i}|$ sont les normes des vecteurs de flot estimé et réel. L'arc cosinus donne l'angle entre les deux vecteurs.

L'aAE mesure l'écart moyen entre les directions des vecteurs de flot estimés et réels. Une faible valeur de aAE indique que les directions estimées sont proches de celles réelles, tandis qu'une valeur élevée suggère que les directions sont mal estimées. Cette métrique est particulièrement utile lorsque l'on s'intéresse à la précision des orientations des mouvements, indépendamment de leur amplitude. Comme l'aEPE, l'aAE est souvent utilisée pour évaluer et comparer la performance des différentes méthodes d'estimation du flot optique.

2.1.3 Mean Squared Error (MSE)

L'erreur "Mean Squared Error" (MSE) est une métrique utilisée pour évaluer la différence globale entre l'image d'origine et l'image compensée par le flot optique estimé. Cette erreur est définie comme la moyenne des carrés des différences entre les pixels correspondants des deux images. Si I_1 représente l'image d'origine et I'_2 l'image compensée, la MSE est donné par la formule suivante :

$$\text{MSE} = \frac{1}{N} \sum_{i=1}^N (I_1(x_i, y_i) - I'_2(x_i, y_i))^2,$$

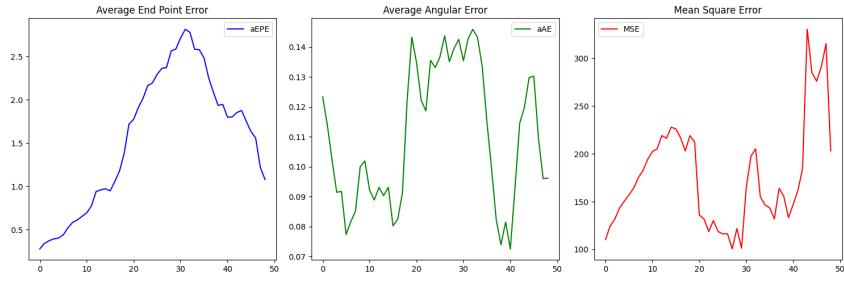
où N est le nombre total de pixels dans l'image, et (x_i, y_i) représente les coordonnées de chaque pixel. L'image compensée I'_2 est obtenue en utilisant le flot optique estimé pour transformer l'image I_2 afin qu'elle corresponde à l'image I_1 .

La MSE mesure la déviation quadratique moyenne entre les deux images, et plus cette valeur est faible, plus les images sont similaires. Dans le contexte du flot optique, un MSE faible indique que l'estimation du flot optique a bien compensé l'image initiale, c'est-à-dire que les déplacements des pixels sont correctement prédits. Cette métrique est souvent utilisée pour évaluer la qualité de la compensation d'image, et elle est particulièrement utile lorsque l'on veut minimiser l'erreur globale entre l'image d'origine et l'image déformée.

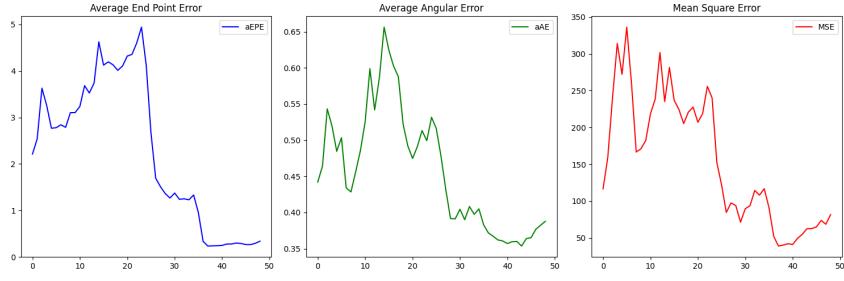
2.2 Évolution des Erreurs

En regardant les performances des 2 méthodes sur le dataset 'Sintel' (visibles sur les FIG.1 et 3), on remarque que la méthode RAFT surpassé toujours la méthode classique en terme de *aEPE*, avec des valeurs 3 à 10 fois inférieures et de *AAE*, diminuant elle d'un facteur variant entre 2 et 7 selon les séquences. La MSE est élevée dans les 2 cas mais ce n'est pas très étonnant étant donné que cette erreur n'est pas très pertinente sur ce dataset.

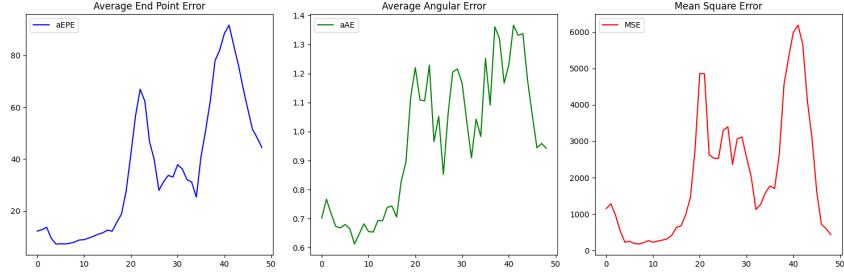
De manière assez surprenante, les FIG.2 et 4 montrent que pour les séquences appelées *Can* et *Rice*, il semblerait que la méthode classique surpassé la méthode RAFT en terme de MSE, signifiant que cette méthode classique tendrait à réaliser de meilleures prédictions. Mais étant donné la valeur qui reste assez haute, il est possible que la méthode RAFT n'ait juste pas été entraînée assez longtemps, ce qui pourrait entraîner un apprentissage partiel du RNN.



(a) Alley

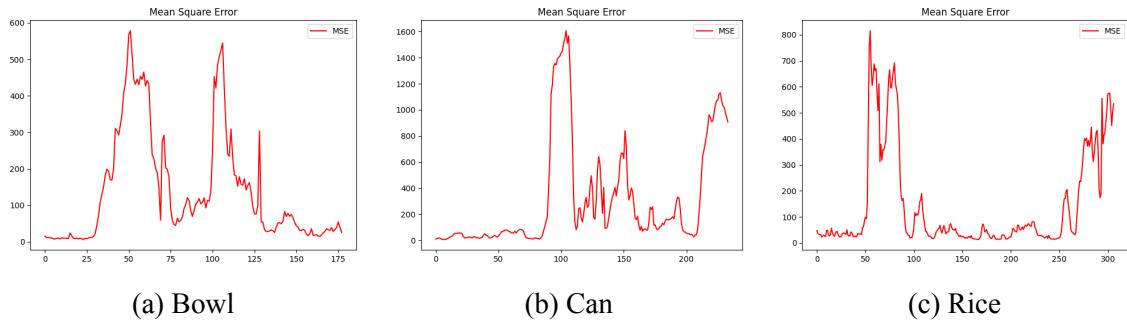


(b) Market



(c) Temple

Figure 1 – Evolution des trois erreurs par la méthode de Lukas-Kanade (Farneback) sur le dataset "Sintel" sur les 3 images d'entrée sélectionnées.

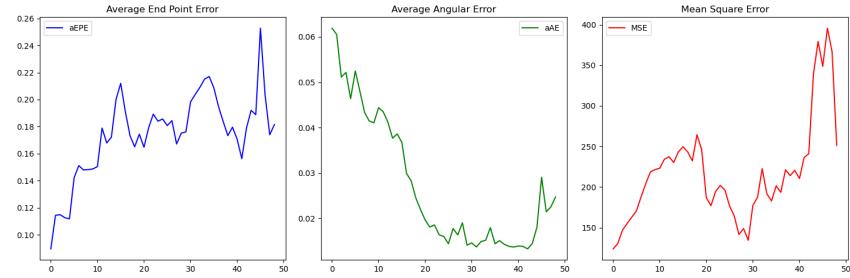


(a) Bowl

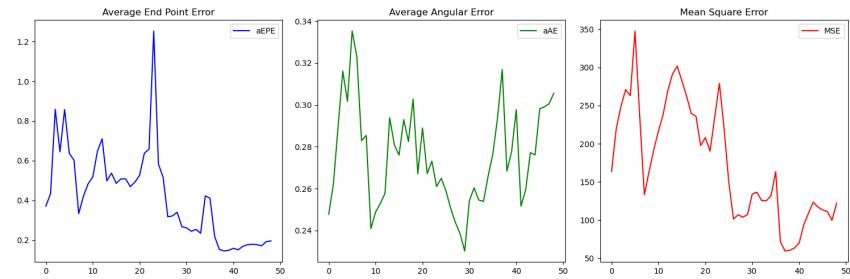
(b) Can

(c) Rice

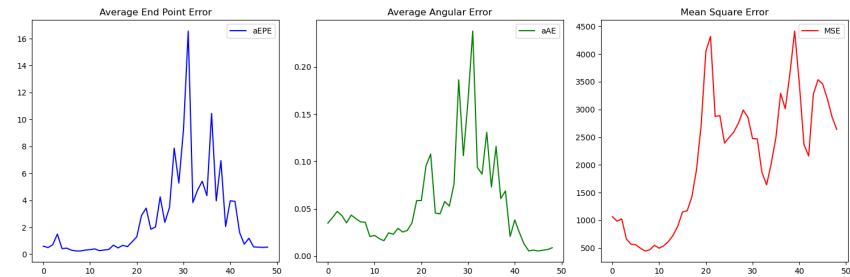
Figure 2 – Evolution de la MSE par la méthode de Lukas-Kanade (Farneback) sur le dataset "GITW" sur les 3 images d'entrée sélectionnées.



(a) Alley

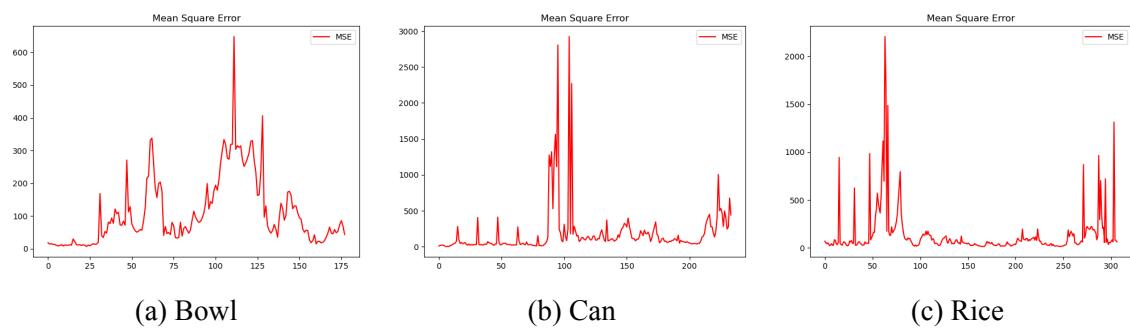


(b) Market



(c) Temple

Figure 3 – Evolution des trois erreurs par la méthode de RAFT sur le dataset "Sintel" sur les 3 images d'entrée sélectionnées.



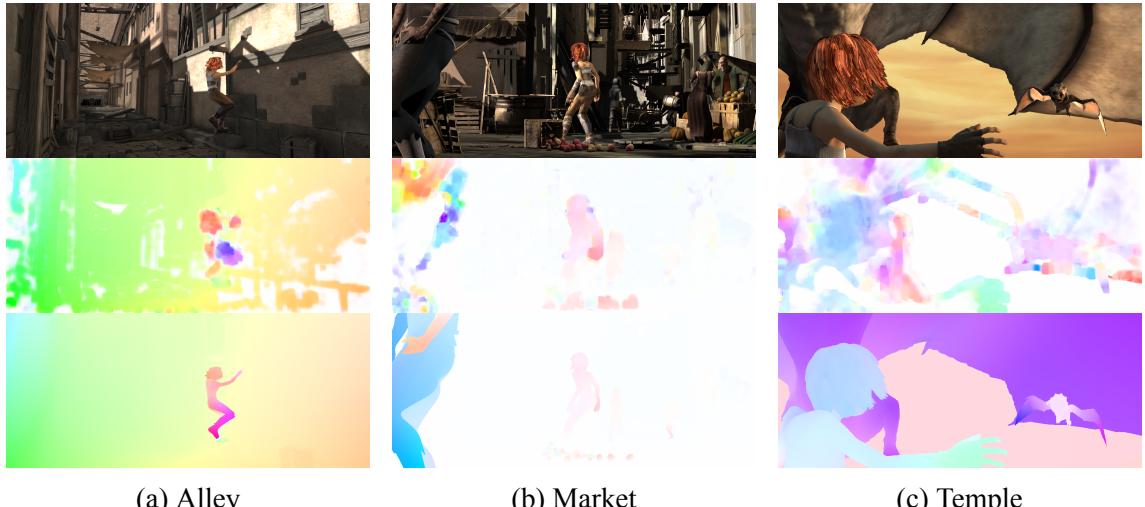
(a) Bowl

(b) Can

(c) Rice

Figure 4 – Evolution de la MSE par la méthode de RAFT sur le dataset "GITW" sur les 3 images d'entrée sélectionnées.

3 Images d'exemple

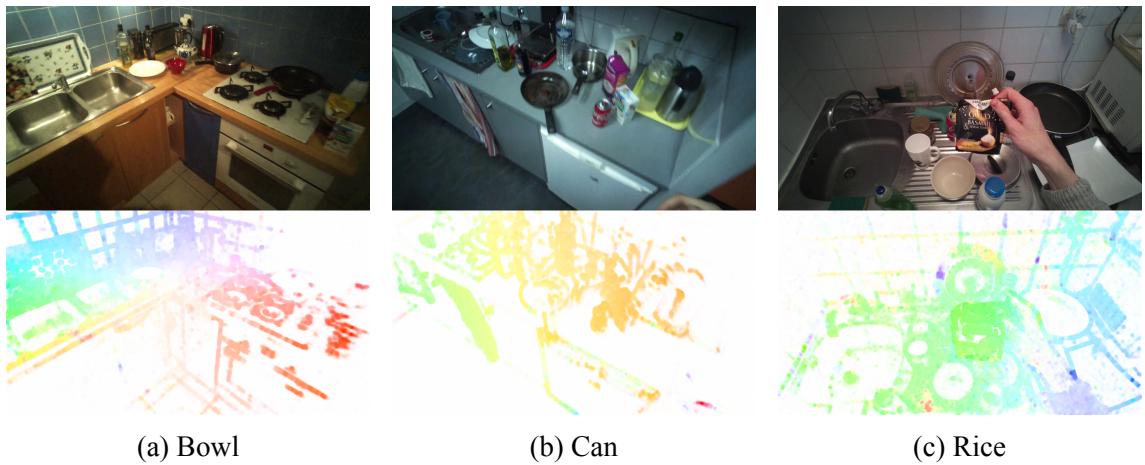


(a) Alley

(b) Market

(c) Temple

Figure 5 – Résultats obtenus par la méthode de Lukas-Kanade (Farneback) sur le dataset "Sintel". Pour chaque image, la première représente l'image d'origine, celle du milieu représente la prédiction du mouvement tandis que la dernière représente le mouvement réel.

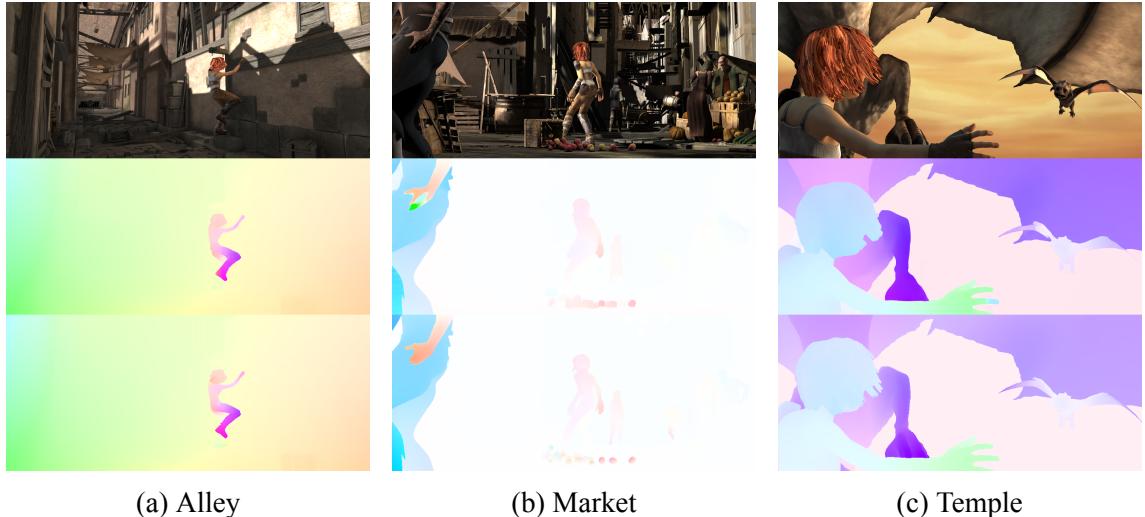


(a) Bowl

(b) Can

(c) Rice

Figure 6 – Résultats obtenus par la méthode de Lukas-Kanade (Farneback) sur le dataset "GITW". Pour chaque image, celle du dessus représente l'image d'origine tandis que celle du milieu représente la prédiction du mouvement.

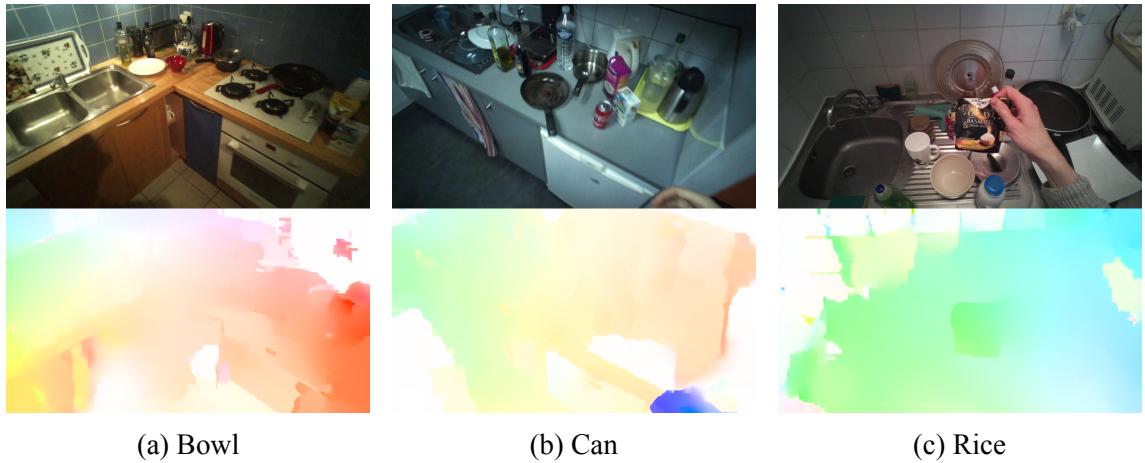


(a) Alley

(b) Market

(c) Temple

Figure 7 – Résultats obtenus par la méthode de RAFT sur le dataset ”Sintel”. Pour chaque image, la première représente l’image d’origine, celle du milieu représente la prédiction du mouvement tandis que la dernière représente le mouvement réel.



(a) Bowl

(b) Can

(c) Rice

Figure 8 – Résultats obtenus par la méthode de RAFT sur le dataset ”GITW”. Pour chaque image, celle du dessus représente l’image d’origine tandis que celle du milieu représente la prédiction du mouvement.

4 Annexes

Listing 1 – Compute metrics

```
1 def compute_metrics(img1, img2, flow_estimated, flow_gt):
2     """
3         Compute the average End Point Error (aEPE),
4         average Angular Error (aAE), and
5         Mean Square Error (MSE) between the original and compensated images.
6
7     Parameters:
8         img1 (torch.Tensor): The first image (1, 3, H, W).
9         img2 (torch.Tensor): The second image (1, 3, H, W).
10        flow_estimated (torch.Tensor): Estimated optical flow (2, H, W).
11        flow_gt (torch.Tensor): Ground-truth optical flow (2, H, W).
12
13    Returns:
14        tuple: (aEPE, aAE, mse)
15            - aEPE: Average End Point Error.
16            - aAE: Average Angular Error.
17            - mse: Mean Square Error between img1 and the compensated
18                img2.
19        """
20
21    # Remove batch dimension
22    img1 = img1[0]    # Shape: (3, H, W)
23    img2 = img2[0]    # Shape: (3, H, W)
24
25    # Image dimensions
26    _, H, W = img1.shape
27
28    # Meshgrid for pixel coordinates
29    x = torch.arange(W, device=flow_estimated.device).view(1, -1)
30        .expand(H, W)
31    y = torch.arange(H, device=flow_estimated.device).view(-1, 1)
32        .expand(H, W)
33
34    x_comp = x + flow_estimated[0]    # Add horizontal flow
35    y_comp = y + flow_estimated[1]    # Add vertical flow
36
37    # Interpolate img2 to create the compensated image for each channel
38    compensated_img = torch.zeros_like(img1)
39    for c in range(3):
40        compensated_img[c] = bilinear_interpolation(
41            img2[c], x_comp, y_comp
42        )
43
44    # Mean Square Error (MSE)
45    mse = torch.mean((img1 - compensated_img) ** 2).item()
46
47    # End Point Error (EPE)
48    epe = torch.sqrt(
49        (flow_estimated[0] - flow_gt[0]) ** 2
50        + (flow_estimated[1] - flow_gt[1]) ** 2
51    )
52    aEPE = torch.mean(epe).item()
```

```

53     # Angular Error (AE)
54     dot_product = flow_estimated[0] * flow_gt[0] +
55         flow_estimated[1] * flow_gt[1]
56     norm_est = torch.sqrt(flow_estimated[0] ** 2 +
57         flow_estimated[1] ** 2)
58     norm_gt = torch.sqrt(flow_gt[0] ** 2 + flow_gt[1] ** 2)
59     cos_theta = torch.clamp(dot_product /
60         (norm_est * norm_gt + 1e-6), -1.0, 1.0)
61     angular_error = torch.acos(cos_theta)
62     aAE = torch.mean(angular_error).item()
63
64     return aEPE, aAE, mse

```

Listing 2 – Bilinear interpolation

```

1 def bilinear_interpolation(img, x, y):
2     """
3     Perform bilinear interpolation for image sampling.
4
5     Parameters:
6         img (torch.Tensor): Input image (H, W).
7         x (torch.Tensor): X-coordinates for sampling.
8         y (torch.Tensor): Y-coordinates for sampling.
9
10    Returns:
11        torch.Tensor: Interpolated image.
12    """
13    H, W = img.shape
14
15    x0 = torch.floor(x).long()
16    x1 = x0 + 1
17    y0 = torch.floor(y).long()
18    y1 = y0 + 1
19
20    x0 = torch.clamp(x0, 0, W - 1)
21    x1 = torch.clamp(x1, 0, W - 1)
22    y0 = torch.clamp(y0, 0, H - 1)
23    y1 = torch.clamp(y1, 0, H - 1)
24
25    Ia = img[y0, x0]
26    Ib = img[y1, x0]
27    Ic = img[y0, x1]
28    Id = img[y1, x1]
29
30    wa = (x1 - x) * (y1 - y)
31    wb = (x1 - x) * (y - y0)
32    wc = (x - x0) * (y1 - y)
33    wd = (x - x0) * (y - y0)
34
35    return wa * Ia + wb * Ib + wc * Ic + wd * Id

```

Listing 3 – Lukas-Kanade

```

1 def compute_optical_flow_lk(images_path, flo_path):
2     aEPEs = []
3     aAEs = []

```

```

4     MSEs = []
5
6     images = sorted(
7         glob.glob(os.path.join(images_path, "*.png"))
8         + glob.glob(os.path.join(images_path, "*.jpg"))
9     )
10
11    flows = sorted(glob.glob(os.path.join(flo_path, "*.flo")))
12
13    if len(images) < 2:
14        raise ValueError("Needs at least 2 images.")
15
16    optical_flows = []
17
18    prev_img = cv2.imread(images[0], cv2.IMREAD_GRAYSCALE)
19    for i in range(1, len(images)):
20        next_img = cv2.imread(images[i], cv2.IMREAD_GRAYSCALE)
21
22        flow = cv2.calcOpticalFlowFarneback(
23            prev_img,
24            next_img,
25            None,
26            pyr_scale=0.5,
27            levels=3,
28            winsize=15,
29            iterations=3,
30            poly_n=5,
31            poly_sigma=1.2,
32            flags=0,
33        )
34
35        flow_data = read_flo_file(flows[i - 1])
36        flow_gt = flow_data.float().to(DEVICE)
37
38        torch_flow = torch.from_numpy(flow).permute(2, 0, 1).float()
39        .to(DEVICE)
40
41        if i == len(images) // 2:
42            viz(
43                load_image(images[i - 1]), torch_flow.unsqueeze(0), flow_gt
44            )
45
46        optical_flows.append(flow)
47
48        aEPE, aAE, MSE = compute_metrics(
49            load_image(images[i - 1]),
50            load_image(images[i]),
51            torch_flow,
52            flow_gt,
53        )
54        aEPEs.append(aEPE)
55        aAEs.append(aAE)
56        MSEs.append(MSE)
57
58        prev_img = next_img
59

```

```

60     print("aEPE:", np.mean(aEPEs))
61     print("aAE:", np.mean(aAEs))
62     print("MSE:", np.mean(MSEs))
63
64     # Ploting
65     plt.figure(figsize=(15, 5))
66
67     # Plot aEPE
68     plt.subplot(1, 3, 1)
69     plt.plot(aEPEs, label="aEPE", color="b")
70     plt.legend()
71     plt.title("Average End Point Error")
72
73     # Plot aAE
74     plt.subplot(1, 3, 2)
75     plt.plot(aAEs, label="aAE", color="g")
76     plt.legend()
77     plt.title("Average Angular Error")
78
79     # Plot MSE
80     plt.subplot(1, 3, 3)
81     plt.plot(MSEs, label="MSE", color="r")
82     plt.legend()
83     plt.title("Mean Square Error")
84
85     plt.tight_layout()
86     plt.show()
87
88     return optical_flows

```

Listing 4 – RAFT

```

1 def compute_raft_optical_flow(args):
2     model = torch.nn.DataParallel(RAFT(args))
3     model.load_state_dict(torch.load(args.model))
4
5     model = model.module
6     model.to(DEVICE)
7     model.eval()
8
9     aEPEs = []
10    aAEs = []
11    MSEs = []
12
13    with torch.no_grad():
14        images = sorted(
15            glob.glob(os.path.join(args.path, "*.png"))
16            + glob.glob(os.path.join(args.path, "*.jpg"))
17        )
18        flows = sorted(glob.glob(os.path.join(args.path_flo, "*.flo")))
19
20        for idx, (imfile1, imfile2) in enumerate(
21            zip(images[:-1], images[1:])
22        ):
23            image1 = load_image(imfile1)
24            image2 = load_image(imfile2)

```

```

25
26     flow_data = read_flo_file(flows[idx])
27     flow_gt = flow_data.float().to(DEVICE)
28
29     padder = InputPadder(image1.shape)
30     image1, image2 = padder.pad(image1, image2)
31
32     flow_low, flow_up = model(
33         image1, image2, iters=20, test_mode=True
34     )
35     if idx == len(images) // 2:
36         viz(image1, flow_up, flow_gt)
37
38     flow_up = flow_up.squeeze()
39
40     aEPE, aAE, MSE = compute_metrics(
41         image1, image2, flow_up, flow_gt
42     )
43     aEPEs.append(aEPE)
44     aAEs.append(aAE)
45     MSEs.append(MSE)
46
47     print("aEPE:", np.mean(aEPEs))
48     print("aAE:", np.mean(aAEs))
49     print("MSE:", np.mean(MSEs))
50
51     # Ploting
52     plt.figure(figsize=(15, 5))
53
54     # Plot aEPE
55     plt.subplot(1, 3, 1)
56     plt.plot(aEPEs, label="aEPE", color="b")
57     plt.legend()
58     plt.title("Average End Point Error")
59
60     # Plot aAE
61     plt.subplot(1, 3, 2)
62     plt.plot(aAEs, label="aAE", color="g")
63     plt.legend()
64     plt.title("Average Angular Error")
65
66     # Plot MSE
67     plt.subplot(1, 3, 3)
68     plt.plot(MSEs, label="MSE", color="r")
69     plt.legend()
70     plt.title("Mean Square Error")
71
72     plt.tight_layout()
73     plt.show()

```