Image Panorama and Inpainting

ENEE 631

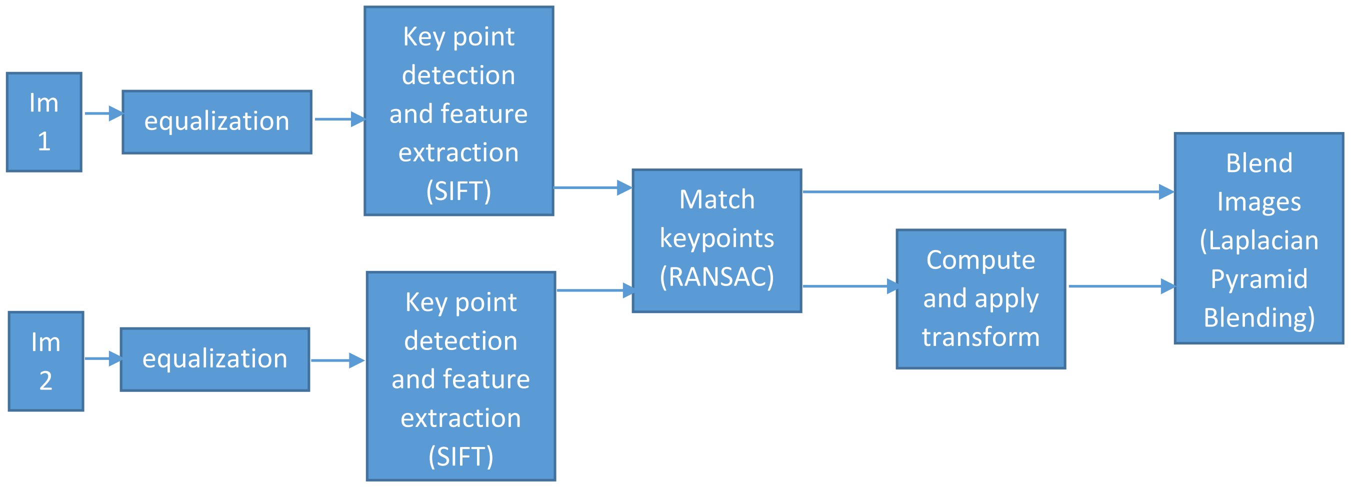
Sayantan Sarkar

Contents

1 Image Panorama

1.1 Introduction

Image stitching is the process of combining multiple overlapping images to produce a single image (panorama) that spans a wide view angle. Broadly the process of stitching of two images that has been implemented can be summarized by the following block diagram.


Im 1
equalization
Key point detection and feature extraction (SIFT)
Im 2
equalization
Key point detection and feature extraction (SIFT)
Match keypoints (RANSAC)
Compute and apply transform
Blend Images (Laplacian Pyramid Blending)

Given two images we equalize them both if necessary. Then we apply a key point detection and feature extraction algorithm like SIFT or SURF on both images. With the keypoints we get we perform a matching using RANSAC. Using the best match we get out of RANSAC, the best transformation is computed. Finally the two images are blended together using Laplacian Pyramid Blending.

I have implemented the pipeline (and each of its blocks like equalization, SIFT, RANSAC, Blending etc) in MATLAB. I have used Mr. Ganesh Kumar's code (with whom I had worked during my summer internship) to serve as a guide line for the MATLAB implementation. In addition I started implementing it in OpenCV but could not finish. Also I tried MATLAB's inbuilt functions to implement stitching [4].

Let us first discuss these blocks individually, then we will discuss the pipeline as a whole later.

1.2 Brief description of blocks

1.2.1 Equalization

Since we deal with RGB colour images, simple equalization does not work. Also equalizing each channel separately is not a good idea as RGB channels do not contain the image structure individually. Instead it is better to convert to HSV colour space and then equalize only the V channel. Finally the image is converted back to RGB.

Sometimes equalization gives better results in stitching but at other times, it is not useful. The user can choose to equalize the image before processing or just skip it.

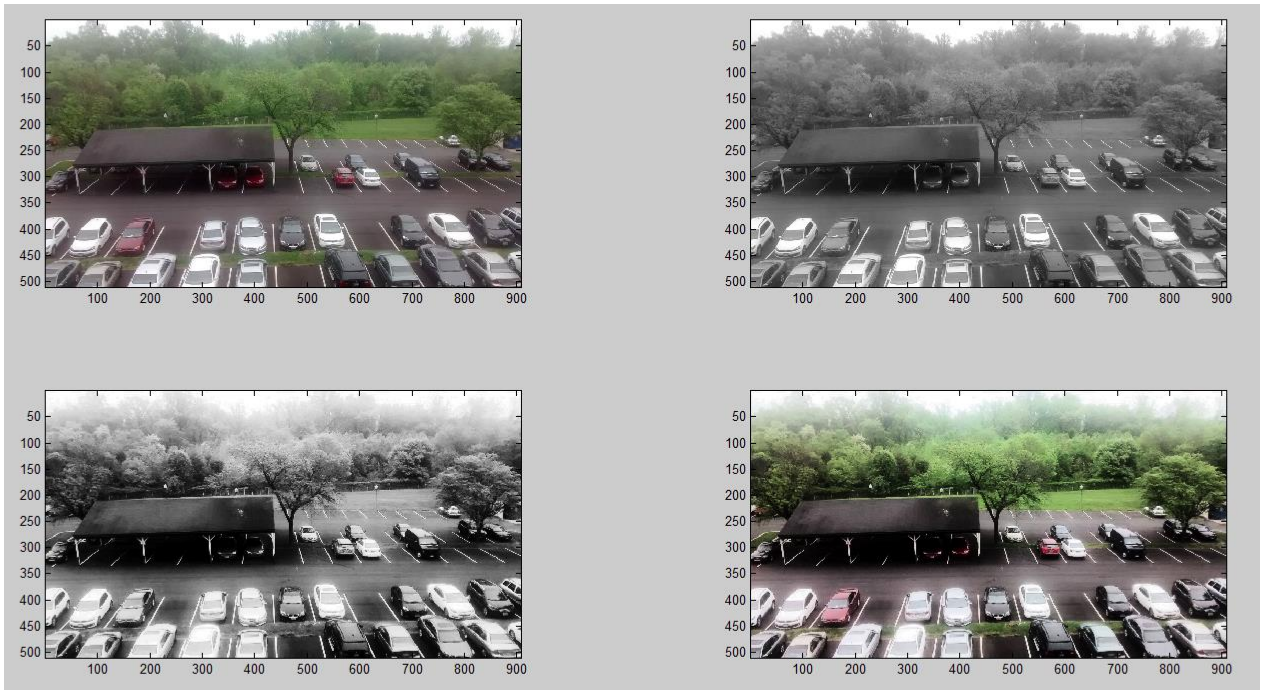


Fig: Top Left: Original image. Top Right: V channel of HSV representation. Bottom Left: V channel equalized. Bottom Right: Equalized colour image.

1.2.2 SIFT for keypoint detection and feature extraction

Scale-Invariant Feature Transform (SIFT) is a very popular computer vision algorithm to detect and describe local keypoints in images. SIFT features can be described by the following block diagram.

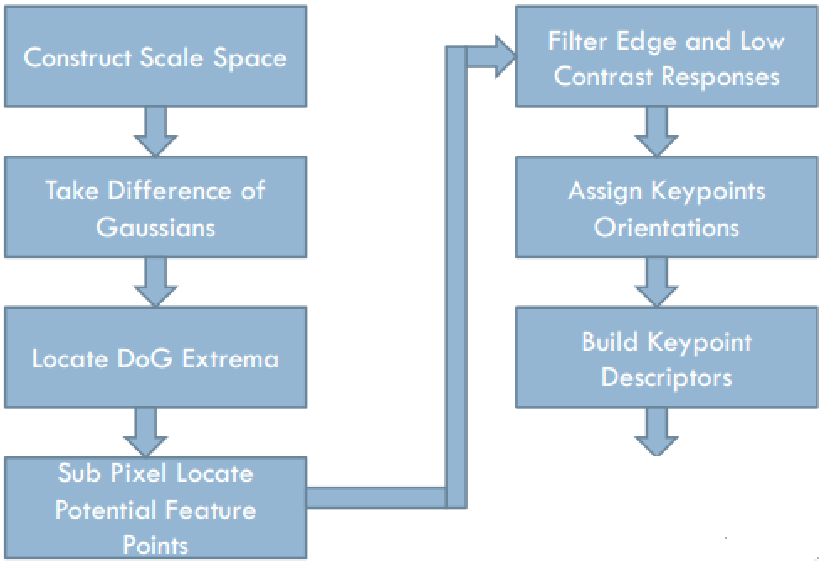

Construct Scale Space
Take Difference of Gaussians
Locate DoG Extrema
Sub Pixel Locate Potential Feature Points
Filter Edge and Low Contrast Responses
Assign Keypoints Orientations
Build Keypoint Descriptors

Fig: SIFT block diagram

The first step of SIFT is to create a Laplacian pyramid at different scale spaces. What it means is that we find the Laplacian of the image for different values and then downsample the image and repeat the same operations. Each downsampled version of the image is called an octave and for each different value of σ for which we calculate the laplacian is called a level. Usually we approximate the laplacian operation by performing difference of Gaussian filtered images which are filtered with increasing values of σ.

After we have built this scale space, we find extremas in it (maxima or minima) by comparing it each pixel to its neighbours in its level and the levels above and below it. After applying a sub pixel correction and filtering out some of the keypoints based on some thresholds, we calculate the SIFT features of the remaining keypoints. The features are basically histograms of edge orientations weighted by edge intensities and a Gaussian window.

1.2.3 RANSAC for best match between keypoints

Random Sampling Consensus is an iterative non deterministic algorithm to estimate parameters in a mathematical model from a set of observed data which may contain outliers. In our setting we need at least 4 points to fit (using least squares solution) the model using the projective geometry equation.

The RANSAC algorithm works as follows:

1: Select randomly the minimum number of points required to determine the model parameters.
2: Solve for the parameters of the model.
3: Determine how many points from the set of all points fit with a predefined tolerance.
4: If the fraction of the number of inliers over the total number points in the set exceeds a predefined threshold, re-estimate the model parameters using all the identified inliers and terminate.
5: Otherwise, repeat steps 1 through 4 (maximum of N times).

1.2.4 Image blending

For merging first we transform one of the images (based on the model generated by RANSAC) and find its corner points. These are useful to define the dimensions of the stitched image. Then we stitch each channel individually. For stitching I used Laplacian Pyramid Blending technique.

In this technique Laplacian (approximated by difference of Gaussian) of the image at different scales are taken much like in SIFT features.

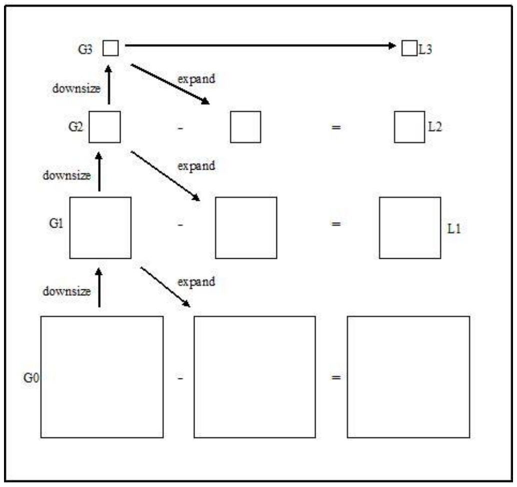


Fig: Image showing Difference of Gaussian Pyramid that approximates Laplacian operator

After generating Laplacian pyramids for the overlap images, we combine the two images in different Laplacian levels by combining partial images from each of them.

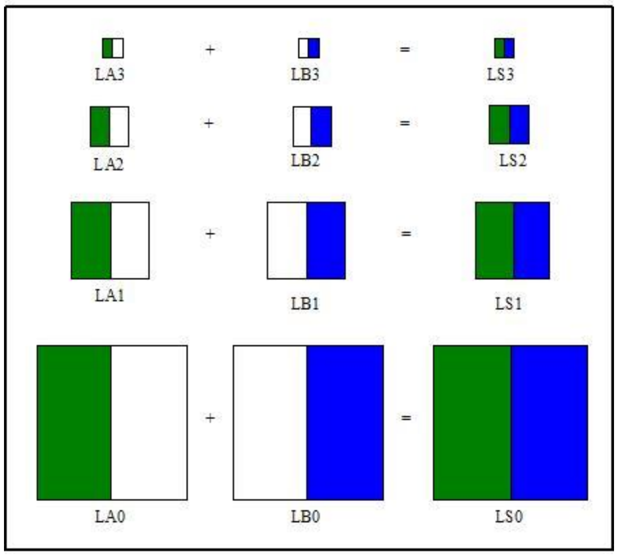


Fig: Combining images at different levels with appropriate masks

Finally we expand the pyramid and reconstruct the blended image as shown in the figure below.

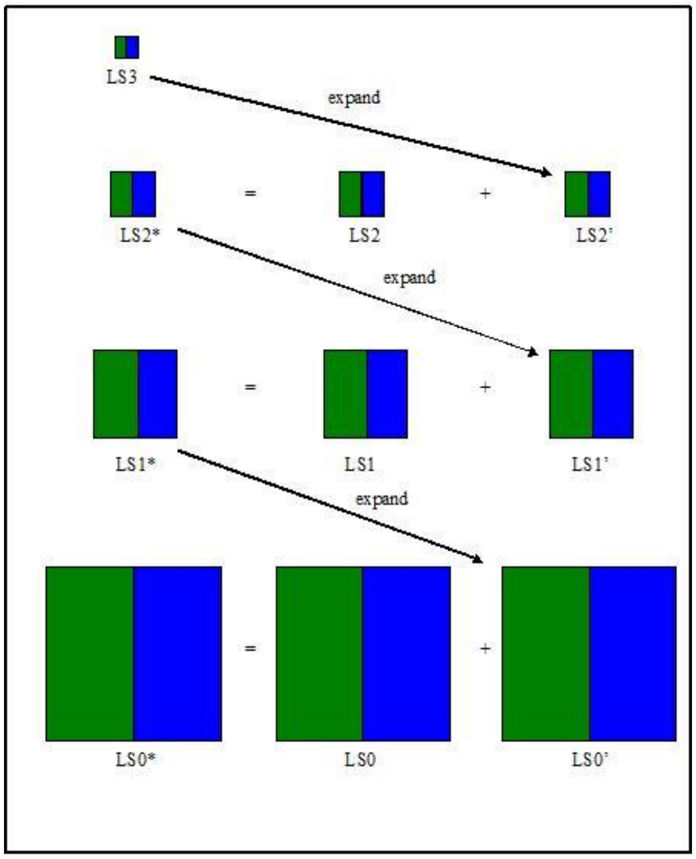


Fig: Reconstruction of pyramids

1.3 Implementation details and results

The image stitching pipeline has been implemented in MATLAB. It has two modes. The first mode is basic 2 image stitching, while the second mode extracts frames form a video and stitches them together. The user can specify how many frames should be extracted from the video.

Also I have implemented a similar pipeline based on MATLAB's inbuilt functions. It is faster and performs better than my code. One probable reason for its speed is that MATLAB uses SURF features which are faster to compute than SIFT descriptors. But I think my blending algorithm works better as I do not see visible seam lines.

The image below shows two images, their key points and the panorama generated by my implementation.

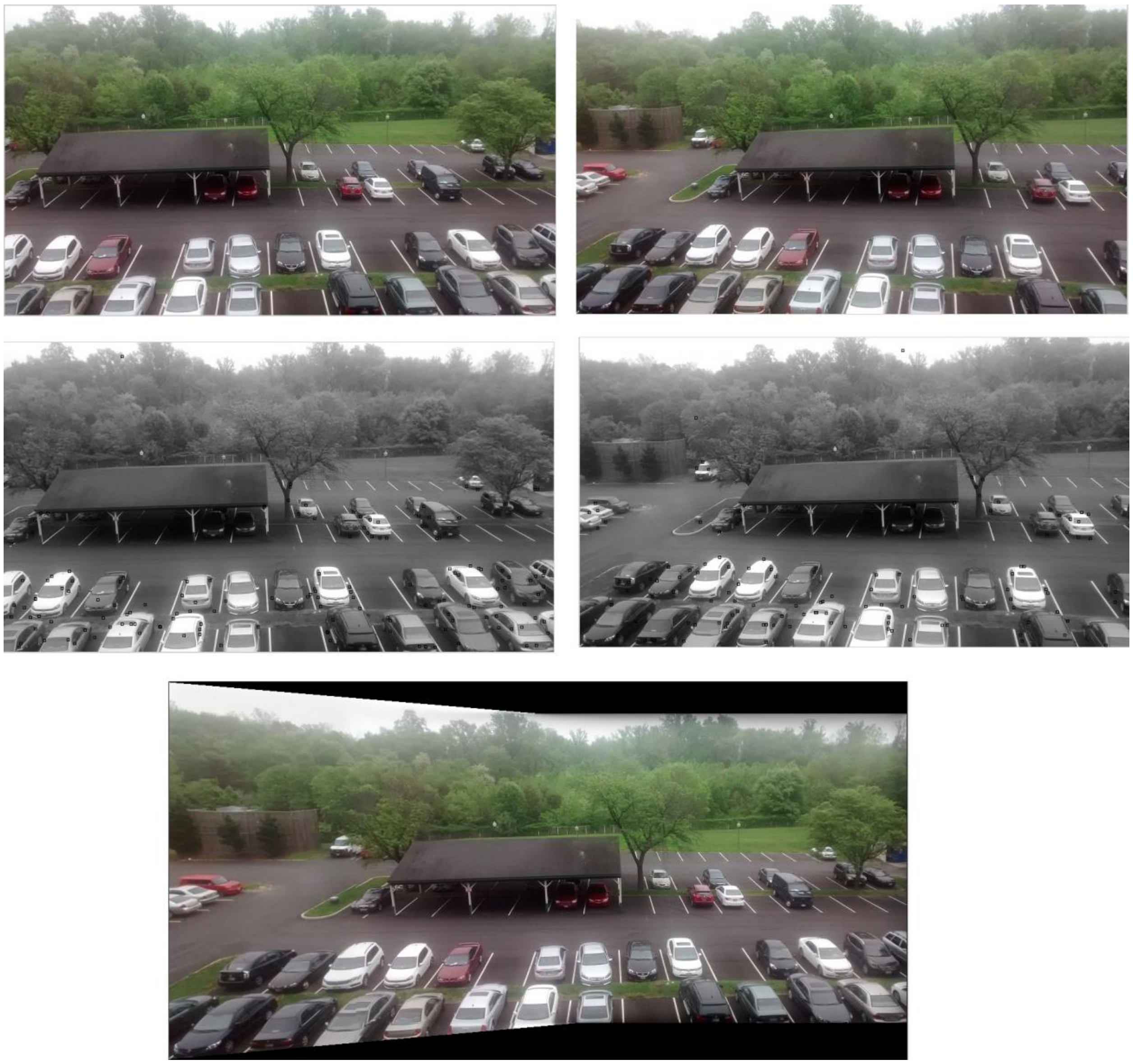


Fig: Panorama example using my algorithm

The following image shows the same two images, their SURF detected keypoints and the stitched image using MATLAB’s inbuilt functions

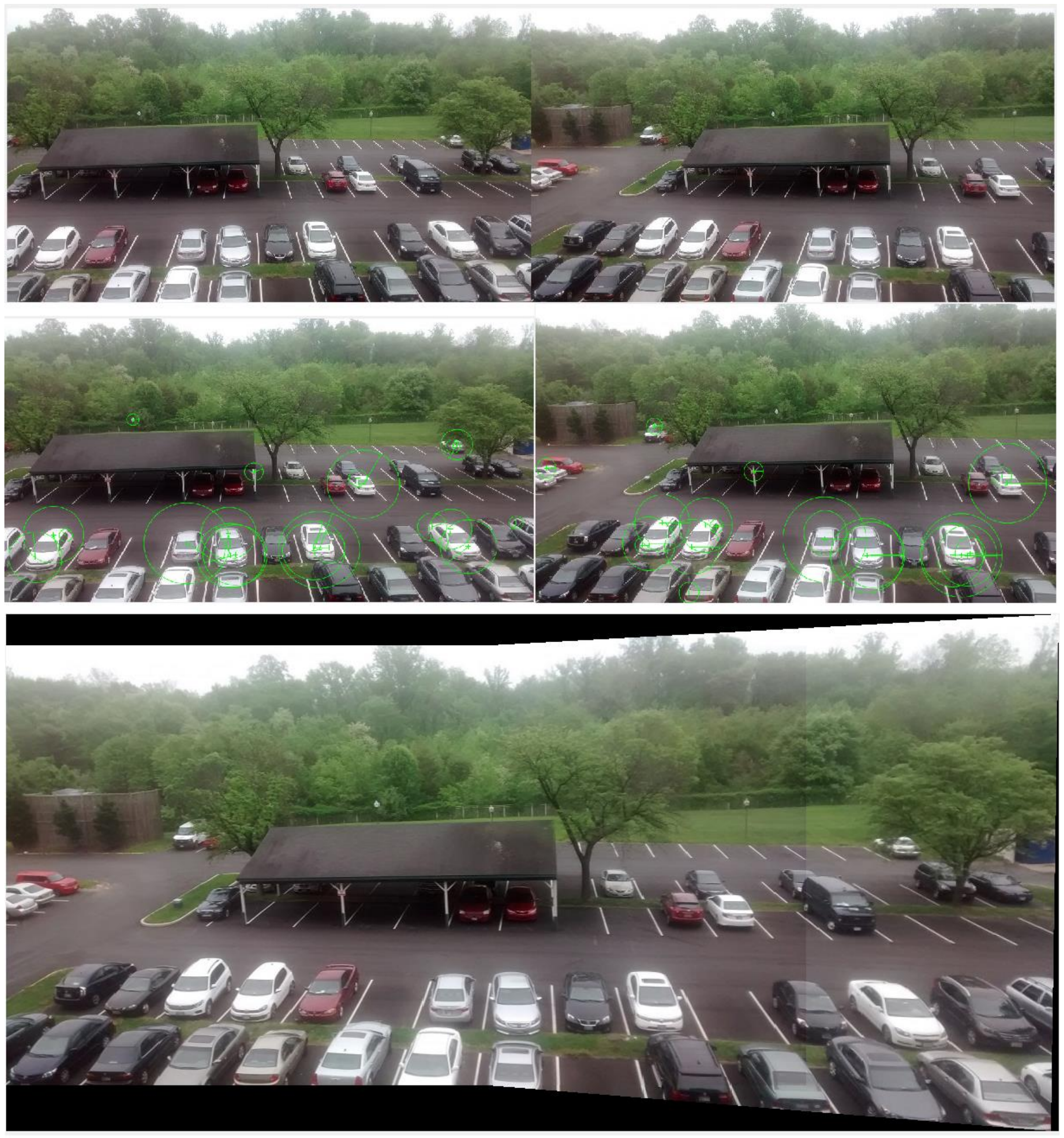


Fig: Panorama example using MATLAB’s inbuilt functions

Comparing the two I see that there is a visible seam in the second panorama, but the second one is much faster probably because it uses SURF features.

Here is another example showing the two original images, their keypoints and the panorama stitched using my algorithm

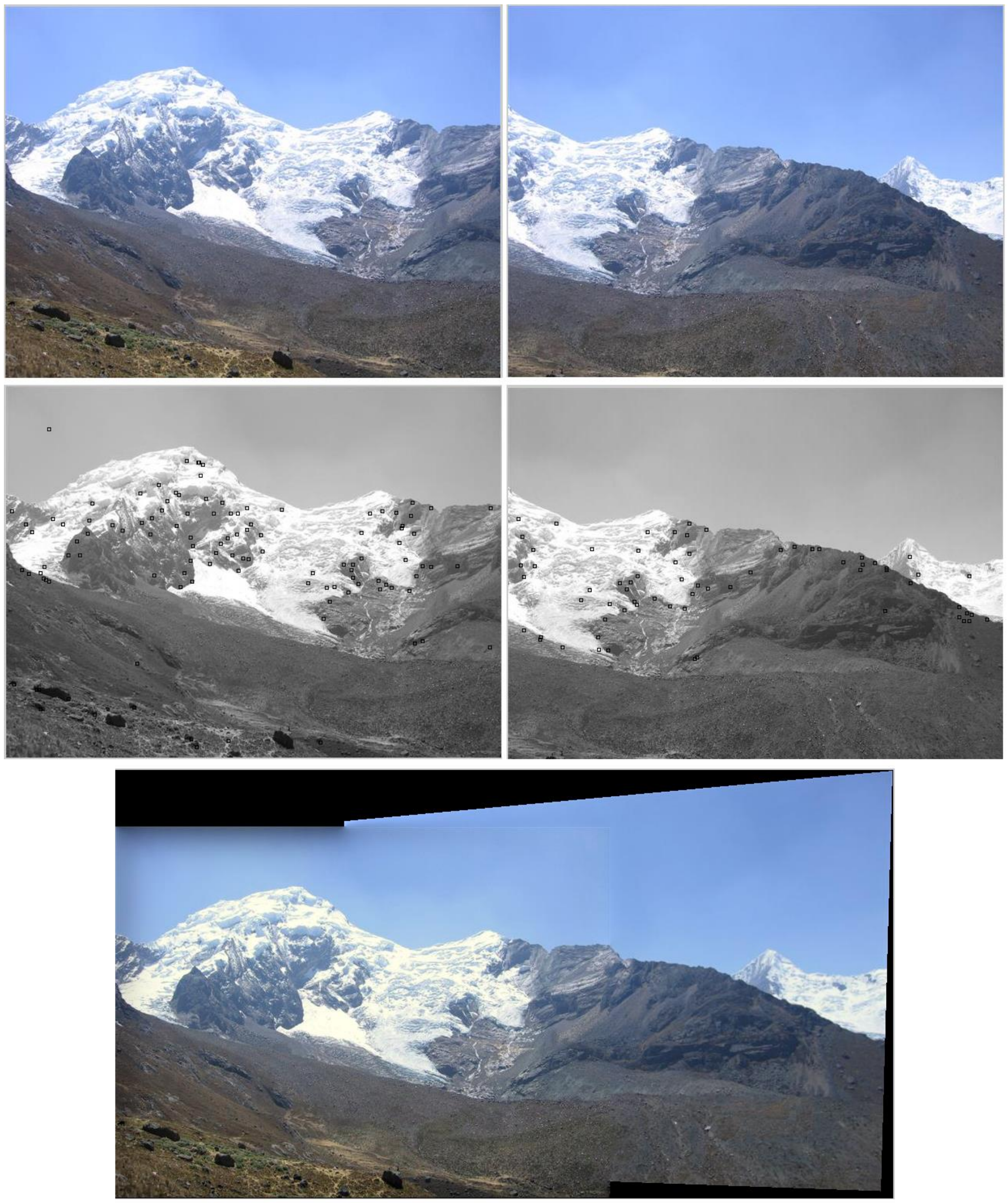


Fig: Panorama example using my algorithm

Here is the output from the MATLAB inbuilt function using code.

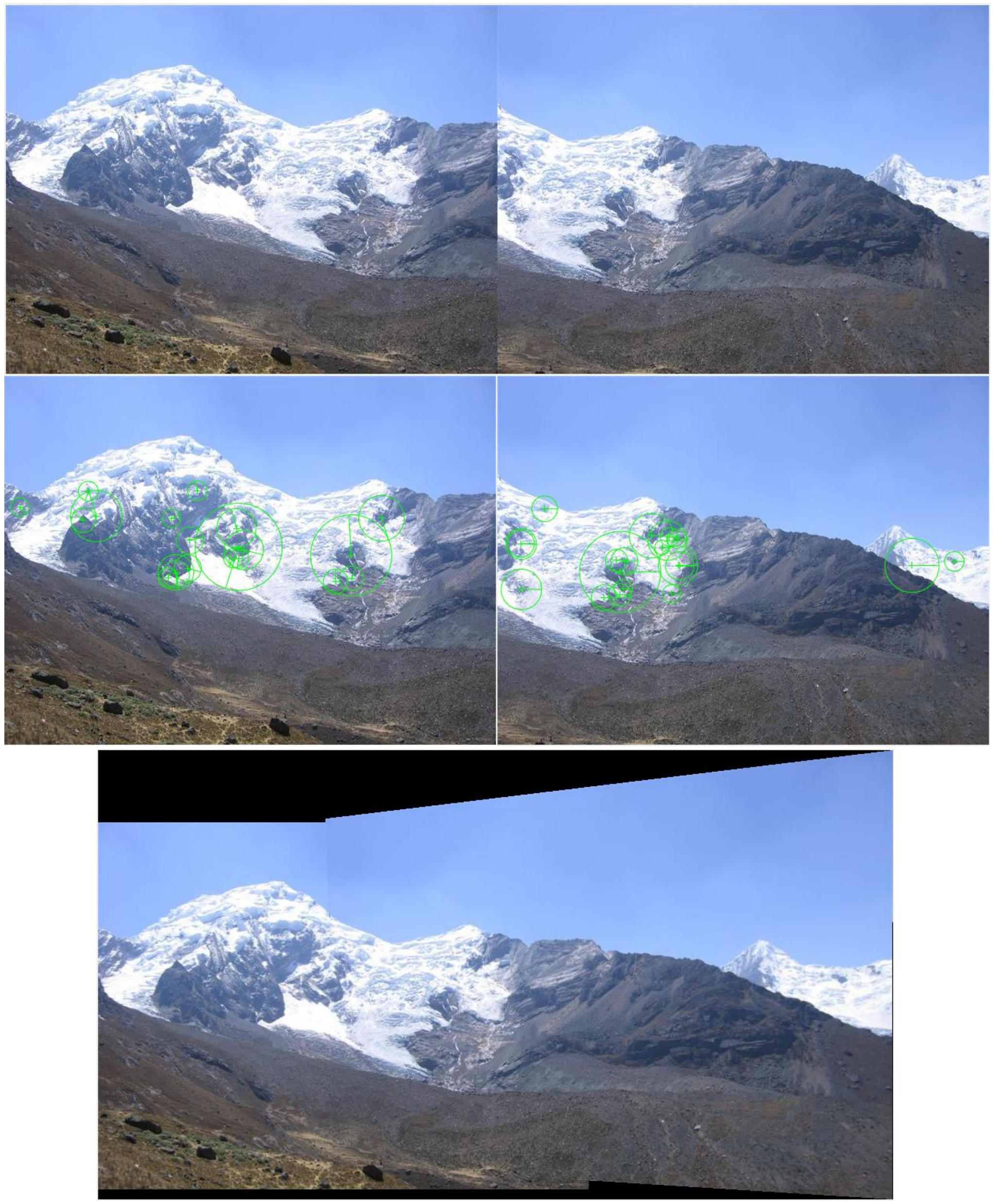


Fig: Panorama example using MATLAB’s inbuilt functions

Here are some more examples

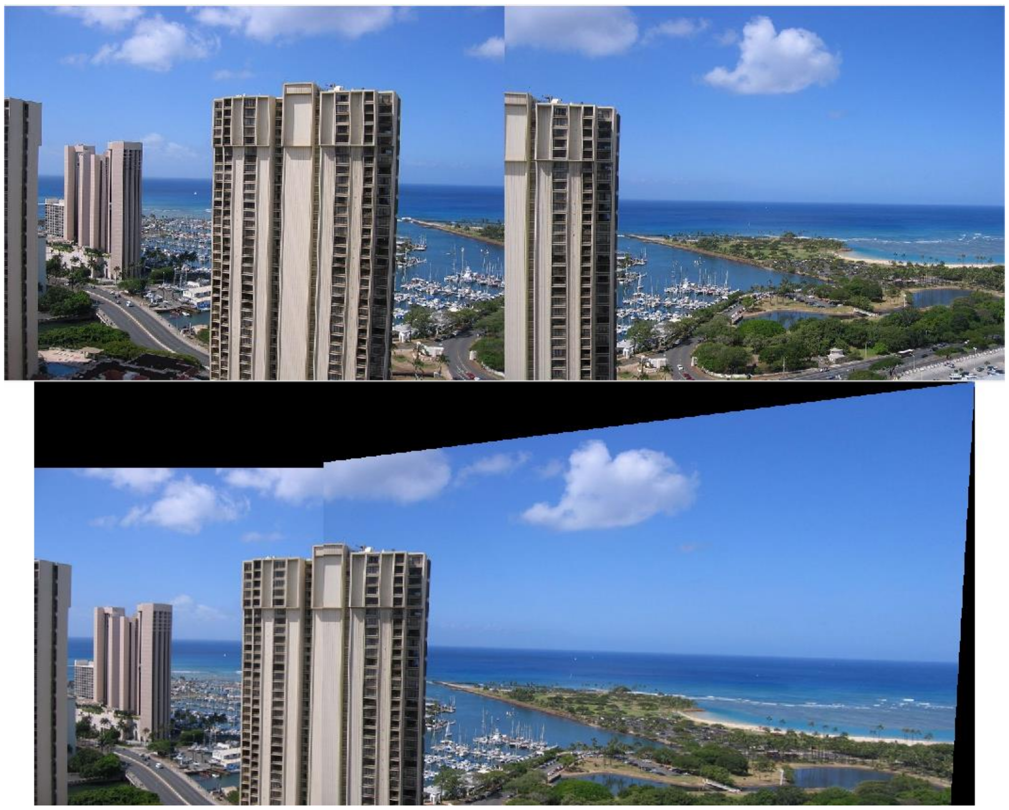


Fig: Panorama example using MATLAB’s inbuilt functions

Finally here is an example of a panorama constructed from a video. It shows 6 frames from the video that were used for the fusion and the final output image

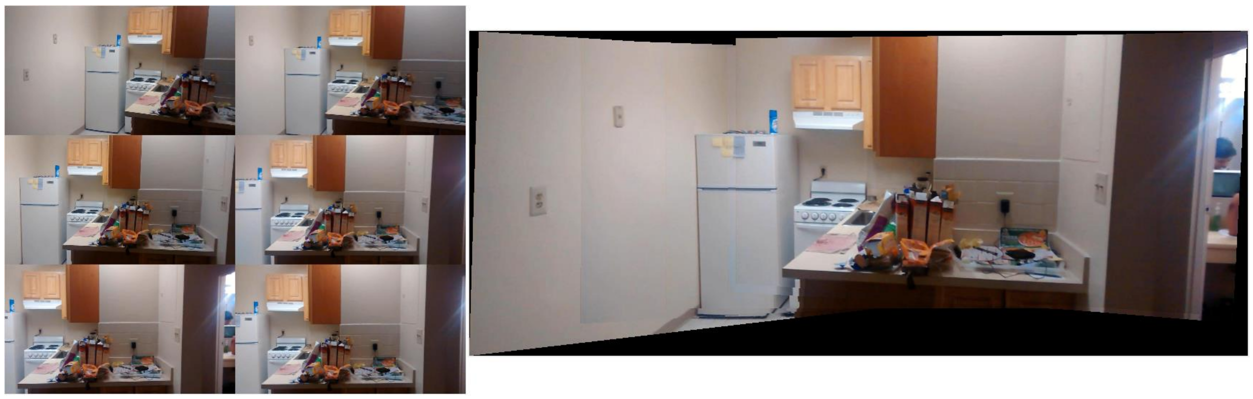


Fig: Panorama from video

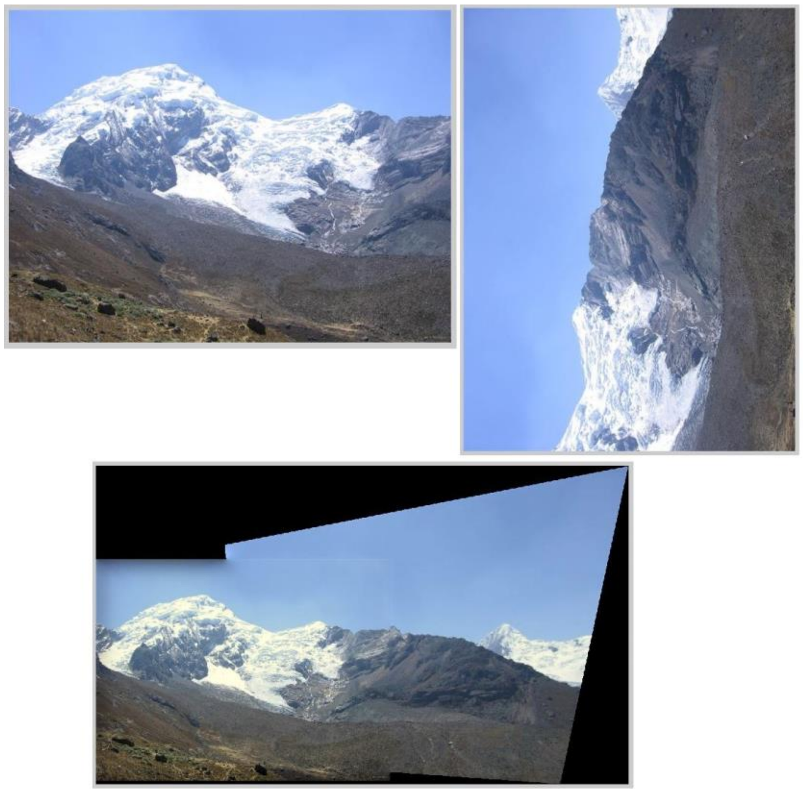


Fig: Stitching of rotated images

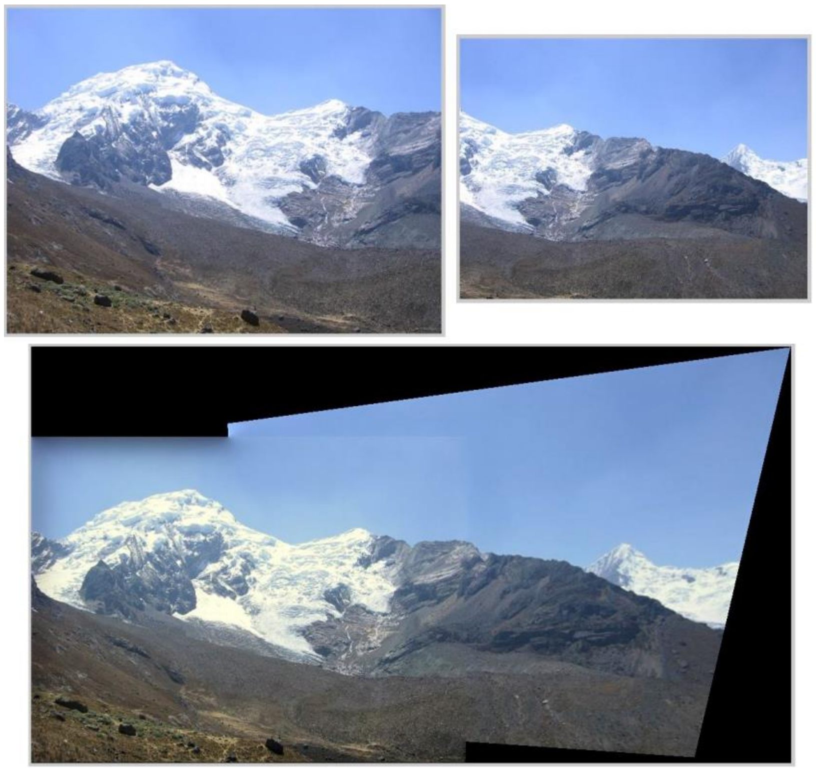


Fig: Stitching of images of different scales

2 Inpainting

2.1 Introduction

Image Inpainting is the process of reconstructing lost or deteriorated parts of images. Of course here we deal with automatic computerized digital inpainting rather than inpainting by hand. Inpainting has the following uses:

1. Restore old or damaged images
2. Restore pixels lost during transmission
3. Remove an unwanted object from the image

Formally the inpainting problem can be defined thus: Given an image I, and a mask Ω that defines the region to be inpainted, we wish to automatically interpolate pixel values for the region defined by Ω based on the source region $I - \Omega$. The resulting image should look reasonable to human eyes.

Notice that the inpainting problem requires a predefined mask Ω. When developing inpainting systems, we need to provide different ways for choosing the mask Ω. In this report we shall three different techniques for choosing Ω.

After defining a few terms, I shall describe the underlying inpainting algorithm that I have implemented, then describe some variations of it to handle different usecases. Finally I shall describe its implementation in openCV on windows and Android.

2.2 Definitions

First let us define a few terms.

Isophotes are linear structures in images which are specially useful for inpainting as they provide the basic skeleton if images. Isophotes get high priority and are ideal candidates for propogaion into the unknown mask.

The Image is denoted by I which is basically an array of pixel values (with 3 channels if it is a colour image). The unknown region or mask is denoted by Ω. The Source is $\Phi = \mathrm{I} - \Omega$. We use the pixels in the source to fill in the mask Ω. The fill front is the contour or border of the mask Ω and is denoted by $\delta\Omega$.

Patches are square blocks of pixels centered at a point p and denoted by ψ_p

2.3 Underlying Inpainting Algorithm

I have chosen to implement Criminisi, Perez and Toyoma’s inpainting algorithm as described in their paper “Object Removal by Exemplar-Based Inpainting”. It is an exemplar based technique, which means it fills the mask Ω with patches drawn from the already filled source region $\Phi = \mathrm{I} - \Omega$. Exemplar based techniques tend to give crisper results compared to diffusion based techniques, which tend to blur the colours and edges.

We start with a mask Ω. At every iteration we identify the border of the remaining unfilled region (fill front) and for each pixel in the border we calculate two terms (the data term $D(p)$ and the confidence term $C(p)$) and multiply them to get the priority for the patch. The two terms are defined as:

$$D(p) = \frac{|\nabla I_p^{\perp}.n_p|}{\alpha}, C(p) = \frac{\sum_{q\epsilon\psi_p\cap\bar{\Omega}} C(q)}{|\psi_p|}$$

Where ψ_p is the image patch centered at a point p in the fill front, $|\psi_p|$ is the area of the patch, n_p is the normal to the fill front at the point p, $\nabla I_p^{\perp}$ is the isophote direction and α is a normalization constant (set to 255 for 8 bit grayscale images).

Once we have found the patch with the highest priority, we find the best match among patches in the source region and fill the high priority patch. Then the confidence values of the filled patches is set equal to the patch that was used to fill it. Finally we update the fill front and continue the process. The image below from the paper of Criminisi explains the process for one patch.

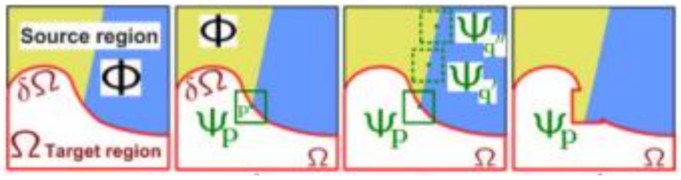


Fig: Patch based region filling

The key idea in the algorithm is to have two terms (confidence and data) to determine which patch should be filled next in the fill front. The data term finds the gradient of the image at that point and multiplies it with the normal of the fill front at that point. Therefore if an edge in the image is perpendicular to the fill front that gets a high priority as edges are important defining characteristics and it is desirable that they be propagated faithfully.

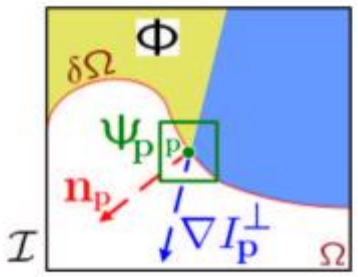
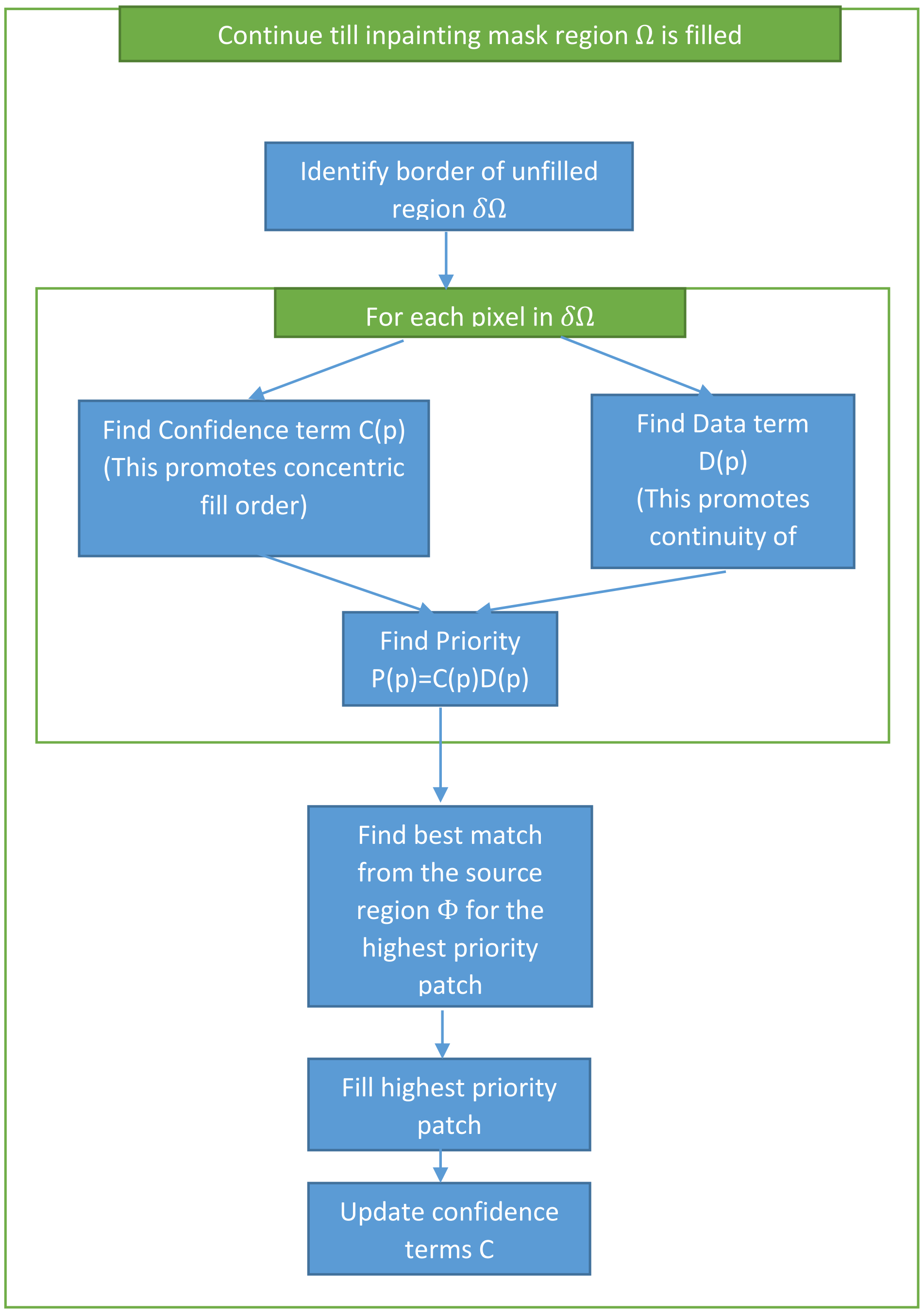


Fig: Isophote propagation using Data term in priority calculation

The confidence of the source region is set to 1 when the algorithm starts, but then as patches fill up the mask region, the confidence term decays. Thus confidence of regions that were filled first is higher compared to those filled later. This enforces a desirable concentric fill order.

Together the data term tends to push isophotes rapidly into the mask region while the confidence term tends to suppress this kind of propagation, instead preferring a concentric fill order.

A block diagram for the algorithm is shown below.


Continue till inpainting mask region Ω is filled
Identify border of unfilled region δΩ
For each pixel in δΩ
Find Confidence term C(p) (This promotes concentric fill order)
Find Data term D(p) (This promotes continuity of
Find Priority P(p)=C(p)D(p)
Find best match from the source region Φ for the highest priority patch
Fill highest priority patch
Update confidence terms C
Fig: Block diagram of inpainting algorithm

2.4 Modifications

Building up on the underlying algorithm described in the last section, I added a few modifications to it to make it more user friendly. Since it is a very computationally intensive algorithm, it takes quite long to run. Therefore in the search for the best match I did not scan every possible candidate instead opting to scan candidates by skipping a few pixels. If a match is found that is better than the best match so far then I do a finer search in that region.

Another possible modification that can be made to speed things up is to search for candidates only in a region centered at where the last match was found and if a very good match is not found only then search in the entire image. I have not implemented and tested this idea though.

When finding best match patches often we find patches that have same distance from the patch we are interested in filling. In that case I have chosen the patches that are nearer to the patch being filled. This gives better results. Also in finding the distance of patches I use simple Euclidean distance between the channels (RGB or YUV) and add them with equal weights for RGB and more weight for Y channel for YUV case. Also I ensure that the best patch should have low overall distance as well as its distance in each channel should be below a certain threshold.

2.5 Usage modes

I have implemented this inpainting algorithm in C++ using OpenCV libraries using the Microsoft Visual Studio 2010 IDE in Windows. The program when run displays the image under consideration. Now we three modes to operate on it.

2.5.1 Mode 1: Polygon Mask

The user can left click multiple points on the image in a clockwise or anticlockwise order to define vertices of a polygon. Once the user is done he/she can press any button and a new image is shown with a black polygon defined by the click points superimposed on the image. The polygon defines the mask for the fill region and the inpainting algorithm is called on this image with the polygon mask. This mode is useful to define arbitrary regions in the image and the user can handpick any region of arbitrary shape using this. This can be useful for masking out certain objects so that they can be inpainted out.

2.5.2 Mode 2: Thresholding in Polygon Mask

In this mode the user can left click to define a polygonal region of interest where the object to be inpainted lies. Then after the polygon is defined the user right clicks on a point that is inside the region to be inpainted and presses any button to continue. Since the point that was right clicked is inside the region to be inpainted, a thresholding based on colour intensities is done to get a mask. The mask is dilated to remove discontinuities in the mask region and the inpainting algorithm is called. This mode works well when the region to be inpainted has almost constant colour so that the mask can be inferred through simple thresholding. For example this is useful for removing timestamps from images.

2.5.3 Mode 3. Global inpainting

In this mode the user control + left clicks on a pixel and presses any button to continue. All pixels that are of similar colour to the clicked pixel are considered part of the mask and attempted to be replaced by inpainting. This is useful if the inpainting region has a specific colour and it is spread all over the region, hence making defining the mask difficult.

2.6 Results

I will be showing the results for a few illustrative examples and the captions below the images describe the feature of the implemented algorithm that is visible in that image.

2.6.1 Mode 1

In the following set of images, Mode 1 is in use (where a polygonal region is blacked out). The examples show the original image, the image to be inpainted (with the black polygonal mask) and the final output.

2.6.1.1 Isophote propagation

The next three images show how isophotes are propagated satisfactorily inside the mask, due to the data term in the priority function.

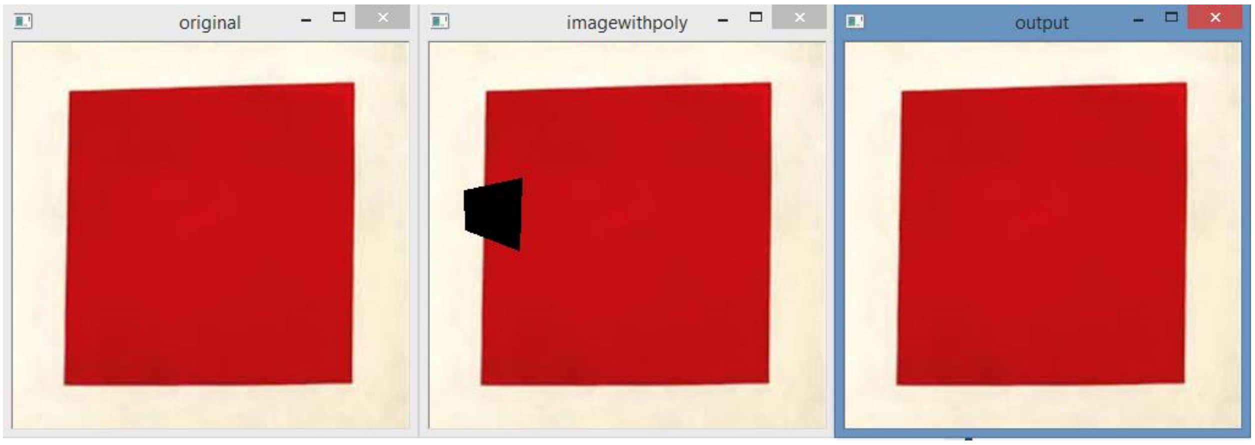


Fig: Simple edge completion

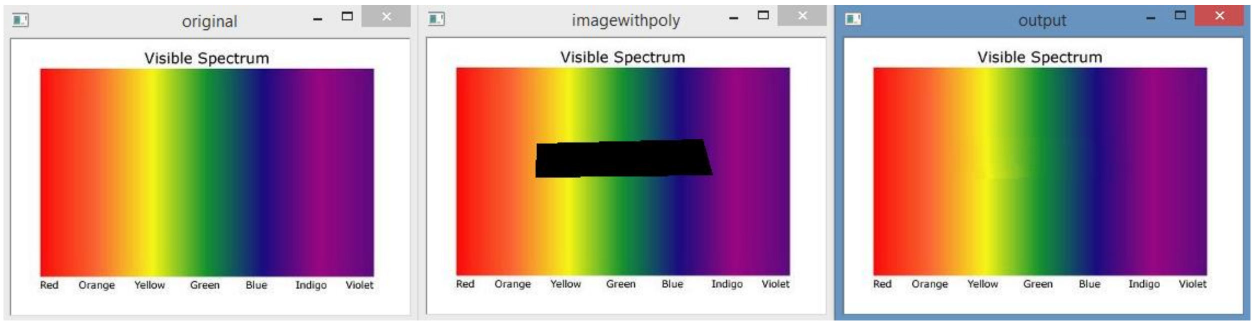


Fig: Multiple edge completion

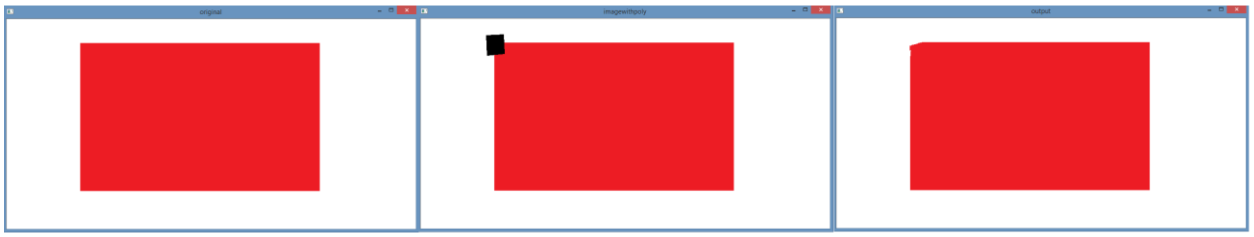


Fig: Corner Completion

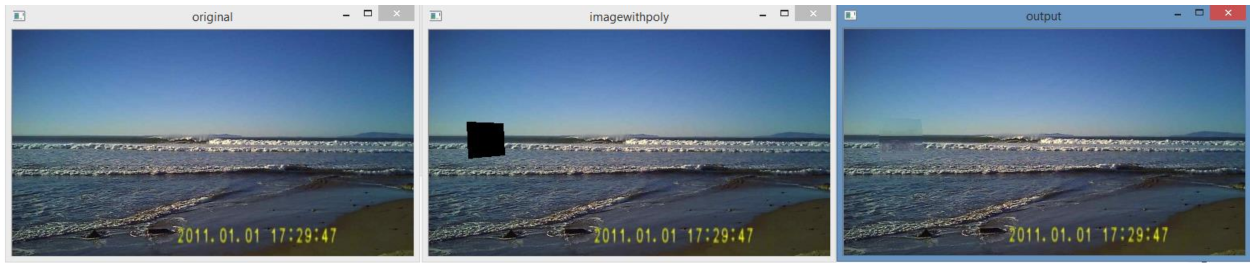


Fig: Edge continuity in a natural image

2.6.1.2 Object removal

The next few image shows examples of object removal

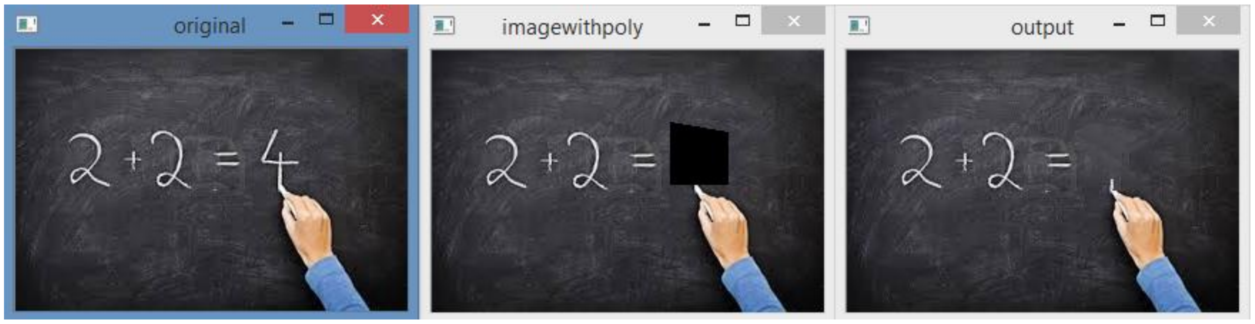


Fig: Simple object removal

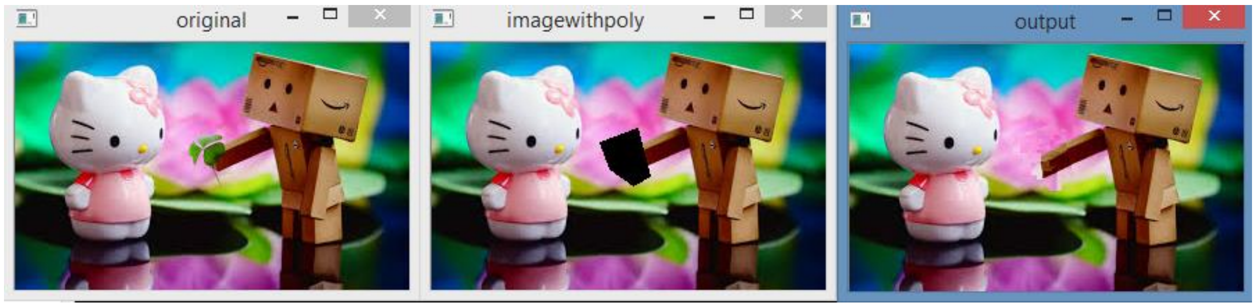


Fig: Object removal with edge continuity (Notice the arms of the block man is continued into the infilled region)

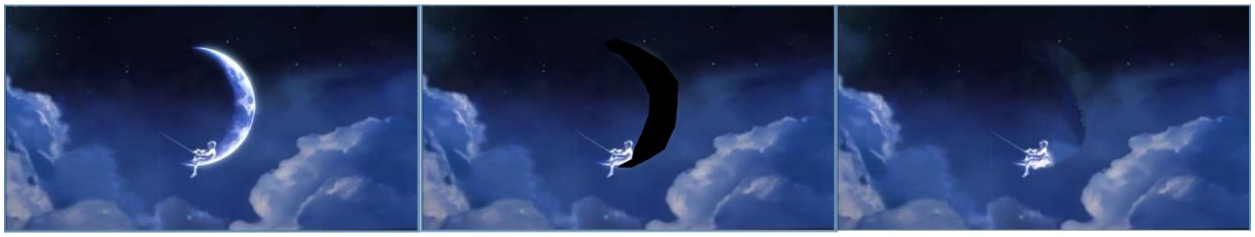


Fig: Removing an arbitrarily shaped object

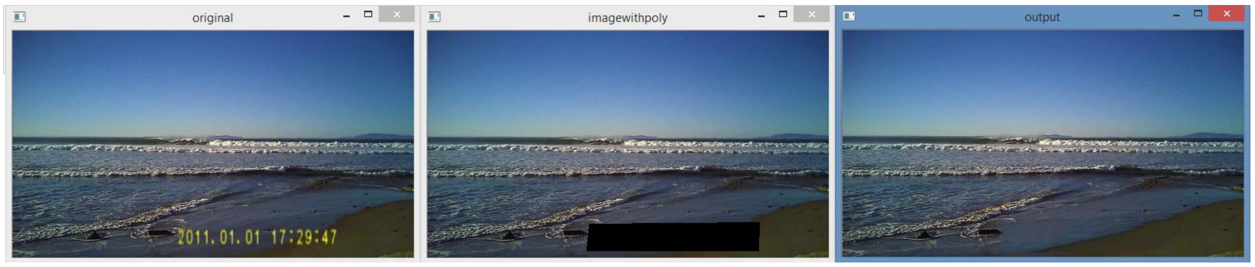


Fig: Timestamp removal

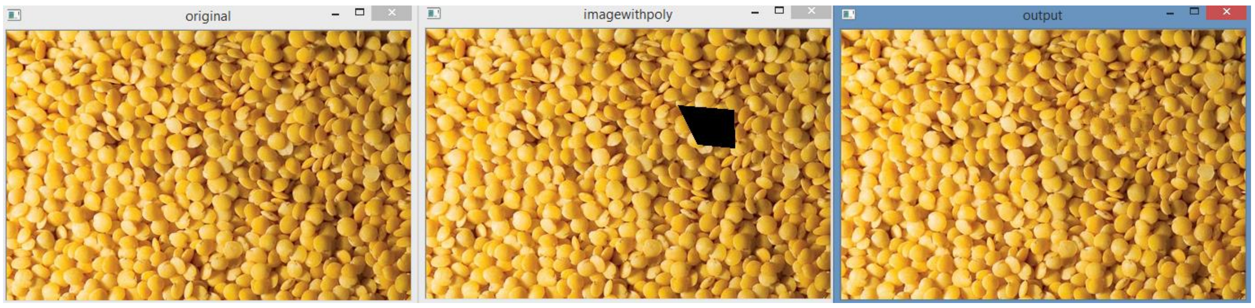


Fig: Inpainting a highly textured region

2.6.2 Mode 2

In this mode I use a polygon to specify a region of interest but not the mask. The mask is obtained by thresholding with respect to a known mask region pixel which is input by the user as a right click as described earlier.

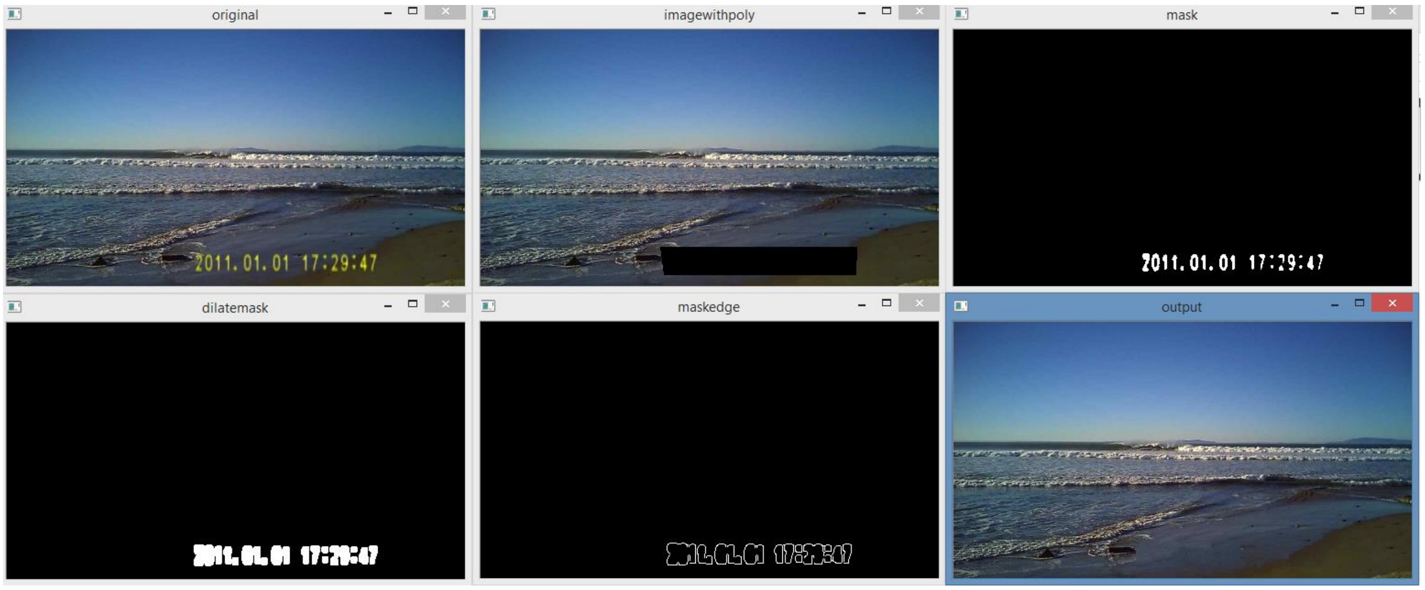


Fig: Top left: Original image with timestamp. Top Centre: User defined region of interest. Top Right: Thresholding based on known point (which is a yellow point of the timestamp, input by the user by right clicking). Note the mask is somewhat discontinuous. Bottom Left: Dilated mask to remove discontinuities. Bottom centre: Edges of the mask which form the initial fill front. Bottom Right: Output image

Next I show two examples on two Barbara images on textured regions.

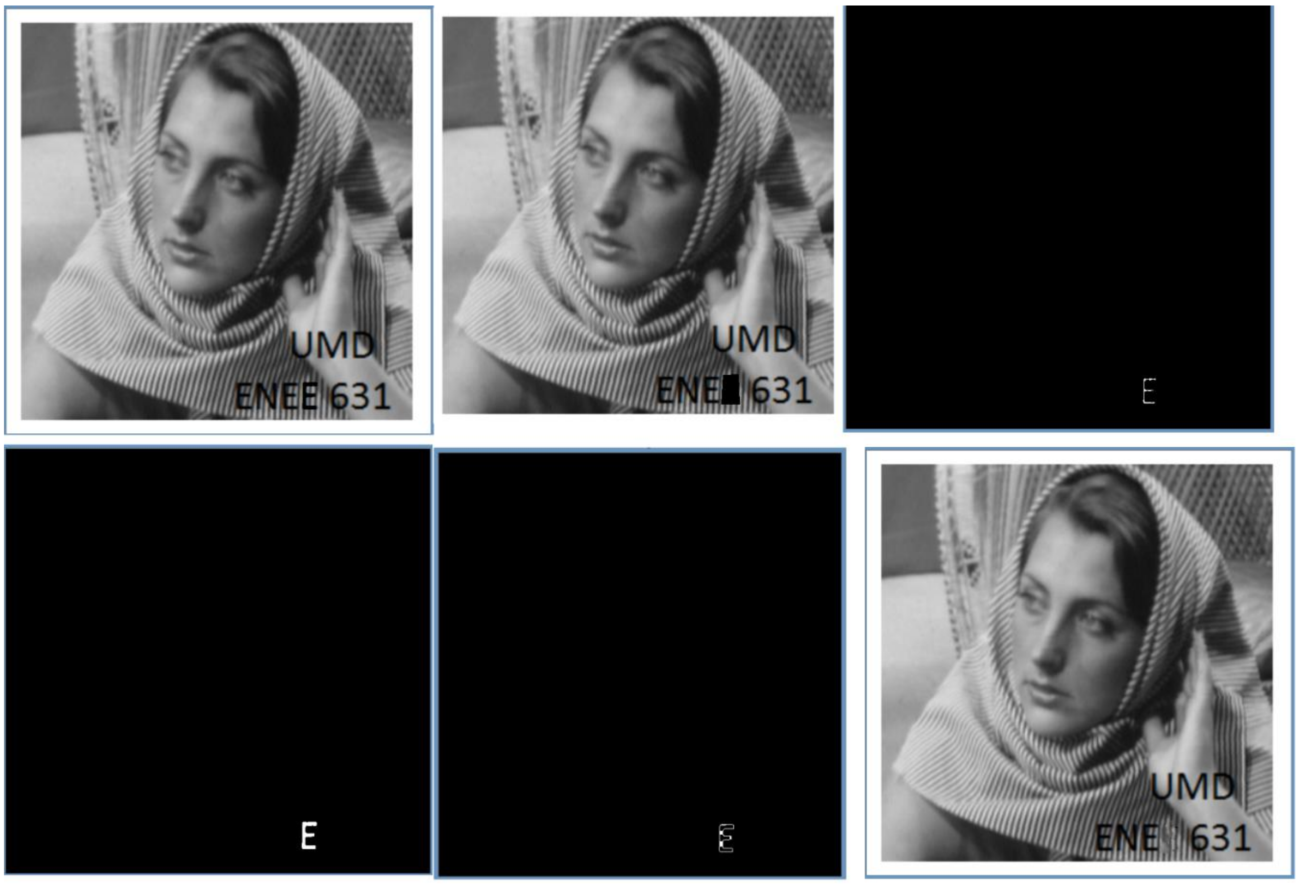


Fig: Inpainting a letter ‘E’. Note the crisp mask formed by the thresholding process

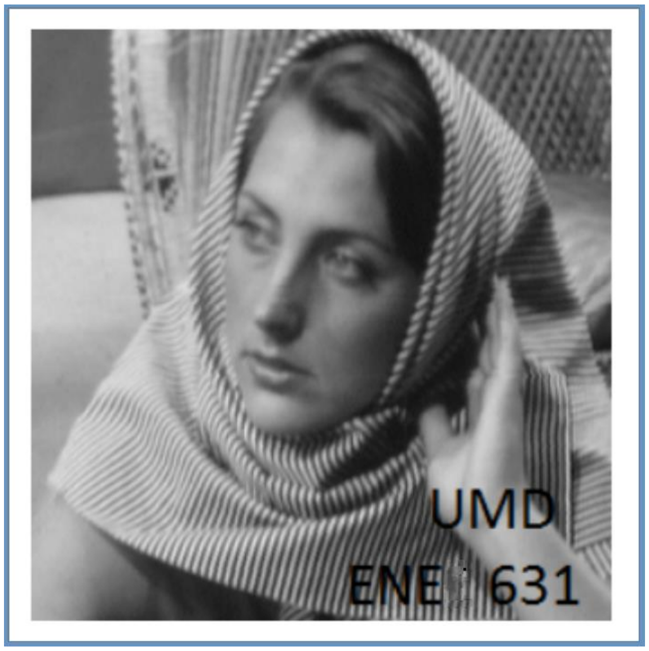


Fig: The output image of the last example

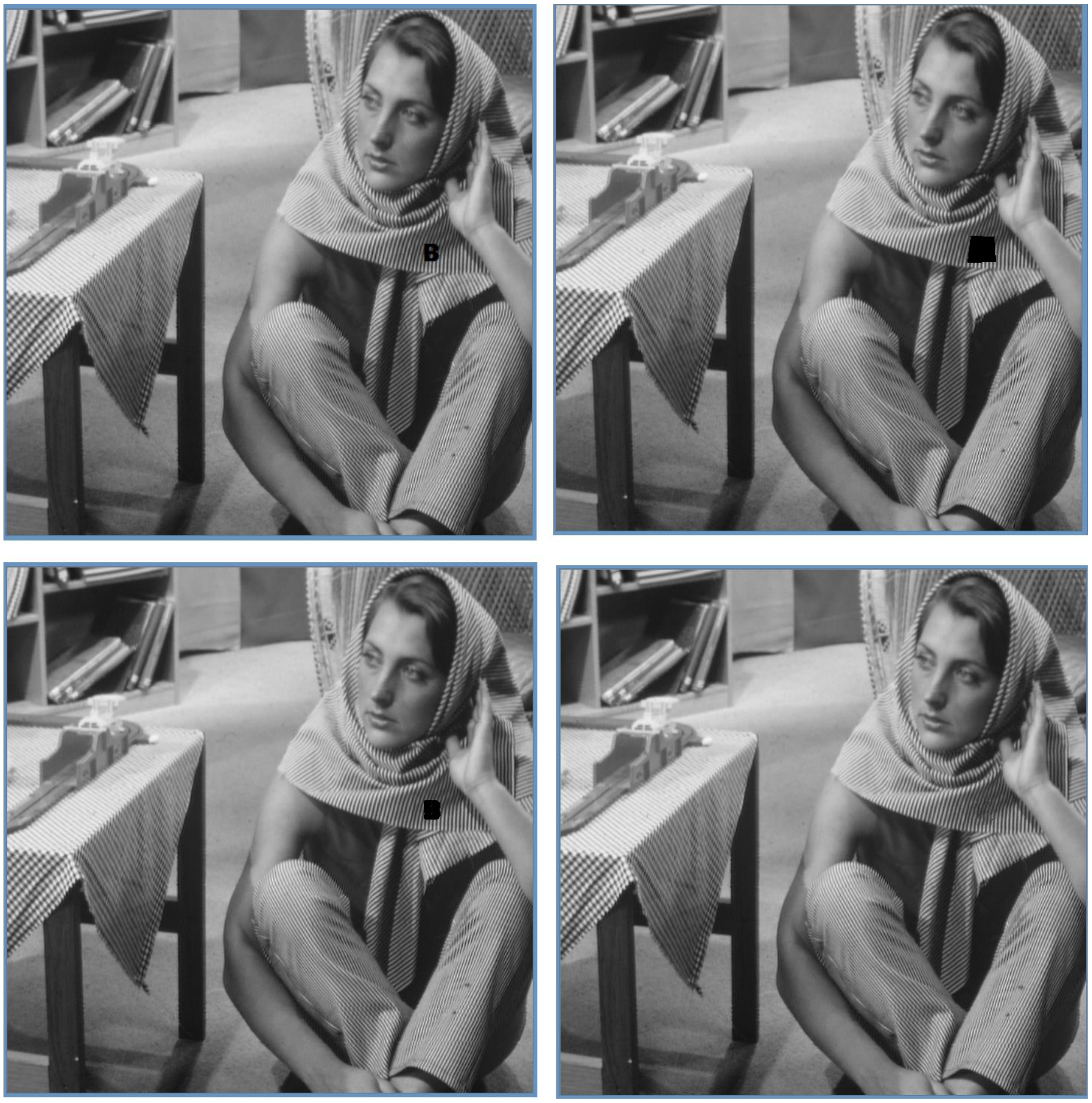


Fig: Top left: Barbara Image with ‘B’ written on it. Top Right: User defined region Bottom Left: Image with thresholded mask. Note the mask looks like a ‘B’ thus it is an improvement over a general polygonal shape that was initially provided by the user. Bottom Right: output image. Note the good approximation of the texture

2.6.3 Mode 3

This mode is useful when there are multiple regions to be inpainted and it is cumbersome to define masks for each region one by one.

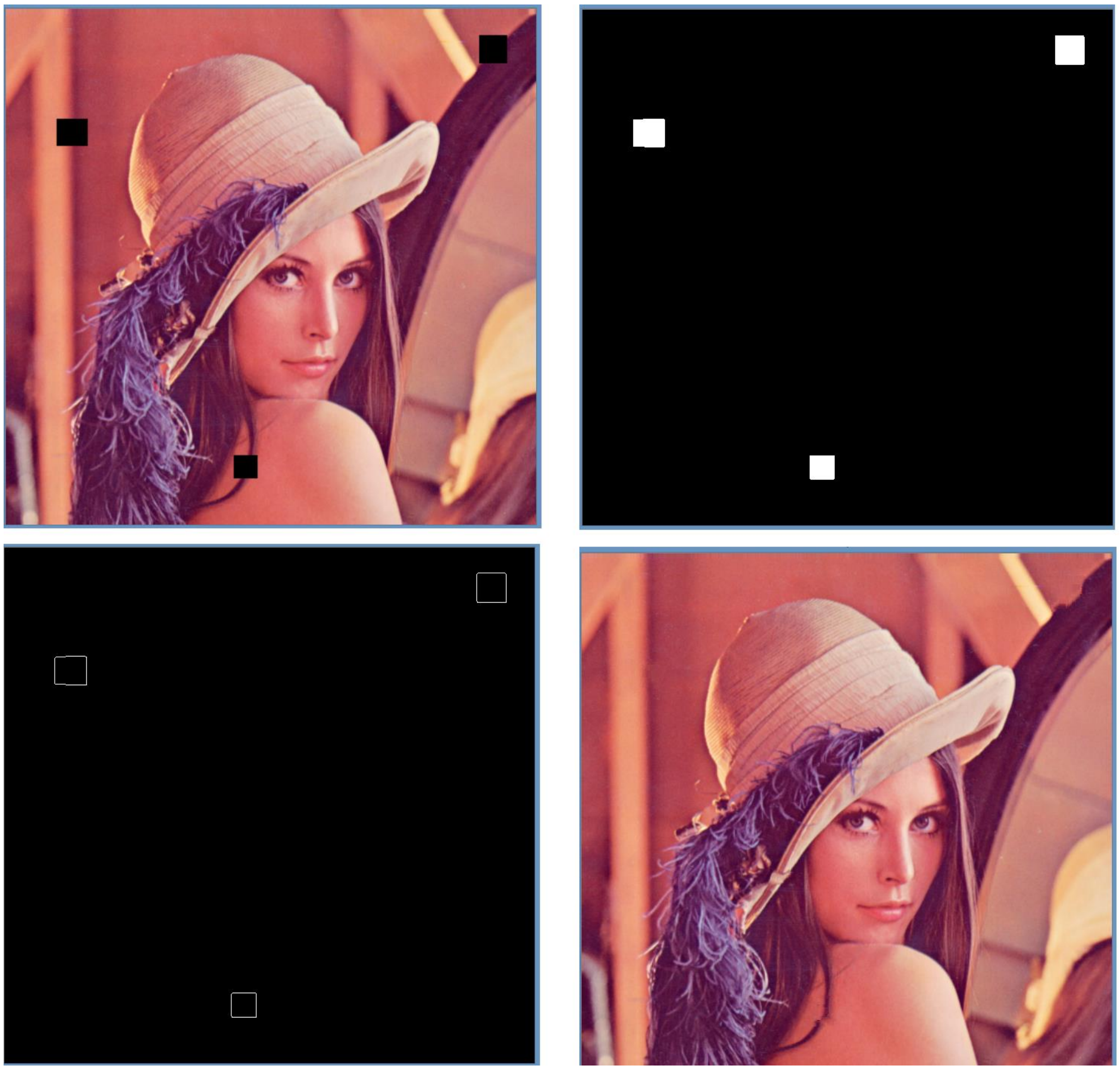


Fig: Top Left: Original image with 3 black region. Top Right: Masks generated by thresholding based on a single pixel that the user clicked on to indicate the colour to be inpainted (In this case the user clicked on one of the black boxes). Bottom Left: Initial fill front. Bottom Right: Output image. Note the edges present in the inpainted region show reasonable continuity.

2.7 Android Implementation

I also developed an android app which uses Java for the GUI development and C++ with OpenCV for the image processing part. Note I used OpenCV's inbuilt functions for inpainting in the android implementation. In the app the user selects an image and can select a rectangular region of interest by dragging his/her finger from the top left to the bottom right. When the region has been selected, the region is zoomed and displayed. If the user is satisfied, he/she can tap once to let the inpainting process start. First Otsu's thresholding is done to infer a mask, then we start the inpainting. On finishing the results are displayed. Below are some screenshots of the app running.

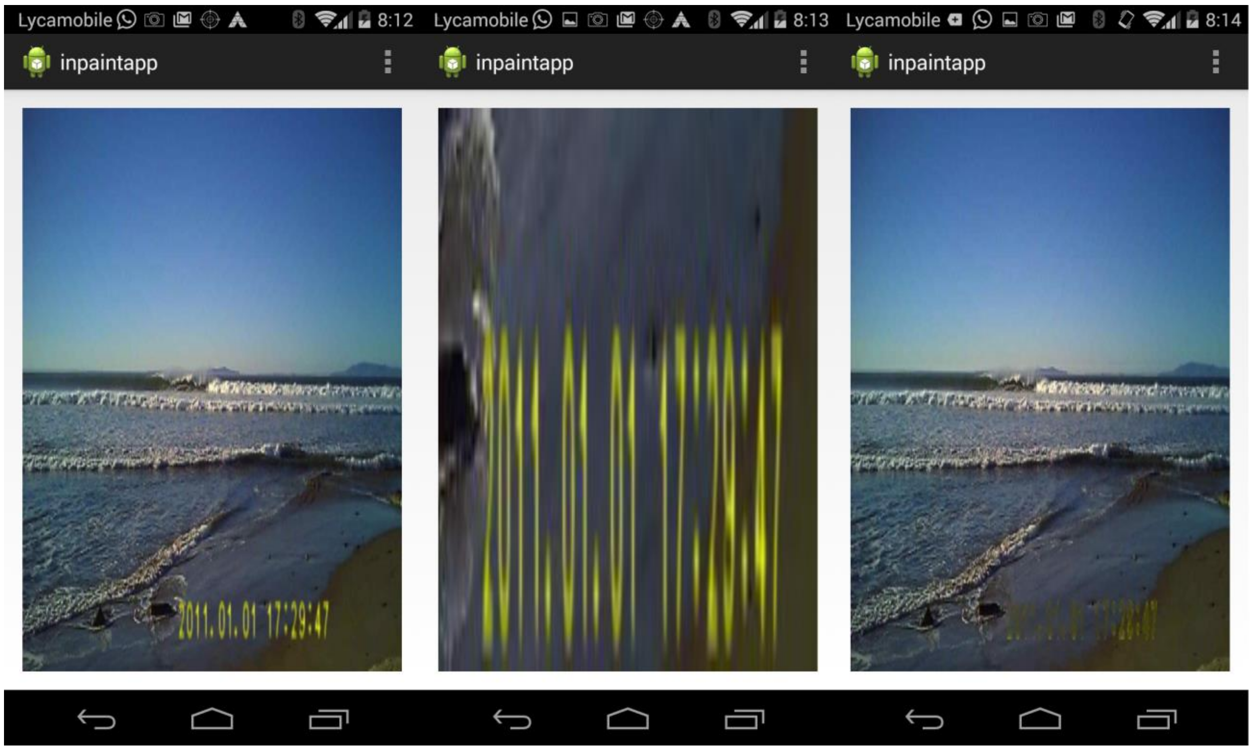
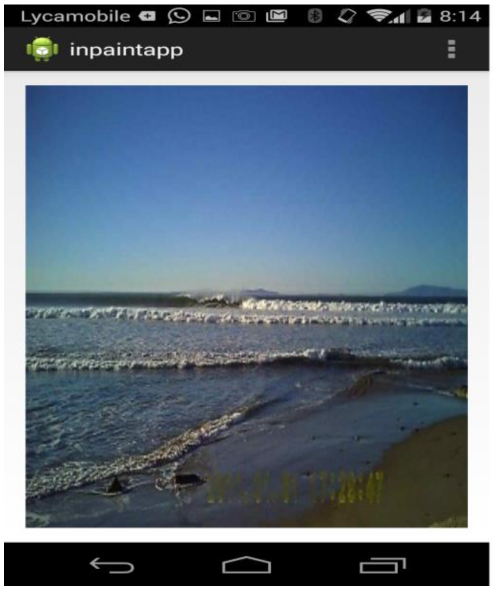


Fig: Left: Original image. Centre: Zoomed in view of the region of interest selected by the user. Right: Output



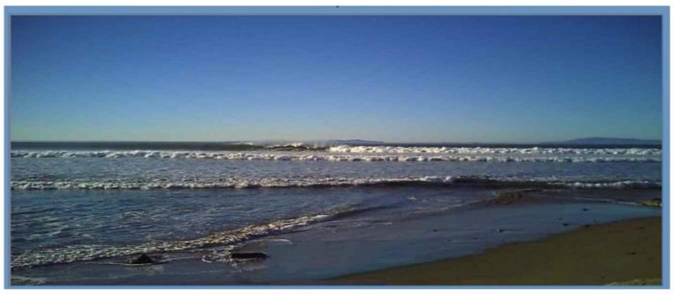


Fig: Comparision between the Android implementation which uses OpenCV's inbuilt inpainting function (Left) and my implementation of inpainting (Right)

The above image shows a comparision between OpenCV's inbuilt inpainting and my implementation. Note the poor performance of OpenCV's implementation, where the time stamp is still visible. But the OpenCV implementation is much faster compared to my implementation.

References

General

[1] http://en.wikipedia.org/wiki/Image_stitching

[2] http://en.wikipedia.org/wiki/Inpainting

[3] Szeliski, Richard. "Image alignment and stitching: A tutorial." Foundations and Trends® in Computer Graphics and Vision 2.1 (2006): 1-104.

[4] http://www.mathworks.com/help/vision/examples/feature-based-panoramic-image-stitching.html

SIFT

[5] http://www.inf.fu-berlin.de/lehre/SS09/CV/uebungen/uebung09/SIFT.pdf

[6] https://www.youtube.com/watch?v=NPcMS49V5hg

[7] http://web.eecs.umich.edu/~silvio/teaching/EECS598/lectures/lecture10_1.pdf

RANSAC

[8] http://en.wikipedia.org/wiki/RANSAC

[9] http://www.cse.yorku.ca/~kosta/CompVis_Notes/ransac.pdf

Blending

[10] http://graphics.cs.cmu.edu/courses/15-463/2005_fall/www/Lectures/Pyramids.pdf

[11] http://www.cs.toronto.edu/~jepson/csc320/notes/pyramids.pdf

[12] Burt, P. J. and Adelson, E. H., A multiresolution spline with applications to image mosaics, ACM Transactions on Graphics, 42(4), October 1983, 217-236

[13] http://pages.cs.wisc.edu/~csverma/CS766_09/ImageMosaic/imagemosaic.html

Inpainting

[14] Toyama, C. E., Criminisi, A., Pérez, P., & Toyama, K. (2003). Object removal by exemplar-based inpainting.