Poisson Hole Filling in ITK

Release 0.00

David Doria

March 12, 2011

Rensselaer Polytechnic Institute, Troy NY

Abstract

This code provides an implementation of two techniques from "Poisson Image Editing" on ITK images. First, we fill a hole in an image given only the pixel values on the boundary. Second, we copy a patch of an image into another image and make the result as smooth as possible. We also explain how these techniques can be used to reconstruct an image after perform editing operations in the gradient domain.

Latest version available at the Insight Journal [http://hdl.handle.net/10380/3253]

Contents

1 Introduction

This code provides an implementation of two techniques from "Poisson Image Editing" ([1]) on ITK images. First, we fill a hole in an image given only the pixel values on the boundary. Second, we copy a patch of an image into another image and make the result as smooth as possible.

2 Hole Filling

As a motivating example of the power of Poisson methods in image editing, we attempt to fill in a missing region in an image in a plausible way. In this example, our goal is to remove the jet from the image in Figure 1(a). To do this, we must manually specify the region to be removed (shown in Figure 1(b)). The result of the filling is shown in Figure 1(c).

(a) Original image (b) Mask (c) Filled image

Figure 1: A demonstration of Poisson hole filling

2.1 Input

The inputs to this algorithm are:

- An image, I.
- A mask. Non-zero pixels indicate the region to fill. The mask must be the same size as the image, and must not have non-zero pixels on its border.

2.2 Concept

It has been shown using Calculus of Variations that the best solution for the region inside the hole, H, is given by the solution to

$$\nabla^2 H = 0 \tag{1}$$

while ensuring $H = I$ on the boundary of the hole. This setup is a Laplace equation with Dirichlet (aka first order) boundary conditions.

2.3 Discrete Solution

The Laplace operator is given by

$$\nabla^2 f = \sum_{i=1}^{n} \frac{\partial^2 f}{\partial x_i^2} \tag{2}$$

A discrete approximation to this function at a pixel (x,y) is given by

$$\nabla^2 f(x,y) \approx f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y) \tag{3}$$

Another way to write this is to multiply the pixel and its neighbors by a kernel:

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{4}$$

From this, for each unknown pixel $u_{i,j}$, the equation is:

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} = 0 \tag{5}$$

To solve the Laplace equation over the entire image, we can write a linear system of equations. We create a variable for every pixel to be filled. If the pixels are vector-valued (e.g. an RGB image), N of these systems must be solved independently (where N is the dimensionality of each pixel).

A sparse matrix U is constructed row by row, one row per variable. In each row, the value in the Laplacian kernel corresponding to the pixel's location relative to the current variable pixel is placed in the column corresponding to the variable id. When one of the pixels appearing in Equation 5 is on the border of the hole (and is therefore known), $u_{(\cdot,\cdot)}$ is replaced with $p_{(\cdot,\cdot)}$, the value of the pixel from the original image. In this case, rather than place the value of the Laplacian kernel in the U matrix, we instead multiply it by the image value and subtract the result from the corresponding entry of the b vector. That is, we move the known value to the right side of the equation.

A vector H_v, the vectorized version of the solution to the set of hole pixels, is created as the unknown vector to be solved in a system of equations.

The linear system is then

$$U^T H_v = b \tag{6}$$

After solving for H_v, the resulting H_v is then remapped back to the pixels corresponding to each variable id to construct H. U is incredibly sparse, so a sparse solver should definitely be used.

2.4 Code Snippet

We have packaged this implementation in a class called *PoissonEditing*. Using this class is very straight forward. The *Image* and *Mask* must be set. The *FillMaskedRegion*() function does the work. If the Image consists of vector-valued pixels, it is internally decomposed, filled, recombined and returned by the *GetOutput*() function.

```
PoissonEditing<FloatVectorImageType> poissonEditing;
poissonEditing.SetImage(imageReader->GetOutput());
poissonEditing.SetMask(maskReader->GetOutput());
poissonEditing.FillMaskedRegion();

FloatVectorImageType::Pointer outputImage = poissonEditing.GetOutput();

Helpers::ClampImage<FloatVectorImageType>(outputImage);
Helpers::CastAndWriteImage<FloatVectorImageType>(outputImage, "output.png");
```

3 Region Copying

In the problem of region copying (aka seamless cloning), we are interested in copying a region from one image into another image in a visually pleasing way. This is again best motivated with an example. In our example, the goal is to copy the jet from Figure 2(a) into the image of the canyon shown in Figure 2(c). This source and target pair along with the mask and the result are shown in 2.

(a) Original image (b) Mask (c) Target Image (d) Resulting Image

Figure 2: A demonstration of Poisson region copying

3.1 Input

The inputs to this algorithm are:

- A source image, S.
- A target image, T.
- A mask, M. Non-zero pixels indicate the region to fill. The mask must be the same size as the image, and must not have non-zero pixels on its border.

3.2 Concept

In [1], it was argued that a good way to copy a region of one image into another image is to do something very similar to hole filling, but additionally introduce a "guidance field", G. That is, the way to copy a region of one image into another is to solve the equation

$$\nabla^2 H = div(G) \tag{7}$$

The boundary condition this time is again first order and specifies that the resulting H be equal to the target image, T, at the hole boundary.

The suggested guidance field G is the gradient of the source image. That is,

$$G = \nabla S \tag{8}$$

In this case, the right hand side of the equation has become $div(\nabla S)$, which is exactly the Laplacian of S.

3.3 Discrete Solution

Just as before, the discrete Laplacian equation is written for each pixel, but this time the right hand side is set to $\nabla^2 S(i,j)$. When the right side of this equation is non-zero, it is referred to as a Poisson equation.

$$-4u_{i,j} + u_{i+1,j} + u_{i-1,j} + u_{i,j+1} + u_{i,j-1} \approx \nabla^2 S(i,j) \tag{9}$$

The linear system

$$U^T H_v = b \tag{10}$$

is created and solved identically as before.

3.4 Code Snippet

We have packed this functionality into a class called *PoissonCloning*. Using this class is very straight forward. In fact, *PoissonCloning* derives from *PoissonEditing* and the only addition is that *PoissonCloning* provides a *SetTargetImage* function. The rest of the functionality is identical.

```
PoissonCloning<FloatVectorImageType> poissonCloning;
poissonCloning.SetImage(sourceImageReader->GetOutput());
poissonCloning.SetTargetImage(targetImageReader->GetOutput());
poissonCloning.SetMask(maskReader->GetOutput());
poissonCloning.PasteMaskedRegionIntoTargetImage();

FloatVectorImageType::Pointer outputImage = poissonCloning.GetOutput();

Helpers::ClampImage<FloatVectorImageType>(outputImage);
Helpers::CastAndWriteImage<FloatVectorImageType>(outputImage, "output.png");
```

4 Manipulation in the Gradient Domain

There are tricks that can be done by performing operations to the derivatives of an image. The goal of such manipulations is often to produce an effect in the original image. To get back to the image from the gradient, one must find the least squares solution to a system of equations involving the derivatives of the image (see Section 4.1). However, this technique is not often used. More commonly, the derivatives are recombined into the Laplacian and the reconstruction is done using the methods described in previous sections. We will outline both techniques below.

4.1 Reconstructing an Image from Its Derivatives

For the task of reconstructing an image directly from its derivatives, we can use a very similar method to solving the Laplace equation. This time, however, there are two equations to be solved simultaneously:

$$\frac{d}{dx}(U) = D_x \tag{11}$$

$$\frac{d}{dy}(U) = D_y \tag{12}$$

where D_x and D_y are the known derivative images. Of course the first order boundary conditions must again be known. This time the boundary is the actual border of the image.

We can construct the same type of linear system to solve for the components of U that best satisfy both equations. Any discrete derivative operator can be used. We have chosen to use the Sobel operators:

$$S_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{13}$$

$$S_y = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix} \tag{14}$$

By applying these operators to every pixel in the unknown image U, we can write a system of equations, two for each pixel:

$$-u_{i-1,j-1} - 2u_{i-1,j} - u_{i-1,j+1} + u_{i+1,j+1} + 2u_{i+1,j} + u_{i,j+1} = D_x(i,j) \tag{15}$$

$$-u_{i-1,j-1} - 2u_{i,j-1} - u_{i+1,j+1} + u_{i-1,j-1} + 2u_{i,j+1} + u_{i+1,j+1} = D_y(i,j) \tag{16}$$

Again, we simply place the coefficient of each term in the column of the matrix corresponding the variable id of the pixel. As usual, we move the term to the right side of the equation (to contribute to the b vector) if the pixel is known. Figure 3 shows an image, its derivatives, and the image reconstructed using only the border of the image in 3(a) and the derivatives.

(a) Original image (b) D_x (c) D_y (d) Resulting Image

Figure 3: A demonstration of reconstructing an image from its derivatives

4.2 Reconstructing an Image from Its Laplacian

The technique to reconstruct an image from its Laplacian is identical to the procedure described in Section 3. We specify the Laplacian in the unknown region as the guidance field. If the manipulation was done

directly on the Laplacian, this process was straight forward. However, if the manipulation was done on the derivative images, there is an additional step of going from the derivatives to the Laplacian that requires special attention.

The Laplacian is defined as the divergence of the gradient:

$$\nabla^2 I = div(\nabla I) \tag{17}$$

Here, we have the gradient

$$\nabla I(i,j) = (D_x(i,j), D_y(i,j)) \tag{18}$$

so the Laplacian is

$$\nabla I^2(i,j) = \frac{\partial D_x(i,j)}{\partial x} + \frac{\partial D_y(i,j)}{\partial y} \tag{19}$$

That is, we must take the x derivative of the x derivative that we already have, and the y derivative of the y derivative that we already have, and add them together. NOTE: the discrete operators that you choose is critically important. If you use the 5 points Laplacian operator

$$\begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{20}$$

you MUST use the forward difference operator to take the first set of derivatives (which produce the input to the algorithm):

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \tag{21}$$

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{22}$$

and the backward difference operator to take the second derivatives:

$$\begin{pmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{23}$$

$$\begin{pmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{24}$$

It possible to choose different operators, but you must choose the derivative operators so that if they are applied successively they produce exactly the Laplacian operator that you choose. If you do not do this, the result of the reconstruction will be wildly incorrect.

5 Notes

5.1 Clamping outputs

It is important to note that everywhere in this document the linear systems have been solved with no constraints. This allows the resulting image to take invalid values (i.e. greater than 255 or less than 0). The

resulting images must therefore be clamped or scaled to ensure the output has an appropriate range.

5.2 Efficiency

The hole filling example in this document took 1m30s using VNL's sparse solver (from the VXL library) on a single core machine. However, when it was replaced with the UMFPACK interface provided by Eigen, and identical result was obtained in only 0m1.5s. The VNL team is working to provide a way to specify a tuned BLAS library through CMake to achieve a more comparable result.

References

[1] P. Perez, M. Gangnet, and A. Blake. Poisson image editing. *ACM Transactions on Graphics*, 22(3):313–318, 2003. 1, 3.2