

2020 年度 Computer Vision 最終レポート

慶應義塾大学 理工学研究科 専攻 修士 1 年 佐久間拓哉 (82019167)

2020 年 7 月 20 日

1 序論・問題文

本レポートは「2D match move/augmented reality」を選択した。

近年、スポーツに使用されるスタジアムのバックネット広告に、バーチャル広告が採用されているケースがある。例えば 2018 年のボルシア・ドルトムントではビッヂわきの広告版にこの技術を採用し、ヨーロッパやアメリカ、アジアそれぞれの試合中継で異なる広告を表示することで広告板を増やすことなく広告スペースを拡大することに成功している [?].

本レポートではこのバーチャル広告のベースとなる技術を実装する。具体的には画像のワーピング、特微量抽出及びマッチング、ホモグラフィ行列の算出を組み合わせて、対象画像の一部を差し替える。

問題の全体像を以下の図に示す。まず、図 1(a) の部分画像 ABCD(テンプレート画像) とマッチする部分画像 EFGH(適用画像) を図中 (c) から探索する。その後図中 (b) の差し替え先画像から差し替え画像へのホモグラフィ行列を計算し、ワーピング処理によって差し替えを行う。

以降の章の構成は次の通りである。2 章では本アプリケーションで使用するアルゴリズムおよび実装を示す。3 章では実装したアプリケーションの検出精度や実行時間などを評価する。最後に 4 で本レポートの結論を述べる。

2 実装

本章では、各操作の詳細アルゴリズム及び実装を述べる。本レポートでは図 1 における頂点 ABCD の選定は手動で行うが、頂点 EFGH は明示的に選択せず、特微量マッチングによって暗黙的に行う。この場合、差し替え先画像から適用画像へのホモグラフィ行列は、各ピクセルの対応が分からぬいため、直接求めることはできない。そこで、テンプレート画像と差し替え先画像の画像サイズを一致させてから、テンプレート画像と適用画像との特微量マッチングを行うことで、ここから求められるテンプレート画像から適用画像へのホモグラフィ行列が差し替え先画像から適用画

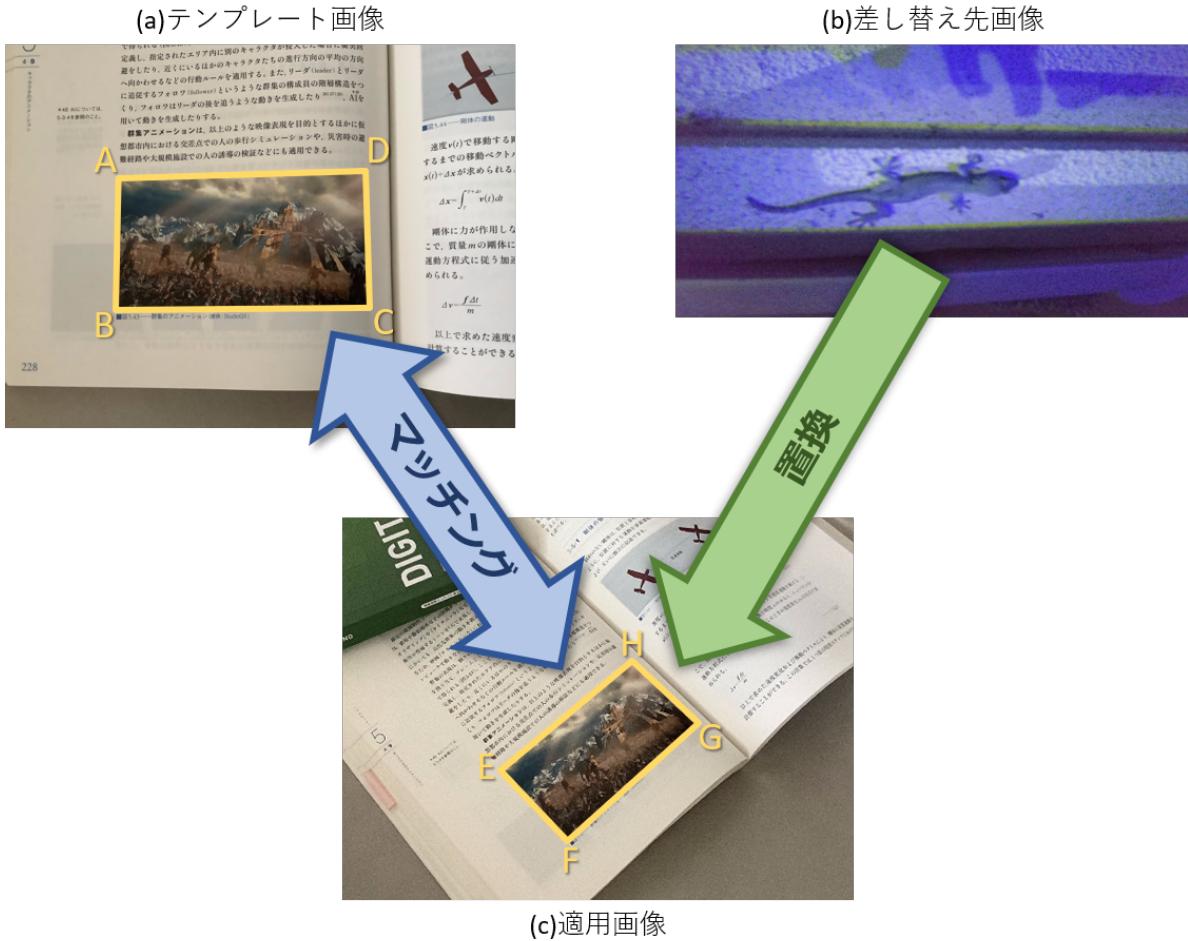


図 1 問題の全体像

像へのホモグラフィ行列に一致する。このようにして各ピクセルの変換方法が求められるため画像を置換できる。

2.1 実装環境

本レポートは次の実行環境で動作させている。

- Python 3.7
- kivy 1.11.1
- OpenCV-python 3.4.2.16
- Numpy 1.19.0

ここで kivy は GUI のアプリケーションを作成するライブラリであり、作成したアプリケーションのコントロールや UI 処理を担っている。OpenCV は一部の画像処理アルゴリズムを使用するために導入している。Numpy は数値計算を目的として導入している。

詳細の実装については GitHub ページに公開している。また、アプリケーションのデモンストレーションは添付した「demo.mp4」を参照されたい。

2.2 テンプレート画像と差し替え先画像の画像サイズ

2.2.1 ワーピング処理

本問題では図 1(a) の頂点 ABCD には任意性があるため、単純なリサイズ処理ではテンプレート画像と差し替え先画像のサイズを一致させることができない。4 点の対応点が与えられるばあにはホモグラフィ行列を用いて変換することが一般的であるが、本レポートではワーピング処理によって頂点 ABCD を矩形に変換する。変換によって画像にピクセルの抜けが生じないようにするために Inverse warping を実装する。具体的には以下の図 2 に示す、四角形 MNOP 内の任意の点 $p_1 = (x_1, y_1)$ から四角形 IJKL の対応する点 p_0 を求める。

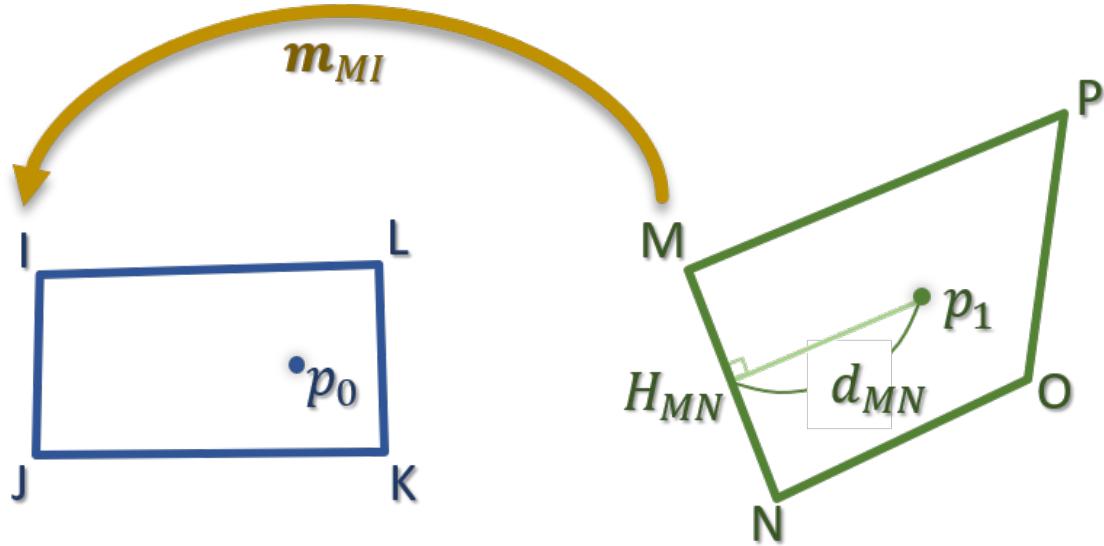


図 2 ワープ処理

まず、頂点 M から I への移動量を $\mathbf{m}_{MI} = (dx_{MI}, dy_{MI})$ とする。 \mathbf{p}_1 から辺 MN への距離及び垂線の足をそれぞれ d_{MN} 及び H_{MN} とする。また、 $\overrightarrow{MH_{MN}} = t_{MN}\overrightarrow{MN}$ とすると、Line-based image warping[] に従って \mathbf{x}_0 は以下の式で求める。

$$\mathbf{p}_1 = \begin{bmatrix} x_1 + \left(1 - \frac{d_{MN}}{d_{MN} + d_{OP}}\right) ((1 - t_{MN}) dx_{MI} + t_{MN} dx_{NJ}) + \frac{d_{MN}}{d_{MN} + d_{OP}} ((1 - t_{OP}) dx_{PL} + t_{OP} dx_{OK}) \\ y_1 + \left(1 - \frac{d_{PM}}{d_{NO} + d_{PM}}\right) ((1 - t_{PM}) dy_{MI} + t_{PM} dy_{PL}) + \frac{d_{PM}}{d_{NO} + d_{PM}} ((1 - t_{NO}) dy_{NJ} + t_{NO} dy_{OK}) \end{bmatrix} \quad (1)$$

python における実装は以下のソースコード 1 に示す。

ソースコード 1 ワーピング

```
1 # 4つの点を指定した影響度でブレンドする関数
2 def bilinear(
3     value,
4     left_bottom,
5     right_bottom,
6     left_top,
7     right_top,
8     main_ratio,
9     cross_ratio):
10    left_ratio = 1-main_ratio
11    right_ratio = main_ratio
12    return value \
13        + (1-cross_ratio) * (left_bottom * left_ratio + right_bottom * right_ratio) \
14        + cross_ratio * (left_top * left_ratio + right_top * right_ratio)
15
16
17 def warp(
18     image,
19     from_bottom_left,
20     from_bottom_right,
21     from_top_right,
22     from_top_left,
23     to_bottom_left,
24     to_bottom_right,
25     to_top_right,
26     to_top_left,
27     return_height,
28     return_width):
29
30    # 任意の点から各辺への距離(d)とその垂線の足が辺のどのあたりに位置しているかの割合(t)を
31    # 戻す関数を作成
32    def td(f, t):
33        a = np.array([f, t])
34        # fとtの差分 = fを基準としたtへのベクトル
35        dif = np.diff(a, axis=0)[0]
36        det = np.linalg.det(a)
37        sq = dif @ dif
38        f = f[:,None,None]
39        def ret(x):
40            # 直線を表す式 : dif[1] * x[0] - dif[0] * x[1] - det=0
41            up = (dif[1] * x[0] - dif[0] * x[1] - det)
42            r = -up / sq
43            # fを基準とした垂線の足へのベクトル
44            h_dif = f - np.array([r * dif[1] + x[0], -r * dif[0] + x[1]])
45            # ベクトルの大きさの比と点と直線の距離を返戻
46            return np.sqrt(np.sum(h_dif**2, axis=0)) / sq, (np.abs(up) / np.sqrt(sq))
47        return ret
48    td_bottom = td(to_bottom_left, to_bottom_right)
49    td_top = td(to_top_left, to_top_right)
50    td_left = td(to_bottom_left, to_top_left)
51    td_right = td(to_bottom_right, to_top_right)
52
53    # 各頂点の移動量を計算
54    move_bottom_left = from_bottom_left - to_bottom_left
55    move_bottom_right = from_bottom_right - to_bottom_right
56    move_top_left = from_top_left - to_top_left
57    move_top_right = from_top_right - to_top_right
58
59    h, w, *_ = image.shape
60
61    # 関数の本体 : i, jはnp.meshgridで与えられる
62    def func(i, j):
63        nonlocal to_bottom_left, to_bottom_right, to_top_left, to_top_right
64        pos = np.array([i, j])
65
66        # ピクセルが変換先の領域内に含まれているか判定するために外積を求める
67        crs_bl = np.cross(
68            pos - to_bottom_left[:,None, None],
69            to_bottom_right - to_bottom_left,
70            axis=0)
```

```

70         crs_br = np.cross(
71             pos - to_bottom_right[:,None, None],
72             to_top_right - to_bottom_right,
73             axis=0)
74         crs_tr = np.cross(
75             pos - to_top_right[:,None, None],
76             to_top_left - to_top_right,
77             axis=0)
78         crs_tl = np.cross(
79             pos - to_top_left[:,None, None],
80             to_bottom_left - to_top_left,
81             axis=0)
82
83     # pos から 各頂点への外積が負なら内部
84     mask = np.where(
85         (crs_bl < 0) & (crs_br < 0) & (crs_tr < 0) & (crs_tl < 0),
86         1, 0)
87     t_bottom, d_bottom = td_bottom(pos)
88     t_top, d_top = td_top(pos)
89     t_left, d_left = td_left(pos)
90     t_right, d_right = td_right(pos)
91
92     t_vert = d_bottom / (d_bottom + d_top)
93     t_hori = d_left / (d_left + d_right)
94
95     v = (i \
96         +((1 - t_bottom) * move_bottom_left[0] + t_bottom * move_bottom_right[0]) * (1 \
97             - t_vert)\ \
98         + ((1 - t_top) * move_top_left[0] + t_top * move_top_right[0]) * t_vert)
99
100    u = (j \
101        + ((1 - t_left) * move_bottom_left[1] + t_left * move_top_left[1]) * (1 - \
102            t_hori)\ \
103        + ((1 - t_right) * move_bottom_right[1] + t_right * move_top_right[1]) * t_hori \
104        )
105
106    # バイリニア補完するために隣接ピクセルの影響度を計算
107    v_int = v.astype(np.int16)
108    u_int = u.astype(np.int16)
109    bottom = np.where((v_int < 0) | (v_int > h-2), h-2, v_int)
110    left = np.where((u_int < 0) | (u_int > w-2), w-2, u_int)
111    top = bottom+1
112    right = left+1
113    v_ratio = (v - v_int)[:, :, None]
114    u_ratio = (u - u_int)[:, :, None]
115
116    return np.where((
117        # 内部判定
118        (mask == 0) | \
119        # 辺の外側に出ていないか判定
120        (((t_bottom < 0) | (t_bottom > 1)) & \
121        ((t_top < 0) | (t_top > 1))) & \
122        ((t_left < 0) | (t_left > 1)) & \
123        ((t_right < 0) | (t_right > 1)) & \
124        ((t_vert < 0) | (t_vert > 1)) & \
125        ((t_hori < 0) | (t_hori > 1))) | \
126        # 存在するピクセルを参照しているか判定
127        ((v_int < 0) | (v_int > h-2) | (u_int < 0) | (u_int > w-2)) \
128        )[:, :, None], \
129        0, \
130        # バイリニア補完で色を決定
131        bilinear( \
132            0, \
133            image[bottom, left], \
134            image[top, left], \
135            image[bottom, right], \
136            image[top, right], \
137            v_ratio, u_ratio).astype(np.uint8))
138
139
140     return np.fromfunction(func, shape=(return_height, return_width))

```

ソースコード 1 は for 文を使用せず、Numpy の性質を活かしてテンソルをまとめて操作すること

で実装している。処理の本体は func 関数内の処理であり、 i , j は画素値を表す行列として与えられる。つまり 62 行目の pos は (2, 画像の高さ, 画像の幅) の形をした三階のテンソルである。

30 行目の td 関数は引数として与えられた二点 f , t でできる辺への距離と、垂線の足が f を基準としてどのくらいの位置にいるか割合で返す関数を作成する。なお、二点 $f = (x_f, y_f)$, $t = (x_t, y_t)$ を通る直線の式は次のように与えられる。

$$(y_t - y_f)x - (x_t - x_f)y + \det(\mathbf{f}t) = 0 \quad (2)$$

任意点 (x_0, y_0) が与えられたとき、点 $\mathbf{f}t$ を通る直線垂線の足 $\mathbf{h} = (x_h, y_h)$ は式 (2) における法線から、次の式を見たす。

$$(x_h - x_0, y_h - y_0) = r(y_t - y_f, -x_t + x_f)) \quad (3)$$

但し、 r は媒介変数であり、 $r = -\frac{(y_t - y_f)x - (x_t - x_f)y + \det(\mathbf{f}t)}{\|\mathbf{f} - \mathbf{t}\|_2^2}$ を満たす。また、任意点 (x_0, y_0) から点 $\mathbf{f}t$ を通る直線への距離 d は次の式で与えられる。

$$d = \frac{\|(y_t - y_f)x - (x_t - x_f)y + \det(\mathbf{f}t)\|_1}{\|\mathbf{f} - \mathbf{t}\|_2} \quad (4)$$

66 から 81 行目ではある点を基準とした任意点と各頂点へのベクトルの外積を計算している。この計算により変換先領域の内外判定をする。この処理をしなければ変換先の辺を基準とした線対称に無駄な変換が行われてしまう。

95 101 行目は式 (1) にしたがって画素の変換を適用する。変化した点は整数ではないため、2 から 14 行目で表されるバイリニア補完を行う。104 から 111 行目はこの変換を施すために、各画素にどのくらい近いかを割合で計算している。

2.2.2 画像サイズの合わせ方

一致させる画像サイズの決め方は次の二種類の方法が考えられる。

- 差し替え先画像の高さ・幅に合わせる
- テンプレート画像の高さ・幅に合わせる

前者については、差し替え先画像は必ず長方形の形をしているため、簡単に高さや幅を取得することができる。しかし、テンプレート画像のアスペクト比が変化してしまうため、特徴点のマッチングが上手く行えない可能性がある。例えば特徴点の記述方法として頻繁に使用される SIFT[?] は DoG によってスケールに頑健に頑健に作用するが、DoG でスケールする際には等方的な縮小しか行わないため、アスペクト比が変化するような異方的なスケールよりも、比を保った等方的なスケールに対しての方が頑健であると推察される。なお、この主張については 3.1 節にて実験を行う。

後者については、テンプレート画像のアスペクト比が変化しないため、前者よりも高い精度でマッチングできると考えられる。しかし、ユーザが入力した四角形は歪んでおり、元の矩形を復元することができないため、正しい高さ及び幅を取得できない。

それぞれの python での実装は以下の通りである。なお、テンプレート画像の頂点 ABCD は tmp_points、差し替え先画像は dest である。

ソースコード 2 画像サイズ

```
1 # 差し替え先画像の高さ・幅に合わせる
2 height, width, *_ = dest.shape
3
4 # テンプレート画像の高さ・幅に合わせる
5 height = np.sqrt(np.sum((tmp_points[1] - tmp_points[0])**2)).astype(np.int16)
6 width = np.sqrt(np.sum((tmp_points[2] - tmp_points[1])**2)).astype(np.int16)
```

ソースコード 2において、後者の実装はテンプレート画像の元の矩形を復元する代替手段として、互いに交わる二辺を高さ及び幅として採用し、辺の L2 ノルムが高さ及び幅となる。

2.3 マッチングアルゴリズム

マッチングアルゴリズムには様々な手法が提案されているが、代表的なものにテンプレートマッチング及び対応点マッチングがある。

このマッチングによってテンプレート画像と適用画像のピクセルの対応を見つけ、ホモグラフィ行列を計算することができる。本レポートではホモグラフィ行列を OpenCV を利用して算出したため、実装の詳細は省く。

2.3.1 テンプレートマッチング

テンプレートマッチングは画像の画素値そのものを特徴として扱うパターンマッチングの一種である [?]. この手法では特徴となる画素値をテンプレートとして用意しておき、SSD を始めとする類似度を使用してマッチングを行う [?].

本問題においても予めテンプレート画像は用意しておくが、サイズが大きいため、精度が低下してしまうことが予想される。さらに根本的な問題として、テンプレートマッチングでは各ピクセルの対応が取れるわけではなく、一致する領域が得られるに過ぎないため、本問題の目的には合致しない。この問題を解決する方法として、ハリスのコーナー検出 [?] を利用して特徴点を検出した後、その点を中心とした部分画像をテンプレートとしてマッチングを行う実装が考えられる。

しかし、全体としてテンプレートマッチングは回転やスケールに対して虚弱であるため、本アプリケーションには適していない。そこで、本レポートでは後述する対応点マッチングによって回転やスケールに頑健なマッチングを行う。

2.4 対応点マッチング

対応点マッチングは異なる画像間で検出された各特徴点の特徴量を比較することで画像間の対応付けを行う。例えば、画像 I_1 及び I_2 の特徴量を x 及び y としたとき、その類似度 dist は L2 ノルムで算出される。

$$\text{dist}(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (x_i - y_i)^2} \quad (5)$$

典型的には、画像 I_1 のある特徴点に対して画像 I_2 から検出された全特徴点との距離を算出する。その後、 n 番目に距離が小さいものを $\text{dist}_n(\mathbf{x}, \mathbf{y})$ とし、以下の式を満たす特徴点を信頼度の高い対応点として判定する。[?]

$$\text{dist}_1(\mathbf{x}, \mathbf{y}) < r\text{dist}_2(\mathbf{x}, \mathbf{y}) \quad (6)$$

OpenCV を利用した実装を以下に示す。なお、実装は OpenCV のチュートリアル [?] を参考にしている。

ソースコード 3 対応点マッチング

```

1 sift = cv2.xfeatures2d.SIFT_create()
2 i1_kp, i1_des = sift.detectAndCompute(i1, None)
3 i2_kp, i2_des = sift.detectAndCompute(i2, None)
4
5 index_params = dict(algorithm=flann_index_kdtree, trees=5)
6 search_params = dict(checks=50)
7 flann = cv2.FlannBasedMatcher(index_params, search_params)
8 matches = flann.knnMatch(i2_des, i1_des, k=2)
9 good = []
10 for m, n in matches:
11     if m.distance < 0.7*n.distance:
12         good.append([m])
13
14 if len(good) > 10:
15     src_pts = np.float32([ i1_kp[m[0].queryIdx].pt for m in good ]).reshape(-1,1,2)
16     dst_pts = np.float32([ i2_kp[m[0].trainIdx].pt for m in good ]).reshape(-1,1,2)

```

二行目の `detectAndCompute` を利用することで、SIFT のキーポイントと特徴量を得られる。7 行目にあるように、マッチングには Fast Library for Approximate Nearest Neighbors[?] を利用する。11 行のように本実装では $r = 0.7$ としており、条件を満たす点を信頼度の高い対応点として記録する。ホモグラフィ行列は対応点を 4 つ与えることで導出できるが、対応点事態に誤差が存在しているため、それ以上の点を入力し、最小二乗法によって尤もらしい行列を求める。本実装では 14 行目のように 10 点以上の対応点を使用する。

2.5 画像の置換

得られたホモグラフィ行列を各画素に適用することで変換を行う。本節ではテンプレート画像から適用画像へのホモグラフィ行列 H を使用して、適用画像を差し替え先画像で置換する実装を示す。以下のソースコード 4 に画像置換の Python 実装を示す。

ソースコード 4 画像の置換

```

1 def replace(
2     reference,
3     source,
4     mat):
5     ref_h, ref_w, *_ = reference.shape

```

```

6     src_h, src_w, *_ = source.shape
7
8     def func(i, j):
9         pos = np.einsum('ij,ujwh->iwh',
10                         mat, np.array([j, i, np.ones(shape=j.shape)]))
11         pos = (pos[0:2] // pos[2]).astype(np.int16)
12         u = np.clip(pos[0], 0, ref_w-1)
13         v = np.clip(pos[1], 0, ref_h-1)
14         return np.where((
15             (pos[0] >= 0) & (pos[0] < ref_w) & \
16             (pos[1] >= 0) & (pos[1] < ref_h))[:, :, None],
17             reference[v, u],
18             source)
19
20     return np.fromfunction(func, shape=(src_h, src_w))
21
22 replaced = replace(dest, frame, H).astype(np.uint8)

```

ソースコード 4 はソースコード 1 と同様に for 文を使用せず、Numpy の性質を活かしてテンソルをまとめて操作することで実装している。ここで dest(reference) は差し替え先画像、frame(source) は適用画像である。9 行目に示すように、各画素に対して行列積を施す。なお、einsumにおいて、 $i=j=3$ である。また、得られたホモグラフィ行列は $(x, y, 1)$ の形をしている画素に対して適用する形になっていることに注意されたい。12、13 行目のように変換先をクリップしているのは、変換先が必ずしも reference の画素内に収まっているとは限らず、17 行目においてアクセスする際にエラーが生じてしまうのを回避するためである。15、16 行目のように変換先が reference の外に出ている場合は元の source の画素値を使用することで、適用画像の一部分のみを変換する。

2.6 精度向上と高速化

本問題で使用するアプリケーションの特徴を活かして、さらなる改善を試みる。本アプリケーションは動画に対して部分画像の差し替えを行うことを目的としている。そこで、動画という時系列データの性質を活用する。時系列データはフレーム間での変化は大きくない。

そのため、あるフレームで画像の置換が完了したとき、置換した領域の付近のみを探索し、特徴点をマッチングすればよい。これによって無駄な領域の特徴点の検出及び、マッチングにかかる計算量を削減できる上に、誤ったマッチングを防げる。以下の実装の詳細を示す。

ソースコード 5 対応点マッチング

```

1 def replace2(
2     reference,
3     source,
4     mat,
5     offset_h=0,
6     offset_w=0):
7     ref_h, ref_w, *_ = reference.shape
8     src_h, src_w, *_ = source.shape
9
10    def func(i, j):
11        pos = np.einsum('ij,ujkl->ikl',
12                         mat, np.array([j, i, np.ones(shape=j.shape)]))
13        pos = (pos[0:2] // pos[2]).astype(np.int16)
14        u = np.clip(pos[0]+offset_w, 0, ref_w-1)
15        v = np.clip(pos[1]+offset_h, 0, ref_h-1)
16        return np.where((
17             (pos[0] >= 0) & (pos[0] < ref_w) & \
18             (pos[1] >= 0) & (pos[1] < ref_h))[:, :, None],

```

```

19         reference[v, u],
20         0)
21
22     return np.fromfunction(func, shape=(src_h, src_w))
23
24 while True:
25     _, frame = cap.read()
26
27     # 前フレームで設定した領域のみ特徴点を検出する
28     i1_kp, i1_des = sift.detectAndCompute(frame[minh:maxh, minw:maxw], None)
29
30     # -----
31     # 略
32     # -----
33
34     # 部分画像で検出した分だけキーポイントの位置がずれているため移動するその分だけ移動する
35     # 必要がある。
36     replaced = replace2(dest, frame, H, minh, minw)
37     mask = np.sum(replaced > 0, axis=2, dtype=bool)
38
39     # ピクセル値が0である画素を元のピクセル値で上書きする
40     frame = np.where(mask[:, :, None], replaced, frame).astype(np.uint8)
41
42     # ピクセル値が0でない画素のインデックスを取得する
43     mask_id = np.array(np.where(mask))
44
45     # 適用画像の領域を矩形として切り出し、次のフレームではこの領域のみ特徴点を検出する
46     minh = min(np.min(mask_id[0]) - radius, 0)
47     minw = min(np.min(mask_id[1]) - radius, 0)
48     maxh = min(np.max(mask_id[0]) + radius, frame.shape[0])
        maxw = min(np.max(mask_id[1]) + radius, frame.shape[1])

```

ソースコード 5 の replace2 関数は 4 の replace 関数とほとんど同じ実装であるが、14、15 行目のようにオフセット分だけ移動させる処理が含まれている。これはソースコード 5においては部分画像に対して特徴点を検出するため、ホモグラフィ行列での変換先に位置ずれが生じてしまうからである。さらに、適用画像の領域を抜き出すために、一遍に出力画像を生成せずに、変換のみを行った画像を作成しておき、後で元の画像と合成する。行目のように適用画像の領域の最小及び最大位置の周辺を取り出しておく。その後 行目に示されるように取り出した区間で画像を切り出し特徴点検出を行う。ここで、radius はどの程度周辺領域を取り出すかを示しており、この値が大きいほどより広い領域を探索する。処理結果の一例を以下に示す。図 3において radius は 0 とする。

図 3(a) のように replaced は差し替え先画像の変換先以外がマスクされた画像となっている。また、(b) のように mask は差し替え先画像が存在している領域を切り出す処理をしており、後の探索領域の指定、すなわち、minh、minw、maxh、maxw の設定に利用する。

3 評価及び考察

本章では作成したアプリケーションの検出精度や実行速度を測り、アプリケーションの実用性を評価する。本評価で使用するテンプレート画像及び差し替え先画像を図 4 に示す。なお、適用画像については特に言及しない限り、本レポートに添付した「target.mov」を使用する。なお、テンプレート画像については無駄な領域を省くために図 5 で示されるように、アプリケーション内でユーザが領域を選択しクリッピングしておく。図 5(a) のアプリケーション画面において、水色のユーザが指定したものであり、各頂点をクリックすることで選択できるようになっている。(b) は

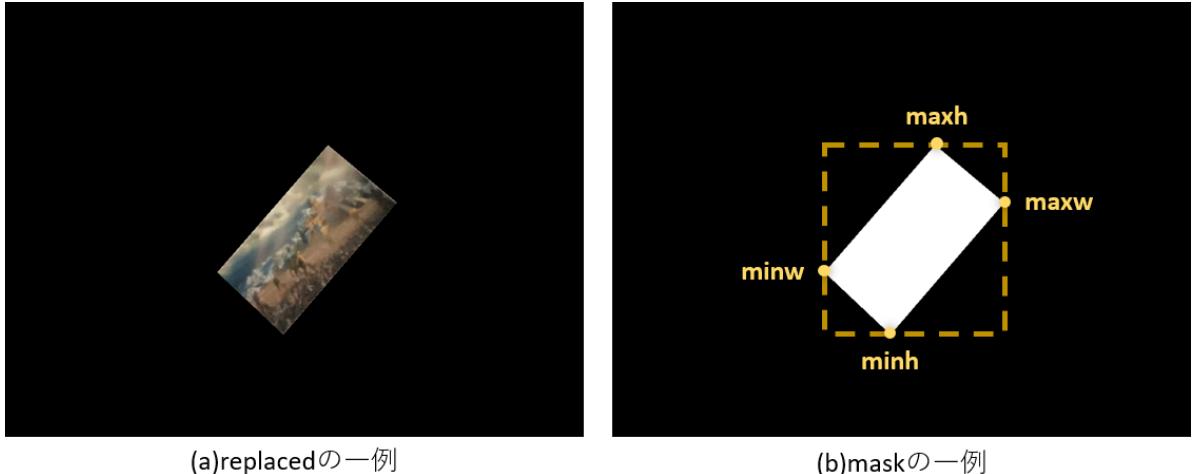


図3 探索領域の限定

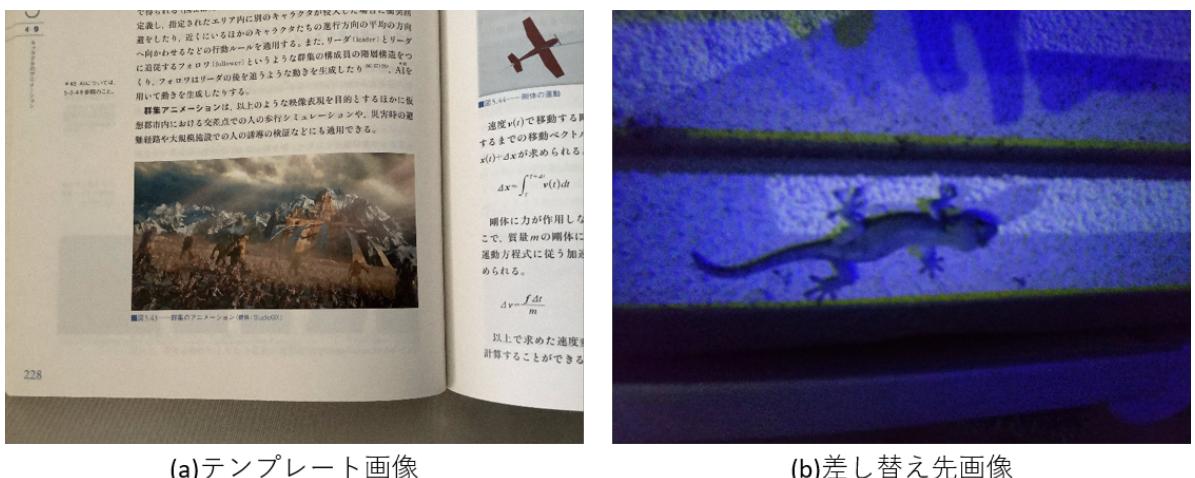


図4 評価データ

クリッピングした結果である。なお変形先については2.2.2節で述べたように二種類の方法で変形する。

3.1 テンプレート画像のサイズと精度

本節では2.2.2節で述べた二種類のサイズとマッチングの精度を比較する。適用先画像には「target.mov」における1フレーム目を使用する。また、マッチングアルゴリズムにはSIFTを使用する。以下にそれぞれにマッチング処理を適用した結果を図7及び6に示す。

図??及び6から分かるように、アスペクト比を変化させないほうが明らかに検出精度が高いことが分かる。この図から、マッチング数、ミスマッチ数を数えた結果を表1に示す。なお、ミスマッチ数については正確に数えることが困難であるため、明らかに適用画像から外れた点のみを数



(a) アプリケーション画面

(b) 処理後

図 5 テンプレート画像のクリッピング

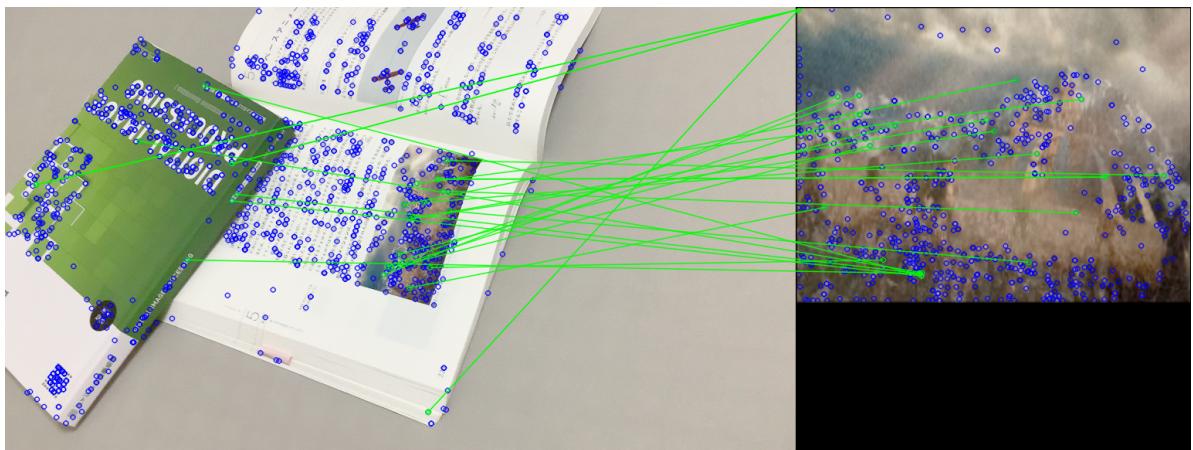


図 6 差し替え先画像の高さ・幅に合わせるマッチング

えた。

表 1 テンプレート画像のサイズと精度

	マッチング数	ミスマッチ数	精度 [%]
差し替え先画像の高さ・幅に合わせる	28	8	71.4
テンプレート画像の高さ・幅に合わせる	68	2	97.1

表 1 から分かるように、マッチング数においてもミスマッチ数においてもテンプレート画像の高さ・幅に合わせた方が優れている。結果として後者の方が高い精度を記録している。

この結果を踏まえて、以降の評価ではテンプレート画像の高さ・幅に画像サイズを合わせて行う。

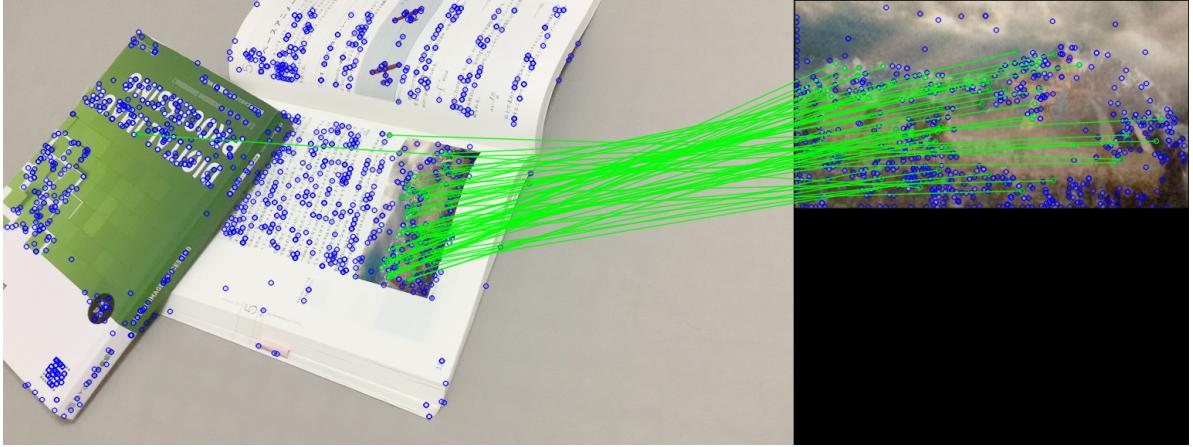


図 7 テンプレート画像の高さ・幅に合わせるマッチング

3.2 置換精度

本節では、様々な状況における置換の精度を評価する。具体的には図 8 に示す、通常の画像、ブレの画像、斜めから見た画像、そして 90° の回転を与えた画像に対して正しく置換できるかを評価する。

この評価は数値で数値で表すことが難しいため、視覚的に比較する。図 9 に実行結果を示す。図 9(a)(b)(d) から分かるように、多少のブレや回転に対しては、ある程度頑健に作用しているまた、(a)(b)(d) はどの画像もスケールが異なっているのにもかかわらず、置換できていることからスケールに対しても頑健であることが分かる。これは本アプリケーションが特徴量検出アルゴリズムとして SIFT を採用していることに起因している。SIFT 特徴量はスケールや移動、回転に対して頑健な性質をもつ。そのために (a) や (d) において問題なくマッチングを行い、置換できたと言える。また、今回発生したブレは等方向的であることから、ある種のスケールと捉えられるため、SIFT で上手くマッチングできたと考えられる。一方で (c) のようにライティングが変化した際には上手く置換できていないことが分かる。これは、今回与えたライティングは対象領域を不均一に変化させていることが原因であると考えられる。これは画像のテクスチャが変化していることを意味するため、正しくマッチングを行えなかったと考えられる。

3.3 部分領域抽出による精度及び速度向上

本節では 2.6 で説明した手法によって、動画データに対して、精度向上及び速度向上を達成できたかを評価する。本評価においてソースコード 5 の radius は 0 とした。

表 2 に 1 フレーム当たりの実行時間を示す。なお、この評価にはフレームの読み出しから、置換結果の書き込みまでを含めた全体の時間である。表 2 から分かるようにアルゴリズムを適用することで、1 フレーム当たりの実行時間は約 84.7% になっている。本評価で使用した動画は全体で 131

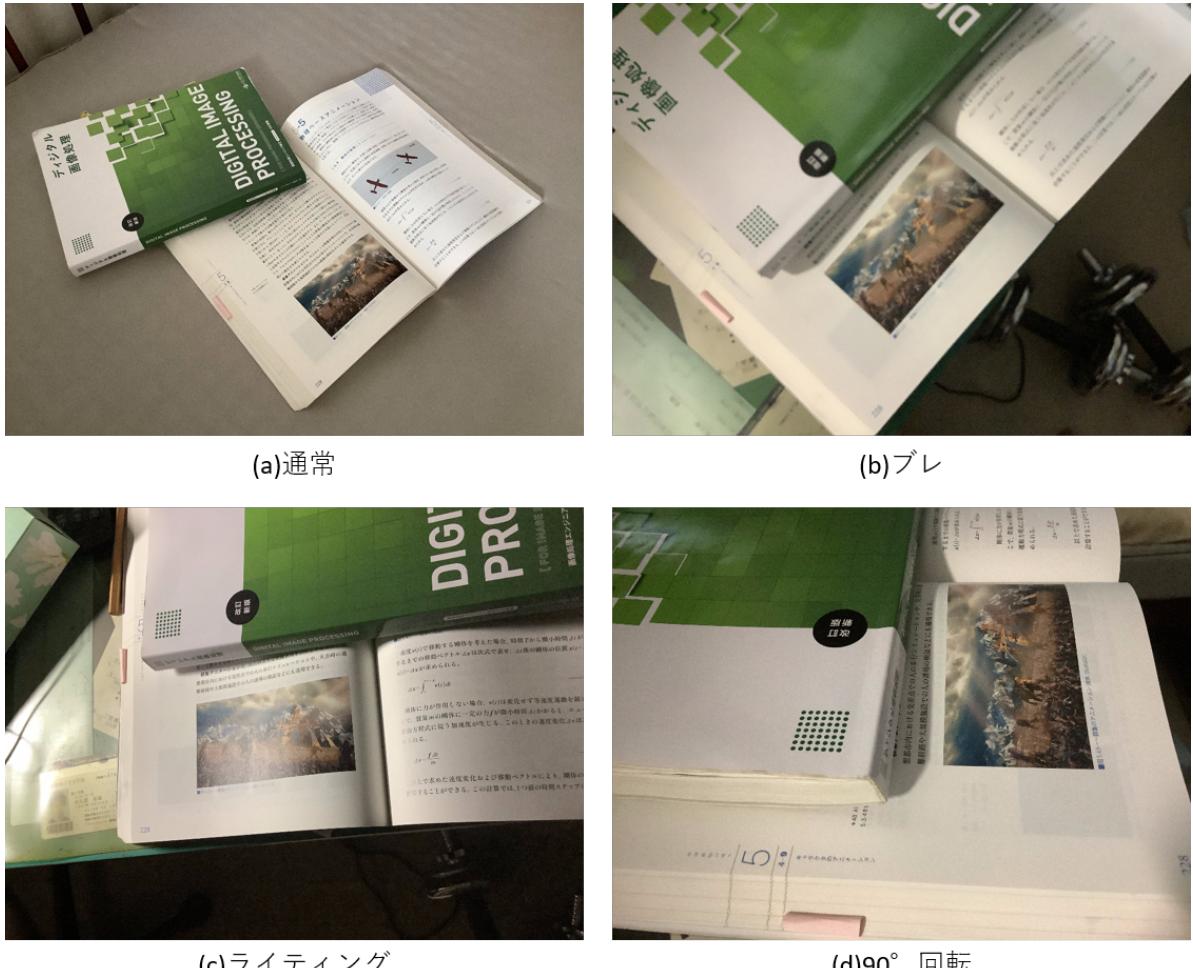


図 8 環境の異なる各データ

表 2 テンプレート画像のサイズと精度

実行時間 [sec/frame]	
適用前	0.287
適用後	0.243

フレームあるため、処理時間は 5.76 秒だけ削減することができた。アルゴリズムを適用することによって探索領域は約 40% になっており、その分だけ計算が削減されるが、実行時間のはそれほど減少しないのは、領域を切り出すために、ソースコード 5 の 36 から 48 行目のような処理を行う必要があるためであると考えられる。

精度向上に関しては添付した「result_before.mp4」「result_after.mp4」を参照されたい。精度に関しては一長一短であるように見られる。適用前の「result_before.mp4」においては、前半はより安定しているが、後半にカメラの移動が速くなると、上手く検出できなくなっている一方で、

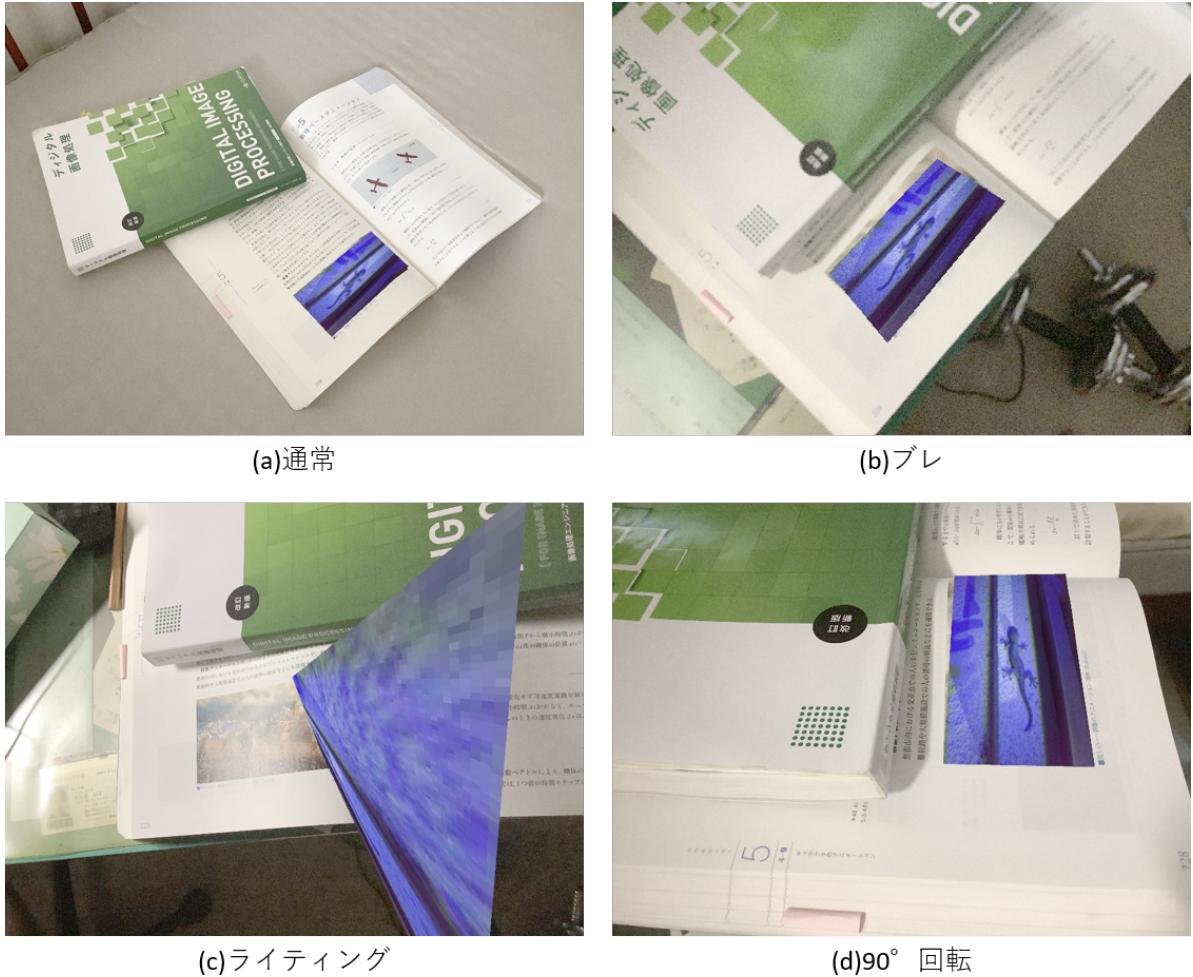


図 9 各環境下での置換結果

「result_after.mp4」は前半の安定感こそ多少劣るものの、後半の誤差は比較的小さいように思われる。特に結果が顕著であったフレームを図 10 に実行結果を示す。

図から分かるようにこのフレームは素早くカメラを移動したことにより、ある方向にブレが生じている。これによって、適用前の結果では本来対応しているはずの画素同士の類似度が低下し、誤った対応をとってしまったためであると考えらえる。適用後の場合は探索領域を絞ったために誤った対応をとる確率が低下し、比較的上手く置換できたと言える。

4 結論

本レポートはスポーツ中継のバーチャル広告などで使用される、画像の差し替え技術を利用したアプリケーションを作成した。

SIFT を利用した特徴量によって並進・スケール・回転にある程度頑健に置換できることを示した。また、動画を対象としているという特徴を活かして、精度を保つ、あるいは向上させながらも



図 10 適用前後で変化が顕著であったフレーム

80% 程度の計算時間で実行できた。

今後の展望としては、次のような工夫が考えられる。

- 特徴点検出アルゴリズムの検討
- 処理の高速化

現在の実装では SIFT 特微量を利用しているが、このアルゴリズムにはライセンスが必要であり、商用のアプリケーションとしてリリースするには向いていない。この問題を解決するアルゴリズムとして、AKAZE[?] などのアルゴリズムが提案されている。AKAZE は商用利用できるだけでなく、SIFT で使用されている、スケールスペースではガウシアンフィルタが等方的に作用するために、エッジもぼやかしてしまうという欠点を解決できるとされている[?]

また、本実装は Python を利用しているために、C++ 等の低級言語と比較して実行速度に難がある。この問題を解決するために、本実装ではできるだけ for 文を使用せずに、テンソル演算を活用した。テンソル演算を活用できるということはインデックス間で処理が独立していることを示しており、GPGPU や OpenMP などを活用できると言える。