Poisson Hole Filling in ITK and VNL

Release 0.00

David Doria

Rensselaer Polytechnic Institute, Troy NY

March 5, 2011

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.

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

The idea here is to fill in a missing region in an image in a plausible way. This is best motivated with an example. In Figure 1, the goal is to remove the plane from the image in Figure 1(a). To do this, we simply 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 class 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 Dirchlet (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

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

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 component of every pixel to be filled. That is, if there are N pixels to be filled and each pixel consists of C components (i.e. 3 in the case of an RGB image), there are NC variables in the linear system.

A sparse matrix U is constructed row by for, one row per variable. In each row, a 4 is placed in the column corresponding to the variable id. A -1 is placed in the column corresponding to the variable id of any non-border 4-connected neighbor.

When one of the pixels appearing in equation 9 is on the border of the hole (and is therefore known), $u_{..}$ is replaced with $p_{..}$, the value of the pixel from the original image. In this case, a -1 is not placed in the U matrix, but instead the value is moved to the right side of the equation to construct the b vector, the right hand side of the linear system equation.

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

The linear system is then

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

is then solved for H_v. The resulting H_v is then remapped back to the pixels corresponding to each variable id to construct H.

2.4 Code Snippet

Using this class is very straight forward, as shown below:

```
PoissonEditing poissonEditing;
poissonEditing.SetImage(imageReader->GetOutput());
```

```
poissonEditing.SetMask(maskReader->GetOutput());
poissonEditing.FillRegion(outputImage);

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 Figure 2, the goal is to copy the plane from the image with the sky background into the image of the canyon.

(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 class 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 do 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 the $\nabla S(i, j)$.

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

The linear system

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

is created and solved identically as before.

3.4 Code Snippet

Using this class is very straight forward, as shown below:

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

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

4 Note

It is important to note that in both problems, the linear system is solved in a least squared sense 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.

References

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