

3x3 Median Filter

Tuesday, 26 April, 2022 9:53 AM

① PNG → .dat

```
img_to_dat.py 4 X
img_to_dat.py > ...
1 import numpy as np
2 from PIL import Image, ImageOps } Import libraries
3
4 # image can be png or jpg format
5 loc = "./data/" ← parent directory
6 fname = "input.png" ← image png to be converted to .dat
7 im = Image.open(loc + fname) ← open the image
8 width, height = im.size ← get dimensions/resolution
9 outdat = "input.dat" ← output file
10
11 with open('./data/dimension.dat', 'w') as f: } save dimension/resolution info in dimension.dat to be used/read
12     for item in [width, height]:
13         f.write("%s\n" % item)
14
15 def write_dat(f, pix_val): } callable function: write the pixel values in file variable f
16     for i in range(0, len(pix_val)):
17         f.write(str(pix_val[i])+"\n")
18     f.close()
19
20 # get pixel values
21 print(">> Opening and processing image "+loc+fname+" width "+str(width)+", height "+str(height))
22 im = Image.open(loc+fname, 'r').convert('RGB') ← ignore the RGB, this is written by Dr. Hadi
23 pix_val = list(ImageOps.grayscale(im).getdata()) ← grayscale the input image.
24 # pix_val = list(im.getdata())
25 print(">> Total number of pixels: "+str(len(pix_val)))
26
27 # write dat file
28 print(">> Dat file written at "+loc+outdat)
29 f = open(loc + outdat, 'w')
30 write_dat(f, pix_val) ← call the function above
31
32 # grayscale input image
33 newimage = Image.new('L', (width,height)) } create a blank png and store/write the pixels into it
34 newimage.putdata(np.array(pix_val))
35 newimage.save(f"{loc}gray_{fname}")
```

② 3x3 Median Filter in C++

```
3x3_median_filter.cpp > InsertionSort(int i, int)
1 #include <iostream> } Include libraries
2 #include <chrono>
3 #include <cstdlib>
4 #include <cmath>
5
6 using namespace std;
7
8 void bubbleSort (int inputArray[], int arraySize)
9 {
10     int i, j;
11
12     for (i = 0; i < arraySize - 1; i++){
13         for (j = 0; j < arraySize - i - 1; j++){
14             if (inputArray[j] > inputArray[j + 1]) {
15                 swap(inputArray[j], inputArray[j + 1]);
16             }
17         }
18     }
19 }
20
21 void insertionSort (int inputArray[], int arraySize)
22 {
23     int i, j, key;
24
25     for (i = 1; i < arraySize; i++) {
26
27         key = inputArray[i];
28         j = i - 1;
29
30         while (j >= 0 && inputArray[j] > key) {
31             inputArray[j + 1] = inputArray[j];
32             j = j - 1;
33         }
34
35         inputArray[j + 1] = key;
36     }
37 }
38
39 int main(){
40
41     cout << ">> Reading data..." << endl;
42     FILE *fin, *fdim, *fout, *fout_m, *ftime, *fspeed; ← declare files as variables
43
44     // store image dimensions as dim
45     int dim[2];
46     fdim=fopen("./data/dimension.dat","r");
47     fscanf(fdim, "%d", &dim[0]);
48     fscanf(fdim, "%d", &dim[1]);
49     fclose(fdim);
50
51     // declare width and height
52     int width = dim[0], height = dim[1], row, col, pixel_pointer=0;
53
54     cout << ">> Allocating memory..." << endl;
55     // store input.dat as vec
56     int *vec = (int*) malloc(width*height * sizeof(int)); // Dynamically allocated (width x height) empty matrix space } create dynamic matrix named vec
57
58     fin=fopen("./data/input.dat","r");
59     for (int i = 0; i < width*height; i++) { } store input.dat into vec
60 }
```

```

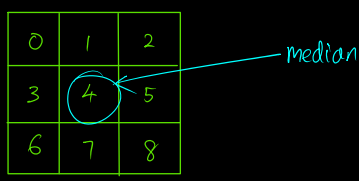
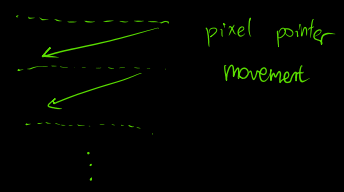
55 cout << ">> Allocating memory..." << endl;
56 // store input.dat as vec
57 int *vec = (int*) malloc(width*height * sizeof(int)); // Dynamically allocated (width x height) empty matrix space } create dynamic matrix named vec
58
59 fin=fopen("../data/input.dat","r");
60 for (int i = 0; i < width*height; i++) { } store input.dat into vec
61     fscanf(fin, "%d", &vec[i]);
62 }
63 fclose(fin);
64
65 // declare storage to store image data (not working directly on image data in vec,
66 // allowing the choice of whether to use preprocessing such as padding)
67 int N = 2000;
68 int *img_array = (int *)malloc(N * N * sizeof(int)); // Dynamically allocated (2000x2000) empty matrix space } create a 2000x2000 dynamic
69 // empty matrix space for the input
70 // to be placed and worked on
71 cout << ">> Preprocessing data..." << endl;
72 // set all elements = 0
73 for (row = 0; row < N; row++){
74     for (col = 0; col < N; col++){
75         img_array[row*N + col] = 0; // equivalent to img_array[row, col]
76     }
77 }
78
79 // store pixel data into img_array matrix
80 // // // With zero padding
81 // for (row = 1; row < height-1; row++){
82 //     for (col = 1; col < width-1; col++){
83 //         img_array[row*N + col] = vec[pixel_pointer];
84 //         pixel_pointer++;
85 //     }
86 // }
87
88 // // No padding
89 for (row = 0; row < height; row++){
90     for (col = 0; col < width; col++){
91         img_array[row*N + col] = vec[pixel_pointer];
92         pixel_pointer++;
93     }
94 }
95
96 // 3x3 median filter main algorithm
97
98 // int window[9];
99 int *window = (int*)malloc(9 * sizeof(int)); // create the 3x3 window
100 double t, speed, sum_t=0, sum_speed=0; // variables for speed and time profiling
101
102 cout << ">> Performing 3x3 median filter, please wait..." << endl;
103
104 fout=fopen("../data/out.gold.dat","w");
105 fout_m=fopen("../data/out.m.dat","w");
106 ftime=fopen("../data/time_ns.dat","w");
107 fspeed=fopen("../data/speed_ns.dat","w");
108 // create those files
109
110 for (row = 1; row <= height; row++)
111 {
112     for (col = 1; col <= width; col++)
113     {
114         // Window = 3x3 matrix, centered at (row,col)
115         window[0] = img_array[(row-1)*N + (col-1)];
116         window[1] = img_array[(row-1)*N + (col)];
117         window[2] = img_array[(row-1)*N + (col+1)];
118         window[3] = img_array[(row)*N + (col-1)];
119         window[4] = img_array[(row)*N + (col)];
120         window[5] = img_array[(row)*N + (col+1)];
121         window[6] = img_array[(row+1)*N + (col-1)];
122         window[7] = img_array[(row+1)*N + (col)];
123         window[8] = img_array[(row+1)*N + (col+1)];
124
125         // sort window array (pick any sorting algorithm above)
126         auto start = chrono::steady_clock::now(); // Start timer
127         insertionSort(window, 9);
128         auto end = chrono::steady_clock::now(); // Stop timer
129         t = chrono::duration_cast<chrono::nanoseconds>(end - start).count(); // get nanoseconds taken
130         sum_t += t; // add to total time taken (still in nanoseconds) ← total time taken
131         speed = std::isfinite(9.0/t) ? 9.0/t : speed; // Speed = (number of pixels in the window / time taken) = pixels per nanoseconds ← speed to get that median
132         sum_speed += speed; // total speeds
133         fprintf(ftime, "%0.8f\n", t); // Write nanoseconds (per window) to time.dat
134         fprintf(fspeed, "%0.8f\n", speed); // write speed (pixels per nanosecond) to speed.dat
135         fprintf(fout, "%d\n", window[4]); // Write median into out.gold.dat
136         fprintf(fout_m, "%d\t", window[4]); // Write median into out.m.dat
137     }
138 }
139
140 double pps = sum_speed*1e9/(width*height); // total speed / total pixels = average pixels per second
141
142 fclose(fout);
143 fclose(fout_m);
144
145 free(img_array);
146 free(vec);
147 free(window);
148
149 cout << ">> Time taken per pixel (nanoseconds) written as /data/time_ns.dat." << endl;
150 cout << ">> Speed per pixel (pixels per nanoseconds) written as /data/speed_ns.dat." << endl;
151 cout << ">> Saved processed output pixels matrix as /data/out.m.dat and flattened as /data/out.gold.dat." << endl;
152 cout << endl << " - - Time profiling - - " << endl << ">> Total time taken = " << sum_t*1e-9 << " seconds." << endl;
153 cout << ">> Processing speed = " << pps << " pixels per second." << endl;
154
155 return 0;
156 }
157

```

with zero padding

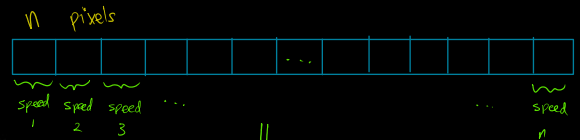
no zero padding

store the input image pixels one by one, row by row into the imp_array.



← elapsed time in nanoseconds (end minus start)

← speed to get that median



$$\frac{1}{time_1} + \frac{1}{time_2} + \frac{1}{time_3} + \dots + \frac{1}{time_n}$$

speed = $\frac{1}{time}$ = average speed

num. pixels

③ .dat → PNG

③ .dat → PNG

```

dat_to_img.py 3 x
dat_to_img.py > ...
1 import numpy as np
2 from PIL import Image
3
4 input_dat = './data/v_outdata.dat' ← read the vertical .dat output file
5 im = Image.open('./data/input.png')
6 width, height = im.size
7 output_png = "output.png" ← declare output png
8
9 with open(input_dat, 'r') as file: ← read the .dat line-by-line (default is read as string format)
10     data = file.readlines()
11
12 for i in range(len(data)):
13     # data[i] = int(data[i].replace(';','\n'))
14     data[i] = int(data[i])
15 } convert data into integer format
16
17 data = np.array(data) ← converted into Numpy array
18 newimage = Image.new('L', (width,height)) ← create new blank PNG ('L' for grayscale, dunno why)
19 print(f">>> Opening and processing dat file {input_dat} width {width}, height {height}")
20 print(f">>> Total number of pixels: {width*height}")
21 print(f">>> PNG file generated as {output_png}")
22 newimage.putdata(data) ← put the pixels in
23 newimage.save(f"./{output_png}")
24 newimage.save(f"./data/{output_png}") } save at 2 separate locations
25

```

④ Plot speed with Python

```

plot_speed.py
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def smooth(y, box_pts):
5     box = np.ones(box_pts)/box_pts
6     y_smooth = np.convolve(y, box, mode='same')
7     return y_smooth
8
9 print(">>> Reading speed data...")
10
11 speedData = [element * 1000 for element in list(map(float, open('./data/speed_ns.dat', 'r').read().splitlines()))] ← read speed.dat and x1000 for all,
12
13 print(">>> Plotting the data...")
14
15 plt.figure(figsize=(20,15)) ← graph size
16 plt.plot(speedData, color='blue', linewidth=1) ← original data, blue line
17 plt.plot(smooth(speedData, 5000), color='red', linewidth=1) ← smoothing window size = 5000, red line
18 plt.legend(['Speed', 'Smoothed'], loc='upper right')
19 # plt.ylim(0, 0.1)
20 plt.grid(True) ← grid on
21 plt.title("3x3 Median Filter - Processing Speed\n", fontsize=30) ← graph title
22 plt.xlabel("\nPixel number", fontsize=20)
23 plt.ylabel("Speed (megapixels per second)\n", fontsize=20) } axis labels
24 plt.savefig("Speed.png") ← save plot as png
25 # plt.show()
26
27 print(">>> Speed per pixel plotted and saved as Speed.png.")

```

so pixels per ns becomes megapixels per second

Example:

$$\frac{0.03 \text{ px}}{1 \text{ ns}} = \frac{0.03 \text{ px}}{1 \times 10^{-9} \text{ s}} = \frac{0.03 \times 10^9 \text{ px}}{1 \text{ s}}$$

$$\frac{0.03 \times 10^3 \times 10^6 \text{ px}}{1 \text{ s}} = \frac{0.03 \times 10^9 \text{ megapixels}}{1 \text{ s}}$$

Here why x1000.

DONE