Assignment 7

Junyi Liao 20307110289

December 8, 2022

1 Free-Form Deformation

1.1 Algorithm

Free-form deformation (FFD) is a common method for non-linear transformation in image processing. To operate the shape of objects in an image or a 3D-model, FFD employs a B-spline mesh to estimate the deformation field, which can be controlled by grid points in the mesh. These grid points scattered uniformly in the deformation field are referred as control points, whose effect on surrounding pixels (in 3D model, voxels) is defined by the cubic B-spline function:

$$\beta^{(3)}(a) = \begin{cases} \frac{1}{6}(4 - 6a^2 + 3|a|^3), & 0 \le |a| < 1, \\ \frac{1}{6}(2 - |a|)^3, & 1 \le |a| < 2, \\ 0, & |a| \ge 2; \end{cases}$$

thus one pixel or voxel is controlled by 4 control points along one orthogonal direction.

Now we concentrate on image transformation under 2D case. Suppose there're $n_x n_y$ control points

$$\{p_{ij} = (il_x,\ jl_y)\,|\,i = 0, 1, \cdots, n_x - 1,\ j = 0, 1, \cdots, n_y - 1\},$$

which are uniformly spaced in the field, with length of vertical intervals l_x and of horizontal intervals l_y.
An FFD $\mathcal{T}$ drags the control point p_{ij} with offset ϕ_{ij}, a 2-dimension vector indicating the moving distances along $+x$ and $+y$ directions, then the transformation of point $\boldsymbol{z} = [x, y]$ is given by

$$\mathcal{T}(\boldsymbol{z}) = \boldsymbol{z} + \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \phi_{ij} \beta^{(3)} \left(\frac{x - p_{ij}^{(x)}}{l_x} \right) \beta^{(3)} \left(\frac{y - p_{ij}^{(y)}}{l_y} \right);$$

Notice that $\boldsymbol{z}$ is only controlled by its 4 proximal control points along one direction, denote

$$i = \lfloor x/l_x \rfloor,\ j = \lfloor y/l_y \rfloor,$$

then $\boldsymbol{z}$ is controlled by $\{p_{i+k,j+l}\,|\,k = -1, 0, 1, 2,\ j = -1, 0, 1, 2\}$; meanwhile the lattice coordinate of $\boldsymbol{z}$ is

$$[u, v] = \left[\frac{x}{l_x} - i,\ \frac{y}{l_y} - j \right] \in [0, 1] \times [0, 1],$$

thus $\mathcal{T}$ can be simplified as

$$\mathcal{T}(\boldsymbol{z}) = \boldsymbol{z} + \sum_{k=-1}^{2} \sum_{l=-1}^{2} \phi_{i+k,j+l} B_{k+1}(u) B_{l+1}(v),$$

where B is modified B-spline family:

$$B_0(u) = \beta^{(3)}(1+u) = (1-u)^3/6, \qquad B_1(u) = \beta^{(3)}(u) = (4-6u^2+3u^3)/6,$$

$$B_2(u) = \beta^{(3)}(u-1) = (1+3u+3u^2-3u^3)/6, \quad B_3(u) = \beta^{(3)}(2-u) = u^3/6;$$

Define the local offset term

$$Q_{local}(x,y) = \sum_{k=-1}^{2}\sum_{l=-1}^{2} \phi_{i+k,j+l} B_{k+1}(u) B_{l+1}(v),$$

then transformed image $X' = X + Q_{local}$.

1.2 Code

In implement, we use matrix operation to accelerate the computation. We define a object `FFD` to execute the transformation. The code for FFD transformation is shown below.

```
class FFD:
    def __init__(self, lx=64, ly=64, ncpx=9, ncpy=9):
        # Number of pixels in each grid. (length of intervals)
        self.lx = lx
        self.ly = ly
        # Number of control points in each direction.
        self.ncpx = ncpx
        self.ncpy = ncpy
        # Original coordinates of control points.
        x = np.linspace(0, lx * (ncpx - 1), ncpx)
        y = np.linspace(0, ly * (ncpy - 1), ncpy)
        xx, yy = np.meshgrid(x, y, indexing='ij')
        self.cp_org = np.stack((xx, yy), axis=-1)
        # Offset of control points.
        self.cp_offset = np.zeros((ncpx, ncpy, 2))

    @staticmethod
    def Bspline(u: np.float, idx: int):
        # B-spline function.
        if idx == 0:
            return (1 - u) ** 3 / 6
        elif idx == 1:
            return (3 * u ** 3 - 6 * u ** 2 + 4) / 6
        elif idx == 2:
            return (-3 * u ** 3 + 3 * u ** 2 + 3 * u + 1) / 6
        else:
            return u ** 3 / 6

```

```
    def update_offset(self, mat):
        # Read in the offset of control points from a text file.
        print('Read in offset data.')
        with open(mat, 'r') as fp:
            txt = fp.read()
            rows = txt.split('\n')
            i = 0
            for row in rows:
                j = 0
                cols = row.split(',')
                for col in cols:
                    self.cp_offset[i, j, :] = np.array(col.split()).astype(float)
                    j += 1
                i += 1

    def FFDtransform(self, coord: np.float):
        """
        2D FFD transform.
        Input: Coordinate matrix of shape 2 * N, each column corresponds to a pixel point.
        Output: Transformed coordinate matrix.
        """
        x, y = coord[0, :], coord[1, :]
        flr_x, flr_y = np.floor([x / self.lx, y / self.ly]).astype(int)

        # Lattice coordinates.
        u, v = x / self.lx - flr_x, y / self.ly - flr_y

        q_local = np.zeros_like(coord)
        # Get the proximal control point.
        get_cp = lambda a, b: self.cp_offset[a.clip(0, self.ncpx - 1), b.clip(0, self.ncpy - 1)].T
        # Local offset.
        for i in range(-1, 3):
            for j in range(-1, 3):
                bunch = self.Bspline(u, idx=i+1) * self.Bspline(v, idx=j+1)
                bunch = np.array([bunch] * 2)
                q_local += bunch * get_cp(flr_x + i, flr_y + j)
        return coord + q_local
```

2 Inverse Transformation

2.1 Algorithm

To generate the transformed image from $\mathcal{T}$, we need to find the value of each pixel (x, y) in transformed image X'. A general idea is to find the corresponding position in the original image X for each pixel in X' based on the inverse transformation

$$X = \mathcal{T}^{-1}(X'),$$

thus the value of pixel in transformed image X' is

$$I'(x, y) = I(\mathcal{T}^{-1}(x, y)).$$

That's the method of generate transformed image based on inverse transformation.

Notice that the inverse $\mathcal{T}^{-1}(x, y)$ is not always integer, we use linear interpolation to estimate the value of position $(x^*, y^*) = \mathcal{T}^{-1}(x, y)$ in the original image based on 4 proximal pixels:

$$I(x^*, y^*) = w_{11}I(\lfloor x^* \rfloor, \lfloor y^* \rfloor) + w_{12}I(\lfloor x^* \rfloor, \lceil y^* \rceil) + w_{21}I(\lceil x^* \rceil, \lfloor y^* \rfloor) + w_{22}I(\lceil x^* \rceil, \lceil y^* \rceil)$$

$$w_{11} = (\lceil x^* \rceil - x^*)(\lceil y^* \rceil - y),$$

$$w_{12} = (\lceil x^* \rceil - x^*)(y^* - \lfloor y^* \rfloor),$$

$$w_{21} = (x^* - \lfloor x^* \rfloor)(\lceil y^* \rceil - y^*),$$

$$w_{22} = (x^* - \lfloor x^* \rfloor)(y^* - \lfloor y^* \rfloor);$$

Also, sometimes the inverse $\mathcal{T}^{-1}(x, y)$ is out of range in the original image. In such case we set it directly as the closest pixel positioned at the boundary of original image.

Thus we can generate the transformed image by inverse transformation and linear interpolation. To accelerate computation we also apply matrix operations. The code is shown below.

```
def generate_coord(shape):
    """
    Generate the coordinates of pixels with a given shape.
    Input: the shape of image (h, w), height and width;
    Output: an array of dimension 2 * N, N = h * w, with each column the
    coordinate of a pixel.
    """
    h = np.linspace(0, shape[0] - 1, shape[0]).astype(float)
    w = np.linspace(0, shape[1] - 1, shape[1]).astype(float)
    hh, ww = np.meshgrid(h, w, indexing='ij')
    coord = np.stack((hh, ww), axis=-1)
    return coord.reshape((-1, 2)).T


```

```
def bilinear_interp(img: np.ndarray, coord: np.ndarray):
    """
    Based on inverse transformation, compute pixel values.
    Input: image with shape h * w, transformed coordinate matrix with shape 2 *
     hw;
    Output: transformed image with shape
    """
    h, w = img.shape
    coord_floor = np.floor(coord).astype(int)
    offset = coord - coord_floor
    get_pixel = lambda a, b: img[a.clip(0, h - 1), b.clip(0, w - 1)]
    la = (1 - offset[0, :]) * (1 - offset[1, :]) * get_pixel(coord_floor[0, :],
     coord_floor[1, :])
    lb = offset[0, :] * (1 - offset[1, :]) * get_pixel(coord_floor[0, :] + 1,
    coord_floor[1, :])
    ra = (1 - offset[0, :]) * offset[1, :] * get_pixel(coord_floor[0, :],
    coord_floor[1, :] + 1)
    rb = offset[0, :] * offset[1, :] * get_pixel(coord_floor[0, :] + 1,
    coord_floor[1, :] + 1)
    return (la + lb + ra + rb).reshape((h, w))
```

2.2 Estimate of Inverse

An inverse transformation is required in the method above, however the inverse of FFD is unknown. Note that FFD is a non-linear transformation, it's difficult even impossible to derive an inverse transformation of analytical form. Thus we need to find some methods to estimate the inverse.

2.2.1 A Rough Approximation

We can assume that the transformed image X' doesn't diverge much from the original image X. In fact, this assumption can be often satisfied, since many topological structures aren't damaged after transformation. Under the assumption, $Q_{local}(X) = X' - X = o(1)$ is a first-order small quantity;
Moreover, $Q_{local}(X) - Q_{local}(X') = J^{\top}(X - X')$, where Jacobian matrix $J = o(I)$, hence

$$Q_{local}(X) - Q_{local}(X') = o(X - X')$$

is a second-order small quantity, and

$$X = X' - Q_{local}(X') - o(X - X'),$$

$$\mathcal{T}^{-1}(X) \approx X - Q_{local}(X) = 2X - \mathcal{T}(X);$$

2.2.2 Optimization Method

The assumption above is satisfied in varying degrees under different case, and the approximation $2X - \mathcal{T}(X)$ can be inaccurate when the original image is severely deformed.

To improve the accuracy, we apply optimization methods to estimate the inverse of FFD.
For a latent solution (x, y) and the objective point (α, β), it implies $(x, y) = \mathcal{T}^{-1}(\alpha, \beta)$, where RHS is not accessible. An easy idea is to compare $\mathcal{T}(x, y)$ and (α, β), therefore the loss function can be defined as the squared distance between them, and the problem can be written as:

$$\min_{x,y} \|\mathcal{T}(x, y) - (\alpha, \beta)\|_2^2;$$

The objective function is approximately convex, so we can solve it by gradient descent or Newton's method. Denote the objective function as f, in gradient method, the descent direction at point x is

$$p_{gd} = -\nabla f(x),$$

while in Newton's method,

$$p_{nt} = -\left[\nabla^2 f(x)\right]^{-1} \nabla f(x);$$

Here the gradient $\nabla f(x)$ and Hessian matrix $\nabla^2 f(x)$ can't be written in a closed form, so we estimate them using finite-difference approach:

$$\frac{\partial}{\partial x} f \approx \frac{f(x + h) - f(x - h)}{2h},$$

where h is a relatively small quantity.
Set the learning rate as t, the step size along descent direction, the renewal formula in k-th iteration is

$$x^{(k)} = x^{(k-1)} + tp^{(k-1)},\ k = 1, 2, \cdots;$$

A properly large learning rate often leands to quick convergence, while too large rate may "leap over" the optimal solution. To guarantee that the value of objective function is descending, we introduce the Armijo line search rule:

$$f(x + tp) \leq f(x) + \alpha t p^\top \nabla f(x)$$

Here α is a small positive constant. Judge the condition above before update the current solution x as $x + tp$. If the condition is not satisfied, set the step size as $t \leftarrow t/2$ and repeat until it's satisfied.
Now we consider the terminate condition. It's known that the optimal value of objective function $f(x, y) = \|\mathcal{T}(x, y) - (\alpha, \beta)\|_2^2$ is 0. Therefore we set a small constant $\varepsilon > 0$, and accept (x^*, y^*) as a solution if $f(x^*, y^*) < \varepsilon$, say the transformed point $\mathcal{T}(x^*, y^*)$ is sufficiently close to the goal point (α, β).

2.2.3 Implement: A heuristic method

We conbine the two methods above and derive a heuristic method to solve the inverse of FFD. We know the selection of initial point $(x^{(0)}, y^{(0)})$ significantly affects the efficiency of optimization method, even the result. Since we can get an approximate inverse by $\mathcal{F}^{-1}(X) \approx 2X - \mathcal{T}(X)$, we can set this as the initial point. In contrast to random initialization, it reduces the number of iterations needed to find the optimal point.
For the hyperparameters, we set the learning rate $t = 1$, the Armijo constant $\alpha = 10^{-4}$, and the acceptable error $\varepsilon = 1$. In implement, $\varepsilon = 1$ means the error from transformed point $\mathcal{T}(x, y)$ to goal point (α, β) is no more than one pixel, which is not observable by naked eye.
For the optimization method, we choose Newton's method, which is a second-order method and converges faster than gradient descent.

The code for inverse estimation is shown below.

(i) Methods `compute_gradient` and `compute_hessian` estimate the gradient and Hessian matrix of loss function using finite-difference approach, with difference $h = 5 \times 10^{-3}$.

(ii) Methods `optim_gd` and `optim_newton` are optimization algorithms (gradient descent and Newton's method).

(iii) Method `FFDinvtransform` solves the inverse of the defined FFD.

```
class FFD:
    def __init__(self, lx=64, ly=64, ncpx=9, ncpy=9):...

    @staticmethod
    def Bspline(u: np.float, idx: int):...

    def update_offset(self, mat):...

    def FFDtransform(self, coord: np.float):...

    def compute_gradient(self, x, y, xt, yt, h=5e-3):
        """
        Use finite-difference approach to estimate the gradient:
        df/dx = [f(x + h) - f(x - h)] / (2 * h)
        Here the objective function f is the squared distance between
        transformed pixel point and objective pixel point.
        Input: (x, y) -> current point, (xt, yt) -> transformed objective point
    ;
        Output: column vector (df/dx, df/dy).
        """
        f = lambda x, y, x0, y0: ((self.FFDtransform(np.array([[x, y]]).T) - np
    .array([[x0, y0]]).T) ** 2).sum()
        dfdx = (f(x + h, y, xt, yt) - f(x - h, y, xt, yt)) / (2 * h)
        dfdy = (f(x, y + h, xt, yt) - f(x, y - h, xt, yt)) / (2 * h)
        return np.array([[dfdx, dfdy]]).T

    def compute_hessian(self, x, y, xt, yt, h=5e-3):
        """
        Use finite-difference approach to estimate the hessian matrix:
        d2f/d2x = [f'(x + h) - f'(x - h)] / (2 * h)
        Here the objective function f is the squared distance between
        transformed pixel point and objective pixel point.
        Input: (x, y) -> current point, (xt, yt) -> transformed objective point
    ;
        Output: Hessian matrix (d2f/dx2, d2f/dxdy; d2f/dxdy, d2f/dy2).
        """
```

```
        f = lambda x, y, x0, y0: ((self.FFDtransform(np.array([[x, y]]).T) - np.array([[x0, y0]]).T) ** 2).sum()
        d2fdx2 = (f(x + 2 * h, y, xt, yt) + f(x - 2 * h, y, xt, yt) - 2 * f(x, y, xt, yt)) / (4 * h * h)
        d2fdy2 = (f(x, y + 2 * h, xt, yt) + f(x, y - 2 * h, xt, yt) - 2 * f(x, y, xt, yt)) / (4 * h * h)
        d2fdxdy = (f(x + h, y + h, xt, yt) - f(x + h, y - h, xt, yt) - f(x - h, y + h, xt, yt) + f(x - h, y - h, xt, yt)) / (4 * h * h)
        return np.array([[d2fdx2, d2fdxdy], [d2fdxdy, d2fdy2]])

    def optim_gd(self, origin, goal, lr=2, epsilon=1, alpha=1e-04):
        """
        Use gradient descent method for optimization.
        Compute the point, which is mapped onto the goal point after FFD transformation.
        Input:  origin -> start point, goal -> a,
                lr -> learning rate, epsilon: acceptable error;
        Output: optimal solution (x*, y*), error, number of iterations.
        """
        # Initialization: compute an approximate solution.
        origin, goal = origin.reshape((2, 1)), goal.reshape((2, 1))
        error = ((self.FFDtransform(origin) - goal) ** 2).sum()
        p = origin
        ite = 0
        # Iterations.
        while error > epsilon and ite < 1e3:
            grad = self.compute_gradient(p[0, 0], p[1, 0], goal[0, 0], goal[1, 0])
            s = 1
            p_new, error_new = np.zeros_like(p), error
            # Use Armijo line search.
            while error_new - error + alpha * lr * s * (grad ** 2).sum() > 0:
                p_new = p - lr * grad * s
                error_new = ((self.FFDtransform(p_new) - goal) ** 2).sum()
                s *= 0.5
            if np.linalg.norm(p - p_new) < 1e-2:
                break
            p, error = p_new, error_new
            ite += 1
        return {'sol': p, 'error': error, 'ite': ite}

    def optim_newton(self, origin, goal, lr=1, epsilon=1, alpha=1e-04):
```

```
        """
        Use newton method for optimization.
        Compute the point, which is mapped onto the goal point after FFD transformation.
        Input:  origin -> start point, goal -> a,
                lr -> learning rate, epsilon: acceptable error;
        Output: optimal solution (x*, y*), error, number of iterations.
        """

        # Initialization.
        origin, goal = origin.reshape((2, 1)), goal.reshape((2, 1))
        error = ((self.FFDtransform(origin) - goal) ** 2).sum()
        p = origin
        ite = 0
        # Iterations.
        while error > epsilon and ite < 1e3:
            grad = self.compute_gradient(p[0, 0], p[1, 0], goal[0, 0], goal[1, 0])
            hess = self.compute_hessian(p[0, 0], p[1, 0], goal[0, 0], goal[1, 0])
            decrement = np.linalg.solve(hess, grad)
            s = 1
            p_new, error_new = np.zeros_like(p), error
            # Use Armijo line search.
            while error_new - error + alpha * lr * s * (grad * decrement).sum() > 0:
                p_new = p - lr * decrement * s
                error_new = ((self.FFDtransform(p_new) - goal) ** 2).sum()
                s *= 0.5
            if np.linalg.norm(p - p_new) < 1e-2:
                break
            p, error = p_new, error_new
            ite += 1
        return {'sol': p, 'error': error, 'ite': ite}

    def FFDinvtransform(self, coord: np.float):
        """
        Based on given coordinate matrix, solve the original coordinate matrix before FFD.
        Input: transformed coordinate matrix of shape 2 * N;
        Output: original matrix of shape 2 * N, mean square error.
        """
```

```
        # Initialization.
        origin = 2 * coord - self.FFDtransform(coord)

        # Compute the inverse.
        inv = np.zeros_like(origin)
        mse = 0
        sum_iter = 0
        stone = coord.shape[1] // 100
        for i in range(coord.shape[1]):
            # opt = self.optim_gd(origin[:, i], coord[:, i])
            opt = self.optim_newton(origin[:, i], coord[:, i])
            inv[:, i] = opt['sol'].flatten()
            mse += opt['error']
            sum_iter += opt['ite']
            if i % stone == 0:
                print('Computing inverse,\titerations: {0:7d},\t{1}% completed;
    '.format(sum_iter, i / stone))
        mse = np.sqrt(mse / coord.shape[1])
        return {'inv': inv, 'mse': mse}
```

For evaluation, we use mean square error (MSE) to measure the accuracy of our solution. Suppose we get the inverse $\hat{X}' \approx \mathcal{T}^{-1}(X)$, then

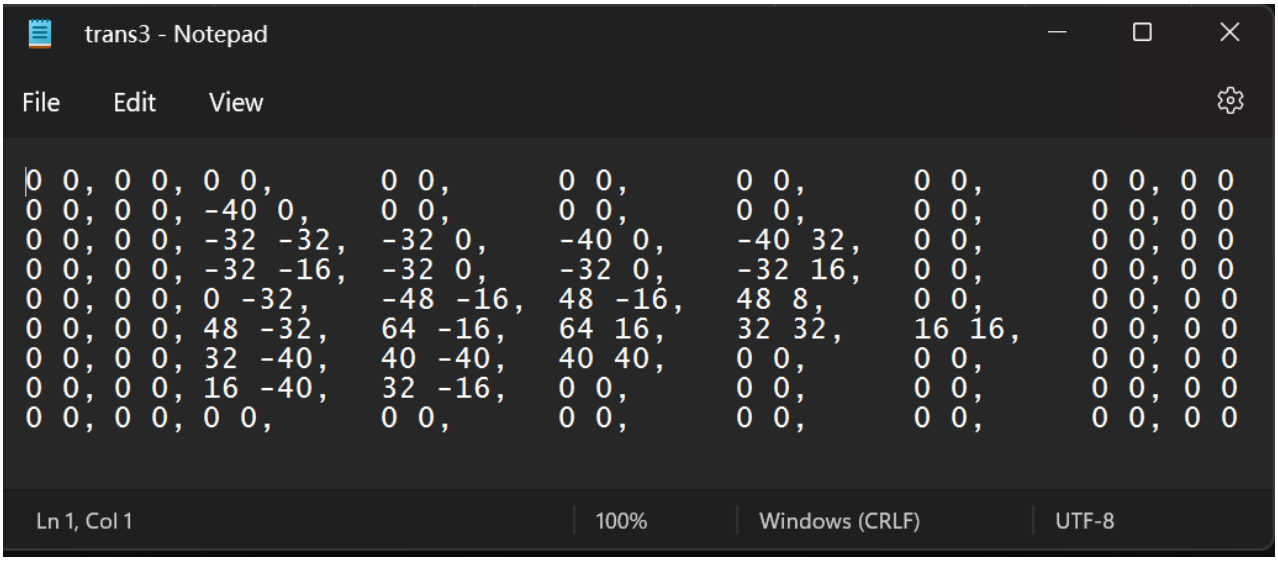
$$MSE = \frac{1}{\sqrt{N}} \|\mathcal{T}(\hat{X}') - X\|_F = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \|\mathcal{T}(\hat{X}_i') - X_i\|_2^2},$$

where N is the number of pixels in the image. MSE indicates the mean error of transformed pixels. It holds that $MSE \leq \sqrt{\epsilon}$.

3 Experiment

We tested our code on several images and different input data. In our experiment, we set the number of control points along one axis as $n_x = 9,\ n_y = 9$ (in total 81 points), with 8 intervals of length $l_x = 64,\ l_y = 64$. All our input images are resized to 513×513.

Our input data includes images and offset of control points. The offset of control points is a $9 \times 9 \times 2$ tensor, referred as $[\phi_{ij}]_{i=0,1,\cdots,8,\ j=0,1,\cdots,8}$ with each $\phi_{ij} \in \mathbb{R}^2$. A preview for data is shown below.



The code to generate transformed image is shown below. It read in images and offset data, then compute the FFD inverse and restore the image by linear interpolation. We save the inverse FFD in our data folder so that redundant computation is not required next time.

```
def main():
    parser = argparse.ArgumentParser(description='Transformation')
    parser.add_argument('-i', '--img_path', type=str, default='images/cat.png',
                        help='the path of source image.')
    parser.add_argument('-o', '--offset_path', type=str,
                        default='data/trans1.txt',
                        help='the path of txt file which stores the offset
    matrix of control points.')
    args = parser.parse_args()

    img = cv2.imread(args.img_path)
    r, g, b = img[:, :, 2], img[:, :, 1], img[:, :, 0]
    coord = generate_coord(r.shape)
    npy_path = osp.splitext(args.offset_path)[0] + '_inv.npy'

    if osp.exists(npy_path):
        # If the data has been fitted, use the saved result directly.
```

```
        ffdinv = np.load(npy_path, allow_pickle=True).item()

    else:
        transformer = FFD()
        transformer.update_offset(args.offset_path)
        ffdinv = transformer.FFDinvtransform(coord)
        # Save the inverse transformation result.
        np.save(npy_path, ffdinv)

    print('Restore the image by interpolation.')
    img_ffd_r = bilinear_interp(r, ffdinv['inv'])
    img_ffd_g = bilinear_interp(g, ffdinv['inv'])
    img_ffd_b = bilinear_interp(b, ffdinv['inv'])
    img_ffd = np.zeros_like(img)
    img_ffd[:, :, 2], img_ffd[:, :, 1], img_ffd[:, :, 0] = img_ffd_r, img_ffd_g
    , img_ffd_b
    print('mse: {0}'.format(ffdinv['mse']))

    # Visualization.
    ...

    tr = osp.split(osp.splitext(args.offset_path)[0])[-1]
    cv2.imwrite(osp.splitext(args.img_path)[0] + f'_ffd_{tr}.png', img_ffd)
```

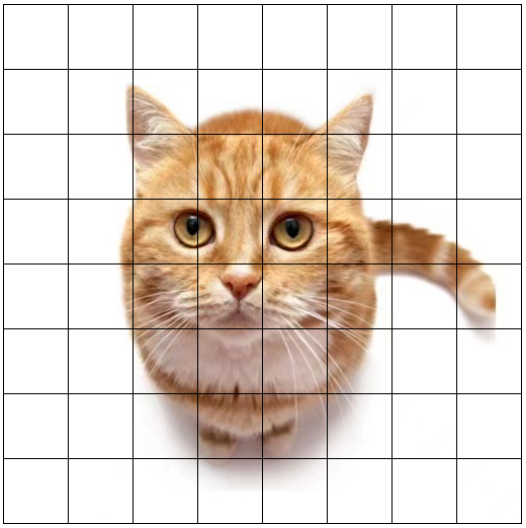
In experiment, we tested both gradient descent method and Newton's method. It turns out Newton's method needs much less iterations (often $10^4 \sim 10^6$) than gradient method (often $10^5 \sim 10^7$).

The visualization result is shown below.

From the result it's seen the effect of deformation is satisfactory. The topological structure within grid doesn't change much, and the original image is correctly transformed according to the deformation field defined by FFD (The grid points in right images are control points).

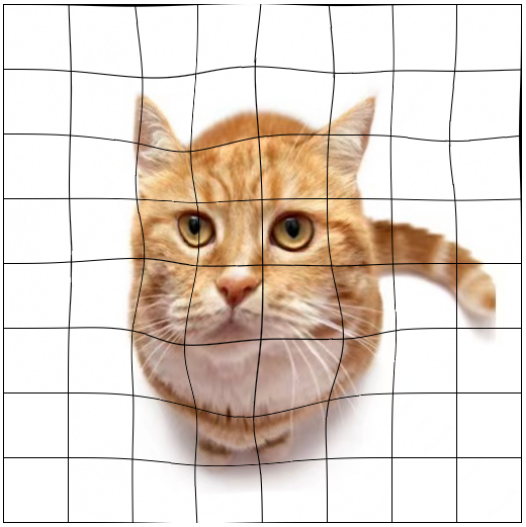
For quantitative evaluation, the MSE of transformed data are all less than 0.5, indicating the accuracy of our estimate of inverse.

(1) Original image cat.png



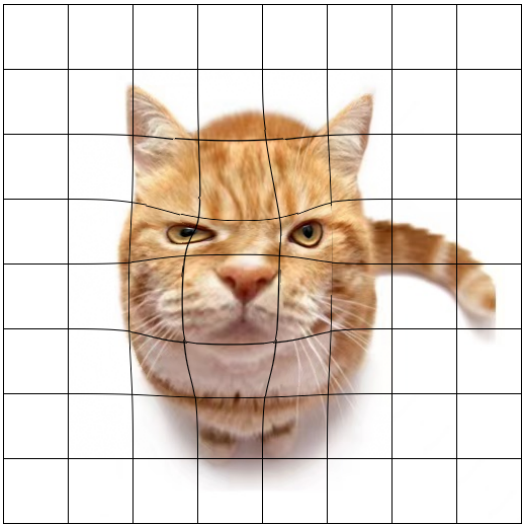
(2) Original image cat.png with grids

(3) FFD transformation result 1



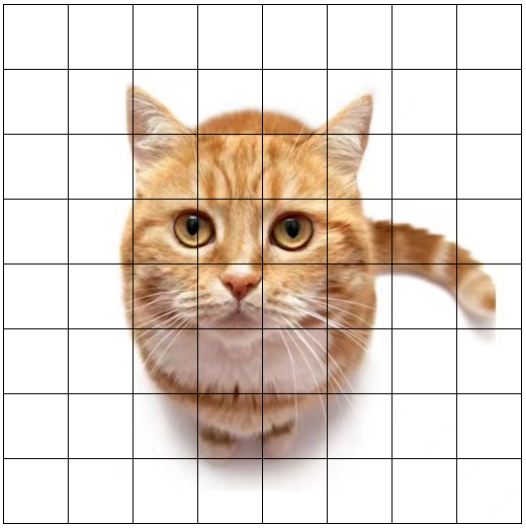
(4) Result 1 with grids, $MSE = 0.1333$

(5) FFD transformation result 2



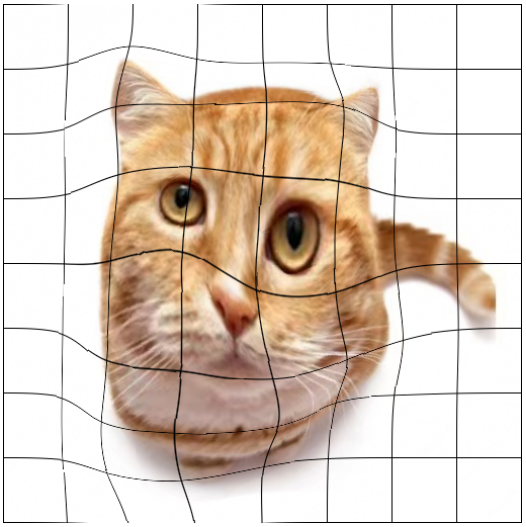
(6) Result 2 with grids, $MSE = 0.1663$

(7) Original image cat.png



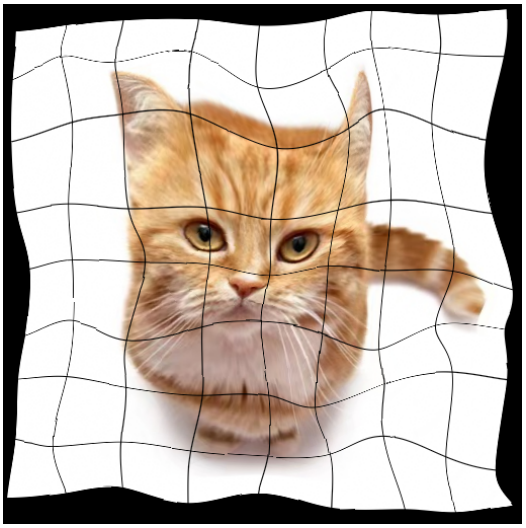
(8) Original image cat.png with grids

(9) FFD transformation result 3



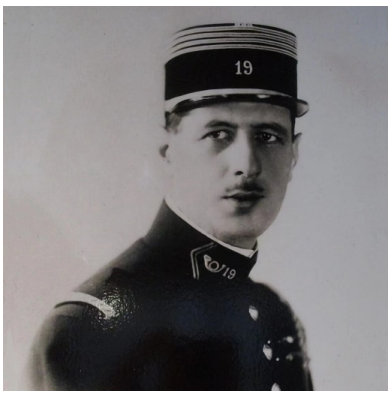
(10) Result 3 with grids, $MSE = 0.3086$

(11) FFD transformation result 4



(12) Result 4 with grids, $MSE = 0.3743$

Additional Results



(13) deGaulle.png

(14) FFD transformation 1

(15) FFD transformation 2

(16) meisje_met_de_parel.png

(17) FFD transformation 3

(18) FFD transformation 4

4 Appendix

4.1 Libraries

We use the `numpy` library to support the large-scale computation of matrices which store the grey level information of images.
We use `opencv` library (`cv2`) to read the input images and save the output images.
We use `matplotlib` to show images before and after processing.
We also use `argparse` to read in the arguments for processing, which can be specified in the command line.

4.2 IDE

All our experiments are done on PyCharm Community Edition 2022.1.2.