ECE634 Project 1

Yezhi Shen, shen397@purdue.edu

Feburary 2023

1 Logic

1.1 Exhaustive block matching algorithm

Exhaustive Block Matching Algorithm (EBMA) is a motion estimation technique used in video compression to determine motion vectors between two consecutive frames. The goal of motion estimation is to find the best match between a block of pixels in the current frame and a similar block in the previous frame, so that motion vectors can be used to describe the motion between two frames. The steps of EBMA are:

- Divide the current frame into small, non-overlapping blocks of pixels, of size NxN.
- Select a block in the current frame: The algorithm starts by selecting the first block in the current frame.
- Calculate the mean absolute differences (MAD) for each block in the previous frame.
- Select the block with the minimum MAD and store motion vectors.
- Warp the block according to the calculated motion vectors.

1.2 Hierarchical block matching algorithm

HBMA is an optimization of the Exhaustive Block Matching Algorithm (EBMA) that reduces the computational complexity of motion estimation by dividing the search space into smaller regions and searching for the best match in each region in a hierarchical manner.
The steps of HBMA are:

- Iteratively performs the following steps
- Divide the current frame into small, non-overlapping blocks of pixels, of size NxN.

- Select a block in the current frame: The algorithm starts by selecting the first block in the current frame.
- Calculate the mean absolute differences (MAD) for each block in the previous frame.
- Select the block with the minimum MAD and store motion vectors.
- Reduce the block number by 2 and repeat the above steps.
- Warp the block according to the calculated motion vectors.

1.3 RANSAC

The fundamentals of the RANSAC algorithm is using randomly selected least amount of data to construct an estimation and the ascertain the extent of support that the rest of the data provides to the estimates. In homography reconstruction, we first select the minimal number of points necessary for the estimation: 4 pairs. Then we use the H obtained from the 4 pairs to project the key points x to its target x'. We reject the outliers that have euclidean distance greater than delta, which is the threshold. After the above steps, the inliers will be kept and the above steps will be done repetitively. The final output will be the most inliers in the previous runs. Finally, with the inliers set, we will estimate the homography using the full filtered set.

1.4 Levenberg-Marquardt Algorithm

The solution of Gauss Newton method is given by

$$\delta_p = (J_f^T J_f)^{-1} J_f^T \epsilon(p)$$

Then we can rewrite the fomular as

$$(J_f^T J_f)\delta_p = J_f^T \epsilon(p)$$

Ideally, the solution given by this method should be the same as the gradient descent method, but in the risk that $J_f^T J_f$ is not purely diagonal. So, a damping coefficient is added to this formula

$$(J_f^T J_f + \mu I)\delta_p = J_f^T \epsilon(p)$$

2 Implementation

2.1 EMBA

Given the two images (Figure.24) as inputs, they are first divided to non-overlapping blocks of size 16x16. Two additional parameters: search range and half pel accuracy are required.

For the half-pel accuracy implementation, the images used for matching are up-sampled to 2x the resolution using bilinear interpolation.

Figure 1: Inputs

2.2 HMBA

The HMBA implementation requires the same input parameters as the EMBA implementation. However, the HMBA process has the additional steps of determining the downsample block size. To avoid unnecessary error checking, the block size input to the HMBA is the size of the smallest output block. The block size for each down sample layer is calculated by multiplying the block size by 2 during each iteration.

2.3 RANSAC

In the implementation of RANSAC algorithm, we need five parameters: $\delta, \epsilon, n, p, N$

- δ is the decision threshold used for rejecting outliers and accepting the inliers. In assumption, the nosie induced displacement for the true locations of a pixel should be modeled by gaussian distribution, so $\delta = 3\sigma$ can capture 90% of all the inliers, where σ is a small number between 0.5 and 2. Thus, we set the delta to be 8 in the code.
- ϵ is the probability of a corresponding is an outlier. Assuming that 10%

of the correspondings are outliers, we set the parameter to be 0.1

- n is the smallest number of pairs needed for the homography calculation. In this experiment, we just need 4 pairs to estimate a homography, so it is set to 4.
- p is the probability that at least one trail is free of outliers, and is set to be 0.99
- N is the number of trails, determined by

$$N = \frac{ln(1-p)}{ln(1-(1-\epsilon)^n)}$$

2.4 Linear Estimation of H

Since only the ratio is important in homography representation, we assume all points are in the form of $x_i = (x, y, 1)^T$. The homography can be computed by Ah = b that

$$\begin{pmatrix} 0 & 0 & 0 & -x_i & -y_i & y_i'x_i & y_i'y_i \\ x_i & y_I & 1 & 0 & 0 & 0 & -x_i'x_i - x_i'y_i \end{pmatrix} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \end{pmatrix} = \begin{pmatrix} -y_i' \\ x_i' \end{pmatrix}$$

The solution is given by $min|b - Ah|$, when $h = (A^TA)^{-1}A^Tb$

2.5 None Linear Estimation of H using LM

The Levenburg-Marquardt method combines the merits of Gradient descent and Gauss-Newton method and gives us a stable and fast computation. The J in the formula is the jacobian of the matrix. We can calculate the error between the estimation using $\epsilon_p = X - f_p$. To refine the linear matrix using the LM method, we repeat the following step until the error becomes small enough.

- First we initialize the computation with the damping coefficient with $\mu = 0.5 \times max\{diag\{J_f^T J_f\}\}$
- Then we compute the δ_p by $(J_f^T J_f)^{-1} J_f^T \epsilon(p)$
- In order to check the sign of change, we need to calculate the ratio by

$$\rho = \frac{C(p_k) - C(p_{k+1})}{(J_f^T J_f)\delta_p}$$

- We update the damping coefficient by $\mu = \mu \times max\{1/3, 1 - (2\rho - 1)^3\}$

3 Results and conclusion

Through the experiments of EBMA, HBMA and using the motion field to estimate the affine transformation matrix, I found out that

- With the same searching range, a larger block size will produce a better PSNR.
- Among the search range of 3, 6, and 9, search range 3 produces the best result.
- When estimating the affine transformation matrix, images with only camera movements but not object movements performs better.

PSNR for EBMA of range 3 and block size 4 is 31.968079846022363
PSNR for EBMA of range 3 and block size 8 is 32.26742206191571
PSNR for EBMA of range 3 and block size 16 is 32.61873503651111
PSNR for EBMA of range 6 and block size 4 is 31.152301792781543
PSNR for EBMA of range 6 and block size 8 is 31.431700157384288
PSNR for EBMA of range 6 and block size 16 is 31.890646736294734
PSNR for EBMA of range 9 and block size 4 is 30.89070503854828
PSNR for EBMA of range 9 and block size 8 is 31.174512972828822
PSNR for EBMA of range 9 and block size 16 is 31.57277265407345

PSNR for half pel EBMA of range 3 and block size 4 is 31.827083586438416
PSNR for half pel EBMA of range 3 and block size 8 is 32.0739540996723
PSNR for half pel EBMA of range 3 and block size 16 is 32.5601042384792
PSNR for half pel EBMA of range 6 and block size 4 is 31.140349290348
PSNR for half pel EBMA of range 6 and block size 8 is 31.42526869262834
PSNR for half pel EBMA of range 6 and block size 16 is 31.86955779594782
PSNR for half pel EBMA of range 9 and block size 4 is 30.86257379301854
PSNR for half pel EBMA of range 9 and block size 8 is 31.153214158823825
PSNR for half pel EBMA of range 9 and block size 16 is 31.437298716427748

PSNR for HBMA of range 6, block size 16 and level 2 is 32.178568927486296
PSNR for HBMA of range 6, block size 16 and level 3 is 30.30389910736544
PSNR for HBMA of range 6, block size 16 and level 4 is 30.286449757171017

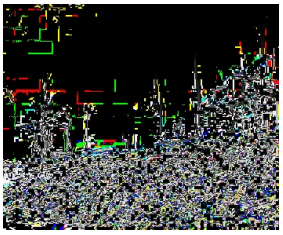
PSNR for transformation matrix estimation is 30.717297637313166 and the matrix between flower0000.jpg and flower0062.jpg (Camera motion pair) is

$$\begin{pmatrix} 9.95516982e-01 & 2.15299808e-02 & 1.04569064e+01 \\ -3.27073615e-03 & 1.02883525e+00 & 9.83147909e+00 \\ -2.18710230e-05 & 4.31816197e-05 & 1.00000000e+00 \end{pmatrix}$$

PSNR for transformation matrix estimation is 30.198071093824872 and the matrix between akiyo0000.jpg and akiyo0062.jpg (Object motion pair) is

$$\begin{pmatrix} 1.00818774e+00 & -1.77038824e-03 & 1.02309960e+01 \\ -5.07101986e-04 & 1.00376278e+00 & 1.04704523e+01 \\ 2.11238467e-05 & 2.48920867e-06 & 1.00000000e+00 \end{pmatrix}$$

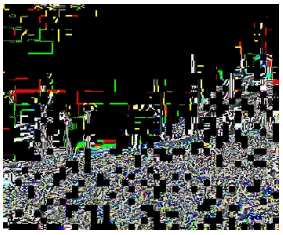




(a) Wrapped Image (b) Motion Field (c) Difference

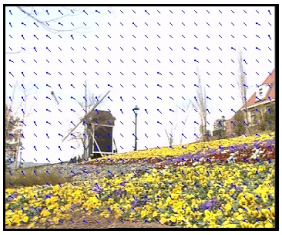
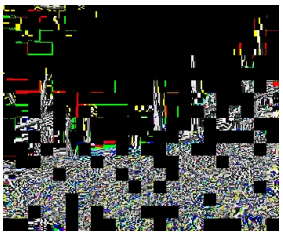
Figure 2: Results from EBMA with range = 3 and blocksize = 4





(a) Wrapped Image (b) Motion Field (c) Difference

Figure 3: Results from EBMA with range = 3 and blocksize = 8

(a) Wrapped Image (b) Motion Field (c) Difference

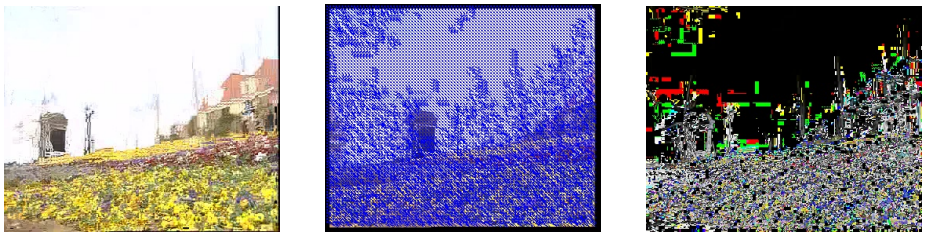
Figure 4: Results from EBMA with range = 3 and blocksize = 16

4 Code

4.1 EBMA

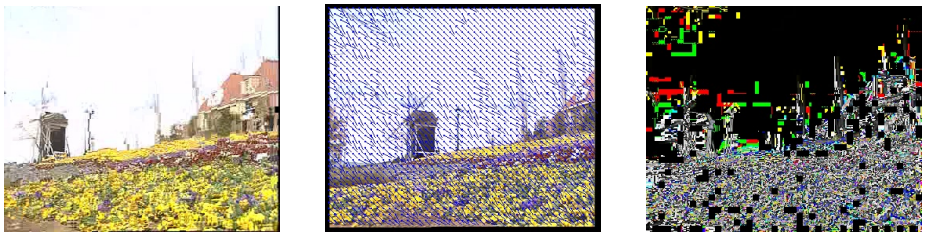
```
import numpy as np
import cv2
from math import log10, sqrt

class ebma:
    def __init__(self, img:np.ndarray, range:int, blockSize:int,
        half_pel:bool=False) -> None:
```



(a) Wrapped Image (b) Motion Field (c) Difference

Figure 5: Results from EBMA with range = 6 and blocksize = 4



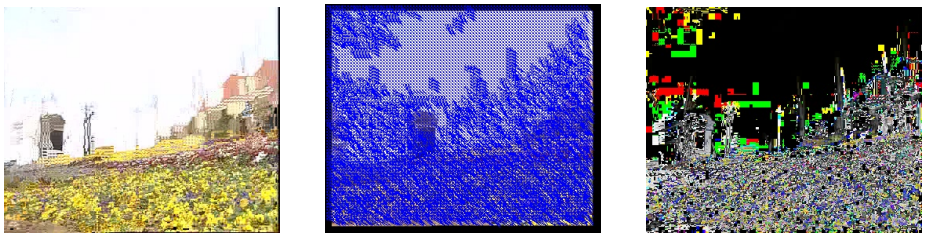
(a) Wrapped Image (b) Motion Field (c) Difference

Figure 6: Results from EBMA with range = 6 and blocksize = 8

(a) Wrapped Image (b) Motion Field (c) Difference

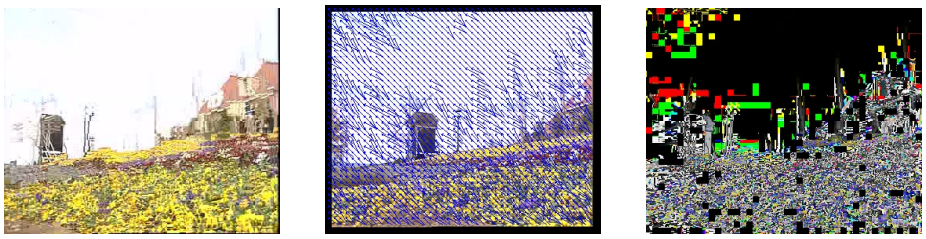
Figure 7: Results from EBMA with range = 6 and blocksize = 16

```
self.half = half_pel
self.range = max(range, 1)
self.height, self.width, _ = img.shape
self.blockS = blockSize
if half_pel:
    self.width *=2
    self.height *= 2
    img = cv2.resize(img, (self.width, self.height),
        interpolation = cv2.INTER_LINEAR)
    self.blockS *=2
    self.range *=2
self.anchor = img # this is acutally the frame
```



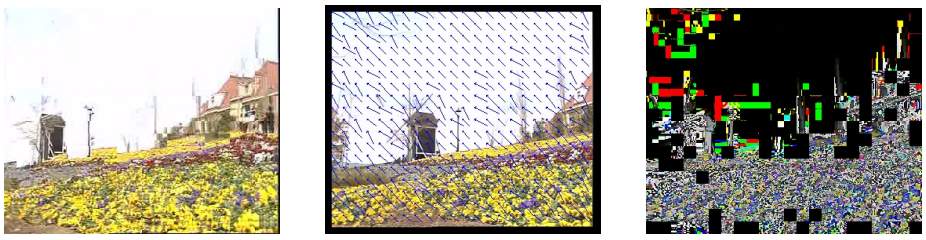
(a) Wrapped Image (b) Motion Field (c) Difference

Figure 8: Results from EBMA with range = 9 and blocksize = 4



(a) Wrapped Image (b) Motion Field (c) Difference

Figure 9: Results from EBMA with range = 9 and blocksize = 8



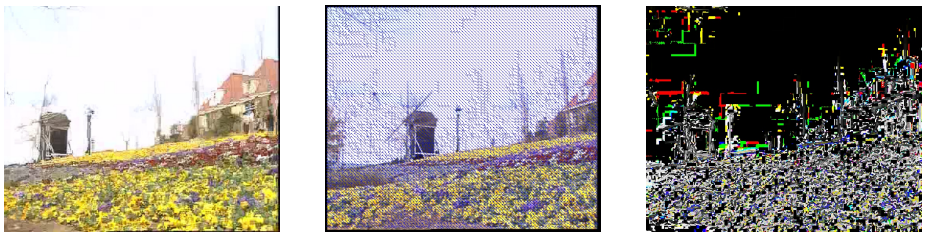
(a) Wrapped Image (b) Motion Field (c) Difference

Figure 10: Results from EBMA with range = 9 and blocksize = 16

```
        self.output = img.copy()

    def setAnchorFrame(self, img:np.ndarray) -> None:
        if self.half:
            img = cv2.resize(img, (self.width*2, self.height*2),
                interpolation = cv2.INTER_LINEAR)
        self.anchor = img
        return

    def PSNR(self, original:np.ndarray, compressed:np.ndarray) -> float:
        mse = np.mean((original - compressed) ** 2)
```

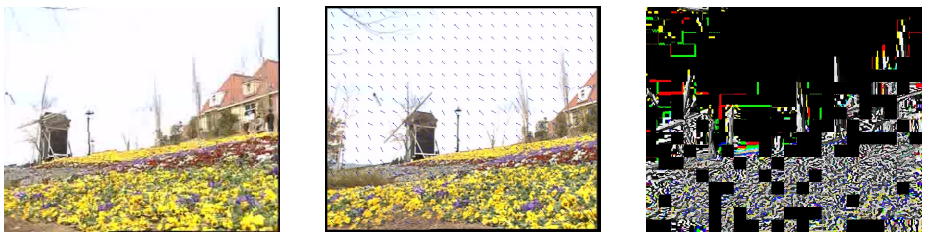


(a) Wrapped Image (b) Motion Field (c) Difference

Figure 11: Results from half pel accuracy EBMA with range = 3 and blocksize = 4

(a) Wrapped Image (b) Motion Field (c) Difference

Figure 12: Results from half pel accuracy EBMA with range = 3 and blocksize = 8

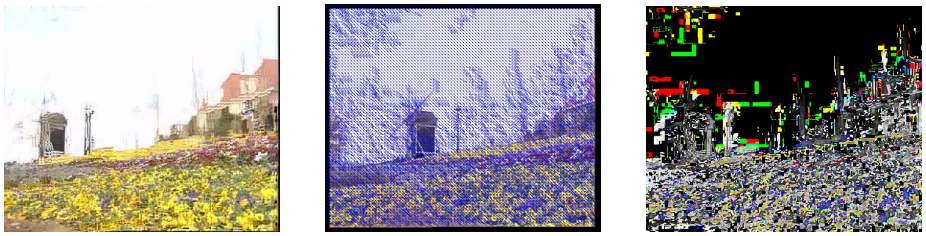


(a) Wrapped Image (b) Motion Field (c) Difference

Figure 13: Results from half pel accuracy EBMA with range = 3 and blocksize = 16

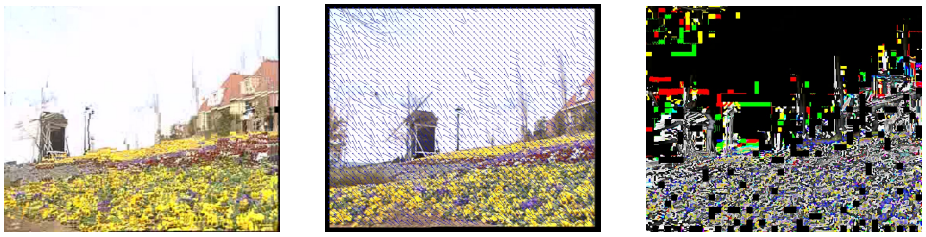
```
        if(mse == 0): # MSE is zero means no noise is present in the
            signal.
            return 100
        max_pixel = 255.0
        psnr = 20 * log10(max_pixel / sqrt(mse))
        return psnr

    def getMAD(self, tBlock:np.ndarray, aBlock:np.ndarray) -> float:
        return np.mean(np.abs(np.subtract(tBlock, aBlock)))
```



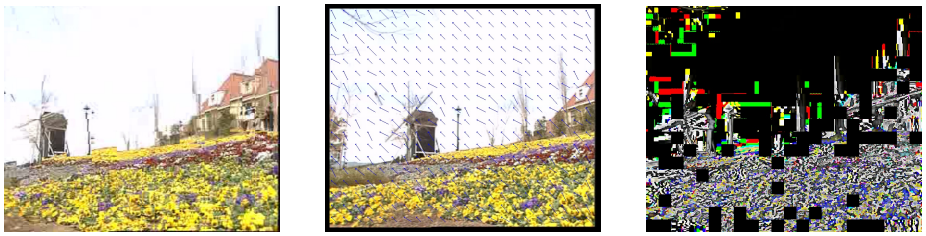
(a) Wrapped Image (b) Motion Field (c) Difference

Figure 14: Results from half pel accuracy EBMA with range = 6 and blocksize = 4



(a) Wrapped Image (b) Motion Field (c) Difference

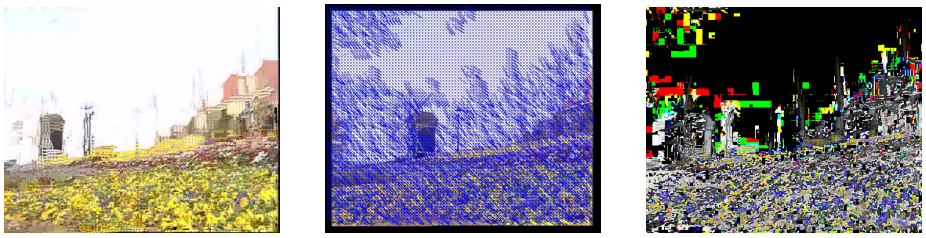
Figure 15: Results from half pel accuracy EBMA with range = 6 and blocksize = 8



(a) Wrapped Image (b) Motion Field (c) Difference

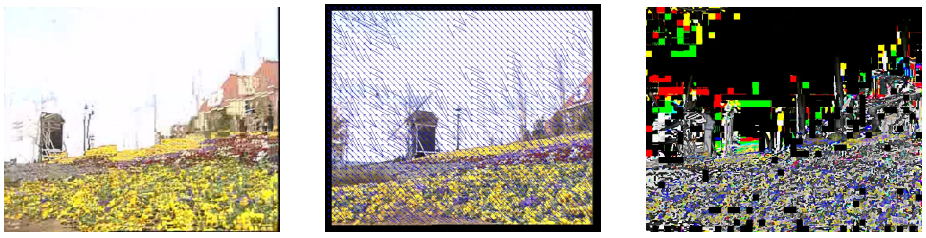
Figure 16: Results from half pel accuracy EBMA with range = 6 and blocksize = 16

```
def pad(self, img:np.ndarray) -> np.ndarray:
    newImg = np.zeros((self.height + 2*self.range, self.width +
        2*self.range, 3))
    newImg[self.range:self.range+self.height,
        self.range:self.range+self.width, :] = img
    return newImg
```



(a) Wrapped Image (b) Motion Field (c) Difference

Figure 17: Results from half pel accuracy EBMA with range = 9 and blocksize = 4



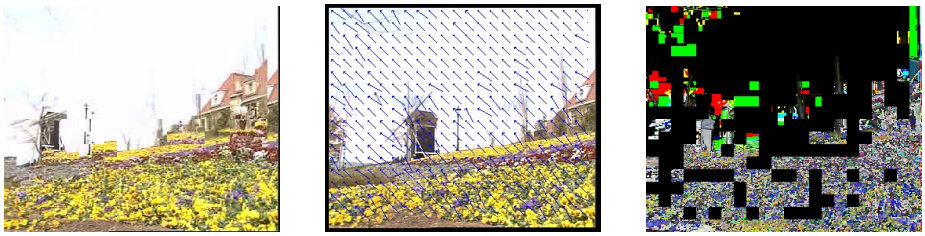
(a) Wrapped Image (b) Motion Field (c) Difference

Figure 18: Results from half pel accuracy EBMA with range = 9 and blocksize = 8

(a) Wrapped Image (b) Motion Field (c) Difference

Figure 19: Results from half pel accuracy EBMA with range = 9 and blocksize = 16

```
def calculate_motion(self, img:np.ndarray) -> np.ndarray:
    img2 = self.pad(img.copy())
    hStep = int(np.ceil(self.height / self.blockS))
    wStep = int(np.ceil(self.width / self.blockS))
    output = np.zeros((hStep, wStep, 2), np.uint8)
    for k in range(hStep):
        for l in range(wStep):
```

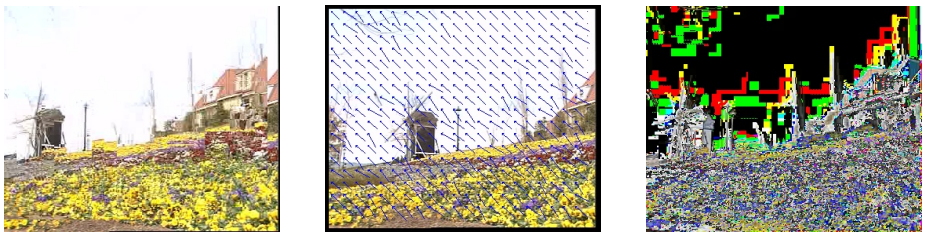


(a) Wrapped Image (b) Motion Field (c) Difference

Figure 20: Results from HBMA with range = 5, blocksize = 16 and level = 2

(a) Wrapped Image (b) Motion Field (c) Difference

Figure 21: Results from HBMA with range = 5, blocksize = 16 and level = 3



(a) Wrapped Image (b) Motion Field (c) Difference

Figure 22: Results from HBMA with range = 5, blocksize = 16 and level = 4

```
anchorBlock =
    self.anchor[k*self.blockS:(k+1)*self.blockS,
    l*self.blockS:(l+1)*self.blockS, :]
highScore = float("inf")
for i in range(self.range, 2*self.range):
    for j in range(self.range, 2*self.range):
        newBlock =
            img2[k*self.blockS+i:(k+1)*self.blockS+i,
            l*self.blockS+j:(l+1)*self.blockS+j, :]
        score = self.getMAD(newBlock, anchorBlock)
        if score < highScore:
            output[k, l, 0] = i
```

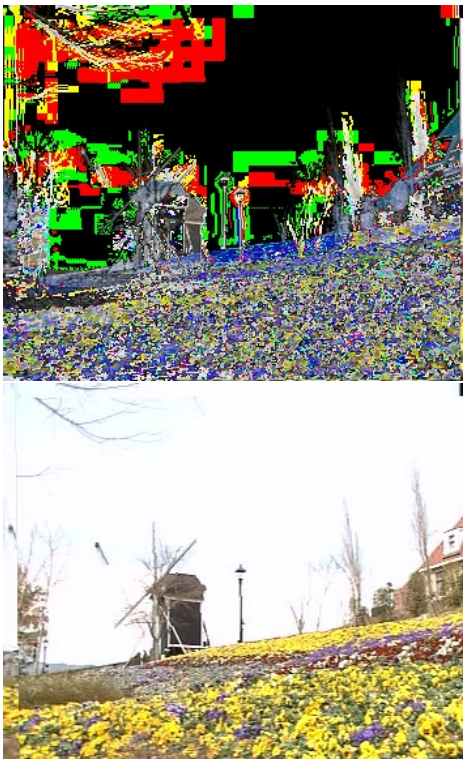


Figure 23: The difference and output using the transformation matrix estimated from motion field on flower image.

```
                        output[k, l, 1] = j
                        highScore = score
                        self.output[k*self.blockS:(k+1)*self.blockS,
                            l*self.blockS:(l+1)*self.blockS, :] =
                            newBlock
        return output

    def warp(self, motion:np.ndarray) -> np.ndarray:
        output = self.pad(self.anchor.copy())

        hStep = int(np.ceil(self.height / self.blockS))
        wStep = int(np.ceil(self.width / self.blockS))
        for k in range(hStep):
            for l in range(wStep):
                anchorBlock =
                    self.anchor[k*self.blockS:(k+1)*self.blockS,
                    l*self.blockS:(l+1)*self.blockS, :]

                dX = motion[k, l, 0]
```

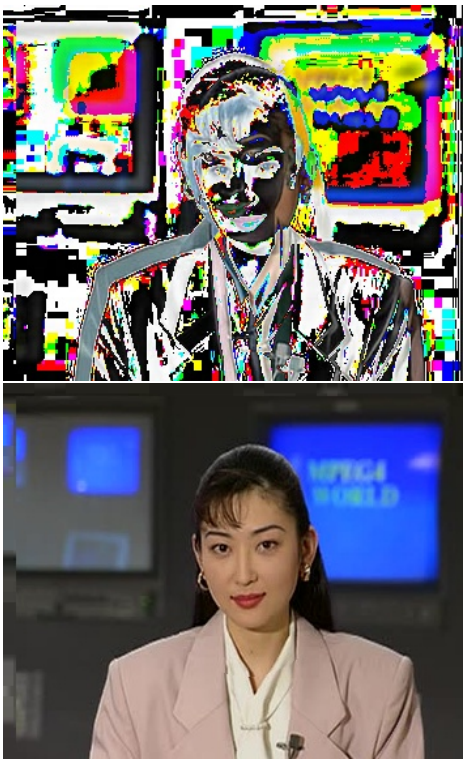


Figure 24: The difference and output using the transformation matrix estimated from motion field on akiyo image.

```
                dY = motion[k, l, 1]
                output[k*self.blockS+dX:(k+1)*self.blockS+dX,
                    l*self.blockS+dY:(l+1)*self.blockS+dY, :] =
                    anchorBlock

        return output[self.range:-self.range,self.range:-self.range,:]

    def diff(self, wrapped:np.ndarray, frame:np.ndarray) -> np.ndarray:
        return np.absolute(wrapped - frame)

    def plot_motion(self, motion:np.ndarray) -> np.ndarray:
        h, w, _ = motion.shape
        output = self.pad(self.anchor.copy())
        pts1 = []
        pts2 = []
        for i in range(h):
            for j in range(w):
                dX = int(motion[i, j, 0])
                dY = int(motion[i, j, 1])
```

```
                startP = [j*self.blockS+self.blockS//2, i*self.blockS +
                    self.blockS//2]
                endP = [j*self.blockS+self.blockS//2 + dY, i*self.blockS
                    + self.blockS//2 + dX]
                pts1.append(endP)
                pts2.append(startP)
                output = cv2.arrowedLine(output, endP, startP, (255, 0,
                    0), 1)
        self.pts1 = pts1
        self.pts2 = pts2
        return output

    def search(self, img:np.ndarray) -> None:
        if self.half:
            img = cv2.resize(img, (self.width, self.height),
                interpolation = cv2.INTER_LINEAR)
        motion = self.calculate_motion(img)
        self.motion = motion
        # output = self.warp(motion)
        output = self.output
        diff = self.diff(output, img)
        plotMotion = self.plot_motion(motion)
        print(self.PSNR(img, output))
        # cv2.imshow('Anchor frame', self.anchor.astype(np.uint8))
        # cv2.imshow('Image', img.astype(np.uint8))
        # cv2.imshow('Warpped output', output.astype(np.uint8))
        # cv2.imshow('Diff', diff.astype(np.uint8))
        # cv2.imshow('Motion', plotMotion.astype(np.uint8))
        # cv2.waitKey(0)
        return

    def getMatchP(self):
        return self.pts1, self.pts2


# img1 = cv2.imread("flower0000.jpg")
# img2 = cv2.imread("flower0062.jpg")
# matcher = ebma(img1, range=10, blockSize=16, half_pel=True)
# matcher.search(img2)
```

4.2 HBMA

```
    import numpy as np
import cv2
from math import log10, sqrt

class hbma:
```

```
def __init__(self, img:np.ndarray, range:int, blockSize:int,
    level:int) -> None:
    self.range = max(range, 1)
    self.setSize = False
    self.lev = level
    self.blockS = blockSize * 2**(self.lev-1)
    self.height, self.width, _ = img.shape
    self.anchor = img
    self.output = img.copy()

def setAnchorFrame(self, img:np.ndarray) -> None:
    if self.half:
        img = cv2.resize(img, (self.width*2, self.height*2),
            interpolation = cv2.INTER_LINEAR)
    self.anchor = img
    return

def PSNR(self, original, compressed):
    mse = np.mean((original - compressed) ** 2)
    if(mse == 0): # MSE is zero means no noise is present in the
        signal .
              # Therefore PSNR have no importance.
        return 100
    max_pixel = 255.0
    psnr = 20 * log10(max_pixel / sqrt(mse))
    return psnr

def getMAD(self, tBlock, aBlock):
    return np.mean(np.abs(np.subtract(tBlock, aBlock)))

def pad(self, img:np.ndarray) -> np.ndarray:
    newImg = np.zeros((self.height + 2*self.range, self.width +
        2*self.range, 3))
    newImg[self.range:self.range+self.height,
        self.range:self.range+self.width, :] = img
    return newImg

def calculate_motion(self, img:np.ndarray) -> np.ndarray:
    img2 = self.pad(img.copy())
    # i, j is the upper left corner of new frame
    # k, l is the upper left corner of anchor frame
    output = None
    for lev in range(self.lev):
        hStep = int(np.ceil(self.height / self.blockS))
        wStep = int(np.ceil(self.width / self.blockS))
        if output is None:
            output = np.zeros((hStep, wStep, 2), np.uint8)
        else:
            output = cv2.resize(output, (wStep, hStep),
                cv2.INTER_NEAREST)
```

```
            output = output * 2
            for k in range(hStep):
                for l in range(wStep):
                    anchorBlock =
                        self.anchor[k*self.blockS:(k+1)*self.blockS,
                        l*self.blockS:(l+1)*self.blockS, :]
                    highScore = float("inf")
                    for i in range(self.range, min(2*self.range,
                        self.height-output[k, l,
                        0]-(k+1)*self.blockS)):
                        for j in range(self.range, min(2*self.range,
                            self.width-output[k, l,
                            1]-(l+1)*self.blockS)):
                            hL = k*self.blockS+output[k, l, 0]+i
                            hH = (k+1)*self.blockS+output[k, l, 0]+i
                            wL = l*self.blockS+output[k, l, 1]+j
                            wH = (l+1)*self.blockS+output[k, l, 1]+j
                            if hL < 0 or wL < 0 or hH >
                                self.height+2*self.range or wH >
                                self.width+2*self.range:
                                continue
                            newBlock = img2[k*self.blockS+output[k, l,
                                0]+i:(k+1)*self.blockS+output[k, l,
                                0]+i, l*self.blockS+output[k, l,
                                1]+j:(l+1)*self.blockS+output[k, l,
                                1]+j, :]
                            score = self.getMAD(newBlock, anchorBlock)
                            if score < highScore:
                                output[k, l, 0] = i
                                output[k, l, 1] = j
                                highScore = score
                                self.output[k*self.blockS:(k+1)*self.blockS,
                                    l*self.blockS:(l+1)*self.blockS, :]
                                    = newBlock
        self.blockS = self.blockS//2
    return output

def warp(self, motion:np.ndarray) -> np.ndarray:
    output = self.pad(self.anchor.copy())

    hStep = int(np.ceil(self.height / self.blockS))
    wStep = int(np.ceil(self.width / self.blockS))
    for k in range(hStep):
        for l in range(wStep):
            anchorBlock =
                self.anchor[k*self.blockS:(k+1)*self.blockS,
                l*self.blockS:(l+1)*self.blockS, :]

            dX = motion[k, l, 0]
            dY = motion[k, l, 1]
```

```
            output[k*self.blockS+dX:(k+1)*self.blockS+dX,
                l*self.blockS+dY:(l+1)*self.blockS+dY, :] =
                anchorBlock

    return output[self.range:-self.range,self.range:-self.range,:]

def diff(self, anchor:np.ndarray, frame:np.ndarray) -> np.ndarray:
    return np.absolute(anchor - frame)

def plot_motion(self, motion:np.ndarray) -> np.ndarray:
    h, w, _ = motion.shape
    self.blockS*=2
    output = self.pad(self.anchor.copy())
    pts1 = []
    pts2 = []
    for i in range(h):
        for j in range(w):
            dX = int(motion[i, j, 0]) * 2
            dY = int(motion[i, j, 1]) * 2
            startP = [j*self.blockS+self.blockS//2, i*self.blockS +
                self.blockS//2]
            endP = [j*self.blockS+self.blockS//2 + dY, i*self.blockS
                + self.blockS//2 + dX]
            pts1.append(endP)
            pts2.append(startP)
            output = cv2.arrowedLine(output, endP, startP, (255, 0,
                0), 1)

    self.pts1 = pts1
    self.pts2 = pts2
    return output

def search(self, img:np.ndarray) -> None:
    motion = self.calculate_motion(img)
    self.motion = motion
    # output = self.warp(motion)
    output = self.output
    diff = self.diff(output, img)
    plotMotion = self.plot_motion(motion)
    print(self.PSNR(img, output))
    # cv2.imshow('Image', img.astype(np.uint8))
    # cv2.imshow('Anchor frame', self.anchor.astype(np.uint8))
    # cv2.imshow('Warpped output', output.astype(np.uint8))
    # cv2.imshow('Diff', diff.astype(np.uint8))
    # cv2.imshow('Motion', plotMotion.astype(np.uint8))
    # cv2.waitKey(0)
    return

def getMatchP(self):
    return self.pts1, self.pts2
```

```
# img1 = cv2.imread("flower0000.jpg")
# img2 = cv2.imread("flower0062.jpg")
# matcher = hbma(img1, range=3, blockSize=16, level=3)
# matcher.search(img2)
```

4.3 Extra Credit

```
    import numpy as np
import cv2
from scipy.optimize import least_squares
from ebma import ebma

def random_subset(set1, set2, num):
    idx_list = np.random.permutation(len(set1))
    out1 = []
    out2 = []
    for i in range(num):
        out1.append(set1[idx_list[i]])
        out2.append(set2[idx_list[i]])
    return out1, out2

def computeH(img1, img2):
    # Compute H according to lecture note 5
    # img1 maps to img2
    x = np.zeros((len(img1) * 2, 8))
    y = np.zeros((len(img1) * 2, 1))
    for i in range(len(img1)):
        x[2 * i] = [img1[i][0], img1[i][1], 1, 0, 0, 0, -img1[i][0] *
            img2[i][0], -img1[i][1] * img2[i][0]]
        x[2 * i + 1] = [0, 0, 0, img1[i][0], img1[i][1], 1, -img1[i][0]
            * img2[i][1], -img1[i][1] * img2[i][1]]
        y[2 * i] = img2[i][0]
        y[2 * i + 1] = img2[i][1]

    h = np.append(np.dot(np.linalg.pinv(x), y), 1)
    return h.reshape((3, 3))

def transform(pt2, H):
    pt = pt2[0:2]
    pt.append(1)
    new_pt = np.dot(H, pt)
    return new_pt / new_pt[2]

def find_inlier(pts1, pts2, H, delta):
    out1 = []
    out2 = []
```

```
    for i in range(len(pts1)):
        if np.linalg.norm(pts2[i] - transform(pts1[i], H)[0:2]) < delta:
            out1.append(pts1[i])
            out2.append(pts2[i])
    return out1, out2

# def find_inlier(pts1, pts2, H, delta):
#     out1 = []
#     out2 = []
#     for i in range(len(pts1)):
#         if np.linalg.norm(pts2[i] - transform(pts1[i], H)[0:2]) >=
    delta:
#             out1.append(pts1[i])
#             out2.append(pts2[i])
#     return out1, out2

def ransac(pts1, pts2, epsilon, n, p, delta):
    N = int(np.log(1-p) / np.log(1 - (1 - epsilon) ** n))
    save1 = []
    save2 = []
    for i in range(100):
        sub1, sub2 = random_subset(pts1, pts2, 4)
        H = computeH(sub1, sub2)
        in1, in2 = find_inlier(pts1, pts2, H, delta)
        if len(in1) > len(save1):
            save1 = in1
            save2 = in2
    return save1, save2

def linearH(pts1, pts2):
    lens = len(pts1)
    if lens % 2 != 0:
        pts1.append ([0, 0])
        pts2.append ([0, 0])

    A = np.zeros((2 * lens, 8))
    b = np.zeros((2 * lens, 1))

    for i in range (lens):
        pt1 = np.array([pts1[i][0], pts1[i][1], 1])
        pt2 = np.array([pts2[i][0], pts2[i][1], 1])
        A[2 * i] = [0, 0, 0, -pt1[0], - pt1[1], -1, pt1[0], pt2[1] *
            pt1[1]]
        b[2 * i] = -pt2[1] * pt1[2]
        A[2 * i + 1] = [pt1[0], pt1[1], 1, 0, 0, 0, - pt2[0] * pt1[0],
            -pt2[0] * pt1[1]]
        b[2 * i + 1] = pt2[0] * pt1[2]

    h = np.dot(np.linalg.pinv(A), b)
    h = np.append(h, 1)
```

```
    return h.reshape((3, 3))

def canvas_size(img , H):
    (h, w, _) = img.shape
    b = np.zeros ((4, 3))
    b[0] = np.dot(H, np.array([0, 0, 1])) / np.dot(H, np.array([0, 0,
        1]))[2]
    b[1] = np.dot(H, np.array([h, 0, 1])) / np.dot(H, np.array([h, 0,
        1]))[2]
    b[2] = np.dot(H, np.array([0, w, 1])) / np.dot(H, np.array([0, w,
        1]))[2]
    b[3] = np.dot(H, np.array([h, w, 1])) / np.dot(H, np.array([w, h,
        1]))[2]

    return b

def empty_canvas(img0, img1, img3, img4, H02, H12, H32, H42):
    b1 = canvas_size(img0, H02)
    b2 = canvas_size(img1, H12)
    b3 = canvas_size(img3, H32)
    b4 = canvas_size(img4, H42)

    x_min, y_min, _ = np.amin(np.amin([b1, b2, b3, b4], 0), 0)
    x_max, y_max, _ = np.amax(np.amax([b1, b2, b3, b4], 0), 0)
    w = int(round(x_max)) - int(round(x_min))
    h = int(round(y_max)) - int(round(y_min))
    canvas = np.zeros((h, w, 3))

    return canvas, x_min, y_min

def transform2(img, img2, h, x_min, y_min):
    # img2 to img1 map
    # h is img2 to img1 map
    h = np.linalg.pinv(h) / np.linalg.pinv(h)[2][2]
    img1 = img.copy()
    for i in range(img1.shape[0]):
        for j in range(img1.shape[1]):
            newCor = np.dot(h, np.array([j - y_min, i - x_min, 1]))
            l = round(newCor[0]/newCor[2])
            k = round(newCor[1]/newCor[2])
            if k < img2.shape[0] and k >= 0 and l < img2.shape[1] and l
                >= 0:
                img1[i][j] = img2[k][l]
    return img1

def show_liers(img1, img2, inlier_pt1, inlier_pt2, name):
    img = np.hstack((img1, img2))
    h, w, d = img1.shape
```

```
    for i in range(len(inlier_pt1)):
        pt1 = [int(inlier_pt1[i][0]), int(inlier_pt1[i][1])]
        pt2 = [int(inlier_pt2[i][0] + w), int(inlier_pt2[i][1])]
        cv2.line(img, tuple(pt1), tuple(pt2), (0, 255, 255), 1)
        cv2.circle(img, tuple(pt1), 4, (0, 255, 255), 2)
        cv2.circle(img, tuple(pt2), 4, (0, 255, 255), 2)
    cv2.imwrite(name,img)
    return

def find_M(matcher, knr1, knr2, num):
  # Initialize lists
  list_kp1 = []
  list_kp2 = []
  cnt = 0

  # For each match...
  for mat in matcher:
      if cnt >= num:
          break
      # Get the matching keypoints for each of the images
      img1_idx = mat.queryIdx
      img2_idx = mat.trainIdx

      # x - columns
      # y - rows
      # Get the coordinates
      [x1, y1] = knr1[img1_idx].pt
      [x2, y2] = knr2[img2_idx].pt

      # Append to each list
      list_kp1.append([x1, y1])
      list_kp2.append([x2, y2])
      cnt = cnt + 1

  return list_kp1, list_kp2

def nonLinearH(pts1, pts2, H):

    def func(h):
        loss = []

        for i in range(len(pts1)):
            x = pts1[i][0] * h[0] + pts1[i][1] * h[1] + h[2] /
                (pts1[i][0] * h[6] + pts1[i][7] * h[4] + 1)
            loss.append(pts2[i][0] - x)
            y = pts1[i][0] * h[3] + pts1[i][1] * h[1] + h[5] /
                (pts1[i][0] * h[6] + pts1[i][7] * h[4] + 1)
            loss.append(pts2[i][1] - y)

        return np.asarray(loss)
```

```
    sol = least_squares(func, H.squeeze(), method='lm')
    output = sol.x
    output.append(1)
    return output.reshape((3, 3))

# read image
img0 = cv2.imread("flower0000.jpg")
img1 = cv2.imread("flower0062.jpg")
matcher = ebma(img0, range=10, blockSize=16, half_pel=False)
motion = matcher.search(img1)
pts1, pts2 = matcher.getMatchP()
print("**********Read Image Finished**********")


# RANSAC paramters
epsilon = 0.1
delta = 8
p = 0.99
n = 4

in0, in01 = ransac(pts1, pts2, n, epsilon, p, delta)
print("**********RANSAC Finished**********")

H_01 = linearH(in0, in01)
print("**********H Compute Finished**********")

canvas = transform2(img1, img0, H_01, 0, 0)
print("**********Linear Output Finished**********")

print(matcher.PSNR(img0, canvas))
diff = matcher.diff(canvas, img0)
cv2.imshow('Diff', diff.astype(np.uint8))
cv2.waitKey(0)

cv2.imwrite('output.jpg', canvas)

# H_01 = nonLinearH(in0, in01, H_01)
print("**********NonLinear Finished**********")

# canvas = transform2(img1, img0, H_01, 0, 0)

# cv2.imwrite('output.jpg', canvas)
print("**********Output Finished**********")
```