ADSA Final Project Report

Imran Ibrahimli
imranibrahimli98@gmail.com

Cavid Huseynov
ch4863277@gmail.com

This report covers both the information for users and implementation details about the final project of Advanced Data Structures and Algorithms course. The project focuses on the task of obtaining a projection of the surface of a pot given a set of strips extracted from sequential images of a pot.

1 User Manual

The project is structured into several directories: `include`, `src`, and `build`. `include` contains header files, `src` contains the only source file: `main.cpp`, and `build` contains some dependency files generated by `make` and the executable `flt`. The script `run.sh` is located in the project directory. None of the files should be moved since they are referenced throughout the project by their paths relative to the project directory.

1.1 Compilation

The project is written in C++ without using any 3rd party libraries. The BMP library is implemented as a header-only library in the file `bmp.hpp` in the directory `include`. All of the header files are located in that directory as well - namely, `pixel.hpp` which contains a structs representing 3- and 4-channel pixels and `util.hpp` which contains utility functions used in the project. Compilation from the source files is done using the `make` utility which is installed by default in most Unixes and the included `makefile`. After changing into project directory, run

```
$ make
```

This should generate an executable named `flt` located in `build` directory. To test that the compilation has been successful, you can launch the executable without any arguments:

```
$ build/flt
```

which should output usage information.

NOTE: Everything was tested on Ubuntu 18.04.1 LTS. Unfortunately, we did not have access to a Mac, so if anything does not work on your system, please let us know.

1.2 Usage

It is recommended to use the script `run.sh` *with argument* `-f` *like shown down in the Examples section, as it is the simplest way to run the whole pipeline*
The executable itself works with individual images and can be run in one of the 4 modes:

1. **Calculate psf between 2 images**
   Option: `--psf <img1> <img2> <out_psf>`
   Calculates shifts and psf values between `<img1>` and `<img2>`, and outputs a psf file called `<img1>.psf`

2. **Resize and cut an image**
   Option: `--scale <img_in> <psf_file> <img_out>`
   Resizes `<img_in>` according to `<psf_file>` and saves the output image as `<img_out>`

3. **Merge multiple images into one**
   Option: `--merge <img_out> <img_1> ... <img_N>`
   Merges images `<img_1>` to `<img_N>` into a single image and saves it as `<img_out>`

4. **Display help**
   Option: `--help`
   Displays help

Of course, it is very inconvenient to do this manually for a large number of images, so it is better to use the provided script `run.sh` for batch processing. Notice that unlike the executable, this script works with short option style. The script has modes to do batch psf calculation, batch resizing, and batch merging, also a full automatic mode. Usually user only needs to use the automatic mode. Possible options:

1. **Calculate psf for images in a directory**
   Option: `-p <img_dir> <psf_dir>`
   Calculates psf values for each image in `<img_dir>` and saves psfs in `<psf_dir>`

2. **Resize and cut images in a directory**
   Option: `-s <img_dir> <psf_dir> <resized_dir>`
   Scales all images in `img_dir` according to their psf profile and saves them under name `xxx_resized.bmp` in `<resized_dir>`

3. **Merge multiple images into one**
   Option: `-m <img_dir> <out_img>`
   Merges all images in given directory and saves the result as `<out_img>`

4. **Full automatic**
   Option: `-f <img_dir> <out_img>`
   Does what options 1, 2, and 3 do, combined. Given strips in `<img_dir>`, generates output image. Stores intermediate results in `temp/psf` and `temp/resized`

5. **Display help**
   Option: `-h`
   Displays help

1.3 Examples

Let us consider images `pot_1.bmp` and `pot_2.bmp`. In order to calculate psf values for `pot_1.bmp` and save the psf file under the name `pot_1.psf` we can use:

```
$ build/flt --psf pot_1.bmp pot_2.bmp pot_1.psf
```

Result is a .psf file which is a custom simple plain-text file format consisting of `\n` separated values. The values are: number of horizontal strips, width of every strip, maximum shift, psf1, psf2 ... psfN.
To resize `pot_1.bmp` according to `pot_1.psf` and save the result as `pot_1_resized.bmp`:

```
$ build/flt --scale pot_1.bmp pot_1.psf pot_1_resized.bmp
```

To merge our 2 initial images into `out.bmp`:

```
$ build/flt --merge pot_1.bmp pot_1.psf pot_1_resized.bmp
```

To generate output image from 360 images in directory `pot_360` and save it as `result.bmp` using the script:

```
$ ./run.sh -f pot_360 out.bmp
```

Please remember that the script will try to delete the directory `temp` in current directory, create it again, and write its intermediate output there in directories `psf` and `resized`. If you have already existing directories under that name, they can get deleted, so it is better to `cd` into project directory and call the script there.

2 Principles of Operation

This section covers details of how the program works, the shift detection algorithm and some visualizations. First thing that we need to be able to do is to detect the amount of shift (in pixels) between two rectangular regions. The problem is simplified by knowing that we are only interested in translation along the x axis.

2.1 Shift Detection

We came up with an algorithm similar to cross-correlation. The algorithm considers all possible shifts with a reasonable area of intersection between the regions, and selects the shift where the overlapping parts match the best. Let us consider input regions A and B with $w_A = width(A) \geq w_B = width(B)$ and $height(A) = height(B)$. The algorithm is "sliding" B over A and scoring each configuration using modulus of scaled pixel-wise difference (PD). The scaled pixel-wise difference is defined as

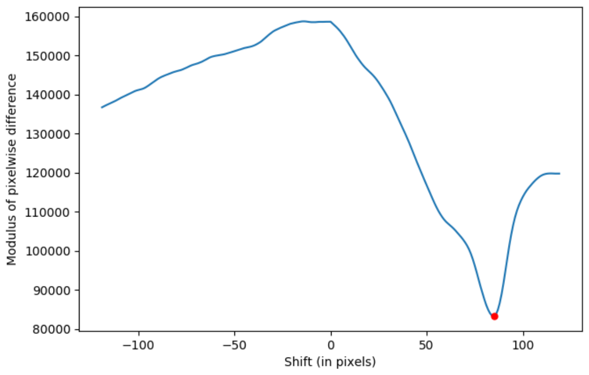


Figure 1: Sum of moduli of PD for each shift. Red point is the best fit

$$diff(pixel1, pixel2) = \frac{|R_{pixel1} - R_{pixel2}| + |G_{pixel1} - G_{pixel2}| + |B_{pixel1} - B_{pixel2}|}{3 \cdot 255}$$

where R, G, and B are red, green, and blue color channels of sub-scripted pixels respectively. The output of this function lies on the interval $[0, 1]$, 0 meaning the pixels are identical, and 1 meaning pixels are as different as possible. To score each shift value $s \in [-w_B + 1, w_A - 1]$, we sum the differences between all pixels of the regions A and B that fall into their intersection when start of B is shifted by s relative to start of A:

$$score(s) = \sum_{p \in A \cap B} diff(p_A, p_B)$$

Notice that a lower *score* means a better fit, so our aim is to find the minimum of *score* with the respect to the shift:

$$\text{best shift} = \arg\min_{s} score(s)$$

In practice, there are possible optimizations, such as reducing the shift search space to $[-\frac{w_b}{4}+1, \frac{w_a}{4}-1]$, assuming that the true shift does not exceed one fourth of the image width. The algorithm is highly parallelizable, that is we wrote its implementation to be multithreaded. The implementation of the algorithm can be found in function `x_correlate_region` in the file `include/util.hpp`.

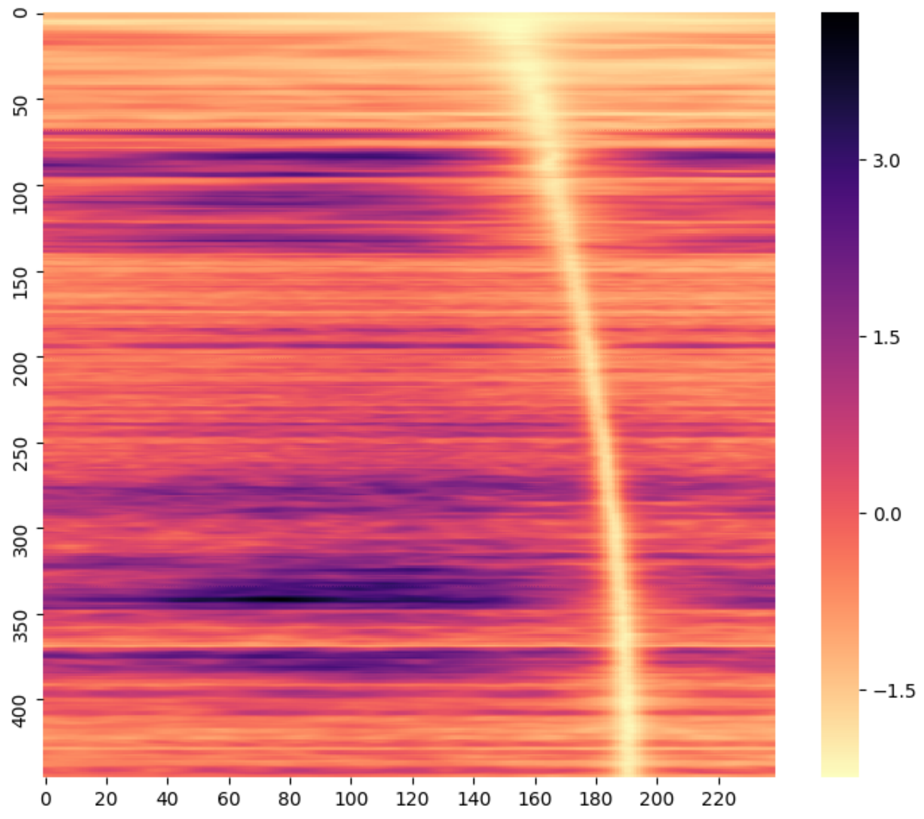


Figure 2: Heatmap of PD scores for horizontal regions for each shift. An approximation for the radius of pot at different heights

In Figure 1 you can see the unscaled value of *score* as a function of shift, and the minimum is marked with a red point. In Figure 2, the heatmap is colored to show the best fit values (lower values of PD) with a brighter color. Values in 2^{nd} figure are normalized. Both of the figures are obtained from `AzCroppedPot25.bmp` and `AzCroppedPot26.bmp`.

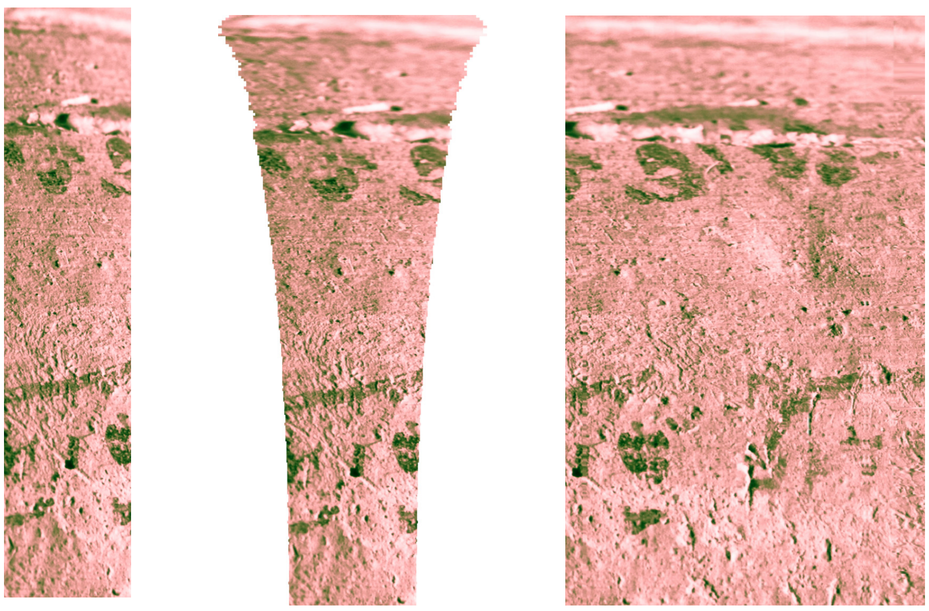
2.2 Scaling

The horizontal upscaling is done by linear interpolation. Let us consider n$^{\text{th}}$ pixel of the target row $t[n]$, that is obtained by scaling source row by a factor of $k \geq 1$, such that l is where $t[n]$ would be in source row: $l \cdot k = n$. We want to know where it came from, but its location in source row $l = \frac{n}{k}$ may not be an integer. If it is not an integer, we take two indexes in source row that l falls between: $i_{left} = floor\left(\frac{n}{k}\right)$ and $i_{right} = ceil\left(\frac{n}{k}\right)$. If we assume that the value changes linearly between these pixels, we can express the color of t_n as a weighted sum of $s[i_{left}]$ and $s[i_{right}]$ (equation of a line passing through 2 points, also we know that distance between x's $(i_{right} - i_{left})$ is 1, allowing us to simplify). Then, this operation is done for each color channel:

$$t[n] = s[i_{left}] \cdot (i_{right} - l) + s[i_{right}] \cdot (l - i_{left})$$

3 Results

Here you can see an example run with `AzCroppedPot25.bmp`: original image, resized image, and a result for 16 provided images. A full image of pot's surface obtained from all 360 images can be downloaded from here (warning: 268 MB file) or you can run `./run.sh -f pot_360/ result.bmp` which takes around 4 minutes and 30 seconds on my machine. Resulting image looks good.



(a) pot25.bmp (b) pot25resized.bmp (c) result16.bmp

Figure 3: Some results