**A Journey Through Image Inpainting With Partial Differential Equations**

Created by Mark Goldwater and Jonah Spicher

Content:

- Inpainting with Heat Equation
- Convolutional Kernels
- Discrete Calculus
- Calculus of Variations
- Inpainting inspired by professional restorator approach

References:


- **Scale-Space and Edge Detection Using Anisotropic Diffusion - Perona and Malik**
- **Image Inpainting with the Heat Equation - Kalish**
- **Math 257: Finite Difference Methods**
- **Image Inpainting - Sapio and Ballester**
- **Digital image processing: p054 - Anisotropic Diffusion (Sapiro)**
- **Digital image processing: p053- Calculus of Variations (Sapiro)**
- **create_image_and_mask.m from Parisotto and Schoenlieb at Cambridge**


## Background

According to a Mathworks article written by Carola-Bibiane Schönlieb, "Inpainting, or image interpolation, is a process used to reconstruct missing parts of images. Artists have long used manual inpainting to restore damaged paintings. Today, mathematicians apply partial differential equations (PDEs) to automate image interpolation. The PDEs operate in much the same way that trained restorers do: They propagate information from the structure around a hole into the hole to fill it in."

Figure 1: Example of restored image using inpainting techniques.

Above in 1, you can see an example of a photograph with missing information (pictured on the left) and its restoration using inpainting techniques (pictured on the right). The pure PDE methods behind

image inpainting are sometimes accompanied with machine learning algorithms; however, for the purpose of this problem set, we will stick with pure PDE inpainting techniques. To start, we are going to apply the well-renown heat equation to the inpainting problem.

**Inpainting Using the Heat Equation**

Recall that the one-dimensional heat equation in continuous space and time along a wire of length $L$ with the initial condition $f(x)$ and Dirichlet boundary conditions is defined as follows.

$$\begin{cases} u_t = ku_{xx} & 0 \leq x \leq L,\ 0 \leq t \\ u(x,0) = f(x) & 0 \leq x \leq L \\ u(0,t) = u(L,0) = 0 & 0 \leq t. \end{cases}$$

Also recall that the same situation with a Neumann boundary condition, which defines the flux of heat on the boundaries of the wire is written as follows.

$$\begin{cases} u_t = ku_{xx} & 0 \leq x \leq L,\ 0 \leq t \\ u(x,0) = f(x) & 0 \leq x \leq L \\ u_x(0,t) = u_x(L,0) = C & 0 \leq t,\ C \in \mathbb{R}. \end{cases}$$

We have learned various methods to solve the heat equation, including separation of variables and convolution with the natural solution, but in order to use a computer to solve the heat equation we need to apply the method of finite differences to attractively produce a solution among a discrete version of space and time. We have previously derived this iterative solution to be the following.

$$u_j^{m+1} = u_j^m + s(u_{j-1}^m - 2u_j^m + u_{j+1}^m) \qquad 0 \leq s \leq \frac{1}{2}$$

1. To start, imagine a 1 dimensional image of a solid black bar, as shown below. The center portion of the bar has been obscured from $x = 0$ to $x = L$.

In order to most accurately get values at the edge of our boundary, we should consider the "heat flux" in the image surrounding our boundary. To find this, we find the difference in the value of the two pixels outside of our boundary. For example, if our boundary is at L, then we would find $u(L+1, m) - u(L+2, m)$.

Assuming that $u_x(0, m) = u_x(-1, m)$ is a more accurate approximation than assuming that $u(0, m) = u(-1, m)$, so Neumann boundary conditions allow our inpainting algorithm to fill in gradients more accurately.

(a) Use a centered difference equation

$$f_x(x,t) = \frac{f(x + \Delta x, m) - f(x - \Delta x, m)}{2\Delta m}$$

to rewrite the Neumann boundary conditions at $x = 0$ and $x = L$ for some heat flux $u_x = C$.

(b) Simplify these expressions to get an expression for $u(-\Delta x, m)$ and $u(N + \Delta x, m)$.

(c) $u(-\Delta x, m)$ and $u(L + \Delta x, m)$ are the values on $u$ of one behind the left-hand boundary of 0 and and one beyond the right-hand boundary of $L$.

Using the discretized heat equation in the information box above, come up with **two equations** to calculate both $u_0^{m+1}$ and $u_L^{m+1}$

2. Now, lets begin to see how we can apply these discretized equations to a MATLAB implementation of the obfuscated bar problem.

   (a) Write an expression for the value of an arbitrary pixel $j$ at time $m+1$ using the discretized version of the heat equation for when $0 < j < L$.

   (b) Using the starter code **here**, insert your expressions for $I_j^{m+1}$ for the endpoints of the inpainting region, from **Problem 1 Part c**, and for the inside of the inpainting region from the previous part of this problem. Set $s = \frac{1}{2}$. (*Note:* the actual bar in the code is a two dimensional image, but we are treating each column as one pixel, which makes it one dimensional and not just a singular line of pixels that you would have to squint at to see).

   (c) Can you think of situations where the heat equation would be less effective? What are potential limitations of the heat equation in image inpainting?

## Image Inpainting Via Information Propagation Along Isophotes

This next image inpainting technique is the final one that we are going to go over. It is inspired by the method that actual resortators utilize when fixing damaged photos or art pieces. It is described using the following equation:

$$I_t = \nabla(\Delta I) \cdot \nabla^{\perp} I$$

This equation may be a little intimidating at first, but let's try and understand what it achieves qualitatively.

The creators of this method have chosen the Laplacian of the image $\Delta I$ as a rough estimation of smoothness of the image. Then we calculate the change in smoothness across the image by taking the gradient of this smoothness measurement which corresponds to $\nabla(\Delta I)$ in our equation.
Then, lastly, we want to project this change in smoothness along the edges that are present in our image, for these are the contours where we want this quantity to be zero (shown in Figure 4). Recall, that the gradient is always perpendicular to contours, so we want to project $\nabla(\Delta I)$ in the direction of the perpendicular gradient, giving us the right-hand side of the equation $\nabla(\Delta I) \cdot \nabla^{\perp} I$

In order to apply this equation to an image, we then set it equal to $I_t$ and run the simulation until this quantity is close to zero. Setting the quantity on the right to zero ensures that the image is smooth along its contours, which only happens when the image is filled in.

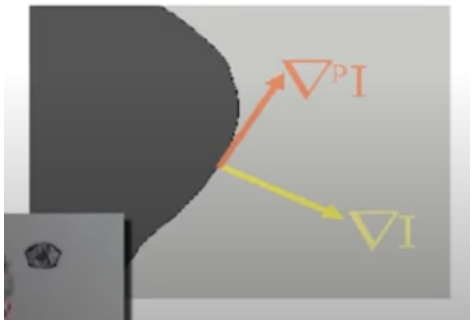


Figure 2: We want to propagate image smoothness information in the direction of the perpendicular contour as shown in the figure.

**Note:** To ensure a correct evolution of the direction field, a diffusion process is interleaved with the image inpainting process. The diffusion process we will use is called anisotropic diffusion and will be described later on in this problem.

Before we start you are going to need to acquire some new tools in order to be able to successfully implement this algorithm such that it will run with sufficient efficiency. The first of these is convolutional kernels for performing operations such as the gradient and Laplacian. The second of these is anisotropic diffusion which we have implemented for you to use in the algorithm and will explain in the next proble.

1. Discrete time convolution can be used to quickly and easily find different derivatives of an image. This technique will be helpful to calculate the terms in the discrete PDE used for inpainting. For example, the kernel [0 -1 1] (which we will call $a$ can be used to find the forward difference approximation for the first derivative of a function. When $y[k] = a * x[k]$, discrete convolution means that we are effectively setting $y[k] = 0 \cdot x[k-1] - x[k] + x[k+1]$.

   (a) Using the same principle, come up with a kernel which gives a centered difference approximation for the second derivative: $x_{tt}[t] \approx x[t-1] - 2 \cdot x[t] + x[t+1]$.

   (b) Let's move to two dimensions. Using the answer to part a, find a kernel which approximates the Laplacian of a 2D function $I$ (like an image!). Note that this kernel will be a 3x3 matrix, because you will need the second derivative in both the x direction and the y direction. (Hint: Try writing out a discrete approximation of the Laplacian of $I(x, y)$ using the centered difference approximation for the second derivative).

   (c) We are also going to need the gradient of the image. The gradient is a little bit trickier, because its result is a vector field, which means our convolution alone does not give us what we want. However, we can write an expression for the x coordinate ($I_x$) and the y coordinate ($I_y$) of the gradient. Write these expressions using discrete approximations for the first derivative.

2. Now let's start implementing the inpainting algorithm! Download **restore_inpaint_algorithm.m**, **guillermo_inpainting.m**, **anisodiff2D.m**, and **ws.mat**.

   guillermo_inpaint.m iteratively calls functions defined in two other files, ws.mat contains the original image, the disrupted image, and the mask used to disrupt the image, and anisodiff2D.m is used to smooth the image (don't worry about that for now we'll go over it in the next problem. It helps to evolve the field correctly).

   Finally, restore_inpaint_algorithm.m is the file you will be editing. This file implements a discrete version of the main inpainting PDE explained above. Lets go through it step-by-step.

   (a) After loading ws.mat into your MATLAB workspace use the **imshow()** function to look at the variables *u*, *orig*, and *mask*. What are they, and how do they relate to each other?

   (b) The matlab function **imfilter()** takes three parameters. The first is the image to be modified, the second is the kernel to be used, and the third is the specific functionality you want to use. This should just be set to 'conv' for convolution.

      i. First, fill in the kernels you found in problem 1 of this section. A few have been done for you. (These are very similar to the gradient kernel you found, but use either a centered or backwards approximation for the first derivative, for example, $x_t \approx x[t] - x[t-1]$ instead of $x_t \approx x[t+1] - x[t]$).

      ii. Now, fill in the expressions for the various derivatives of the image. For each one, use imfilter and the relevant kernel. For the first three, the image you are modifying is called diff_im. A few notes:
      - When using diff_im, you only want the current color slice. The code explains this in more detail.
      - In order to get the gradient of the Laplacian, you will need to modify the variable lap using the kernels provided for the Laplacian.
      - Don't worry about the backwards difference kernels.

   (c) Now that we know all of the relevant derivatives of the image, it is time to stat putting them together into our PDE.

The gradient of the image is actually not the vector we want. In order to propagate information along contour lines, in the direction where the image doesn't change, we want to use the vector which points perpendicular to the gradient (you can get this with a rotation matrix, if you want). Also, we want to normalize it, as we don't care how big the gradient is, we just want to know its direction. This vector can be written as:

$$\hat{N} = \frac{\nabla^{\perp} I}{\|\nabla I\|} = \begin{bmatrix} -I_y \\ I_x \end{bmatrix} \cdot \frac{1}{\sqrt{I_x^2 + I_y^2 + \epsilon}}$$

The $\epsilon$ is added to prevent division by zero in regions where the image is perfectly smooth (so the norm of the gradient is equal to zero). Use the value $\epsilon = 0.1$. Here, $I_y$ is what is called gradY in the code and $I_x$ is called gradX.

Using this equation and the derivatives you calculated in part b, write an expression for $\hat{N}$ in the code (Note that in the code, you need to write its x component and its y component separately).

(d) Finally, we have the two components described by the PDE. There is still some work to be done to update the image, but you can plug in what you found. We will define a variable $\beta = \nabla(\Delta I) \cdot \hat{N}$. This variable tells us how much the smoothness of the image is disrupted when we follow the lines along which it shouldn't change. If the region wasn't missing, this should be zero. Using the x and y coordinates of both $\hat{N}$ and the gradient of the Laplacian, write an expression for $\beta$ using the $x$ and $y$ components of *lapd* and *N_hat*.

(e) Now that the code is filled in, try calling the **guillermo_inpaint()** function from the matlab command prompt. It takes an image (u), some number of iterations to perform, and the mask (mask). Try 800 iterations. Does it work?

What this code is doing is using the beta variable you found (approximately a measure of how rough the image is along its contour lines) as the time derivative of the image, and then advancing time. This lets information flow in to the deleted region along lines, filling it back in. Here is a fun picture which demonstrates roughly what that means.

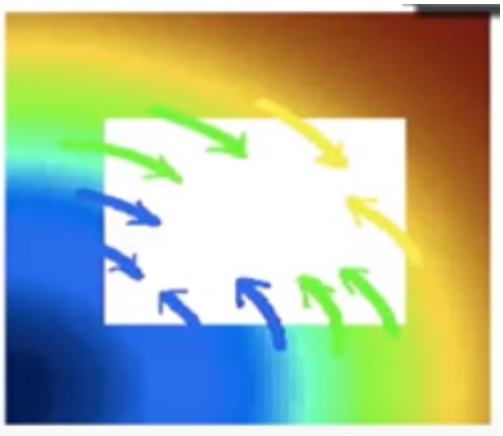

Figure 3: The color flows in towards the center, but only in the direction where the color is staying the same.

## Anisotropic Diffusion

If we think back to the beginning of the problem set where we inpainted using the heat equation, we were performing what is know as *isotropic diffusion*. This approach iteratively averaged all the pixels in the region we were trying to inpaint which was the reason that this method only worked when the boundaries were homogeneous. If they were not, the process would eventually average across edges in the image which is not useful for inpainting.

As a result the diffusion step that we interweave in the *Information Propagation Along Isophotes* method shown in the last problem is called *Anisotropic Diffusion*. This method of diffusion is also iterativly applied, but using the following PDE (which is for a 2D image)

$$I_t = div(c(x, y, t)\nabla I) = c(x, y, t)\nabla I + \nabla c \cdot \nabla I$$

where $c(x, y, t)$ is a scalar field which scales the gradient of the image near edges so it does not diffuse in these areas.

**So, the main take-away for this in the context of the last image inpainting algorithm is that we apply two iterations of this diffusion after 15 iterations of the inpainting algorithm in order to ensure a correct evolution of the field. You can see this applied in the guillermo_inpainting.m file.**

Now for the rest of this problem, we will derive the anisotropic diffusion technique using *Calculus of Variations* which according to **Wikipedia** is "...a field of mathematical analysis that uses variations, which are small changes in functions and functionals, to find maxima and minima of functionals: mappings from a set of functions to the real numbers." In other words: finding function that minimize an objective function.

One tool in this field of mathematics is the Euler-Lagrange Equation:

$$\left(\frac{\partial}{\partial u} - \frac{d}{dx}\frac{\partial}{\partial u_x}\right) F(u, u_x) = 0$$

Whose solution will find extrema values of integrals of the following form:

$$\int F(u, u_x)dx$$

1. Let's consider a classic example in the realm of *Calculus of Variations*. Consider a situation where we have a particle at position $x_0$ at time $t_0$ and at position $x_1$ at time $t_1$ as shown below.

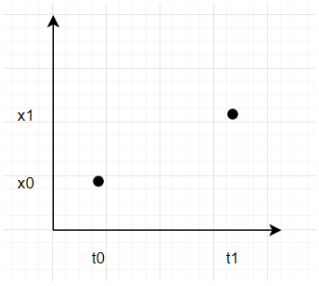


Figure 4: Position vs. Time graph for a particle for which the position is known at $t_0$ and $t_1$.

By applying the Euler-Lagrange equation, derive a function that represents the shortest path between these two points that the particle takes (*Hint:* recall that the length of a curve $u$ can be calculated using the following integral: $\int_{x_0}^{x_1} \sqrt{1+u_x^2}dx$)?

Recall back to your most recent calculus class (probably) and the concept of gradient descent. Let's review this quickly. Say we have a function $f(x)$ which we want to apply gradient descent to in order to find a local minimum. We can determine the "steps" we need to take to work towards this minimum by taking the derivative of this function after it experiences a small perturbation and set it equal to zero as the perturbation becomes infinitesimal, for this defines a local maximum or minimum.

$$\forall n : \lim_{\epsilon \to 0} \left( \frac{df(x+\epsilon n}{d\epsilon} \right) \Leftrightarrow \forall n : f_x(x)n \Leftrightarrow f_x = 0$$

The conclusion reached by the above math might seem trivial. Of course we need the derivative to be zero at a local minimum. But now we can write the following differential equation:

$$x_t = -f_x(x)$$

which allows us to change $x$ in the opposite direction of the derivative of the function allowing us to descend the gradient in the following fashion:

$$x_{t+1} = x_t - dtf_x(x_t)$$

where $dt$ is the rate at which you are descending or the "step size". Exciting!

2. Write out the discrete differential equation used to perform gradient descent on the quadratic function $f(x) = x^2$. Given that $x_0 = 3$, calculate $x_1$, $x_2$, and $x_3$.

Looking back at the Euler-Lagrange equation, we derive it in a similar way to how we just derived gradient descent except instead of slowly stepping with a point, we are stepping with a function. Let's walk through it!

Let's define the integral we are trying to minimize as follows:

$$E(u(x)) = \int F(u, u_x)dx$$

Then, this time the perturbation will be the addition of an entire function because now we are trying to find a minimizing *function*. So if we let $\tilde{u}(x)$ be the perturbed function we get

$$\tilde{u}(x) = u(x) + \epsilon n(x)$$

Thinking analogously to the derivation of gradient descent, we now want to find

$$\forall n(x) : \lim_{\epsilon \to 0} \left( \frac{d}{d\epsilon} \int F(\tilde{u}, \tilde{u_x})dx \right) = 0$$

which eventualy leads us to the following conclusion (which is left as an exercise for the reader)

$$\frac{\delta E(u)}{\delta u} = \left( \frac{\partial}{\partial u} - \frac{d}{dx}\frac{\partial}{\partial u_x} \right) F(u, u_x)$$

It's the Euler-Lagrange equation! We can then cary out the gradient descent process using $u_t = -\frac{\delta E(u)}{\delta u}$

3. Read the Motivation section of the anisotropic diffusion Wikipedia page to see how the initial anisotropic diffusion equation given at the beginning of this section was derived. Nothing to do here, just read!

4. Look at Equation 7 in the Original paper for anisotropic diffusion, otherwise known as Perona-Malik diffusion, which is the discretized version of this type of diffusion.

   (a) Derive this discretized version from the original equation.

   (b) Take a look at the implementation in **anisodiff2D.m** and make sense of it using your knowledge of convolutional kernel tricks from earlier in the problem set and part (a) of this problem.