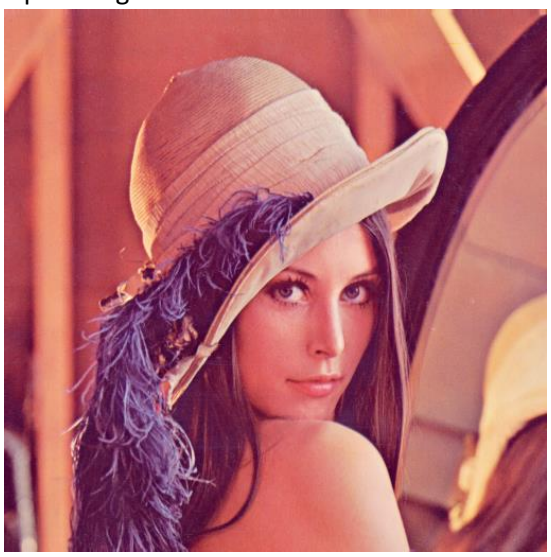
Computer Vision - OpenCV Lab Report

Introduction

The goal of this assignment is to familiarise myself with OpenCV and the different uses of it in a variety of applications. This report contains 4 different practical applications of how it OpenCV can be used to create complex programs more easily.

Lab 1

- Aim: Understand the basic principles behind OpenCV and use them to apply various imaging filters to a picture.
- This first lab was made up of several small tasks that would be coded into 6 separate programs. The first of these, was to simply open an image from a file, and then display the image in a window with a title on the window border. All other sections built off the code used in this first program. Second, was to convert the imported image to greyscale, optionally display the image afterwards, and then save the image to an output file. The next program applied a log scale to a greyscale image, which essentially made areas of a darker contrast in the image appear lighter. The last 3 sections were all about applying different filters to the image then saving and optionally displaying the results. The mean filter and median filter images required an additional input of a number to measure the how strong the filter would be applied, except for Gaussian, which also had 2 optional numbers to adjust other aspects of the filter as well.
- Command line arguments for each section were:
    - displayImage: <input_image>
    - rgb2grey, logScale: <input_image> <output_image> [-display]
    - meanFilter, medianFilter: <input_image> <output_image> <radius> [-display]
    - gaussianFilter: : <input_image> <output_image> <radius> [-display] [sigmaX] [sigmaY]
- Input images used were:



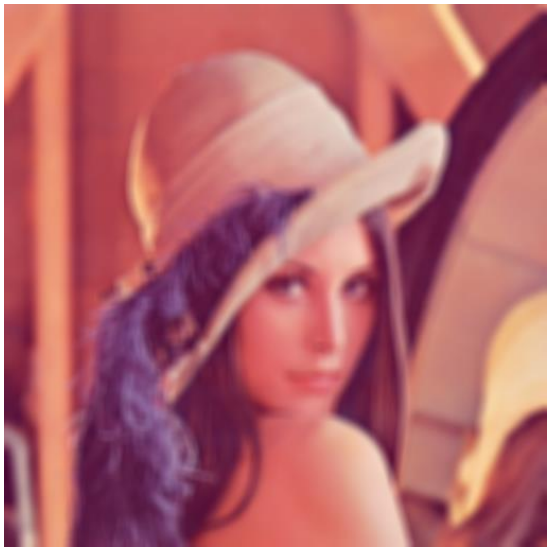
lena_color_512.png
for displayImage, rgb2grey, meanFilter, medianFilter & gaussianFilter

lake.tif
for logScale

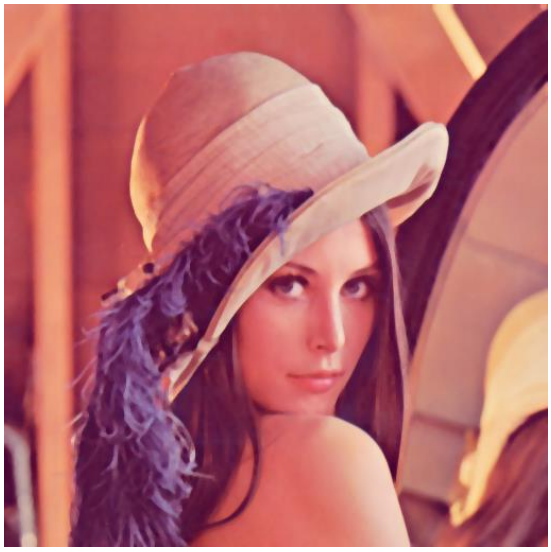
- Output images created from the programs that made visual changes:

rgb2grey

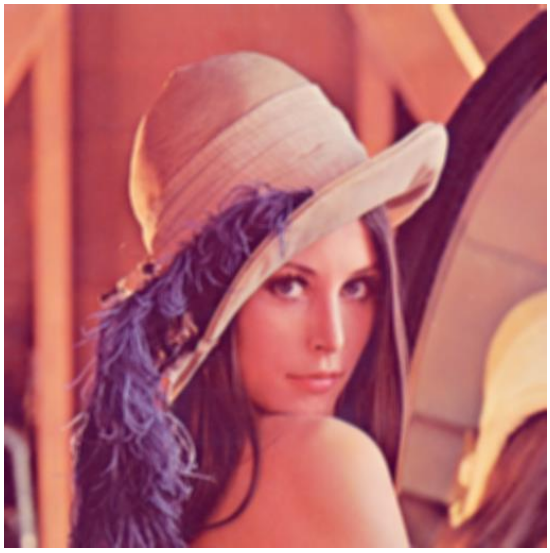
logScale



meanfilter



medianFilter



gaussianFilter

- Key parts of source code taken from each program:

```
// Create an image instance
cv::Mat image;

// Open and read the image
image = cv::imread(input_filename, cv::IMREAD_COLOR);

// The image has not been loaded
if (!image.data)
{
    // Create an error message
    std::string error_message;
    error_message = "Could not open or find the image \"";
    error_message += input_filename;
    error_message += "\".";

    // Throw an error
    throw error_message;
}
```

```
// Create a string to contain the window title
string window_title;
window_title = "Display \"";
window_title += input_filename;
window_title += "\"";

// Create the window
cv::namedWindow(window_title, cv::WINDOW_AUTOSIZE);

// Show the image in the window
cv::imshow(window_title, image);

// Wait for a user input to leave the window
cv::waitKey(0);
```

displayImage

Shows the image being imported from input_filename, which uses the command line argument, and tests to make sure the image name was correct, and the image has loaded. Then creates the window where the image will be displayed.

```
// Convert to grey
cv::Mat grey_image;
cv::cvtColor(image, grey_image, cv::COLOR_RGB2GRAY);

// Write the image
if (!cv::imwrite(output_filename, grey_image))
{
    // The image has not been written

    // Create an error message
    std::string error_message;
    error_message = "Could not write the image \"";
    error_message += output_filename;
    error_message += "\".";

    // Throw an error
    throw error_message;
}
if (display_image) {
    // Create a string to contain the window title
    string window_title;
```

rgb2grey

Show the greyscale image being converted and the process of saving the image to a file before it can be displayed.

```
// Convert to grey
cv::Mat grey_image;
cv::cvtColor(image, grey_image, cv::COLOR_RGB2GRAY);

// Convert to float
cv::Mat float_image;
grey_image.convertTo(float_image, CV_32FC1);

// Log transformation
cv::Mat log_image;
cv::log(float_image + 1.0, log_image);

// Normalisation
cv::Mat normalised_image;
cv::normalize(log_image, normalised_image, 0, 255, cv::NORM_MINMAX, CV_8UC1);

// Write the image
if (!cv::imwrite(output_filename, normalised_image))
```

logScale

Shows the image process for a log transformation applied to the image.

```
float kernel_width = radius * 2 + 1;
float kernel_height = radius * 2 + 1;

// Filter size
cv::Size filter_size(kernel_width, kernel_height);

// Create image instances
cv::Mat image;
cv::Mat filtered_image;

// Open and read the image
image = cv::imread(input_filename, cv::IMREAD_COLOR);
```

```
// Apply filter
cv::blur(image, filtered_image, filter_size);

// Write the image
if (!cv::imwrite(output_filename, filtered_image))
```

meanFilter

Shows how the radius is used and how the filter is made for the blur.

```
// Apply filter
cv::medianBlur(image, filtered_image , radius);

// Write the image
if (!cv::imwrite(output_filename, filtered_image))
```

medianFilter

Shows the radius again, though it is used only as it's pure value from the command line argument this time.

```
float kernel_width = radius * 2 + 1;
float kernel_height = radius * 2 + 1;

// Filter size
cv::Size filter_size(kernel_width, kernel_height);

if (sigmaX == 0) {
    sigmaX = filter_size.width;
}
if (sigmaY == 0) {
    sigmaY = sigmaX;
}
```

```
// Apply filter
cv::GaussianBlur(image, filtered_image, filter_size, sigmaX, sigmaY);

// Write the image
if (!cv::imwrite(output_filename, filtered_image))
```

gaussianFilter

Shows how the filter is made, and how sigma values are determined if they are not specified in the command line arguments.

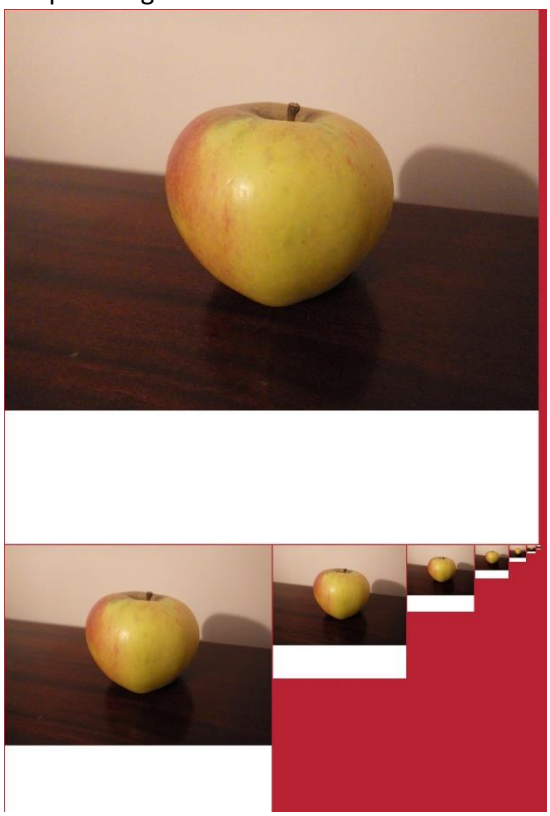
Lab 2

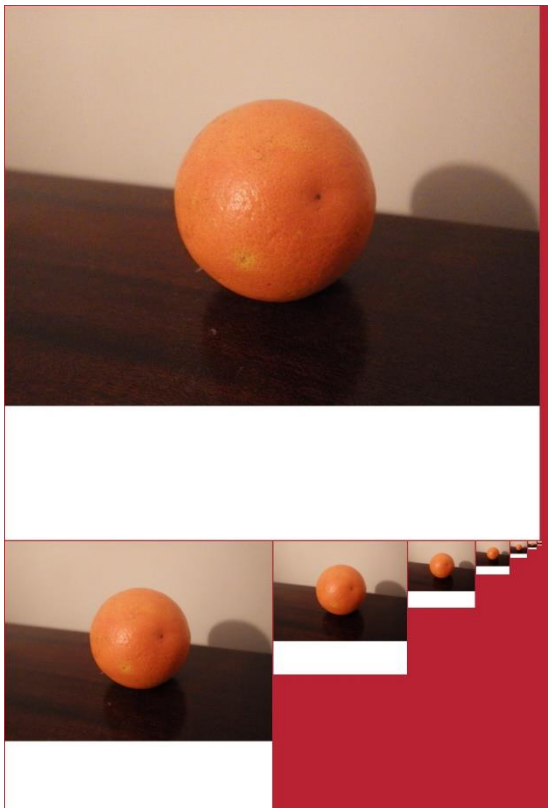
- Aim: Understand pyramid blending in images and create blended images
- This lab was made up of fewer, though more complicated tasks whereby I would compute two different pyramids onto two images, then blend these to make one final image.
- Command line arguments for the Blending program were:
  <input_image_1> <input_image_2> <output_image> <pyramid_levels>
- Input images

apple.jpg

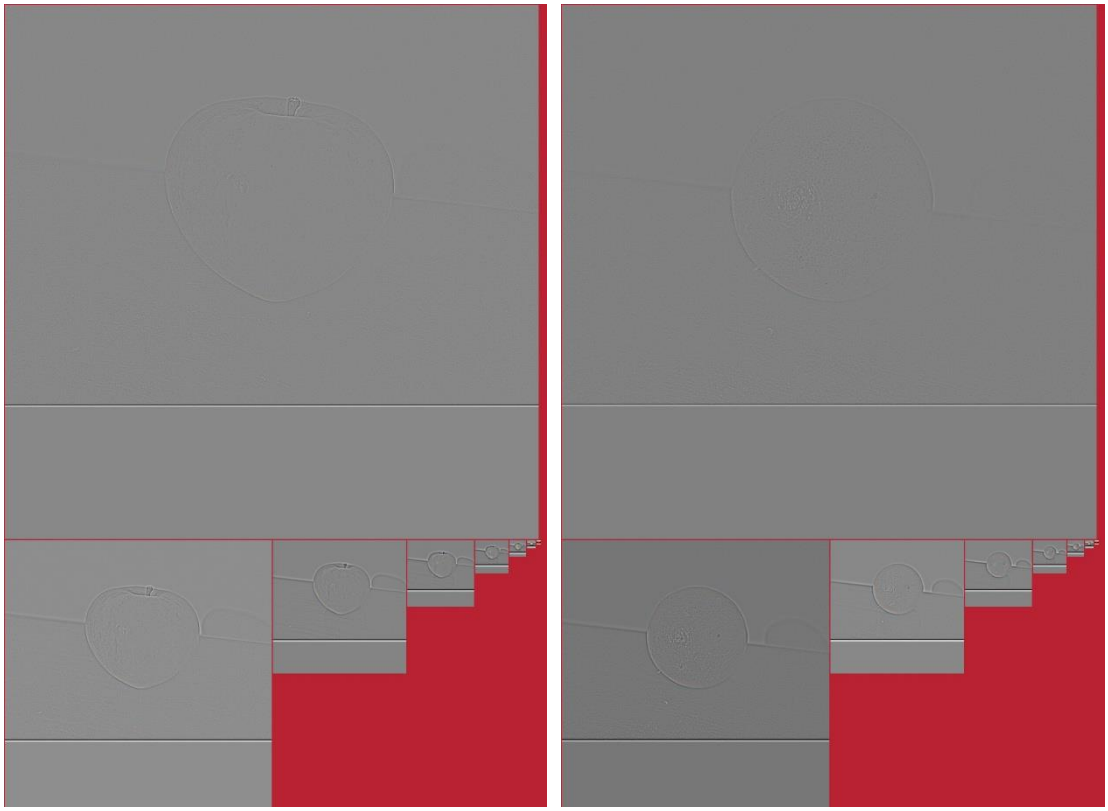
orange.jpg

- Output images

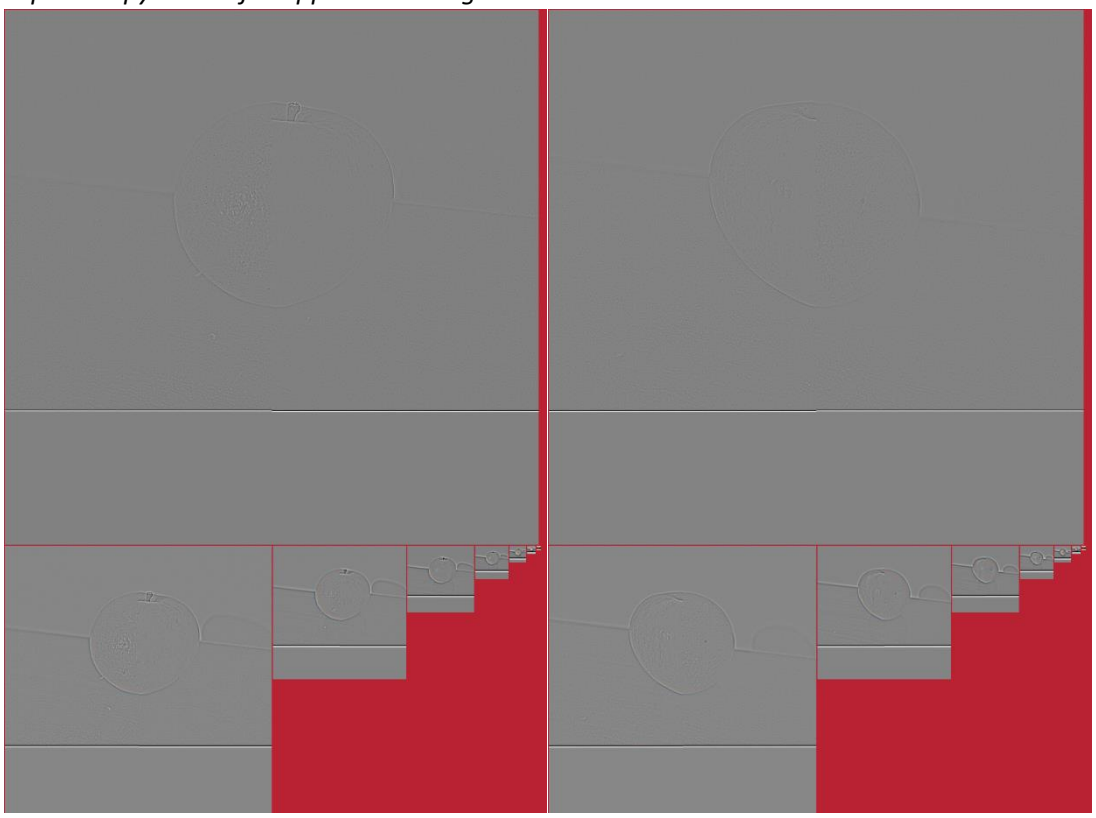




Gaussian Pyramids for apple and orange



Laplacian pyramids for apple and orange



Morphed Laplacian pyramids, first orange then apple, second apple then orange


Synthesised reconstructions of the morphed Laplacian pyramid images

- Key code snippets for creating the different images:

```
// Convert images to float

images[0].convertTo(images[0], CV_32FC3);
images[1].convertTo(images[1], CV_32FC3);

// Check images have the same size

if (images[0].rows != images[1].rows || images[0].cols != images[1].cols)
{
    throw "The two images don't have the same size -- Exiting the program";
}

for (int i = 0; i < 2; i++) {
    size_t width = isPowerOfTwo(images[i].cols);
    size_t height = isPowerOfTwo(images[i].rows);

    if (width != images[i].cols || height != images[i].rows)
    {
        throw "Both image's size must be a power of two. -- Exiting the program";
    }
}
```

This converts the images to a more easily readable numerical format, then checks that both images' sizes are the same and a power of two.

```
// Gaussian Pyramids
vector<Mat> imageGaussian[2];
createGaussianPyramid(images[0], imageGaussian[0], levels);
createGaussianPyramid(images[1], imageGaussian[1], levels);

Mat imageGaussian_vis[2];


for (int i = 0; i < 2; i++) {

    // Display & save

    imageGaussian_vis[i] = displayPyramid(imageGaussian[i]);

    imwrite("gaussian_pyramid_" + input_filename[i], imageGaussian_vis[i]);

    // Write the image
    if (!cv::imwrite("gaussian_pyramid_" + input_filename[i], imageGaussian_vis[i]))
```

This creates a gaussian pyramid using the input images and the number of levels specified, before displaying the pyramids to the user and saving them to a file.

```
//-----------------------------------------------------
void createGaussianPyramid(const Mat& anOriginalImage,
    vector<Mat>& aGaussianPyramid,
    size_t aNumberOfLevels)
    //-----------------------------------------------------
{
    // Make sure the pyramid is empty
    aGaussianPyramid.clear();

    // Add the original image
    Mat source = anOriginalImage;
    aGaussianPyramid.push_back(source);

    // Add the other sizes
    for (unsigned int i(0); i < aNumberOfLevels - 1; ++i)
    {
        // Compute the new image
        Mat dst;
        pyrDown(source, dst, Size(source.cols / 2, source.rows / 2));

        // Store the new image in the pyramid
        aGaussianPyramid.push_back(dst);

        // The new image becomes the new source
        source = dst;
    }
}
```

A more in depth look at the gaussian pyramid creating function.

```
// Laplacian Pyramids
vector<Mat> imageLaplacian[2];
createLaplacianPyramid(imageGaussian[0], imageLaplacian[0]);
createLaplacianPyramid(imageGaussian[1], imageLaplacian[1]);

Mat imageLaplacian_vis[2];


for (int i = 0; i < 2; i++) {

    // Display & save

    imageLaplacian_vis[i] = displayPyramid(imageLaplacian[i]);

    imwrite("laplacian_pyramid_" + input_filename[i], imageLaplacian_vis[i]);

    // Write the image
    if (!cv::imwrite("laplacian_pyramid_" + input_filename[i], imageLaplacian_vis[i]))
```

This creates a Laplacian pyramid using the gaussian pyramids created earlier, before displaying the new pyramids to the user and saving them to a file.

```
//-----------------------------------------------------------------
void createLaplacianPyramid(const vector<Mat>& aGaussianPyramid,
    vector<Mat>& aLaplacianPyramid)
    //-----------------------------------------------------------------
{
    // Create an empty pyramid
    aLaplacianPyramid.clear();

    // Last image ID to have been processed
    unsigned int last_id(aGaussianPyramid.size());

    // Add the original image
    Mat source = aGaussianPyramid[--last_id];
    aLaplacianPyramid.push_back(source);

    for (unsigned int i = 0; i < aGaussianPyramid.size() - 1; ++i)
    {
        Mat dst;
        pyrUp(source, dst, Size(source.cols * 2, source.rows * 2));

        // Get the new source from the Gaussian pyramid
        source = aGaussianPyramid[--last_id];

        // Store the new image in the pyramid
        aLaplacianPyramid.push_back(source - dst);
    }
}
```

A more in depth look at the Laplacian pyramid creating function.

```
// Swapping halves

for (size_t i = 0; i < imageLaplacian[0].size(); ++i)
{
    swapHalves(imageLaplacian[0][i], imageLaplacian[1][i]);
}

Mat morphLaplacian_vis[2];


for (int i = 0; i < 2; i++) {

    // Display & save
```

The first part to create the synthesised images is to swap the halves of the two images in their Laplacian pyramid format.

```
// Reconstruction

Mat reconstruction[2];

reconstruction[0] = reconstruct(imageLaplacian[0], 0);
reconstruction[1] = reconstruct(imageLaplacian[1], 0);

for (int i = 0; i < 2; i++) {
```

Then reconstruct the images to restore the colours.

```
Mat reconstruct(const vector<Mat>& aLaplacianPyramid, int aLevel)
{
    Mat reconstruction;

    if (aLaplacianPyramid.size())
    {
        for (int i = 0; i < aLaplacianPyramid.size() - aLevel; ++i)
        {
            if (i == 0)
            {
                reconstruction = aLaplacianPyramid[i];
            }
            else
            {
                Mat dst;
                pyrUp(reconstruction, dst, Size(reconstruction.cols * 2, reconstruction.rows * 2));

                reconstruction = dst + aLaplacianPyramid[i];
            }
        }
    }

    return reconstruction;
}
```

Deeper look at the reconstruction.

Lab 3

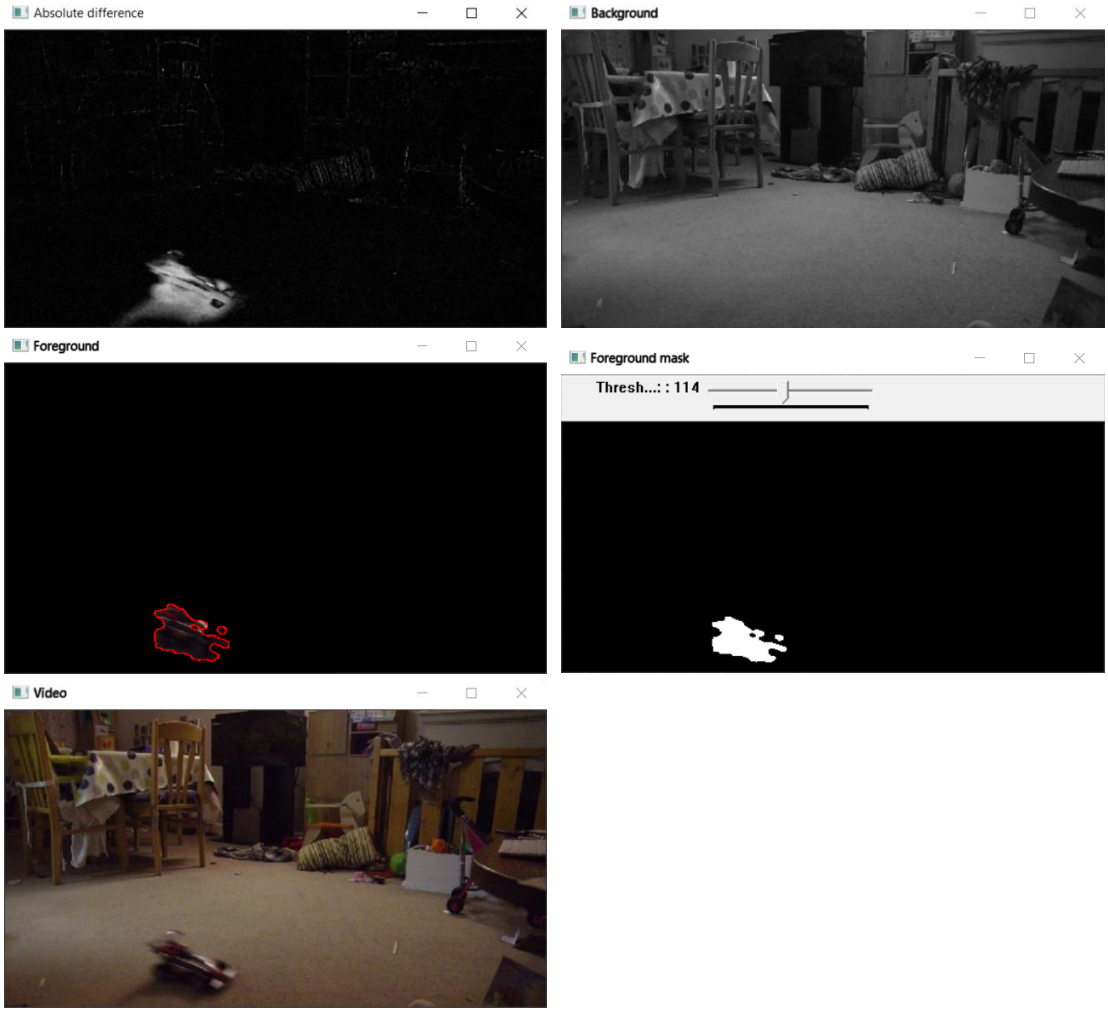
- Detect motion in a video by background subtraction.
- This lab creates multiple windows to view the different ongoing processes to single out the moving objects. A static background is chosen either by the user or by the program, and then it singles out a moving object by removing the background on the output window and showing only objects that are not part of the static background. If an object moves onto the static background area then the program should be able to tell it is not a part of the original background and thus display it to the user. There is also a threshold slider to adjust the tolerance between determining background and moving foreground objects.
- Command line arguments for the MotionDetection program were:
  <input_video_file>
- Input video stills

one_moving_object.avi

change_of_lighting_conditions.avi

two_moving_objects.avi

- Output images
  There are 5 output windows for each input video.
  one_moving_object:



The background was taken on the first frame of the video as it didn't have the car on screen which made a suitable mask to capture the car's motion.

change_of_lighting_conditions:

This video didn’t have a frame where the car was not in view, so the first frame was taken again and there is a visual error where the foreground mask picks up on the car’s original position as movement made since the car moved away. The threshold was lowered to help better capture the motion in the different lighting.

two_moving_objects:

This background was again taken on the first frame, as there were no frames without either car. The threshold captured the two cars decently, though a slightly lowered value (like one_moving_object) may have yielded better results.

- Key source code snapshots:

```
// Open the default camera (see the 0 below)
VideoCapture video_input(0);
if (argc > 1) { // Change webcam input to video file
    video_input = VideoCapture(argv[1]);
}

// Check VideoCapture documentation.
if (!video_input.isOpened())
    throw runtime_error("OpenCV found no webcam or video, the program will terminate");
```

This would use the computer's webcam as video input if no video file was specified or use the video file if one was found. If an invalid video was entered, it would throw an error.

```
// Grab a new frame
Mat frame;
video_input >> frame;

int FPS = video_input.get(CAP_PROP_FPS);

string outputFilename = "output_";
outputFilename += argv[1];
outputFilename += ".avi";
VideoWriter video_output(outputFilename, VideoWriter::fourcc('M', 'J', 'P', 'G'), FPS, Size(frame.cols, frame.rows));

// Set background
background = frame;
cv::cvtColor(background, background, COLOR_BGR2GRAY);
imshow("Background", background);
medianBlur(background, background, 3); //5
background.convertTo(background, CV_32F);
```

This would create a new blank frame to hold the first frame of the video/webcam view, then set this first frame as the background for masking. Also here is the video output initialised, to create a video from the webcam captured footage.

```
int key = -1;
while (key != 27 && key != 'q')
{
    // Grab a new frame
    video_input >> frame;

    // Make sure everything went well
    if (frame.empty())
    {
        if (argc > 1) { // Video
            video_input.set(CAP_PROP_POS_FRAMES, 0);
        }
        else { // Webcam
            video_input.release(); // We are now done with the camera, stop it
            throw runtime_error("OpenCV cannot grab a new frame from the camera, the program will terminate");
        }
    }
```

The following screenshots run inside this loop, which repeats indefinitely until the user manually quits by pressing ‘q’. This first part ensures that the webcam stops capturing after the user is finished, or that the video is constantly playing and loops if it has reached the end.

```
// There is frame
else
{
    // Display the image
    if (argc > 1) { // Video
        imshow("Video", frame);
    }
    else { // Webcam
        imshow("Webcam", frame);
    }

    // Save the frame in the output file
    video_output << frame;
```

If there is a next frame (i.e. the video or webcam stream continues), then that frame is displayed to the user and is saved to the video output.

```
Mat foreground_mask = getForegroundMask(background, frame, threshold_value);

// Apply the foreground mask
Mat clean;
frame.copyTo(clean, foreground_mask);

int thresh = 100;
Mat canny_output;
Canny(foreground_mask, canny_output, thresh, thresh * 2);
vector<vector<Point> > contours;
vector<Vec4i> hierarchy;
findContours(canny_output, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE);
for (size_t i = 0; i < contours.size(); i++)
{
    Scalar color = Scalar(0, 0, 255);
    drawContours(clean, contours, (int)i, color, 2, LINE_8, hierarchy, 0);
}

imshow("Foreground", clean);
```

This section creates (if not already made) and displays (updated version each new frame) the Foreground, Foreground Mask, and Absolute Difference windows. The findContours section is what is used to create the red outline around the moving objects in the foreground window.

```
Mat getForegroundMask(const Mat& aBackground, const Mat& aNewFrame, int aThreshold)
{
    // Convert in greyscale
    Mat grey_frame;
    cv::cvtColor(aNewFrame, grey_frame, COLOR_BGR2GRAY);

    // Blur the image a bit
    medianBlur(grey_frame, grey_frame, 5);

    // Convert to float32
    grey_frame.convertTo(grey_frame, CV_32F);

    // Compute the foreground as the absolute difference
    Mat foreground;
    foreground = aBackground - grey_frame;
    foreground = abs(foreground);

    // Normalise the foreground
    normalize(foreground, foreground, 0, 255, NORM_MINMAX, CV_8UC1);

    // Display the foreground
    imshow("Absolute difference", foreground);

    // Apply a threshold to generate the foreground mask
    Mat foreground_mask;
    threshold(foreground, foreground_mask, aThreshold, 255, THRESH_BINARY);

    // Use mathematical morphology to clean the binary image
    foreground_mask = cleanBinaryImage(foreground_mask, 10); //15

    // Display the foreground mask
    imshow("Foreground mask", foreground_mask);

    // Return the mask
    return foreground_mask;
}
```

A more detailed look at the section that makes the mask and absolute difference windows.

Lab 4

- The aim of this lab is to create a program that can turn multiple images into a single panoramic image.
- This lab starts off by showing a window of the two images that will be stitched together and compares them by matching features to see which parts of the image overlap to determine where the images are to be stitched at. Then one of the images is geometrically transformed to line up with the first image to create the panorama.
- Command line arguments for panorama.cxx:
  <left_input_image> <right_input_image> <output_image>
- Input images

left-1.png

right-1.png

left-2.png

right-2.png

- Output images

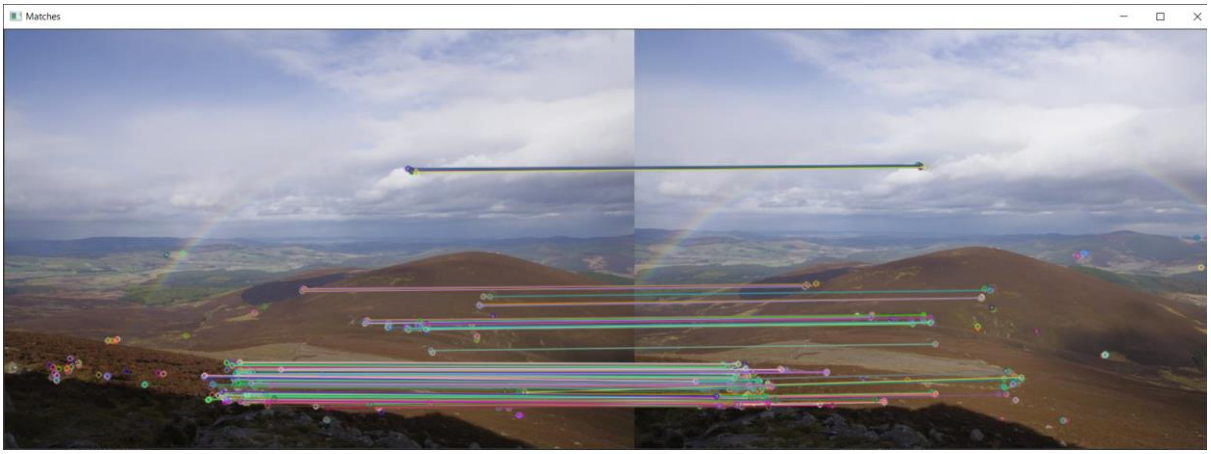
Matching of same points for first two images.

Transformed right image.

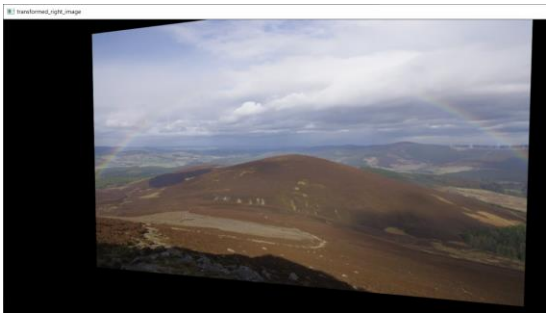
Left image.

Uncropped panorama image.

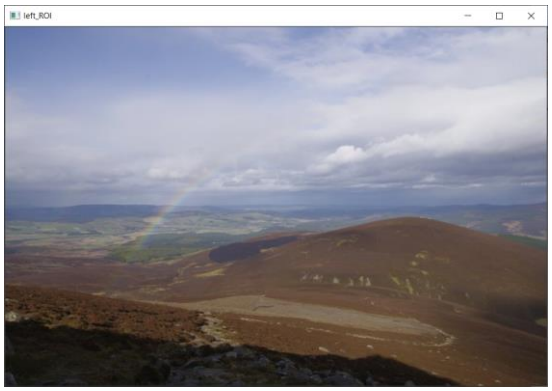
Final cropped panorama image.



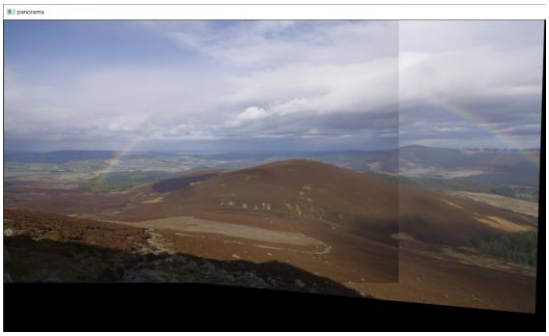
Matching of same points for second two images.



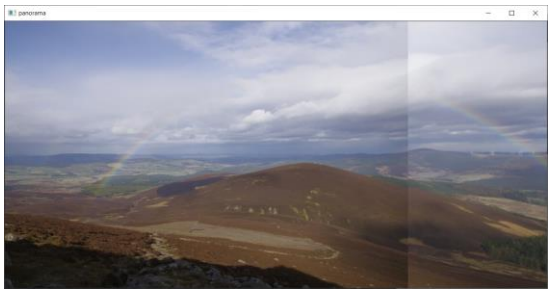
Transformed right image.



Left image.



Uncropped panorama image.



Final cropped panorama image.

- Key source code snapshots:

```
// Features
// Detect
Ptr<FeatureDetector> detector = ORB::create();

vector<KeyPoint> image_keypoints[2];
detector->detect(images[0], image_keypoints[0]);
detector->detect(images[1], image_keypoints[1]);

// Describe
Ptr<DescriptorExtractor> extractor = ORB::create();
Mat image_descriptors[2];
extractor->compute(images[0], image_keypoints[0], image_descriptors[0]);
extractor->compute(images[1], image_keypoints[1], image_descriptors[1]);
```

This section detects and “describes” matches between the two images. This stores the locations of all matched points.

```
// Match
BFMatcher matcher(NORM_HAMMING);
vector<DMatch> matches;
matcher.match(image_descriptors[0], image_descriptors[1], matches);

double min_distance = numeric_limits<double>::max();
double max_distance = -numeric_limits<double>::max();

for (int i = 0; i < matches.size(); i++)
{
    if (matches[i].distance < min_distance) {
        min_distance = matches[i].distance;
    }
    else if (matches[i].distance > max_distance) {
        max_distance = matches[i].distance;
    }
}

vector<DMatch> good_matches;
double thresh = min_distance + (max_distance - min_distance) / 2.0;
thresh = 44;

for (int i = 0; i < matches.size(); i++)
{
    if (matches[i].distance < thresh)
    {
        good_matches.push_back(matches[i]);
    }
}
```

This calculates the distance between the matched points, and determines whether or not a match is “good” based on how accurate each match is.

```
// Draw matches
Mat imageMatches;

drawMatches(images[0], image_keypoints[0], images[1], image_keypoints[1], good_matches, imageMatches);
imshow("Matches", imageMatches);
waitKey(0);
destroyAllWindows();
```

All good matches are highlighted on the image, and have a line drawn between them.

```
// Warping
std::vector<cv::Point2f> image_point_sets[2];

// Look at each good match
for (std::vector< cv::DMatch >::const_iterator ite = good_matches.begin();
    ite != good_matches.end(); ++ite) {
    // Get the keypoints from the good match
    cv::KeyPoint imPoint[2];
    imPoint[0] = KeyPoint(image_keypoints[0][ite->queryIdx]);
    imPoint[1] = KeyPoint(image_keypoints[1][ite->trainIdx]);

    // Add the corresponding 2D points
    image_point_sets[0].push_back(imPoint[0].pt);
    image_point_sets[1].push_back(imPoint[1].pt);
}

Mat homography_matrix(findHomography(image_point_sets[1], image_point_sets[0], RANSAC));

Mat panorama_image;
warpPerspective(images[1], panorama_image, homography_matrix, Size(images[0].cols + images[1].cols, images[0].rows + images[1].rows));
//*
imshow("transformed_right_image", panorama_image);
waitKey(0);
//*/

Mat left_ROI(panorama_image(Rect(0, 0, images[0].cols, images[0].rows)));
images[0].copyTo(left_ROI);
//*
imshow("left_ROI", left_ROI);
waitKey(0);
imshow("panorama", panorama_image);
waitKey(0);
//*/

panorama_image = autoCrop(panorama_image);
imshow("panorama", panorama_image);
waitKey(0);

// Write the image
if (!cv::imwrite(output_filename, panorama_image))
```

This whole section creates the panoramic image. First the right image is warped to match up with the left image, then the left image is added on top of this and the whole thing is cropped down to size. The images are displayed to the user at each stage, and the final image is saved to an output image file.

Conclusion

This assignment had a lot of long complex challenges, though I managed to grasp most concepts fairly quick and create a functional program to achieve many goals set to me. However, there were some areas which I failed to understand.

In the third lab, I couldn't figure out the contourArea in step 14, and how it was used to identify objects that are too small and should be ignored. Because of this, I also was unable to advance to the final steps of the lab assignment.

Also, in the fourth lab, I didn't leave myself enough time to complete the bonus objective to allow 3 images to be stitched together. Although in this case, I feel like I understood well enough how it could be achieved if I had the time to rewrite parts of the program to attempt it.