ICP-3038 Computer Vision

Report

Max Petts - *eeub35*

PRIFYSGOL BANGOR UNIVERSITY

- 07.12.2020

Testing

For unit testing GTest was used, and I also visually compared some of the harder to test images.

Operators

The operators where the first thing I tested. The operators are very important and are the basis for many of the more complex functions. I implemented addition, subtraction, multiplication, division, negative, and their equivalent assignment operators, like +=. The test I ran using GTest were:

Test Name	Description	Expected Result	Actual Result
ImageMultiply	Multiplying an image by 2	pixel x 2	pixel x 2
ImageDivision	Divide an image by 2	pixel / 2	pixel / 2
ImageAddition	Add 2 to the image	pixel + 2	pixel + 2
ImageSubtraction	Subtract 2 from the image	pixel - 2	pixel - 2
ImageNegative	Flip the sign of the image	pixel x -1	pixel x -1

The GTest output was:

```
Running main() from gtest_main.cc
[==========] Running 5 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 5 tests from Operators
[ RUN      ] Operators.ImageMultiply
[       OK ] Operators.ImageMultiply (3 ms)
[ RUN      ] Operators.ImageDivision
[       OK ] Operators.ImageDivision (0 ms)
[ RUN      ] Operators.ImageAddition
[       OK ] Operators.ImageAddition (0 ms)
[ RUN      ] Operators.ImageSubtraction
[       OK ] Operators.ImageSubtraction (0 ms)
[ RUN      ] Operators.ImageNegative
[       OK ] Operators.ImageNegative (0 ms)
[----------] 5 tests from Operators (3 ms total)

[----------] Global test environment tear-down
[==========] 5 tests from 1 test case ran. (3 ms total)
[  PASSED  ] 5 tests.
Program ended with exit code: 0
```

Challenges

The only challenges I encountered when testing was using the incorrect data types. When testing the negative and the division operators I forgot to change the data type from **size_t** which is an unsigned int, into **int** and **float** respectively.

RMSE

The tests I ran using GTest were:

Test Name	Description	Expected Result	Actual Result
SameImage	Comparing the same image	0	0
ImageByThree	Multiply the image by 3	2	2
ImageByNine	Multiply the image by 9	8	8
InverseImage	Doing arithmetic on different images	>0	2
DiffImages	Comparing completely different images	>0	1.7
DiffImagesByThree	Multiply the image by 3 then comparing	>0	6.2

The GTest output was:

```
Running main() from gtest_main.cc
[==========] Running 6 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 6 tests from RMSE
[ RUN      ] RMSE.SameImage
[       OK ] RMSE.SameImage (0 ms)
[ RUN      ] RMSE.ImageByThree
[       OK ] RMSE.ImageByThree (0 ms)
[ RUN      ] RMSE.ImageByNine
[       OK ] RMSE.ImageByNine (0 ms)
[ RUN      ] RMSE.InverseImage
[       OK ] RMSE.InverseImage (0 ms)
[ RUN      ] RMSE.DiffImages
[       OK ] RMSE.DiffImages (0 ms)
[ RUN      ] RMSE.DiffImagesByThree
[       OK ] RMSE.DiffImagesByThree (0 ms)
[----------] 6 tests from RMSE (0 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test case ran. (0 ms total)
[  PASSED  ] 6 tests.
Program ended with exit code: 0
```

Challenges

When implementing the getRMSE() function I had trouble with abstracting the formula into code. I hadn't encountered the sigma sign before but once I read around the topic and through the lecture slides I realised that I should use a for loop. Later on in the project I realised that the for loops are unnecessary and could be done entirely with point operators - unfortunately I could not get this working; perhaps with more time I could. Implementing RMSE in this manner may of drastically improved efficiency.

ZNCC

The tests I ran using GTest were:

Test Name	Description	Expected Result	Actual Result
SameImage	Comparing the same image	1	1
InverseImage	Comparing the inverse of an image	-1	-1
Arith_i1	Doing arithmetic on the same image	1	1
Arith_i2	Doing arithmetic on different images	-1	-1
DiffImages	Comparing completely different images	-1	-1
ArithDiffImages	Comparing completely different images	-1	-1

The GTest output was:

```
Running main() from gtest_main.cc
[==========] Running 6 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 6 tests from ZNCC
[ RUN      ] ZNCC.SameImage
[       OK ] ZNCC.SameImage (0 ms)
[ RUN      ] ZNCC.InverseImage
[       OK ] ZNCC.InverseImage (0 ms)
[ RUN      ] ZNCC.Arith_i1
[       OK ] ZNCC.Arith_i1 (0 ms)
[ RUN      ] ZNCC.Arith_i2
[       OK ] ZNCC.Arith_i2 (0 ms)
[ RUN      ] ZNCC.DiffImages
[       OK ] ZNCC.DiffImages (0 ms)
[ RUN      ] ZNCC.ArithDiffImages
[       OK ] ZNCC.ArithDiffImages (0 ms)
[----------] 6 tests from ZNCC (0 ms total)

[----------] Global test environment tear-down
[==========] 6 tests from 1 test case ran. (1 ms total)
[  PASSED  ] 6 tests.
Program ended with exit code: 0
```

Challenges

This was the most annoying method I implemented as I spent days debugging, only to realise that I had used the wrong operator, and was adding the value to the variable without assigning it. I also learnt that it is bad practice to not declare a number variable as 0, as it won't behave as expected without this.

Blending

In order to test the blending method I imported the outputted image sequence into ImageJ, which allowed me to scroll through each frame individually.

Start of the blend

Middle of the blend

End of the blend

The image fades from the original greyscale image and then into the negative version of this - as expected.

Segmentation

Thanks to polymorphism I implemented 2 segmentation functions using thresholding - one that accepts a single parameter, and one that accepts two.

Threshold

I called the threshold with one value first: 100.

Before threshold

After threshold

Threshold 2

Next I called the threshold function with two values: 100 and 200. The following is the output.

Before threshold with two parameters

After threshold with two parameters

Clamping

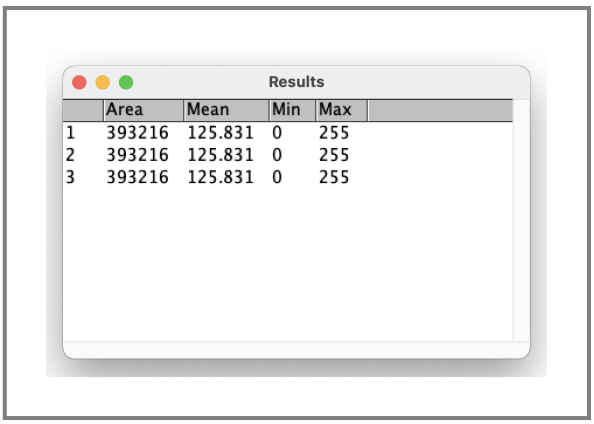
The following images were clamped using 10, as the lower threshold, and 200 as the upper.

Before clamp

After clamp

For each segmentation output I had to import the image into ImageJ and adjust the contrast, I could of normalised the image after calling either of these methods, or even within them - as this would of done roughly the same. All outputs also seem identical, but this is to be expected as they all achieve roughly the same goal

I have included the ImageJ measure results to demonstrate the similarities:



ImageJ measure results

Spacial Convolution

In order to achieve spacial convolution by any kernel, I implemented a function: `Image::conv2d(const Image &aKernel) const`. It accepts a kernel as a parameter, and then checks the kernel size is odd - using a modulus operator. Again I struggled mostly with understanding the formula, and abstracting it into different sections for code - this is akin to my lack of recent maths experience.

I chose to extend the image when the kernel is operating on values outside the image. This was done with the following code:

```
check_x = image_x - (aKernel.m_width / 2) + kernel_x;
check_y = image_y - (aKernel.m_height / 2) + kernel_y;

if (check_x < 0) {
    check_x = 0;
} else if (check_x >= base_image.m_width) {
    check_x = base_image.m_width - 1;
}

if (check_y < 0) {
    check_y = 0;
} else if (check_y >= base_image.m_height) {
    check_y = base_image.m_height - 1;
}
```

I tested this method by filtering images using different kernels, and then checking the results.

Filters

All of the following filters use the previous method: `conv2d()` but all pass a different kernel.

Mean/Box

Kernel:

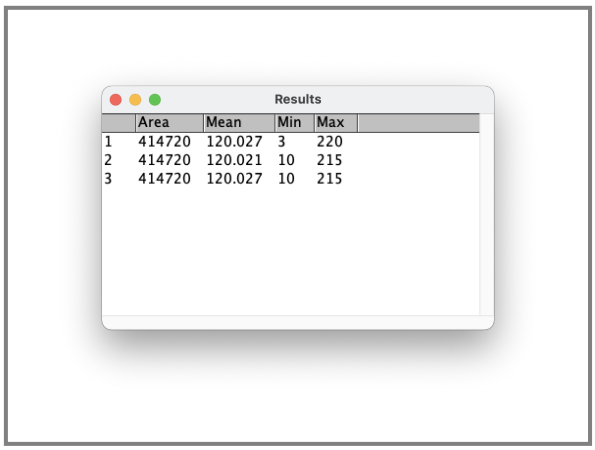
```
1., 1., 1.,
1., 1., 1.,
1., 1., 1.
```

In order to test this method I used ImageJ to box blur the same image and then compared them. Which you can see below, the output images of both programs match - and so I believe this method to be successful.

Before the box blur

After the box blur

The ImageJ blurred image



ImageJ measure results, of each image respectively

Gaussian

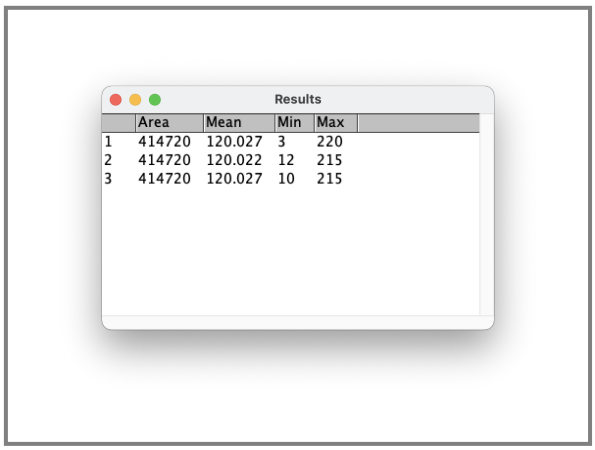
The gaussian blur uses the kernel:

```
1., 2., 1.,
2., 4., 2.,
1., 2., 1.
```

Before the gaussian blur

After the gaussian blur

ImageJ version of gaussian blur

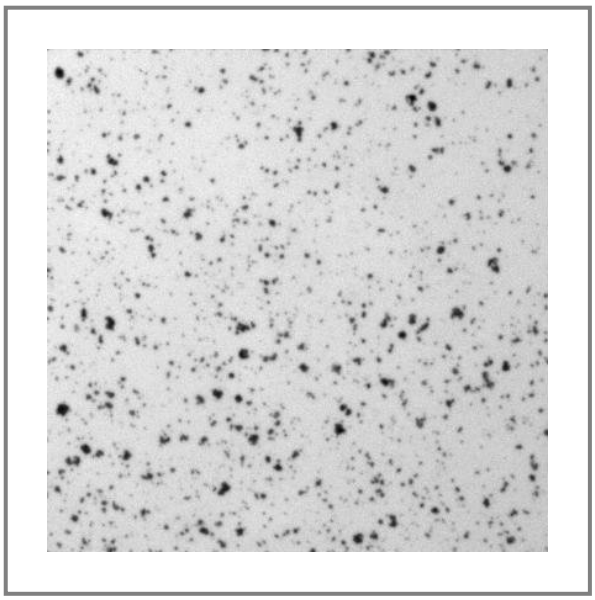


ImageJ measure results for all images

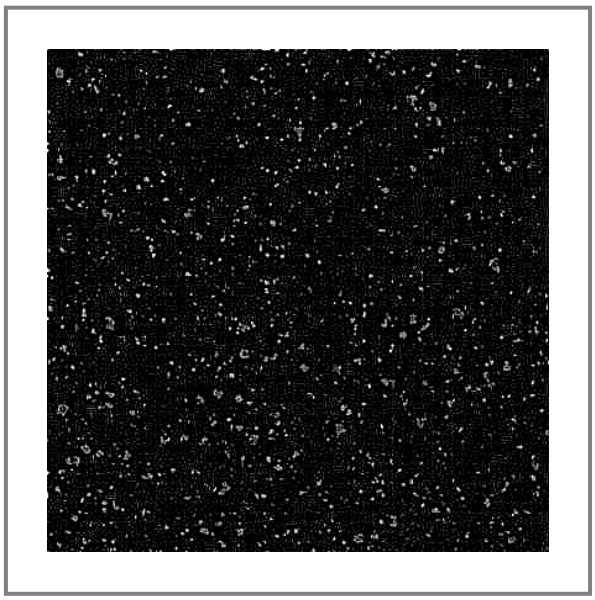
Laplacian

The laplacian filter passes a kernel onto the `conv2d()` function which is:

```
1.,  1., 1.,
1., -8., 1.,
1.,  1., 1.
```



Before laplacian



After laplacian

As you can see from the above images the borders of each cell has been highlighted, as a laplacian finds where the variance is on an image. I have also included some more examples below to demonstrate this filter.

Before laplacian

After laplacian

Sobel Edge

In order to complete this task I implemented two additional methods: `square()` and `squareRoot()`. I passed two separate kernels over the image as to get the vertical and horizontal derivatives - which are the same kernels just transposed. Both additional methods were pretty easy to implement due to the previous operators, *= was used in the `square()` function. I used the sqrt function included in **cmath** to implement `squareRoot()`. Once I'd finished these methods the rest of this task became pretty easy.

Vertical kernel:

```
1., 0., -1.,
2., 0., -2.,
1., 0., -1.
```

Horizontal kernel:

```
1., 2., 1.,
0., 0., 0.,
-1., -2., -1.
```

Before Sobel Edge

After Sobel Edge

Challenges faced

Setting up the tests for each function implemented was one of the hardest parts of the assignment. As coming up with valid methods of testing each was tough, until I began using ImageJ to manipulate the image, and then compare that with the image produced with my code. I began by inputting the kernel being used:

```
1., 2., 1.,
2., 4., 2.,
1., 2., 1.
```

But soon realised, once the test failed, that the actual kernel then gets divided by 9 - so the kernel must also be. The next challenge

Getting the -display flag to be the first command line argument while keeping it entirely optional proved harder than I anticipated, and in hindsight I should of implemented it as the last argument, but I'm used to command line arguments accepting it first.

Future work

If I had more time I would've implemented a way to scroll through the blended images, similar to how the sharpen method functions: with a slider at the top, although the slider affects the opacity of an image, not which image is being displayed, unless it effects the opacity of one image to the next with them all stored in an array - although I can see this getting costly.

I'd like to perhaps improve the efficiency of some of the functions, as when I ran RMSE on an 768×512 image it took way to long to return anything. Due to the terrible performance of ZNCC function it was difficult for me to provide the individual ZNCC or RMSE values for each filter. It seems like Xcode hangs and can't finish executing either function, but works for the smaller images, I'm not sure whether this is due an error in my code or the performance of my computer.

Appendix

The following is all of the code from Image.cxx.

Operators

```
ostream& operator<<(ostream& anOutStream, const Image& anImage) {

    for (size_t row = 0; row < anImage.getHeight(); row++) {
        for (size_t col = 0; col < anImage.getWidth(); col++) {
            anOutStream << anImage(col, row);

            if (col < anImage.getWidth() - 1) {
                anOutStream << " ";
            }
        }

        if (row < anImage.getHeight() - 1) {
            anOutStream << endl;
        }
    }

    return anOutStream;
}

const float& Image::operator()(size_t col, size_t row) const {
    if (col < 0 || col >= m_width || row < 0 || row >= m_height) {
        stringstream err_msg;
        err_msg << "ERROR:";
        err_msg << "\tin File:" << __FILE__ << endl;
        err_msg << "\tin Function:" << __FUNCTION__ << endl;
        err_msg << "\tat Line:" << __LINE__ << endl;
        err_msg << "\tMESSAGE: Pixel(" << col << ", " << row << ") does not exist. The image size
is: " << m_width << "x" << m_height << endl;

        // Throw an exception
        throw out_of_range(err_msg.str());
    }

    return m_pixel_data[row * m_width + col];
}

float& Image::operator()(size_t col, size_t row) {
    // Check if the coordinates are valid, if not throw an error
    if (col < 0 || col >= m_width || row < 0 || row >= m_height) {
        // Format a nice error message
        stringstream error_message;
        error_message << "ERROR:" << endl;
        error_message << "\tin File:" << __FILE__ << endl;
        error_message << "\tin Function:" << __FUNCTION__ << endl;
        error_message << "\tat Line:" << __LINE__ << endl;
        error_message << "\tMESSAGE: Pixel(" << col << ", " << row << ") does not exist. The image
size is: " << m_width << "x" << m_height << endl;

        // Throw an exception
        throw out_of_range(error_message.str());
    }

    // To be on the safe side, turn the flag off
    m_stats_up_to_date = false;

    return m_pixel_data[row * m_width + col];
}
```

```
Image& Image::operator=(const Image& anInputImage) {
    m_pixel_data = anInputImage.m_pixel_data;
    m_width = anInputImage.m_width;
    m_height = anInputImage.m_height;
    m_min_pixel_value = anInputImage.m_min_pixel_value;
    m_max_pixel_value = anInputImage.m_max_pixel_value;
    m_average_pixel_value = anInputImage.m_average_pixel_value;
    m_stddev_pixel_value = anInputImage.m_stddev_pixel_value;
    m_stats_up_to_date = anInputImage.m_stats_up_to_date;
    return *this;
}

Image& Image::operator=(const char* aFileName)
{
    load(aFileName);
    return *this;
}

Image& Image::operator=(const string& aFileName)
{
    load(aFileName);
    return *this;
}

// floating point to image operators. e.g 3 * image_1

Image operator*(float aValue, const Image& anInputImage) {
    Image tmp = anInputImage;

    float* p_data = tmp.getPixelPointer();
    size_t pixels = tmp.getWidth() * tmp.getHeight();

    for (size_t i = 0; i < pixels; ++i) {
        *p_data++ *= aValue;
    }

    return tmp;
}

Image operator+(float aValue, const Image& anInputImage) {
    Image tmp = anInputImage;

    float* p_data = tmp.getPixelPointer();
    size_t pixels = tmp.getWidth() * tmp.getHeight();

    for (size_t i = 0; i < pixels; ++i) {
        *p_data++ += aValue;
    }

    return tmp;
}

Image operator-(float aValue, const Image& anInputImage) {
    // Create a black image of the right size:
    Image temp(0.0, anInputImage.getWidth(), anInputImage.getHeight());

    // Create two pointers on the raw pixel data of the input and output respectively:
    const float* p_input_data = anInputImage.getPixelPointer();
    float* p_output_data = temp.getPixelPointer();

    // Process all the pixels in a for loop:
    size_t number_of_pixels = temp.getWidth() * temp.getHeight();
    for (size_t i = 0; i < number_of_pixels; ++i)
    {
        *p_output_data++ += aValue - *p_input_data++;
```

```
    }

    // Return the new image
    return temp;
}

// Modify an image by a floating point number. e.g image_1 + 5.6

Image Image::operator+(float aValue) const {
    Image tmp = *this;

    tmp.m_stats_up_to_date = false;

    for (size_t i = 0; i < m_width * m_height; ++i)
    {
        tmp.m_pixel_data[i] += aValue;
    }

    return tmp;
}

Image Image::operator-(float aValue) const {
    Image tmp = *this;

    tmp.m_stats_up_to_date = false;

    for (size_t i = 0; i < m_width * m_height; ++i)
    {
        tmp.m_pixel_data[i] -= aValue;
    }

    return tmp;
}

Image Image::operator*(float aValue) const {
    Image tmp = *this;

    tmp.m_stats_up_to_date = false;

    for (size_t i = 0; i < m_width * m_height; ++i)
    {
        tmp.m_pixel_data[i] *= aValue;
    }

    return tmp;
}

Image Image::operator/(float aValue) const {
    Image tmp = *this;

    tmp.m_stats_up_to_date = false;

    for (size_t i = 0; i < m_width * m_height; ++i)
    {
        tmp.m_pixel_data[i] /= aValue;
    }

    return tmp;
}

// Image by Image arithmatic. E.G image_1 * image_2

Image Image::operator+(const Image& img) const {
    Image tmp(0.0, min(m_width, img.m_width), min(m_height, img.m_height));
```

```
    for (size_t j = 0; j < tmp.m_height; ++j)
    {
        for (size_t i = 0; i < tmp.m_width; ++i)
        {
            tmp(i,j) = (*this)(i, j) + img(i, j);
        }
    }
    tmp.m_stats_up_to_date = false;

    return tmp;
}

Image Image::operator-(const Image& img) const {
    Image tmp(0.0, min(m_width, img.m_width), min(m_height, img.m_height));

    for (size_t j = 0; j < tmp.m_height; ++j)
    {
        for (size_t i = 0; i < tmp.m_width; ++i)
        {
            tmp(i,j) = (*this)(i, j) - img(i, j);
        }
    }
    tmp.m_stats_up_to_date = false;

    return tmp;
}

Image Image::operator*(const Image& img) const {
    Image tmp(0.0, min(m_width, img.m_width), min(m_height, img.m_height));

    for (size_t j = 0; j < tmp.m_height; ++j) {
        for (size_t i = 0; i < tmp.m_width; ++i) {
            tmp(i,j) = (*this)(i, j) * img(i, j);
        }
    }
    tmp.m_stats_up_to_date = false;

    return tmp;
}

Image Image::operator/(const Image& img) const {
    Image tmp(0.0, min(m_width, img.m_width), min(m_height, img.m_height));

    for (size_t j = 0; j < tmp.m_height; ++j) {
        for (size_t i = 0; i < tmp.m_width; ++i) {
            tmp(i,j) = (*this)(i, j) / img(i, j);
        }
    }
    tmp.m_stats_up_to_date = false;

    return tmp;
}

Image& Image::operator+=(const Image& img) {
    for (size_t j = 0; j < min((*this).m_height, img.getHeight()); ++j) {
        for (size_t i = 0; i < min((*this).m_width, img.getWidth()); ++i) {
            (*this)(i,j) = (*this)(i, j) += img(i, j);
        }
    }
    m_stats_up_to_date = false;

    return *this;
}

Image& Image::operator-=(const Image& img) {
```

```
    for (size_t j = 0; j < min((*this).m_height, img.getHeight()); ++j) {
        for (size_t i = 0; i < min((*this).m_width, img.getWidth()); ++i) {
            (*this)(i,j) = (*this)(i, j) -= img(i, j);
        }
    }
    m_stats_up_to_date = false;

    return *this;
}

Image& Image::operator*=(const Image& img) {
    for (size_t j = 0; j < min((*this).m_height, img.getHeight()); ++j) {
        for (size_t i = 0; i < min((*this).m_width, img.getWidth()); ++i) {
            (*this)(i,j) = (*this)(i, j) *= img(i, j);
        }
    }
    m_stats_up_to_date = false;

    return *this;
}

Image& Image::operator/=(const Image& img) {
    for (size_t j = 0; j < min((*this).m_height, img.getHeight()); ++j) {
        for (size_t i = 0; i < min((*this).m_width, img.getWidth()); ++i) {
            (*this)(i,j) = (*this)(i, j) /= img(i, j);
        }
    }
    m_stats_up_to_date = false;

    return *this;
}

Image Image::operator!() {
    return m_min_pixel_value + m_max_pixel_value - *this;
}

Image& Image::operator+=(float aValue) {
    *this = *this + aValue;
    m_stats_up_to_date = false;

    return *this;
}
Image& Image::operator-=(float aValue) {
    *this = *this - aValue;
    m_stats_up_to_date = false;

    return *this;
}
Image& Image::operator*=(float aValue) {
    *this = *this * aValue;
    m_stats_up_to_date = false;

    return *this;
}
Image& Image::operator/=(float aValue) {
    *this = *this / aValue;
    m_stats_up_to_date = false;

    return *this;
}
```

Blending

```
Image blending(float alpha, const Image& img1, const Image& img2) {
    return (1.0f - alpha) * img1 + (alpha * img2);
}
```

Normalise

```
Image Image::normalise() {
    return (*this - m_min_pixel_value) / (m_max_pixel_value - m_min_pixel_value);
}
```

Getters

```
float Image::getMinValue() {
    if (!m_stats_up_to_date) updateStats();

    return m_min_pixel_value;
}

float Image::getMaxValue() {
    if (!m_stats_up_to_date) updateStats();

    return m_max_pixel_value;
}

float Image::getMean() {
    if (!m_stats_up_to_date) updateStats();

    return m_average_pixel_value;
}

float Image::getStdDev() {
    if (!m_stats_up_to_date) updateStats();

    return m_stddev_pixel_value;
}

size_t Image::getWidth() const {
    return m_width;
}

size_t Image::getHeight() const {
    return m_height;
}

const float* Image::getPixelPointer() const {
    // There are pixels
    if (m_pixel_data.size() && m_width && m_height)
        return &m_pixel_data[0];
    // There is no pixel
    else
        return 0;
}

float* Image::getPixelPointer() {
    // To be on the safe side, turn the flag off
    m_stats_up_to_date = false;

    // There are pixels
    if (m_pixel_data.size() && m_width && m_height)
        return &m_pixel_data[0];
```

```
    // There is no pixel
    else
        return 0;
}

void Image::updateStats() {
    // Need to update the stats
    if (!m_stats_up_to_date && m_width * m_height) {
        m_min_pixel_value = m_pixel_data[0];
        m_max_pixel_value = m_pixel_data[0];
        m_average_pixel_value = m_pixel_data[0];

        for (size_t i = 1; i < m_width * m_height; ++i) {
            if (m_min_pixel_value > m_pixel_data[i]) m_min_pixel_value = m_pixel_data[i];
            if (m_max_pixel_value < m_pixel_data[i]) m_max_pixel_value = m_pixel_data[i];

            m_average_pixel_value += m_pixel_data[i];
        }
        m_average_pixel_value /= m_width * m_height;

        m_stddev_pixel_value = 0;
        for (size_t i = 0; i < m_width * m_height; ++i) {
            m_stddev_pixel_value += (m_pixel_data[i] - m_average_pixel_value) * (m_pixel_data[i] -
m_average_pixel_value);
        }
        m_stddev_pixel_value /= m_width * m_height;
        m_stddev_pixel_value = sqrt(m_stddev_pixel_value);

        m_stats_up_to_date = true;
    }
}
```

Square & Root

```
Image Image::square() const {
    Image base_image(*this);

    base_image *= base_image;

    return base_image;
}

Image Image::squareRoot() const {
    Image base_image(*this);

    for (size_t x = 0; x < base_image.m_width; ++x) {
        for (size_t y = 0; y < base_image.m_height; ++y) {
            base_image(x, y) = sqrt(float(base_image(x, y)));
        }
    }

    return base_image;
}
```

Load

```
void Image::load(const char* aFileName) {
    string tmp_name = aFileName;
    string capital_name;

    // Capitalise
```

```
    for (int i = 0; i < tmp_name.size(); ++i)
        capital_name += toupper(tmp_name[i]);

    if (string(aFileName).size() > 4) {
        // Load a text file
        if(capital_name.substr(capital_name.length() - 4 ) == ".TXT") {
            // Open the file
            ifstream input_file (aFileName);

            // The file is not open
            if (!input_file.is_open()) {
                // Format a nice error message
                stringstream error_message;
                error_message << "ERROR:" << endl;
                error_message << "\tin File:" << __FILE__ << endl;
                error_message << "\tin Function:" << __FUNCTION__ << endl;
                error_message << "\tat Line:" << __LINE__ << endl;
                error_message << "\tMESSAGE: Can't open " << aFileName << endl;
                throw runtime_error(error_message.str());
            }

            // Empty the image
            m_pixel_data.clear();
            m_width = 0;
            m_height = 0;

            // Load the data into a vector
            string line;
            int number_of_rows(0);
            int number_of_columns(0);

            // Read evely line
            while (getline(input_file, line)) {
                number_of_columns = 0;
                float intensity;
                stringstream line_parser;
                line_parser << line;
                while (line_parser >> intensity)
                {
                    m_pixel_data.push_back(intensity);
                    ++number_of_columns;
                }
                ++number_of_rows;
            }

            // Wrong number of pixels
            if (number_of_rows * number_of_columns != m_pixel_data.size()) {
                // Format a nice error message
                stringstream error_message;
                error_message << "ERROR:" << endl;
                error_message << "\tin File:" << __FILE__ << endl;
                error_message << "\tin Function:" << __FUNCTION__ << endl;
                error_message << "\tat Line:" << __LINE__ << endl;
                error_message << "\tMESSAGE: The file " << aFileName << " is invalid" << endl;
                throw runtime_error(error_message.str());
            }

            // Allocate memory for file content
            m_width = number_of_columns;
            m_height = number_of_rows;
            m_stats_up_to_date = false;
        }
        // Use OpenCV
        else {
#ifdef HAS_OPENCV
```

```
            // Open the image in greyscale
            cv::Mat temp_image = cv::imread(aFileName, cv::IMREAD_GRAYSCALE);

            // The image did not load
            if ( !temp_image.data ) {
                // Format a nice error message
                stringstream error_message;
                error_message << "ERROR:" << endl;
                error_message << "\tin File:" << __FILE__ << endl;
                error_message << "\tin Function:" << __FUNCTION__ << endl;
                error_message << "\tat Line:" << __LINE__ << endl;
                error_message << "\tMESSAGE: Can't open " << aFileName << endl;
                throw runtime_error(error_message.str());
            }

            // Save the size of the image
            m_width = temp_image.cols;
            m_height = temp_image.rows;
            m_pixel_data.resize(m_width * m_height);

            // Copy the pixel data
            cv::Mat img_float;
            temp_image.convertTo(img_float, CV_32F);
            for (int i = 0; i < m_width * m_height; ++i) {
                int x = i / m_width;
                int y = i % m_width;

                m_pixel_data[i] = img_float.at<float>(x, y);
            }


            // The statistics is not up-to-date
            m_stats_up_to_date = false;
#else
            // Format a nice error message
            stringstream error_message;
            error_message << "ERROR:" << endl;
            error_message << "\tin File:" << __FILE__ << endl;
            error_message << "\tin Function:" << __FUNCTION__ << endl;
            error_message << "\tat Line:" << __LINE__ << endl;
            error_message << "\tMESSAGE: OpenCV not supported" << endl;
            throw runtime_error(error_message.str());
#endif
        }
    }
    // Don't know the file type
    else {
        // Format a nice error message
        stringstream error_message;
        error_message << "ERROR:" << endl;
        error_message << "\tin File:" << __FILE__ << endl;
        error_message << "\tin Function:" << __FUNCTION__ << endl;
        error_message << "\tat Line:" << __LINE__ << endl;
        error_message << "\tMESSAGE: Can't open " << aFileName << ", I don't understand the file
type." << endl;
        throw runtime_error(error_message.str());
    }
}

void Image::load(const string& aFileName) {
    load(aFileName.c_str());
}
```

Save

```
void Image::save(const char *aFileName) {
    string temp_filename = aFileName;
    string capital_filename;

    // Capitalise
    for (int i = 0; i < temp_filename.size(); ++i)
    capital_filename += toupper(temp_filename[i]);

    if (string(aFileName).size() > 4) {
        // Load a text file
        if(capital_filename.substr( capital_filename.length() - 4 ) == ".TXT") {
            // Open the file
            ofstream output_file (aFileName);

            // The file is not open
            if (!output_file.is_open()) {
                // Format a nice error message
                stringstream error_message;
                error_message << "ERROR:" << endl;
                error_message << "\tin File:" << __FILE__ << endl;
                error_message << "\tin Function:" << __FUNCTION__ << endl;
                error_message << "\tat Line:" << __LINE__ << endl;
                error_message << "\tMESSAGE: Can't open " << aFileName << endl;
                throw runtime_error(error_message.str());
            }

            // Write content to file
            float* p_data(getPixelPointer());
            for (unsigned int j(0); j < m_height; ++j) {
                for (unsigned int i(0); i < m_width; ++i) {
                    output_file << *p_data++;

                    // This is not the last pixel of the line
                    if (i < m_width - 1) {
                        output_file << " ";
                    }
                }

                // This is not the last line
                if (j < m_height - 1) {
                    output_file << endl;
                }
            }
        }
        // Use OpenCV
        else {
#ifdef HAS_OPENCV
            // Convert the data into an OpenCV Mat instance.
            cv::Mat temp_image(m_height, m_width, CV_32FC1, (float*)getPixelPointer());

            // Write the data
            cv::imwrite(aFileName, temp_image);

#else
            // Format a nice error message
            stringstream error_message;
            error_message << "ERROR:" << endl;
            error_message << "\tin File:" << __FILE__ << endl;
            error_message << "\tin Function:" << __FUNCTION__ << endl;
            error_message << "\tat Line:" << __LINE__ << endl;
            error_message << "\tMESSAGE: OpenCV not supported" << endl;
            throw runtime_error(error_message.str());
```

```
#endif
        }
    }
    // Don't know the file type
    else {
        // Format a nice error message
        stringstream error_message;
        error_message << "ERROR:" << endl;
        error_message << "\tin File:" << __FILE__ << endl;
        error_message << "\tin Function:" << __FUNCTION__ << endl;
        error_message << "\tat Line:" << __LINE__ << endl;
        error_message << "\tMESSAGE: Can't save " << aFileName << ", I don't understand the file
type." << endl;
        throw runtime_error(error_message.str());
    }
}

void Image::save(const string& aFileName) {
    save(aFileName.c_str());
}
```

Display

```
void Image::display(string windowName, bool aNormaliseFlag) const {
#ifdef HAS_OPENCV
    Image display_image = *this;

    if (!display_image.m_stats_up_to_date)
        display_image.updateStats();

    // Normalise the image if needed
    if (aNormaliseFlag)
        display_image = display_image.normalise();

    // Convert the data into an OpenCV Mat instance.
    cv::Mat cv_image(display_image.getHeight(), display_image.getWidth(), CV_32FC1, (float*)
display_image.getPixelPointer());

    // Display the image
    imshow(windowName, cv_image);

    // Wait for a keystroke in the window
    cv::waitKey(0);

#else
    cout << "OpenCV cannot be found" << endl << endl;
#endif // HAS_OPENCV

}
```

RMSE

```
double Image::getRMSE(const Image& image_2) const {
    Image image_1 = *this;

    if (!image_1.m_stats_up_to_date)
        image_1.updateStats();

    // check image size is equivelant, if not throw an error
    if (image_1.m_width == image_2.m_width || image_1.m_height == image_2.m_height) {
        double sse;
```

```
        double total_pixels = image_1.m_width * image_1.m_height;

        for (size_t y = 0; y < image_1.m_height; ++y) {
            for (size_t x = 0; x < image_1.m_width; ++x) {
                sse += pow((double)image_1(x, y) - (double)image_2(x, y), 2);
            }
        }

//        image_1 -= image_2;
//        image_1.square();
//        return sqrt(image_1.getMean());

        sse /= total_pixels;
        return sqrt(sse);

    } else {
        // throw error
        stringstream error_message;
        error_message << "ERROR:" << endl;
        error_message << "\tin File:" << __FILE__ << endl;
        error_message << "\tin Function:" << __FUNCTION__ << endl;
        error_message << "\tat Line:" << __LINE__ << endl;
        error_message << "\tMESSAGE: The size of the images are different." << endl << endl;
        throw runtime_error(error_message.str());
    }
}
```

ZNCC

```
double Image::getZNCC(const Image& anImage) const {
    Image image_1 = *this; // is this more expensive ???
    Image image_2 = anImage;

    // Check image size matches, if not throw error
    if (image_1.m_width == image_2.m_width || image_1.m_height == image_2.m_height) {
        size_t total_pixels = image_1.m_width * image_1.m_height;
        double denominator = image_1.m_stddev_pixel_value * image_2.m_stddev_pixel_value;
        double numerator = 0;
        double result = 0;

        for (size_t y = 0; y < image_1.m_height; ++y) {
            for (size_t x = 0; x < image_1.m_width; ++x) {
                numerator = (image_1(x, y) - image_1.getMean()) * (image_2(x, y) -
image_2.getMean());

                result += numerator / denominator;
            }
        }

        result /= total_pixels;
        return result;
    } else {
        // throw error
        stringstream error_message;
        error_message << "ERROR:" << endl;
        error_message << "\tin File:" << __FILE__ << endl;
        error_message << "\tin Function:" << __FUNCTION__ << endl;
        error_message << "\tat Line:" << __LINE__ << endl;
        error_message << "\tMESSAGE: The size of the images are different." << endl << endl;
        throw runtime_error(error_message.str());
    }
}
```

Convolution

```
Image Image::conv2d(const Image &aKernel) const {
    // Check kernel size
    if (aKernel.m_width % 2 == 0 || aKernel.m_height % 2 == 0) {
        // throw error
        stringstream error_message;
        error_message << "ERROR:" << endl;
        error_message << "\tin File:" << __FILE__ << endl;
        error_message << "\tin Function:" << __FUNCTION__ << endl;
        error_message << "\tat Line:" << __LINE__ << endl;
        error_message << "\tMESSAGE: Kernel sizes should be odd: 3x3, 5x5, 7x7 etc." << endl <<
endl;
        throw runtime_error(error_message.str());
    } else {
        Image base_image = *this;
        Image out_image(0.0, base_image.m_width,  base_image.m_height);

        for(size_t image_x = 0; image_x < base_image.m_width; ++image_x) {
            for(size_t image_y = 0; image_y < base_image.m_height; ++image_y) {
                float accumulator = 0;
                unsigned check_x = 0;
                unsigned check_y = 0;

                for(size_t kernel_x = 0; kernel_x < aKernel.m_width; ++kernel_x) {
                    for(size_t kernel_y = 0; kernel_y < aKernel.m_height; ++kernel_y) {
                        // Check borders
                        check_x = image_x - (aKernel.m_width / 2) + kernel_x;
                        check_y = image_y - (aKernel.m_height / 2) + kernel_y;

                        if (check_x < 0) {
                            check_x = 0;
                        } else if (check_x >= base_image.m_width) {
                            check_x = base_image.m_width - 1;
                        }

                        if (check_y < 0) {
                            check_y = 0;
                        } else if (check_y >= base_image.m_height) {
                            check_y = base_image.m_height - 1;
                        }

                        accumulator += aKernel(kernel_x, kernel_y) * base_image(check_x, check_y);
                    }
                }
                out_image(image_x, image_y) = accumulator;
            }
        }

//        // preserve range
//        out_image.clamp(max(out_image.getMinValue(), base_image.getMinValue()),
max(out_image.getMaxValue(), base_image.getMaxValue()));
//
//        out_image.normalise() * 255;

        return out_image;
    }
}
```

Gaussian Blur

```
Image Image::gaussianFilter() const {
    // create kernel
    Image kernel({
        1., 2., 1.,
        2., 4., 2.,
        1., 2., 1.
    }, 3, 3);

    // Normalise the kernel so that the sum of its coefficients is 1.
    kernel /= 16.0;

    // Filter the image
    return conv2d(kernel);
}
```

Mean, Average & Box Filter

```
Image Image::meanFilter() const {
    // create kernel
    Image kernel(
                 {
        1., 1., 1.,
        1., 1., 1.,
        1., 1., 1.
    }, 3, 3);

    // Normalise the kernel so that the sum of its coefficients is 1.
    kernel /= 9.0;

    // Filter the image
    return conv2d(kernel);
}

Image Image::averageFilter() const {
    return meanFilter();
}

Image Image::boxFilter() const {
    return meanFilter();
}
```

Laplacian

```
Image Image::laplacianFilter() const {
    // create kernel
    Image kernel({
        1.,  1., 1.,
        1., -8., 1.,
        1.,  1., 1.
    }, 3, 3);

    // Filter the image
    return conv2d(kernel);
}
```

Gradient Magnitude

```
Image Image::gradientMagnitude() const {
    Image kernel_x({
        1., 0., -1.,
        2., 0., -2.,
        1., 0., -1.
    }, 3, 3);

    Image kernel_y({
        1., 2., 1.,
        0., 0., 0.,
        -1., -2., -1.
    }, 3, 3);

    Image vert_derivative = conv2d(kernel_x);
    Image horiz_derivative = conv2d(kernel_y);

    Image sobel_image = vert_derivative.square() + horiz_derivative.square();

    return sobel_image.squareRoot();

}
```

Sharpen

```
Image Image::sharpen(double alpha) {
    Image gaussian_kernel({
        1.,  4.,  7.,  4., 1.,
        4., 16., 26., 16., 4.,
        7., 26., 41., 26., 7.,
        4., 16., 26., 16., 4.,
        1.,  4.,  7.,  4., 1.
    }, 5, 5);

    gaussian_kernel /= 273.;

    // Create an image of the details
    Image blur = conv2d(gaussian_kernel);
    Image details = *this - blur;

    // Add details back into original
    Image output = *this + alpha * details;

    // Preserve dynamic range
    return output.clamp(getMinValue(), getMaxValue());
}
```

Clamp

```
Image Image::clamp(float aLowerThreshold, float anUpperThreshold) const {
    Image base_image(*this);

    for (size_t x = 0; x < base_image.m_width; ++x) {
        for (size_t y = 0; y < base_image.m_height; ++y) {
            if (base_image(x, y) < aLowerThreshold)
                base_image(x, y) = aLowerThreshold;
            else if (base_image(x, y) > anUpperThreshold)
                base_image(x, y) = anUpperThreshold;
        }
    }
```

```
    return base_image;
}
```

Threshold

```
Image Image::threshold(float aThreshold) const {
    Image base_image(*this);

    // TODO: Check threshold is within image range
    for (size_t x = 0; x < base_image.m_width; ++x) {
        for (size_t y = 0; y < base_image.m_height; ++y) {
            if (base_image(x, y) < aThreshold) {
                base_image(x, y) = 0;
            } else {
                base_image(x, y) = 1;
            }
        }
    }

    return base_image;
}

Image Image::threshold(float aLowerThreshold, float anUpperThreshold) const {
    Image base_image(*this);

    // TODO: Check threshold is within image range
    for (size_t x = 0; x < base_image.m_width; ++x) {
        for (size_t y = 0; y < base_image.m_height; ++y) {
            if (base_image(x, y) < aLowerThreshold) {
                base_image(x, y) = 0;
            } else if (base_image(x, y) > anUpperThreshold) {
                base_image(x, y) = 0;
            } else {
                base_image(x, y) = 1;
            }
        }
    }

    return base_image;
}
```