**PROJECT REPORT**

**ON**

**BLENDING IMAGES : GRADIENT DOMAIN FUSION**

Submitted in partial fulfillment of the requirements

for the award of the degree of

**Bachelor of Technology**

**In**

**Computer Science and Engineering**

Guide(s): Submitted by:

Ms. Teena Verma Surbhi Bhatnagar -20313302710

Assistant Professor (HMRITM) PawanThareja-20513302710

Vidit Jain-21413302710



**HMR Institute of Technology & Management**

**Hamidpur, Delhi-110 036**

**Affiliated to**

**GURU GOBIND SINGH INDRAPRASTHA UNIVERSITY**

**Sector - 16C Dwarka, Delhi - 110075, India**

**2010-2014**

**DECLARATION**

We hereby declare that the project work entitled “Blending images: Gradient domain fusion” submitted to Guru Gobind Singh Indraprastha University, is a record of bonafide work done by me under the guidance of Ms. Teena, Assistant Professor, HMRITM, Delhi.

Surbhi Bhatnagar (20313302710)

Pawan Thareja (20513302710) Vidit Jain (21413302710)

**ACKNOWLEDGEMENT**

It is not possible to prepare a project report without the assistance & encouragement of other people. This one is certainly no exception.

On the very outset of this report, we would like to extend our sincere & heartfelt obligation towards all the personages who have helped us in completion of our project. Without their active guidance, help, cooperation & encouragement, we would not have made headway in the project.

We respect and thank Ms. Teena Verma, our guide, for giving us an opportunity to do the project work in HMRITM and providing us with all the support and guidance which helped us complete the project on time .We are extremely grateful to her for providing such a nice support and guidance though she had busy schedule managing the college affairs.

We would not forget to remember Mr. Ravinder Beniwal, our proctor, CSE-B HMRITM, for his unlisted encouragement and more over for his timely support and guidance till the completion of our project work.

I also acknowledge with a deep sense of reverence, our gratitude towards our respective parents and members of our family, who have always supported us morally.

At last but not least gratitude goes to all of our friends who directly or indirectly helped us to complete this project report. Any omission in this brief acknowledgement does not mean lack of gratitude.

Surbhi Bhatnagar

Pawan Thareja Vidit Jain

**CERTIFICATE**

This is to certify that the project report entitled “ Blending Images : Gradient Domain Fusion” done by Ms.Surbhi Bhatnagar (20313302710), Mr. Pawan Thareja (20513302710) & Mr. Vidit Jain (21413302710) is an authentic work carried out by him/her at HMRITM under my guidance. The matter embodied in this project work has not been submitted earlier for the award of any degree or diploma to the best of my knowledge and belief.

Date: 26/11/2013 Ms. Teena Verma

Assistant Professor

HMRITM

**ABSTRACT**

The basic idea of this project is based on the paper *Poisson Image Editing by Patrick Perez, Michel Gangnet, Andrew Blake* where the authors tell about the concept involved behind seamless blending of the source image to the target image based on the mathematics involved behind the image intensity. We plan to extend the project further by including the mixed blending and see if the final image results get any better.

First method we use is based on Poisson’s formula. Its main objective is to seamlessly blend target and source images using Poisson blending. Poisson blending can be viewed as an interpolation problem guided by somegradient field.

For a source image, every pixel that we have, we find out 4 neighbors of each of them. The summation that we do matches those of the source image that we have selected. Two major considerations for gradient need to be made – one is for the pixel values in the source image region and other is the gradient pixel values that are at the boundary. Once we have calculated these values, we transfer these values from the source image onto the target or background image and then rest of the values outside the mask are kept as is. Since we are working on the RGB images, we copied the RGB channels directly.

Second, we use mixed blending. Its main objective is to seamlessly blend target and source images using Mixed blending. This method would involve using the RGB channels. We calculate the larger gradient of the 2 images either source or target and then construct the final image out of these.

**1.** **INTRODUCTION**

**1.1 PROBLEM SPECIFICATION**

* This project is to compose a source image and a target image in the gradient domain based application for the purpose of combining a source image to target image.
* The primary goal of this assignment is to integrate/ blend a source image into a target image making sure that there is seamless blending between the two so that the final image that we obtain looks like a single image. The focus is to maintain the gradient of the source image to be copied rather than keeping our focus on the intensity, this way we are assuring that the image though changes little bit of it's color but since we are maintaining the gradient, the overall image looks as if it's seamlessly blended.
* This system will provide an interface for the customer to blend source image into the target image.
* The efficient design of the application will allow the user to quickly blend images.
* System must be informative, robust, responsive, user-friendly and secure.
* System should be designed to allow possible future expansion, for an example more product lines, and for potential porting to blending techniques.

**1.2 INPUT/ OUTPUT SPECIFICATIONS**

**1.2.1 INPUT SPECIFICATION (USER'S VIEW)**

An application will be designed that allows the user to input the following information from their computer:

* Source Image - We will be selecting the part of image which is to be blended with the target image.
* Target Image - Acts as a background image.

**1.2.2 OUTPUT SPECIFICATION (USER'S VIEW)**

The system will output an image where there would be seamless blending between the source image and the target image so that the final image that we obtain looks likes a single image.

**2. DEVELOPMENT ENVIRONMENT**

**2.1 INTRODUCTION TO MATLAB**

MATLAB (matrix laboratory) is a [numerical computing](http://en.wikipedia.org/wiki/Numerical_analysis) environment and [fourth-generation programming language](http://en.wikipedia.org/wiki/Fourth-generation_programming_language). Developed by [MathWorks](http://en.wikipedia.org/wiki/MathWorks), MATLAB allows [matrix](http://en.wikipedia.org/wiki/Matrix_%28mathematics%29) manipulations, plotting of [functions](http://en.wikipedia.org/wiki/Function_%28mathematics%29) and data, implementation of [algorithms](http://en.wikipedia.org/wiki/Algorithm), creation of [user interfaces](http://en.wikipedia.org/wiki/User_interface), and interfacing with programs written in other languages, including [C](http://en.wikipedia.org/wiki/C_%28programming_language%29), [C++](http://en.wikipedia.org/wiki/C%2B%2B), [Java](http://en.wikipedia.org/wiki/Java_%28programming_language%29), and [Fortran](http://en.wikipedia.org/wiki/Fortran).

## 2.1.1 HISTORY

## [Cleve Moler](http://en.wikipedia.org/wiki/Cleve_Moler), the chairman of the [computer science](http://en.wikipedia.org/wiki/Computer_science) department at the [University of New Mexico](http://en.wikipedia.org/wiki/University_of_New_Mexico), started developing MATLAB in the late 1970s. He designed it to give his students access to [LINPACK](http://en.wikipedia.org/wiki/LINPACK) and [EISPACK](http://en.wikipedia.org/wiki/EISPACK) without them having to learn [Fortran](http://en.wikipedia.org/wiki/Fortran). It soon spread to other universities and found a strong audience within the [applied mathematics](http://en.wikipedia.org/wiki/Applied_mathematics) community. [Jack Little](http://en.wikipedia.org/wiki/John_N._Little), an engineer, was exposed to it during a visit Moler made to [Stanford University](http://en.wikipedia.org/wiki/Stanford_University) in 1983. Recognizing its commercial potential, he joined with Moler and Steve Bangert. They rewrote MATLAB in [C](http://en.wikipedia.org/wiki/C_%28programming_language%29) and founded [MathWorks](http://en.wikipedia.org/wiki/MathWorks) in 1984 to continue its development. These rewritten libraries were known as JACKPAC. In 2000, MATLAB was rewritten to use a newer set of libraries for matrix manipulation, [LAPACK](http://en.wikipedia.org/wiki/LAPACK).

MATLAB was first adopted by researchers and practitioners in [control engineering](http://en.wikipedia.org/wiki/Control_engineering), Little's specialty, quickly spread to many other domains. It is now also used in education, in particular the teaching of [linear algebra](http://en.wikipedia.org/wiki/Linear_algebra) and [numerical analysis](http://en.wikipedia.org/wiki/Numerical_analysis), and is popular amongst scientists involved in [image processing](http://en.wikipedia.org/wiki/Image_processing)

**2.2 KEY FEATURES**

MATLAB provides a range of numerical computation methods for analyzing data, developing algorithms, and creating models. The MATLAB language includes mathematical functions that support common engineering and science operations.

Core math functions use processor-optimized libraries to provide fast execution of vector and matrix calculations.

Available methods include :

* Interpolation and regression
* Differentiation and integration
* Linear systems of equations
* Fourier analysis
* Eigen values and singular values
* Ordinary differential equations (ODEs)
* Sparse matrices

**2.3 DATA ANALYSIS AND VISUALISATION**

MATLAB provides tools to acquire, analyze, and visualize data enabling you to gain insight into your data in a fraction of the time it would take using spreadsheets or traditional programming languages.

MATLAB lets you access data from files, other applications, databases, and external devices. You can read data from popular file formats such as Microsoft Excel; text or binary files; image, sound, and video files; and scientific files such as netCDF and HDF. File I/O functions lets you work with data files in any format. Using MATLAB with add-on products, you can acquire datafrom hardwaredevices, such as your computer’s serial port or sound card, as well as stream live, measured data directly into MATLAB for analysis and visualization. You can also communicate with instruments such as oscilloscopes, function generators.

**2.4 DEPLOYMENT**

MATLAB tools and add-on products provide a range of options to develop and deploy applications. You can share invalidly algorithms and applications with other MATLAB users or deploy them royalty-free to others who do not have MATLAB.

Using GUIDE (Graphical User Interface Development Environment), you can layout, design, and edit custom graphical user interfaces. You can include common controls such as list boxes, pull-down menus, and push buttons, as well as MATLAB plots. Graphical user interfaces can also be created programmatically using MATLAB functions. Deploying applications to distribute an application directly to other MATLAB users, you can package it as a MATLAB app, which provides a single file for distribution. Apps automatically install in the MATLAB apps gallery for easy access. To share applications with others who do not have MATLAB, you can use application deployment products. These add-on products automatically generate standalone applications, shared libraries, and software components for integration in C, C++, Java, .NET, and Excel environments. The executable and components can be distributed royalty-free.

### 2.4.1 VARIABLES

Variables are defined using the assignment operator, =. MATLAB is a [weakly typed](http://en.wikipedia.org/wiki/Strong_and_weak_typing) programming language because types are implicitly converted. It is a dynamically typed language because variables can be assigned without declaring their type, except if they are to be treated as symbolic objects, and that their type can change. Values can come from [constants](http://en.wikipedia.org/wiki/Constant_%28computer_science%29), from computation involving values of other variables, or from the output of a function.

### 2.4.2 VECTORS / MATRICES

A simple array is defined using the colon syntax: *init*: *incrementt: terminator*. For instance:

>>array = 1:2:9

array = 13579

Defines a variable named array (or assigns a new value to an existing variable with the name array) which is an array consisting of the values 1, 3, 5, 7, and 9. That is, the array starts at 1 (the *init* value), increments with each step from the previous value by 2 (the *increment* value), and stops once it reaches (or to avoid exceeding) 9 (the *terminator* value).

>>array = 1:3:9

array = 147

The *increment* value can actually be left out of this syntax (along with one of the colons), to use a default value of 1.

>>ari = 1:5

ari = 12345

assigns to the variable named ari an array with the values 1, 2, 3, 4, and 5, since the default value of 1 is used as the incrementer.

[Indexing](http://en.wikipedia.org/wiki/One-based_indexing) is one-based, which is the usual convention for [matrices](http://en.wikipedia.org/wiki/Matrix_%28mathematics%29) in mathematics, although not for some programming languages such as C, C++, and Java.

Matrices can be defined by separating the elements of a row with blank space or comma and using a semicolon to terminate each row. The list of elements should be surrounded by square brackets: []. Parentheses: () are used to access elements and subarrays (they are also used to denote a function argument list).

>> A = [163213; 510118; 96712; 415141]

A = 163213

510118

96712

415141

>>A(2,3)

ans11

Most MATLAB functions can accept matrices and will apply themselves to each element. For example, mod(2\*J,n) will multiply every element in "J" by 2, and then reduce each element modulo "n". MATLAB does include standard "for" and "while" loops, but (as in other similar applications such as [R](http://en.wikipedia.org/wiki/R_%28programming_language%29)), using the [vectorized](http://en.wikipedia.org/wiki/Vectorization_%28parallel_computing%29) notation often produces code that is faster to execute. This code, excerpted from the function *magic.m*, creates a [magic square](http://en.wikipedia.org/wiki/Magic_square)*M* for odd values of *n* (MATLAB function *meshgrid* is used here to generate square matrices I and J containing 1:n).

[J,I] = meshgrid(1:n);

A = mod(I + J - (n + 3) / 2, n);

B = mod(I + 2 \* J - 2, n);

M = n \* A + B + 1;

### 2.4.3 STRUCTURES

MATLAB has structure data types. Since all variables in MATLAB are arrays, a more adequate name is "structure array", where each element of the array has the same field names. In addition, MATLAB supports dynamic field names (field look-ups by name, field manipulations, etc.). Unfortunately, MATLAB JIT does not support MATLAB structures, therefore just a simple bundling of various variables into a structure will come at a cost.

### 2.4.4 FUNCTION HANDLES

MATLAB supports elements of [lambda calculus](http://en.wikipedia.org/wiki/Lambda_calculus) by introducing function handles,or function references, which are implemented either in .m files or anonymous/nested functions.

### 2.4.5 CLASSES

Although MATLAB has classes, the syntax and calling conventions are significantly different from other languages. MATLAB has value classes and reference classes, depending on whether the class has *handle* as a super-class (for reference classes) or not (for value classes).

Method call behavior is different between value and reference classes. For example, a call to a method object.method();can alter any member of *object* only if *object* is an instance of a reference class.

## 2.4.6 GRAPHICS AND GRAPHCAL USER INTERFACE PROGRAMMING

MATLAB supports developing applications with graphical user interface features. MATLAB includes GUIDE (GUI development environment) for graphically designing GUIs.It also has tightly integrated graph-plotting features. For example the function *plot* can be used to produce a graph from two vectors *x* and *y*.

**3. TYPES OF BLENDING**

**3.1 POISSON BLENDING**

The objective of this "Poisson Blending" algorithm is to compose a source image and a target image in the gradient domain. The reason "Poisson Blending" achieves a more realistic looking composition than naively pasting two similarly colored images together is because the human visual system is more sensitive to contrast than intensity values.

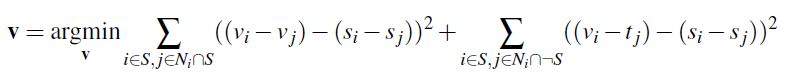
Image editing tasks concern either global changes (color/intensity corrections, filters, deformations) or local changes confined to a selection. Here we are interested in achieving local changes, ones that are restricted to a region manually selected, in a seamless and effortless manner. The extent of the changes ranges from slight distortions to complete replacement by novel content. Classic tools to achieve that include image filters confined to a selection, for slight changes, and interactive cut-and-paste with cloning tools for complete replacements. With these classic tools, changes in the selected regions result in visible seams, which can be only partly hidden, subsequently, by feathering along the border of the selected region.

We propose here a generic machinery from which different tools for seamless editing and cloning of a selection region can be de-rived. The mathematical tool at the heart of the approach is the Poisson partial differential equation with Dirichlet boundary conditions which species the Laplacian of an unknown function over the domain of interest, along with the unknown function values over the boundary of the domain.

First, it is well known to psychologists [Land and McCann 1971] that slow gradients of intensity, which are suppressed by the Laplacian operator, can be superimposed on an image with barely noticeable effect. Conversely, the second-order variations extracted by the Laplacian operator are the most significant perceptually.

So, given methods for crafting the Laplacian of an unknown function over some domain, and its boundary conditions, the Poisson equation can be solved numerically to achieve seamlessing of that domain. This can be replicated independently in each of the channels of a color image. Solving the Poisson equation also has an alternative interpretation as a minimization problem: it computes the function whose gradient is the closest, in the *L*2 -norm, to some prescribed vector field — the *guidance* vector field — under given boundary conditions. In that way, the reconstructed function interpolates the boundary conditions inwards, while following the spatial variations of the guidance field as closely as possible.

The following is an objective function that can help us solve the problem



Given the pixel intensities of the source image "s" and of the target image "t", we want to solve for new intensity values "v" within the source region "S". Here, each "i" is a pixel in the source region "S", and each "j" is a 4-neighbor of "i". Each summation guides the gradient values to match those of the source region. In the first summation, the gradient is over two variable pixels; in the second, one pixel is variable and one is in the fixed target region.

Perhaps a figure can do a better job in explaining,

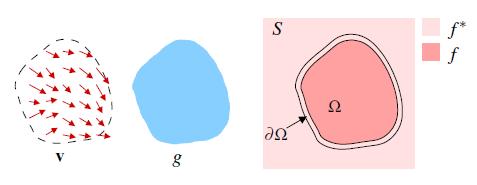
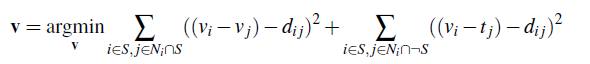


Figure 1 : Poisson Blending

* v – guidance field of image gradient
* g – selected region of source
* f\* – known functions that exist in domain S
* f – unknown functions that exist in domain Ω
* Ω – region g that is now placed on domain S (target background)
* ∂Ω – boundaries between the source and target regions

Under the guidance of gradient vector field v, f interpolates in domain Ω the target function f\*. We want to find f such that the new gradient is closest to the gradient of the source region g.



If the target gradient is greater than the source gradient, then we use the target gradient as the guide. Same vice versa.

* source gradient(i, j) = src(i) – src(j)
* target gradient(i, j) = tar(i) – tar(j)
* d(i, j) = if target > source then target else source

If you were to look at the equation as just solving for fp, you would see that the equation simplifies to the new pixel value equals the average of it's neighborhood plus the gradient of the pixel to it's neighbors. Notice that the value taken for the neighbor is the new value (fq) if the neighbor is under the mask and it is the target image value otherwise. Since the new (unknown) pixels may have neighbors who are also unknown, you can think of the problem recursively or as a large series of constraints. Each pixel in the target image is assigned a variable, P1,1, P1,2, P1,n, P2,1,..., through Pm,n for an m by n image. Thus, the constraints put on the variable Pi,j are:

* If Pi,j ∉ Ω, Pi,j = Targeti,j
* Otherwise:

( # of neighbors ) \* Pi,j =sum for each connected neighbor Q: (i-1,j), (i+1,j), (i,j-1), (i,j+1)

If Q is under the mask,

+QUnknown  
   else   
     +QTarget  
   end   
     +(PSource - QSource)

The above constraints yield m\*n linear equations with m\*n unknowns which can be represented in the [standard matrix form](http://en.wikipedia.org/wiki/System_of_linear_equations#Matrix_equation) Ax = b and solved to get exactly one solution. Here P is a column vector comprised of the m\*n variables (pixels) in the final image, A is composed of the constraint equations listed above and b is a combination of known pixel values and calculated values.

The coefficient matrix will be mostly sparse since the only pixels affecting the output value are the given pixel and 4 of it's neighbors. Also instead of using inv(A) in MATLAB to solve Ax=b, use: x = A\b; where A is the coefficient matrix, b is the solution vector, and x is the output values for the pixels. If you don't, you might get annoying errors that makes no sense.

**3.1.1 ALGORITHM**

The objective of this "Poisson Blending" algorithm is to compose a source image and a target image in the gradient domain. The reason "Poisson Blending" achieves a more realistic looking composition than naively pasting two similarly colored images together is because the human visual system is more sensitive to contrast than intensity values. The technique described here is an implementation of the algorithm presented in [(Pérez, et al.)](http://www.cs.brown.edu/courses/csci1290/asgn/proj2/resources/PoissonImageEditing.pdf)

**3.1.1.1 ALGORITHM OVERVIEW**

Prior to running the algorithm, a mask needs to be manually generated that indicates the overlapping region between the source (image being inserted) and the target (image the source is being inserted into).

#### Preprocess Source, Target, and Mask images

The mask and source images are first padded so that they are the same size as the target image. Next, they are translated using parameterized row/column offsets. If the valid pixels in the mask are on the edge of the source image, a 1 pixel symmetric buffer is added to all 3 images to make processing easier. This 1 pixel buffer is then cropped off later from the result.

#### Generate a Sparse Matrix

A linear system of equations is required to compute the resulting image from the source and target gradients. This system is represented by *Ax=b*, where *A* is the sparse coefficients matrix, *x* is the output image, and *b* is the desired gradient matrix. The size of sparse matrix *A* is *N*x*N*, where *N* is target image rows multipled by target image columns.

For pixels outside the masked region, the output image pixel is simply the same as the target image. For these pixels, the row in the sparse matrix *A* is simply the same as the identity matrix. Also for these pixels, the corresponding value in the desired pixel gradient matrix *b* are also the same pixel value as the target image.

For pixels inside the masked region, the output image pixel *x* at (row, col) depends on its neighbors according to the following equation:

*4\*x(row, col) - x(row+1, col) - x(row-1, col) - x(row, col+1) - x(row, col-1) =* Desired pixel gradient

For these pixels inside the masked region, the row in the sparse matrix then contains these coefficients at the corresponding indices. The value of the desired pixel gradient is in matrix *b* and is described further in the Mixing Gradients section.

The creation of this sparse matrix was also optimized by eliminating most of the for loops, avoiding calls to MATLAB's *sub2ind* and *ind2sub* functions, and only calling MATLAB's *sparse* function once.

#### Blend Each Color Channel Seperately

Finally, each image's red, green, and blue color channels are blended separately. While the same sparse coefficients matrix *A* may be used for each color channel, the gradient matrix *b* is different for each color channel.

The final output is calculated using the matrix operation *x = A\b* in MATLAB.

**3.1.1.2 ALGORITHM ANALYSIS**

Let us now present the system of equations that solves the problem. Let's call the image which we're pasting on A (as before), the image which we're pasting B (as before), and the new image to be pasted H (where H should be some improved version of B that blends in with A better). First the easy part: the boundary constraints. We said before the pixels on H's boundary should be exactly the same as the pixels of A on that boundary, so that we can match those pixels on the outside of the selection and blend them inwards. Mathematically, we have the following equation :

*H*(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/144/char3B.png*y*)=*A*(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/144/char3B.png*y*)http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char38.png(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char3B.png *y*)http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char32.pnghttp://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char40.png*B*

So we already know the exact solution to all pixels on the boundary of H (which is also the boundary of B). Now we need to solve for the pixels in the interior of H. We want the gradient of the pixels on the interior of H to equal the gradient of the pixels on the interior of B. A simple way to define a gradient of an image at a spot is the sum of the differences between that pixel and all of its neighbors:

http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char6A.pnghttp://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char72.png*B*(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/144/char3B.png *y*)http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char6A.png=4*B*(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char3B.png *y*)−*B*(*x*−1http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char3B.png *y*)−*B*(*x*+1http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char3B.png *y*)−*B*(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char3B.png *y*−1)−*B*(*x*http://www.ctralie.com/jsMath/fonts/cmmi10/alpha/207/char3B.png *y*+1)

If one of the neighbors happens to be a boundary pixel, then its value is fixed. If one of the neighbors happens to be out of bounds of the selection, then it should be excluded. This is summarized for all cases for every point in H by the following difference equation (lifted from the paper and rearranged in a way that I think makes more sense):

http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char6A.png*N*http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/207/char6A.png*H*(*x, y*)−http://www.ctralie.com/jsMath/fonts/cmex10/alpha/207/char58.png(*dx, dy*)+(*x*, *y*)http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/144/char32.png http://www.ctralie.com/jsMath/fonts/cmr10/alpha/144/char0A.png *H* (*x*+*dx*, *y*+*dy*)−http://www.ctralie.com/jsMath/fonts/cmex10/alpha/207/char58.png((*dx*, *dy*)+(*x*, *y* )http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/144/char32.pnghttp://www.ctralie.com/jsMath/fonts/cmmi10/alpha/144/char40.pnghttp://www.ctralie.com/jsMath/fonts/cmr10/alpha/144/char0A.png

*A*(*x*+*dx*, *y*+*dy*)

=http://www.ctralie.com/jsMath/fonts/cmex10/alpha/207/char58.png(*dx*, *dy*)+( *x*, *y*)http://www.ctralie.com/jsMath/fonts/cmsy10/alpha/144/char32.png (http://www.ctralie.com/jsMath/fonts/cmr10/alpha/144/char0A.pnghttp://www.ctralie.com/jsMath/fonts/cmsy10/alpha/144/char5B.pnghttp://www.ctralie.com/jsMath/fonts/cmmi10/alpha/144/char40.pnghttp://www.ctralie.com/jsMath/fonts/cmr10/alpha/144/char0A.png ) (*B*(*x*+*dx*, *y*+*dy*)−*B*(*x*, *y*))

where (x, y) is the location of the point of interest on the 2D grid, "N" is the number of valid neighbors the pixel actually has within the selection region including the boundary (less than or equal to 4), "Omega" is the selection area of B and H excluding the boundary, "partial Omega" is the boundary of the selection area, and (dx, dy) are the possible neighbor locations that range over {(-1, 0), (1, 0), (0, -1), (0, 1)} (this accounts for the 4 possible neighbors of each point). This looks like a mess but it really isn't, so let us break down this equation :

* The left hand side of the equation is computing the spatial gradient of the unknown point H(x, y) by taking summing the difference between H(x, y) and all of its **N** neighbors. Each difference that goes into the gradient has the form H(x, y) - other(x', y'), where (x', y') is the position of a neighbor. The first sum on the left hand side represents the difference of H(x, y) with other H(x', y') points that are on the interior of the selection (Omega). The second term represents the difference of H(x, y) with border points, which are fixed at the value of the image that we're pasting onto, A, which is why we have to treat them separately (they do not vary and we do not solve for them).
* The right hand side of the equation is simply the gradient of image B at (x, y), which we would like to match with the gradient of our new image H at (x, y). Hence the equality.
* Note that for color images, these equations are set up and solved for the Red, Green, and Blue channels independently.

Basically, we're trying to solve for a new image H that matches the background of A better, and the border of this image H matches up with A exactly, while the difference between adjacent points of H matches up with the difference of the corresponding adjacent points in B.

What we do next is write out this equation for every point in H, and notice that we have a system of linear equations in k variables, where k is the number of pixels we need to solve for in H. To solve the system of equations, the most straightforward thing to do would be to put them in matrix form and invert the matrix. However, k is actually quite large. For instance, in a 200x200 selection, k = 40,000. We do not want to sit around and wait to invert a 40,000 x 40,000 matrix. We should also note that the matrix is *extremely sparse* because every point has at maximum 4 neighbors (and also positive definite for you really observant math heads reading this). So each row has at most 5 nonzero elements and the rest are zero. This lends itself well to an *iterative matrix solving technique*. For simplicity, I decided to use something called the Jacobi Method to solve the sparse linear system. The Jacobi Method is a special case of Gradient Descent . Basically here's how it works:

* Set up your matrix equations in the form

*Ax*=*b*

* Where A is the matrix of equations that I defined above (the equations that related gradients to gradients or set the boundary pixels equal to some constant), x is what we'd like to solve for (the pixels of the H image in this case), and b is what the equations should equal.
* Initialize x to all zeros (this is an all black image)
* Compute the product Ax
* Compute the difference (b - Ax), which measures the error between what the current guess of x (our H image) is and what we need it to be.
* Add the difference (b - Ax) back to x. This is the "gradient descent" part where we try to get our guess of the solution (x) to move in the right direction.
* Repeat steps 3-5 until the error between x and (b-Ax) is small enough.

This process is guaranteed to converge on the correct solution for x if A is positive definite (which it is), and it does so exponentially. What ends up happening with this iterative solution is that the new image starts off with the correct border and a black interior, and then slowly fills in color from the outside .

**3.1.2 MIXING GRADIENTS**

Mixing gradients is performed using a linear combination of the source gradient and target gradient for pixels inside the masked region. A parameterized coefficient *alpha* is used to control how much the source and target influence the gradient according to the following equation:

*gradient = (alpha)\*source\_gradient + (1 - alpha)\*target\_gradient*

Therefore, an *alpha* value of 0.5 would be the same as averaging the two gradients together. A comparison of two blended images processed with and without this technique is shown below.

It is sometimes better, especially for transparent objects, to retain some of the gradient of the target image in the mask area.  The two gradients are mixed by picking the gradient to each neighbor either from the source image or the target image, whichever gradient has a higher absolute value.

|  |  |
| --- | --- |
| http://cs.brown.edu/courses/csci1950-g/results/proj2/edwallac/source_gradient.jpg  The gradient of the source image | http://cs.brown.edu/courses/csci1950-g/results/proj2/edwallac/target_gradient.jpg  The gradient of the target image |
| http://cs.brown.edu/courses/csci1950-g/results/proj2/edwallac/mixed_gradients.jpg Both gradients mixed | http://cs.brown.edu/courses/csci1950-g/results/proj2/edwallac/res_mix_img04.jpg  The final image using gradient mixing |

Figure 2 : Mixing gradients

**3.1.3 POISSION BLENDING EXAMPLES**

**Original Source Target Picture**

**Direct Blend Poisson Blend**

Figure 3 : Poisson Blending Example

Poisson blending solves a linear system of equations, such that the difference between the intensity on boundary between source and target is as small as possible. Also Poisson blending tries to minimize the gradient difference across pixels between the background under source and the source region. I tried to find the source image with the best match of background to the target image, in order to get Poisson blending look as good as possible. The tree trunk under the elephant in source matches the position of the tree in target.

**3.1.3.1 OTHER EXAMPLES**

**Original Source Target Picture**



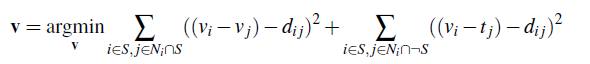
**Direct Blend Poisson Blend**

Figure 4 : Another example of poisson blending

Difficulty comes mostly from finding source image with relative similar background as the target image. Also, placing the source image correctly on the target, such that there is not sharp color transition on the border between source and target. If there is a sharp transition, some region of the final blend would be forced to take a weird gradient. This is considered as a failure case.

**3.2** **MIXED GRADIENT BLENDING**

Follow the same steps as Poisson blending, but use the gradient in source or target with the larger magnitude as the guide, rather than the source gradient:   
  
  
Here "d\_ij" is the value of the gradient from the source or the target image with larger magnitude, i.e. if abs(s\_i-s\_j) > abs(t\_i-t\_j), then d\_ij = s\_i-s\_j; else d\_ij = t\_i-t\_j. One possibility is to blend a picture of writing on a plain background onto another image.

Mixed gradient blending follows the same steps as Poisson blending, but it uses the gradient in source or target with the larger magnitude as the guide, rather than the source gradient.

**3.2.1 MIXED GRADIENT BLENDING IMPLEMENTATION**

Implemented the same code to calculate the sparse matrix A and known vector b and then using lscov to calculate each of the RGB channels and then constructing a new image out of this.

Instead of calculating according to the gradient of the source image, here we are considering for each of the point where either the source image gradient is larger or target image gradient is. Whichever is the largest is taken into consideration.

|  |
| --- |
|  |
| picture 1 | picture 1 | picture 2 |
| (a) Direct cut-and-paste | (b) Poisson blending | (c) Mixed gradient blending |

Figure 5 : Comparison (I) Between the various types of blending

Another result as seen on the paper:

|  |
| --- |
|  |
|  | picture 1 | picture 2 |
|  | (a) Poisson blending | (b) Mixed gradient blending |

Figure 6 : Comparison (II) Between the various types of blending

We can see that in (b), the original background associated with the source object is still present because we only ensure that the gradient to be the same across the object boundary and gradient within the source object follows that of the original source image. On the other hand, the mixed gradient method compares the gradient magnitude between the source and target and chooses the larger one to follow, hence, we can observe that the background texture behind the words (having larger gradient values) are preserved.

**3.3 COMPARISON BETWEEN POISSON BLENDING AND MIXED BLENDING**

Normal Poisson blending is sometimes desired as it can cover the texture of the background. Compare Final Blended Image and Mixed Gradient Blended Image with results given above Without mixed gradient, the first one produces blur effect at the bottom of the balloon and lose original textures. However, in the second picture we want the source image to completely guide the gradient.

(a) Background (b)Source Image



(d) Poisson Blending(e)Mixed Blending

Figure 7 : Comparison Between Poisson Blending And Mixed Blending

### 3.3.1 ANOTHER EXAMPLE FOR COMPARING THE TWO TYPES OF BLENDING :  WALKING ON WATER

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| [http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/mixed/water_good-small.png](http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/mixed/water_good.png) | I first tried to create a man walking on water using poisson blending, but noticed that the pixels in between the mans' legs did not have the same texture as the water. Using mixed blending I saw it produced a much better result as the texture from the water was copied in between the man's legs.   |  |  | | --- | --- | | http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/water-leg-poisson.png | http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/water-leg-mixed.png | | Legs with Poisson Blending | Legs with Mixed Blending | |

|  |  |  |
| --- | --- | --- |
| http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/images/walking3.jpg | http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/images/water-small.jpg | [http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/poisson/water3-small.png](http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15463-f11/www/proj2/www/julenka/poisson/water3.png) |
| **source** | **target** | **copying pixels (water texture not preserved)** |

**4. CODING**

**4.1 MAIN FILE: hw2\_starter.m**

% starter script for hw2

clear;

clc;

close all;

%options to set which functionality to use

DO\_TOY = false;

DO\_BLEND = true;

DO\_MIXED = false;

if DO\_TOY

toyim = im2double(imread('C:\Users\hp\Desktop\toy\_problem.png'));

% im\_out should be approximately the same as toyim

im\_out = toy\_reconstruct(toyim);

disp(['Error: ' num2str(sqrt(sum(toyim(:)-im\_out(:))))])

end

if DO\_BLEND

im\_background =

imresize(im2double(imread('C:\Users\hp\Desktop\HW2\Results\Poisson Results\PenguinChickExample\im2.JPG')), 0.5, 'bilinear');

im\_object = imresize(im2double(imread('C:\Users\hp\Desktop\HW2\Results\Poisson Results\PenguinChickExample\penguin-chick.JPEG')), 0.5, 'bilinear');

% get source region mask from the user

objmask = getMask(im\_object);

% align im\_s and mask\_s with im\_background

[im\_s, mask\_s] = alignSource(im\_object, objmask, im\_background);

im\_blend = Poisson(im\_s, im\_background, mask\_s);

figure(3), hold off, imshow(im\_blend)

end

if DO\_MIXED

im\_background = imresize(im2double(imread('C:\Users\hp\Desktop\HW2\Results\Mixed Blending Results\Handwriting1Example\text2.jpg')), 0.5, 'bilinear');

im\_object = imresize(im2double(imread('C:\Users\hp\Desktop\HW2\Results\Mixed Blending Results\Handwriting1Example\hand.jpg')), 0.5, 'bilinear');

% get source region mask from the user

objmask = getMask(im\_object);

% align im\_s and mask\_s with im\_background

[im\_s, mask\_s] = alignSource(im\_object, objmask, im\_background);

% blend

% im\_blend = poissonBlend(im\_s, mask\_s, im\_background);

im\_blend = Mixed(im\_s, im\_background, mask\_s);

figure(3), hold off, imshow(im\_blend)

end

**4.2 TOY RECONSTRUCTION**

function [ im\_result ] = toy\_reconstruct( input\_image )

%TOY\_RECONSTRUCT Summary of this function goes here.

% Calculate the dimensions of the toy input image and keep a matrix "im2var" that maps each pixel to a variable number.

% Create a sparse matrix A whose dimensions are according to the image whose dimensions we calculated in the above step. Also create a known vector b , initialize its value to 0.

% Next we need to compute v using A and b, for that we fill in the values in A and b using the least squares problem in MATLAB and then finally compute v using lscov(A,b) and then reconstruct the image using the MATLAB function reshape.

[imh, imw , nb] = size( input\_image );

im2var = zeros(imh, imw);

im2var(1:imh\*imw) = 1:imh\*imw;

%expression counter

e = 1;

%sparse matrix

A = sparse(3\*imh , imw);

%known vector b

b = zeros(3\*imh, 1);

%achieving objective 1

for x=1:imh - 1

for y =1:imw

A(e, im2var(x+1,y))=1;

A(e, im2var(x,y))=-1;

b(e) = input\_image(x+1,y)-input\_image(x,y);

e=e+1;

end

end

for x=1:imh

for y =1:imw - 1

A(e, im2var(x,y+1))=1;

A(e, im2var(x,y))=-1;

b(e) = input\_image(x, y+1)-input\_image(x, y);

e=e+1;

end

end

%achieving objective 3

A(e, im2var(1,1))=1;

b(e)=input\_image(1,1);

%calculating the unknown variable v

v = lscov(A,b);

im\_result = reshape( v ,imh , imw);

subplot(1,2,1);

imshow(input\_image);

subplot(1,2,2);

imshow(im\_result);

end

**4.3 getMask.m**

function [mask, poly] = getMask(im)

% [mask, poly] = getMask(im)

% Asks user to draw polygon around input image. Provides binary mask of

% polygon and a chain of all interior boundary points.

disp('Draw polygon around source object in clockwise order, q to stop')

figure(1), hold off, imagesc(im), axis image;

sx = [];

sy = [];

while 1

figure(1)

[x, y, b] = ginput(1);

if b=='q'

break;

end

sx(end+1) = x;

sy(end+1) = y;

hold on, plot(sx, sy, '\*-');

end

mask = poly2mask(sx, sy, size(im, 1), size(im, 2));

if nargout>1

[poly.x, poly.y] = mask2chain(mask);

end

**4.4 mask2chain.m**

function [x, y] = mask2chain\_tmp(mask)

crack\_img = seg2cracks(double(mask));

fragments = cracks2fragments(crack\_img, mask, 1);

x = round(fragments{1}(:, 1));

y = round(fragments{1}(:, 2));

% x = fragments{1}(:, 1);

% y = fragments{1}(:, 2);

[gy, gx] = gradient(double(mask));

epix = (gy.^2+gx.^2>0) & mask;

e = y + (x-1)\*size(crack\_img, 1);

ind = epix(e);

mean(ind)

x = x(ind);

y = y(ind);

figure(1), hold off, plot(x, y), axis image

%% cracks2fragments

function [fragments, junctions, neighbor\_lookups] = cracks2fragments(crack\_img, seg, isolation\_check)

%

% [fragments, junctions, neighbor\_lookups] = cracks2fragments(crack\_img, seg, isolation\_check)

%

if(nargin<3)

isolation\_check = true;

end

[nrows,ncols] = size(crack\_img);

JUNCTION\_BIT = 5;

% Get a lookup table of where the image border is:

onborder = false(nrows,ncols);

onborder(:,[1 end]) = true;

onborder([1 end],:) = true;

num\_neighbors\_lookup = 4\*ones(nrows,ncols);

num\_neighbors\_lookup([1 end],:) = 3;

num\_neighbors\_lookup(:,[1 end]) = 3;

num\_neighbors\_lookup([1 nrows end-nrows+1 end]) = 2;

% Neighbors: up, down, left, right

neighbor\_offsets = [-1 1 -nrows nrows];

% for the up neighbor, i want to check its down bit (3); for the right

% neighbor, i want to check its left bit (4); etc...

check\_bit\_interior = [3 1 2 4];

% Figure out the valid neighbors on the borders

UP = 1; DOWN = 2; LEFT = 3; RIGHT = 4;

valid\_border\_neighbors = cell(nrows,ncols);

[valid\_border\_neighbors{1,:}] = deal([DOWN LEFT RIGHT]);

[valid\_border\_neighbors{end,:}] = deal([UP LEFT RIGHT]);

[valid\_border\_neighbors{:,1}] = deal([UP DOWN RIGHT]);

[valid\_border\_neighbors{:,end}] = deal([UP DOWN LEFT]);

valid\_border\_neighbors{1,1} = [DOWN RIGHT];

valid\_border\_neighbors{1,end} = [DOWN LEFT];

valid\_border\_neighbors{end,1} = [UP RIGHT];

valid\_border\_neighbors{end,end} = [UP LEFT];

% % Find the junctions that aren't on the image border

junction\_map = bitget(crack\_img, JUNCTION\_BIT);

junction\_index = find(junction\_map);

num\_junctions = length(junction\_index);

junction\_map = double(junction\_map);

junction\_map(junction\_index) = 1:num\_junctions;

junction\_fragmentlist = cell(1,num\_junctions);

fragment\_junctionlist = {};

fragment\_segments = {};

segment\_fragments = cell(1,max(seg(:)));

[junction\_y,junction\_x] = ind2sub([nrows ncols], junction\_index);

junctions = [junction\_x(:)+0.5 junction\_y(:)+0.5];

%% Fragment Chaining

not\_in\_fragment = true(nrows,ncols);

fragment\_indices = zeros(1,nrows\*ncols);

fragment\_ctr = 0;

fragments = {};

% % Randomize the order we go through the starting junctions (mainly for

% % debug/display purposes):

% rand\_index = randperm(num\_junctions);

% for(i\_rand=1:num\_junctions)

% i = rand\_index(i\_rand);

for(i = 1:num\_junctions)

% get the neighbors of the starting junction that have not already been

% made part of another fragment and whose orientation/shape match this

% junction

if(~onborder(junction\_index(i)))

junction\_neighbors = junction\_index(i)+neighbor\_offsets;

check\_bit = check\_bit\_interior;

else

junction\_neighbors = junction\_index(i)+neighbor\_offsets(valid\_border\_neighbors{junction\_index(i)});

check\_bit = check\_bit\_interior(valid\_border\_neighbors{junction\_index(i)});

end

neighbor\_bits = bitget(crack\_img(junction\_neighbors), check\_bit);

which\_junction\_neighbors = find(neighbor\_bits & not\_in\_fragment(junction\_neighbors));

% ...

% & ~onborder(junction\_neighbors));% & ~junction\_map(junction\_neighbors));

% remove any neighbors that are themselves junctions and have a lower

% index than the current starting junction. a single-crack-long

% fragment has already been chained between these two junctions in this

% case, so we don't need to do it again.

which\_junction\_neighbors( junction\_map(junction\_neighbors(which\_junction\_neighbors)) & ...

junction\_neighbors(which\_junction\_neighbors) < junction\_index(i) ) = [];

% create a fragment starting from this junction and heading off along

% each available neighbor

for(j = 1:length(which\_junction\_neighbors))

% in case we just closed a loop, don't start marching back out

% around the same loop in the opposite direction:

if(not\_in\_fragment(junction\_neighbors(which\_junction\_neighbors(j))))

% prevent us from coming right back to the starting junction on

% the next step

not\_in\_fragment(junction\_index(i)) = false;

fragment\_ctr = fragment\_ctr + 1;

junction\_fragmentlist{i}(end+1) = fragment\_ctr;

fragment\_junctionlist{fragment\_ctr} = i;

fragment\_length = 1;

fragment\_indices(1) = junction\_index(i);

% record the segmentation numbers on either side of this

% fragment, store as [leftseg rightseg]

if (~onborder(junction\_index(i)))

step\_dir = which\_junction\_neighbors(j);

else

step\_dir = valid\_border\_neighbors{junction\_index(i)}(which\_junction\_neighbors(j));

end

switch(step\_dir)

case(1) % about to move UP

fragment\_segments{fragment\_ctr} = [seg(junction\_index(i)) seg(junction\_index(i)+nrows)];

case(2) % about to move DOWN

fragment\_segments{fragment\_ctr} = [seg(junction\_index(i)+1+nrows) seg(junction\_index(i)+1)];

case(3) % about to move LEFT

fragment\_segments{fragment\_ctr} = [seg(junction\_index(i)+1) seg(junction\_index(i))];

case(4) % about to move RIGHT

fragment\_segments{fragment\_ctr} = [seg(junction\_index(i)+nrows) seg(junction\_index(i)+1+nrows)];

otherwise

error('Invalid neighbor chosen???')

end

neighbors = junction\_neighbors;

which\_neighbors = which\_junction\_neighbors(j);

while(~isempty(which\_neighbors))

% add current position to fragment

fragment\_length = fragment\_length + 1;

fragment\_indices(fragment\_length) = neighbors(which\_neighbors);

% once we've gone more than one step from the start, re-enable

% the possibility of ending up there, in case of a closed

% fragment around an isolated segment

if(fragment\_length==3)

not\_in\_fragment(junction\_index(i)) = true;

end

% if we've reached another junction, add that junction to this

% fragment and stop

if(junction\_map(neighbors(which\_neighbors)))

% Don't add this fragment if this is a closed loop and

% we already have it listed as a member of this

% junction

if(neighbors(which\_neighbors) ~= junction\_index(i))

junction\_fragmentlist{junction\_map(neighbors(which\_neighbors))}(end+1) = fragment\_ctr;

end

fragment\_junctionlist{fragment\_ctr}(2) = junction\_map(neighbors(which\_neighbors));

break;

end

% Otherwise, mark this position as being part of a fragment and

% continue chaining

not\_in\_fragment(fragment\_indices(fragment\_length)) = false;

% get matching neighbors from this point that aren't already

% used in another fragment

crnt = fragment\_indices(fragment\_length);

if(~onborder(crnt))

neighbors = crnt + neighbor\_offsets;

check\_bit = check\_bit\_interior;

else

neighbors = crnt + neighbor\_offsets(valid\_border\_neighbors{crnt});

check\_bit = check\_bit\_interior(valid\_border\_neighbors{crnt});

end

neighbor\_bits = bitget(crack\_img(neighbors), check\_bit);

which\_neighbors = find(neighbor\_bits & not\_in\_fragment(neighbors)); % & ~onborder(neighbors));

end

% Create the fragment from the list of indices. Offset it by 0.5

% pixels in either direction to put it in image coordinates.

[y,x] = ind2sub([nrows ncols], fragment\_indices(1:fragment\_length));

fragments{fragment\_ctr} = [x(:)+0.5 y(:)+0.5];

% If this fragment didn't end at a junction (but instead at the

% image border), add a "junction" for the end point:

if(numel(fragment\_junctionlist{fragment\_ctr})==1)

num\_junctions = num\_junctions + 1;

junctions(num\_junctions,:) = fragments{fragment\_ctr}(end,:);

fragment\_junctionlist{fragment\_ctr}(2) = num\_junctions;

junction\_fragmentlist{num\_junctions} = fragment\_ctr;

end

end

end

% Allow other fragments to end up at this junction (this \_should\_

% already have been set back to true in most cases, but just in

% case...)

not\_in\_fragment(junction\_index(i)) = true;

end

%% Handle isolated regions

if(isolation\_check)

% Look for any segments that have not had any fragments used yet. They

% must be surrounded entirely by one other segment meaning their boundary

% has no junctions. We need

% to add a fragment for that boundary (and some junctions) to our list.

used\_segments = vertcat(fragment\_segments{:});

used\_segments = unique(used\_segments(:));

num\_segments = max(seg(:));

unused\_segments = true(1,num\_segments);

unused\_segments(used\_segments) = false;

unused\_segments = find(unused\_segments);

if(~isempty(unused\_segments))

stats = regionprops(seg, 'PixelIdxList');

segment\_indices = {stats.PixelIdxList};

isolated\_seg = zeros(size(seg));

for k = 1:numel(unused\_segments)

isolated\_seg(segment\_indices{unused\_segments(k)}) = k;

end

num\_isolated\_seg = numel(unused\_segments);

%isolated\_seg(vertcat(segment\_indices{unused\_segments})) = true;

%[isolated\_seg, num\_isolated\_seg] = bwlabel(isolated\_seg, 8);

% Put a false junction on the border of each isolated segment:

for(i=1:num\_isolated\_seg)

% Get the cracks for this isolated segment

isolated\_cracks = seg2cracks(isolated\_seg==i);

% Put a false junction indicator on the border of the isolated

% segment

crack\_index = find(isolated\_cracks>0);

isolated\_cracks(crack\_index(1)) = bitset(isolated\_cracks(crack\_index(1)), JUNCTION\_BIT);

% Chain the fragments around this isolated segment

[temp\_fragments, temp\_junctions, temp\_neighbor] = ...

cracks2fragments(isolated\_cracks, seg, false);

% Concatenate the results with the existing results, adjusting

% fragment and junction numbering as necessary

fragments = [fragments temp\_fragments];

junctions = [junctions; temp\_junctions];

for(j=1:numel(temp\_neighbor.junction\_fragmentlist))

temp\_neighbor.junction\_fragmentlist{j} = temp\_neighbor.junction\_fragmentlist{j} + fragment\_ctr;

end

junction\_fragmentlist = [junction\_fragmentlist temp\_neighbor.junction\_fragmentlist];

for(j=1:numel(temp\_neighbor.fragment\_junctionlist))

temp\_neighbor.fragment\_junctionlist{j} = temp\_neighbor.fragment\_junctionlist{j} + num\_junctions;

end

fragment\_junctionlist = [fragment\_junctionlist temp\_neighbor.fragment\_junctionlist];

fragment\_segments = [fragment\_segments temp\_neighbor.fragment\_segments];

update\_segments = ~cellfun('isempty', temp\_neighbor.segment\_fragments);

for(j=find(update\_segments))

segment\_fragments{j} = [segment\_fragments{j} temp\_neighbor.segment\_fragments{j}+fragment\_ctr];

end

% Don't want to update these too soon!

fragment\_ctr = fragment\_ctr + numel(temp\_fragments);

junction\_ctr = num\_junctions + size(temp\_junctions,1);

end

end

end

%% Output

neighbor\_lookups.junction\_fragmentlist = junction\_fragmentlist;

neighbor\_lookups.fragment\_junctionlist = fragment\_junctionlist;

neighbor\_lookups.fragment\_segments = fragment\_segments;

neighbor\_lookups.segment\_fragments = segment\_fragments;

% Compute edge adjacency info

num\_fragments = fragment\_ctr;

FORWARD = 2; REVERSE = 1;

directed\_neighbors = cell(2\*num\_fragments,1);

for(i=1:num\_fragments)

% Get forward neighbors

directed\_neighbors{i} = get\_neighbors(i, neighbor\_lookups, FORWARD, num\_fragments);

% Get reverse neighbors

directed\_neighbors{i+num\_fragments} = get\_neighbors(i, neighbor\_lookups, REVERSE, num\_fragments);

end

neighbor\_lookups.directed\_neighbors = directed\_neighbors;

% DEBUG CODE:

% % Check that we're not still having the problem where a fragment has the

% % same superpixel listed on both sides.

% temp = vertcat(fragment\_segments{:});

% if(any(temp(:,1)==temp(:,2)))

% warning('At least one fragment has the same superpixel listed for left and right!')

% keyboard

% end

return;

%% Helper: Find directed neighbors

function neighbors = get\_neighbors(fragment, neighbor\_data, direction, num\_fragments)

% direction==2 -> forward

% direction==1? -> reverse

% Get all the foward neighbors for this fragment:

junction = neighbor\_data.fragment\_junctionlist{fragment}(direction);

neighbors = neighbor\_data.junction\_fragmentlist{junction};

neighbors(neighbors==fragment) = [];

if(~isempty(neighbors))

% Any neighbors that connect to this in the wrong direction should be

% referenced to the reverse fragment

neighbor\_junctions = vertcat(neighbor\_data.fragment\_junctionlist{neighbors});

to\_reverse = (neighbor\_junctions(:,2)==junction);

neighbors(to\_reverse) = neighbors(to\_reverse) + num\_fragments;

end

%% seg2cracks

function [crack\_img] = seg2cracks(seg)

%

% [crack\_img] = seg2cracks(seg)

%

% Converts a segmentation image into a crack-coded image.

%

% Bits:

% |

% | 1

% 4 ----+---- 2

% |

% | 3

% Values:

% tic

UP = 1;

RIGHT = 2;

DOWN = 3;

LEFT = 4;

JUNCTION = 5;

% dx = uint8(seg ~= image\_right(seg));

% dy = uint8(seg ~= image\_down(seg) );

% crack\_img = dx + bitshift(dy,LEFT-1) + ...

% bitshift(image\_right(dy),RIGHT-1) + bitshift(image\_down(dx),DOWN-1);

% Removed dependency on image\_down and image\_right, to make this more

% easily packaged:

dx = uint8(seg ~= seg(:,[2:end end]));

dy = uint8(seg ~= seg([2:end end],:));

crack\_img = dx + bitshift(dy,LEFT-1) + ...

bitshift(dy(:,[2:end end]),RIGHT-1) + bitshift(dx([2:end end],:),DOWN-1);

% Find interior junctions:

junction\_map = (crack\_img==11 | crack\_img==7 | crack\_img==14 | ...

crack\_img==13 | crack\_img==15);

% Find the junctions along the borders:

junction\_map([1 end],:) = junction\_map([1 end],:) | bitget(crack\_img([1 end],:),UP);

junction\_map(:,[1 end]) = junction\_map(:,[1 end]) | bitget(crack\_img(:,[1 end]),LEFT);

% set the junction bit for all these junctions in the crack\_img

crack\_img(junction\_map) = bitset(crack\_img(junction\_map), JUNCTION);

**4.5 alignSource.m**

function [im\_s2, mask2] = alignSource(im\_s, mask, im\_t)

% Asks user for bottom-center position and outputs an aligned source image.

% im\_s represents the source image - here penguin

% mask represents the mask created from the pixels selected - here the shape of the penguin

% im\_t is the target image which has to be used as a background

% im\_s2 represents the space created in the shape of penguin with masked background

% mask2 represents the white area created on the black masked background

figure(1), hold off, imagesc(im\_s), axis image

figure(2), hold off, imagesc(im\_t), axis image

[y, x] = find(mask);

y1 = min(y)-1; y2 = max(y)+1; x1 = min(x)-1; x2 = max(x)+1;

im\_s2 = zeros(size(im\_t));

disp('choose target bottom-center location')

[tx, ty] = ginput(1);

yind = (y1:y2);

yind2 = yind - max(y) + round(ty);

xind = (x1:x2);

xind2 = xind - round(mean(x)) + round(tx);

y = y - max(y) + round(ty);

x = x - round(mean(x)) + round(tx);

ind = y + (x-1)\*size(im\_t, 1);

mask2 = false(size(im\_t, 1), size(im\_t, 2));

mask2(ind) = true;

im\_s2(yind2, xind2, :) = im\_s(yind, xind, :);

im\_t(repmat(mask2, [1 1 3])) = im\_s2(repmat(mask2, [1 1 3]));

figure(1), hold off, imagesc(im\_s2), axis image;

figure(2), hold off, imagesc(im\_t), axis image;

drawnow;

**4.6 Poisson.m**

function [imNew] = Poisson(src\_Img, dest\_Img, mask\_Img)

%Function to achieve poisson blending

% src\_Img - source image

% dest\_Img - destination image

% mask\_Img - a black and white mask to indicate the irregular boundary

% posOffset - offset of corresponding pixel from the source to the destination

% source image must be smaller than or equal to the source

%Using the same principle as used in toy reconstruction , we will use a sparse matrix A and a known vector b which we are going to compute

%using the gradient pixel values of the source image and intensity of the target image

%Creating sparce matrix A, find all those regions in the mask\_Img which are represented by 1 and store its value in n

n = size(find(mask\_Img), 1);

assignin('base', 'num', n);

%Here our n is equivalent to our number of unknowns

fprintf('Number of Unknowns = %d\n', n);

%spalloc is matlab function used for allocating space for sparse matrix, creates an all zero sparse matrix S of size m-by-n with room to hold nzmax nonzeros

A = spalloc(n, n, 5\*n);

% initialized b to contain all 0s

b = zeros(3, n);

%decompose or divide the image into its rgb channels and store each of the channels in R , G and B part of the destination or the background or target image

[destImg\_R destImg\_G destImg\_B] = decomposeRGB(dest\_Img);

%similarly dividing the source image or the image to be copied into its RGB channels, it is basically calling decomposeRGB function defined in decomposeRGB.m

[srcImg\_R srcImg\_G srcImg\_B] = decomposeRGB(src\_Img);

% computation of the height and width of both the source image and the destination image and storing them in resepctive variables

[srcImg\_height srcImg\_width] = size(srcImg\_R);

[destImg\_height destImg\_width] = size(destImg\_R);

tempImIndex = zeros(destImg\_height, destImg\_width);

ctr = 0;

% Main function to fill in those values of the destination or the target image where the mask is 1 , that is we will be filling this area from backgrond image

for y = 1:destImg\_height

for x = 1:destImg\_width

if mask\_Img(y, x) ~= 0

ctr = ctr + 1;

tempImIndex(y, x) = ctr;

end

end

end

%computing the lapcian of the coordinate images

laplacian = [0 1 0; 1 -4 1; 0 1 0];

%compute the convolution of the source image , doing a convolution of a 2D matrix source image gives you the central part of the convolution of same size as

%input src image channels RGB

imLaplacian\_R = conv2(srcImg\_R, -laplacian, 'same');

imLaplacian\_G = conv2(srcImg\_G, -laplacian, 'same');

imLaplacian\_B = conv2(srcImg\_B, -laplacian, 'same');

%similalry compute the same for the target image

imLaplacianDest\_R = conv2(destImg\_R, -laplacian, 'same');

imLaplacianDest\_G = conv2(destImg\_G, -laplacian, 'same');

imLaplacianDest\_B = conv2(destImg\_B, -laplacian, 'same');

ctr = 0;

%now we will iterate over the entire image and see where all in the image mask are the values 1, keep iterating over it, since the image mask is of the same

%size as the target image and hence we can use the same dimensions

for y = 2:destImg\_height-1

for x = 2:destImg\_width-1

if mask\_Img(y, x) ~= 0

ctr = ctr + 1;

%keep storing these x,y values where we have 1 set in the mask\_Image

ydest = y;

xdest = x;

% now we are done checking the pixel, we have to concentrate on the 4 neighbours of that pixel so that we can keep track of the gradient values

% Pixel to the top of x,y

if mask\_Img(y-1, x) ~= 0

column\_index = tempImIndex(ydest-1, xdest);

A(ctr, column\_index) = -1;

else % consider the region which is black in the mask, but will form part of the target image

b(1, ctr) = b(1, ctr) + destImg\_R(ydest-1, xdest);

b(2, ctr) = b(2, ctr) + destImg\_G(ydest-1, xdest);

b(3, ctr) = b(3, ctr) + destImg\_B(ydest-1, xdest);

end

% Pixel to the left of x,y

if mask\_Img(y, x-1) ~= 0

column\_index = tempImIndex(ydest, xdest-1);

A(ctr, column\_index) = -1;

else % consider the region which is black in the mask, but will form part of the target image at the left border

b(1, ctr) = b(1, ctr) + destImg\_R(ydest, xdest-1);

b(2, ctr) = b(2, ctr) + destImg\_G(ydest, xdest-1);

b(3, ctr) = b(3, ctr) + destImg\_B(ydest, xdest-1);

end

% Pixel to the right of x,y

if mask\_Img(y, x+1) ~= 0

column\_index = tempImIndex(ydest, xdest+1);

A(ctr, column\_index) = -1;

else % % consider the region which is black in the mask, but will form part of the target image at the right border

b(1, ctr) = b(1, ctr) + destImg\_R(ydest, xdest+1);

b(2, ctr) = b(2, ctr) + destImg\_G(ydest, xdest+1);

b(3, ctr) = b(3, ctr) + destImg\_B(ydest, xdest+1);

end

% Pixel to the bottom of x,y

if mask\_Img(y+1, x) ~= 0

column\_index = tempImIndex(ydest+1, xdest);

A(ctr, column\_index) = -1;

else % consider the region which is black in the mask, but will form part of the target image at the bottom border

b(1, ctr) = b(1, ctr) + destImg\_R(ydest+1, xdest);

b(2, ctr) = b(2, ctr) + destImg\_G(ydest+1, xdest);

b(3, ctr) = b(3, ctr) + destImg\_B(ydest+1, xdest);

end

%represents the 4 border neighbours of any pixel value and hence intialzied to 4 in the sparse matrix

A(ctr, ctr) = 4;

% construct the vector b depending on each pixel x,y and using the computation done before

b(1, ctr) = b(1, ctr) + imLaplacian\_R(y, x);

b(2, ctr) = b(2, ctr) + imLaplacian\_G(y, x);

b(3, ctr) = b(3, ctr) + imLaplacian\_B(y, x);

end

end

end

%solve the variables as we did in toy reconstruction like A\*(xR)=b

xR = A\( b(1,:)' );

xG = A\( b(2,:)' );

xB = A\( b(3,:)' );

%store the computed R G B values in new respectives variable

imComputedR = destImg\_R;

imComputedG = destImg\_G;

imComputedB = destImg\_B;

% for each of the pixel for the dimensions of the entire target image, fill in the above computed values

for y1 = 1:destImg\_height

for x1 = 1:destImg\_width

if mask\_Img(y1, x1) ~= 0

index = tempImIndex(y1, x1);

imComputedR(y1, x1) = xR(index);

imComputedG(y1, x1) = xG(index);

imComputedB(y1, x1) = xB(index);

end

end

end

%compute the complete RGB image using the composeRGB.m

imNew = composeRGB(imComputedR, imComputedG, imComputedB);

**4.7 mixed.m**

function [imNew] = Mixed(src\_Img, dest\_Img, mask\_Img)

%Function to achieve mixed blending

% src\_Img - source image

% dest\_Img - destination image

% mask\_Img - a black and white mask to indicate the irregular boundary

% posOffset - offset of corresponding pixel from the source to the destination

% source image must be smaller than or equal to the source

%Using the same principle as used in toy reconstruction , we will use a sparse matrix A and a known vector b which we are going to compute

%using the gradient pixel values of the source image and intensity of the target image but here we will use the value of the source or target image,

%whichever is the higher value

%Creating sparce matrix A, find all those regions in the mask\_Img which are represented by 1 and store its value in n

n = size(find(mask\_Img), 1);

assignin('base', 'num', n);

%Here our n is equivalent to our number of unknowns

fprintf('Number of Unknowns = %d\n', n);

%spalloc is matlab function used for allocating space for sparse matrix, creates an all zero sparse matrix S of size m-by-n with room to hold nzmax nonzeros

A = spalloc(n, n, 5\*n);

% initialized b to contain all 0s

b = zeros(3, n);

%decompose or divide the image into its rgb channels and store each of the channels in R , G and B part of the destination or the background or target image

[destImg\_R destImg\_G destImg\_B] = decomposeRGB(dest\_Img);

%similarly dividing the source image or the image to be copied into its RGB channels, it is basically calling decomposeRGB function defined in decomposeRGB.m

[srcImg\_R srcImg\_G srcImg\_B] = decomposeRGB(src\_Img);

% computation of the height and width of both the source image and the destination image and storing them in resepctive variables

[srcImg\_height srcImg\_width] = size(srcImg\_R);

[destImg\_height destImg\_width] = size(destImg\_R);

tempImIndex = zeros(destImg\_height, destImg\_width);

count = 0;

% now fill in the

for y = 1:destImg\_height

for x = 1:destImg\_width

if mask\_Img(y, x) ~= 0

count = count + 1;

tempImIndex(y, x) = count;

end

end

end

laplacian = [0 1 0; 1 -4 1; 0 1 0];

laplacianDest = [0 1 0; 1 -4 1; 0 1 0];

%compute the convolution of the source image , doing a convolution of a 2D matrix source image gives you the central part of the convolution of same size as

%input src image channels RGB

imLaplacianR = conv2(srcImg\_R, -laplacian, 'same');

imLaplacianG = conv2(srcImg\_G, -laplacian, 'same');

imLaplacianB = conv2(srcImg\_B, -laplacian, 'same');

imLaplacianDestR = conv2(destImg\_R, -laplacianDest, 'same');

imLaplacianDestG = conv2(destImg\_G, -laplacianDest, 'same');

imLaplacianDestB = conv2(destImg\_B, -laplacianDest, 'same');

% [sgxR,sgyR] = gradient(srcImg\_R);

% imLaplacianR = sqrt(sgxR.^2 + sgyR.^2);

%

% [sgxG,sgyG] = gradient(srcImg\_G);

% imLaplacianG = sqrt(sgxG.^2 + sgyG.^2);

%

% [sgxB,sgyB] = gradient(srcImg\_B);

% imLaplacianB = sqrt(sgxB.^2 + sgyB.^2);

%

%

% [dgxR,dgyR] = gradient(destImg\_R);

% imLaplacianDestR = sqrt(dgxR.^2 + dgyR.^2);

%

% [dgxG,dgyG] = gradient(destImg\_G);

% imLaplacianDestG = sqrt(dgxG.^2 + dgyG.^2);

%

% [dgxB,dgyB] = gradient(destImg\_B);

% imLaplacianDestB = sqrt(dgxB.^2 + dgyB.^2);

% matrix row count

count = 0; % count is the row index

for y = 2:destImg\_height-1

for x = 2:destImg\_width-1

% if the mask is not zero, then add to the matrix

if mask\_Img(y, x) ~= 0

% increase the counter

count = count + 1;

%keep storing these x,y values where we have 1 set in the mask\_Image

ydest = y;

xdest = x;

% gathering the coefficient for the matrix, checking Neighbours

% Pixel to the top of x,y

if mask\_Img(y-1, x) ~= 0

colIndex = tempImIndex(ydest-1, xdest);

A(count, colIndex) = -1;

else % consider the region which is black in the mask, but will form part of the target image

b(1, count) = b(1, count) + destImg\_R(ydest-1, xdest);

b(2, count) = b(2, count) + destImg\_R(ydest-1, xdest);

b(3, count) = b(3, count) + destImg\_R(ydest-1, xdest);

end

% Pixel to the left of x,y

if mask\_Img(y, x-1) ~= 0

colIndex = tempImIndex(ydest, xdest-1);

A(count, colIndex) = -1;

else % consider the region which is black in the mask, but will form part of the target image at the left border

b(1, count) = b(1, count) + destImg\_R(ydest, xdest-1);

b(2, count) = b(2, count) + destImg\_G(ydest, xdest-1);

b(3, count) = b(3, count) + destImg\_B(ydest, xdest-1);

end

% Pixel to the right of x,y

if mask\_Img(y, x+1) ~= 0

colIndex = tempImIndex(ydest, xdest+1);

A(count, colIndex) = -1;

else % % consider the region which is black in the mask, but will form part of the target image at the right border

b(1, count) = b(1, count) + destImg\_R(ydest, xdest+1);

b(2, count) = b(2, count) + destImg\_G(ydest, xdest+1);

b(3, count) = b(3, count) + destImg\_B(ydest, xdest+1);

end

% Pixel to the bottom of x,y

if mask\_Img(y+1, x) ~= 0

colIndex = tempImIndex(ydest+1, xdest);

A(count, colIndex) = -1;

else % consider the region which is black in the mask, but will form part of the target image at the bottom border

b(1, count) = b(1, count) + destImg\_R(ydest+1, xdest);

b(2, count) = b(2, count) + destImg\_G(ydest+1, xdest);

b(3, count) = b(3, count) + destImg\_B(ydest+1, xdest);

end

A(count, count) = 4;

% if imLaplacianR(y, x) > imLaplacianDestR(y, x)

% % b(1, count) = b(1, count) + imLaplacianR(y, x);

% b(1, count) = imLaplacianR(y, x);

% else

% % b(1, count) = b(1, count) + imLaplacianDestR(y, x);

% b(1, count) = imLaplacianDestR(y, x);

% end

%

% if imLaplacianG(y, x) > imLaplacianDestG(y, x)

% % b(2, count) = b(2, count) + imLaplacianG(y, x);

% b(2, count) = imLaplacianG(y, x);

% else

% % b(2, count) = b(2, count) + imLaplacianDestG(y, x);

% b(2, count) = imLaplacianDestG(y, x);

% end

%

% if imLaplacianB(y, x) > imLaplacianDestB(y, x)

% % b(3, count) = b(3, count) + imLaplacianB(y, x);

% b(3, count) = imLaplacianB(y, x);

% else

% % b(3, count) = b(3, count) + imLaplacianDestB(y, x);

% b(3, count) = imLaplacianDestB(y, x);

% end

% construct the guidance field

% using the average of the pixel gradient values of the target and the source image to get mixed blending

b(1, count) = b(1, count) + 0.5\*imLaplacianR(y, x) + 0.5\*imLaplacianDestR(y,x);

b(2, count) = b(2, count) + 0.5\*imLaplacianG(y, x) + 0.5\*imLaplacianDestG(y,x);

b(3, count) = b(3, count) + 0.5\*imLaplacianB(y, x) + 0.5\*imLaplacianDestB(y,x); end

end

end

%---------------------------------------------

% solve for the sparse matrix

%---------------------------------------------

% xR = A\( b(1,:)' );

% xG = A\( b(2,:)' );

% xB = A\( b(3,:)' );

xR = lscov(A ,b(1,:)' );

xG = lscov(A ,b(2,:)' );

xB = lscov(A ,b(3,:)' );

%---------------------------------------------

% now fill in the solved values

%---------------------------------------------

imNewR = destImg\_R;

imNewG = destImg\_G;

imNewB = destImg\_B;

% now fill in the

for y1 = 1:destImg\_height

for x1 = 1:destImg\_width

if mask\_Img(y1, x1) ~= 0

index = tempImIndex(y1, x1);

imNewR(y1, x1) = xR(index);

imNewG(y1, x1) = xG(index);

imNewB(y1, x1) = xB(index);

end

end

end

imNew = composeRGB(imNewR, imNewG, imNewB);

**5. CONCLUSION**

A source image was integrated/ blended into a target image making sure that there was seamless blending between the two so that the final image that we obtained looked like a single image.

The focus was to maintain the gradient of the source image to be copied rather than keeping our focus on the intensity.

**6. BIBLIOGRAPHY**

* *MATLAB products* [online] available : <http://www.mathworks.in/products/matlab/>
* *MATLAB Wikipedia* [online] link available :<http://en.wikipedia.org/wiki/MATLAB>
* PÉREZ P., GANGNET M., BLAKE A.: Poisson image  
  editing. TOG *(SIGGRAPH ’03) 22 (2003), 313–318.*
* A Levin, A Zomet, S Peleg, Y Weiss - *Seamless image stitching in the gradient domain . Computer Vision-ECCV 2004, 2004 – Springer.*

**7. APPENDIX**

**7.1 SCREENSHOTS**

**Step 1 : Selecting the Source and the Target Image**

****  ****

(a) Source Image (b) Target Image

Figure 8 : Images to be blended together

**Step 2: Selecting the part of the Source Image to be blended into Target Image using Boundary points**

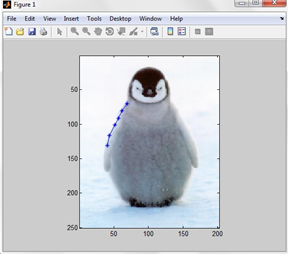


Figure 9 : Selecting points on the boundaries of the penguin

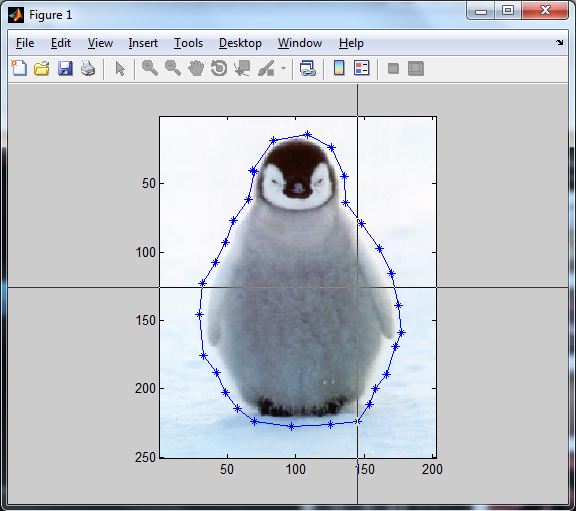


Figure 10 : Finally selected Source Image

**Step 3 : Masking**

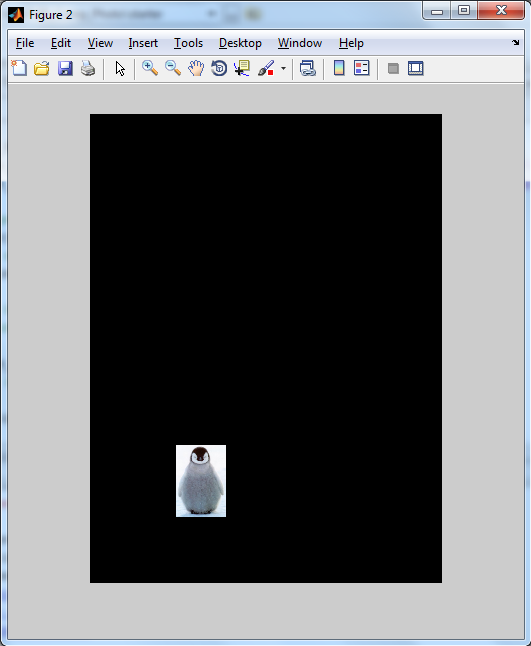
****

Figure 11 : Masking in process

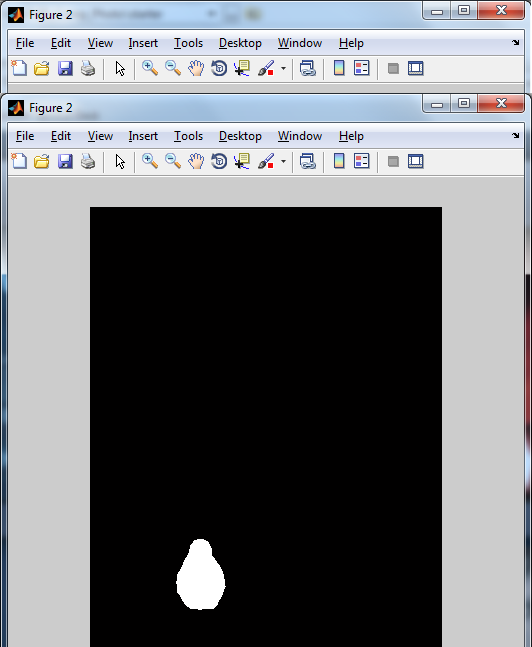
****

Figure 12 : Masked up image

**Step 4 : Click 'q' on the keyboard to exit the point selection process and this displays the Target Image where we click on the point so as to get our Source placed on the Target Image.**

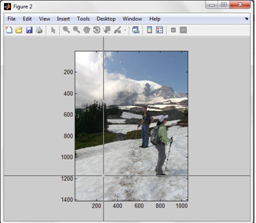
****

Figure 13 : Target Image point selection

**Step 5 : Finally blended Image is ready.**



Figure 14 : Final blended Image

**A Few other examples**

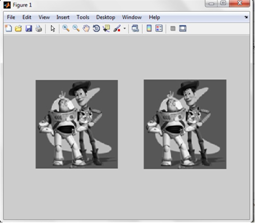
****

Figure 15 : Toy problem



Figure 16 : Source Image for poisson blending



Figure 17 : Target image for Poisson Blending



Figure 18 : Poisson Blending (Final Image )



Figure 19 : Mixed Blending



Figure 20 : Poisson Blending