SK3 Project 1

Benjamin Li, 400130878, lib83

February 2020

1 Bicubic Interpolation

Here, we apply cubic interpolation on a 2D grid (image), using **convolution**. The algorithm consists of three parts:

1. Deriving an **interpolation kernel**, $u(s)$
2. Padding the input-image to handle **boundary-cases**
3. A matrix-operation that executes the bi-cubic **interpolation function**

1.1 Interpolation Kernel

We will first consider one-dimensional bi-cubic interpolation. "For equally spaced data, many interpolation functions can be written in the form:" [1]

$$g(x) = \sum_k c_k u(\frac{x - x_k}{h}) \tag{1}$$

The following is the *cubic convolution kernel.*[1]. Note $u(s) = 0$ unless it is in the interval [2,2]:

$$u(s) = \begin{cases} A_1|s|^3 + B_1|s|^2 + C_1|s| + D_1 & x < 0 < |s| < 1 \\ A_2|s|^3 + B_2|s|^2 + C_2|s| + D_2 & 1 < |s| < 2 \\ 0 & 2 < |s| \end{cases} \tag{2}$$

Consider two pixels on an axis: x_j and x_k (interpolation nodes); and let h be the sampling interval. Therefore:

$$\frac{x_j - x_k}{h} = j - k \tag{3}$$

Substituting x_j into (1), we get:

$$g(x_j) = \sum_k c_k u(j - k) = c_j \tag{4}$$

The sum equals c_j because $u(j - k) = 0$ everywhere except when $k = j$, at which $u(0) = 1$ [1]. Since $g(x_j) = f(x_j)$ by the *interpolation theorem* [2] and $g(x_j) = c_j$, where $f(x)$ is the function we are interpolating, it stands that $c_j = f(x_j)$. Thus, the c_k's in (1) are simply the sampled data (i.e. pixels).

1.2 Solving A_1 ... D_2

From (2) we have 8 unknowns. We derive a system of 7 equation, meaning one coefficient (A_2) must be manually set; and all other coefficients will be relative

to it. For example, we know $u(0) = 1 \therefore D_1 = 1$ from the first piece of (2). The new *interpolation kernel* or *convolution matrix*, after solving the system is:

$$u(s) = \begin{cases} (A_2+2)|s|^3 - (A_2+3)|s|^2 + 1 & 0 < |s| < 1 \\ A_2|s|^3 - 5(A_2)|s|^2 - 4A_2 & 1 < |s| < 2 \\ 0 & 2 < |s| \end{cases} \tag{5}$$

Robert Keys found that $A_2 = -\frac{1}{2}$ allows $g(x)$ to approximate $f(x)$ with third-order precision [1]. In code, where $\alpha \equiv A_2$, this is:

```
def convMatrix(s):
    # not in [-2,2]
    if not 0 <= s <= 2:
        return 0
    # [-1,1]
    elif 0 <= abs(s) <=1:
        return (ALPHA()+2) * abs(s)**3 - (ALPHA()+3) * abs(s)**2 + 1
    # (1,2], [-2,-1)
    else:
        return ALPHA()* abs(s)**3 - 5*ALPHA() * abs(s)**2 + 8*ALPHA() *
            abs(s)- 4*ALPHA()
```

1.3 Interpolation Function $g(x)$

The interpolation function $g(x)$ in (1), through a series of expansions and substitutions, becomes the following sum of powers of s:

$$\begin{aligned} g(x) = -[A_2(c_{j+2} - c_{j-1}) + (A_2+2)(c_{j+1})]s^3 \\ +[2A_2(c_{j+1} - c_{j-1}) + 3(c_{j+1} - c_j) + A_2(c_{j+2} - c_j)]s^2 \\ -A_2(c_{j+1} - c_{j-1})s^1 + c_j s^0 \end{aligned} \tag{6}$$

This resembles the **project-requirement** in one-dimension: $f(x,y) = \sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij}x^i y^j$, where we see a linear-combination of the powers of the argument(s).

1.4 Deriving the Interpolation Function $g(x)$

The first step is to define $u(s)$ for $s = s, s+1, s-1, s+2$. This domain is selected because of the following:

> Let x be a point to-be-sampled between the samples: x_j and x_{j+1}. We can then express (1) as the below, using (3), where $s = (x - x_j)/h$.

$$
\begin{aligned}
g(x) &= \sum_k c_k u(\frac{x - x_k}{h}) \\
&= \sum_k c_k u(\frac{x - x_j + x_j - x_k}{h}) \\
&= \sum_k c_k u(\frac{x - x_j}{h} + j - k) \\
&= \sum_k c_k u(s + j - k)
\end{aligned}
\tag{7}
$$

We know $-2 \leq (j - k) \leq 1$ for nonzero results of $u(s + j - k)$ because $0 < s < 1$. Thus, k can equal the following:

1. k = j+2 $\longrightarrow s + j - k = s - 2$
2. k = j+1 $\longrightarrow s + j - k = s - 1$
3. k = j $\longrightarrow s + j - k = s$
4. k = j-1 $\longrightarrow s + j - k = s + 1$

Substituting the above k's into (7), we get a simple definition of $g(x)$ with the 4 aforementioned non-zero terms:

$$
g(x) = c_{j-1}u(s + 1) + c_j u(s) + c_{j+1}u(s - 1) + c_{j+2}u(s - 2) \tag{8}
$$

After computing and substituting the u(arg) terms, (6) is derived. For clarity, $c_{j-1}, c_j, c_{j+1}, c_{j+2}$ map to $f(x_{j-1}, f(x_j), f(x_{j+1}), f(x_{j+2})$—four adjacent pixels on an axis, used to interpolate x. ***Example:*** *a pixel at $x = 68.66$, uses adjacent-pixels at $x = 67, 68, 69, 70$ per my code for interpolation.* **In any case, (8) along with the convolution-matrix (5) are directly used in the code.**

1.5 Connection to Code

Looking at (8), recall:

$$s = \frac{x - x_j}{h}$$

The interpolation function, $g(x)$ is represented as the dot-product of a u **vector** and c **vector**. However, in two-dimensions, rather than a vector of pixels in one axis, we interpolate with a 4x4 block of pixels. Furthermore, we need two u vectors: one for x and one for y.

$$g(x) = \begin{bmatrix} u(x_0), u(x_1), u(x_2), u(x_3 0 \end{bmatrix} \cdot \begin{bmatrix} f_{0,0} & f_{0,2} & f_{0,3} & f_{0,4} \\ f_{1,0} & f_{1,2} & f_{1,3} & f_{1,4} \\ f_{2,0} & f_{2,2} & f_{2,3} & f_{2,4} \\ f_{3,0} & f_{3,2} & f_{3,3} & f_{3,4} \end{bmatrix} \cdot \begin{bmatrix} u(y_0) \\ u(y_1) \\ u(y_2) \\ u(y_3) \end{bmatrix} \tag{9}$$

```
def interpolationFunction(x):
    ...
    # u-vectors
    ux = np.array([convMatrix((x-x//1+1)), ...])
    uy = np.array([[convMatrix((y-y//1+1))], ...])

    # matrix of samples
    fMatrix = np.ones((4,4))
    fMatrix[0] = [img[ int((j*samplingIncrement+2)-(y-y//1+1)),
        int((i*samplingIncrement+2)-(x-x//1)), channel],...]
    fMatrix[1] = ...
    fMatrix[2] = ...
    fMatrix[3] = ...
    return np.dot(np.dot(ux, fMatrix), uy)
```

1.6 Padding

$g(x, y)$ interpolates pixels by using the pixels around it. Padding the input-image becomes necessary, when interpolating pixels on or near edges, where there are no pixels to its left/right/top/bottom.

1.6.1 numpy.pad

The **pad** function from NumPy was initially used; however, using it with three-channel color-images (R,G,B) resulted in monochromatic output images.

Custom Padding Function Not enough time, sadly.

2 Difference between Hashed/Non-hashed

There is no difference between the *missing pixels* and *center missing pixels*. By observing the code, you will notice pixels were interpolated in the same way.

3 Average Number of Computations

Every pixel has 3 channels. The number of operations per channel is as follows. One-time operations such as padding the image are not included.

1. *interpolation kernel* or convMatrix() executes **8 times**
2. pixel-values are fetched **16 times** from the input-image
3. a dot-products is performed **twice**

Multiplying (8+16+2) by 3, there are **78 operations** per interpolated-pixel; or just **30 operations** if pixel-fetches are discounted.

4 References

1. R. Keys (1981). "Cubic convolution interpolation for digital image processing". IEEE Transactions on Acoustics, Speech, and Signal Processing. 29 (6): 1153–1160. CiteSeerX 10.1.1.320.776. doi:10.1109/TASSP.1981.1163711.
2. Humpherys, Jeffrey; Jarvis, Tyler J. (2020). "9.2 - Interpolation". Foundations of Applied Mathematics Volume 2: Algorithms, Approximation, Optimization. Society for Industrial and Applied Mathematics. p. 418. ISBN 978-1-611976-05-2
3. Wikipedia contributors. (2019, July 17). Bicubic interpolation. In Wikipedia, The Free Encyclopedia.