## Lab 1

# Anisotropic Diffusion

**Lab Objective:** *Demonstrate the use of finite difference schemes in image analysis.*

A common task in image processing is to remove extra static from an image. This is most easily done by simply blurring the image, which can be accomplished by treating the image as a rectangular domain and applying the diffusion (heat) equation:

$$u_t = c\Delta u$$

where $c$ is some diffusion constant and $\Delta$ is the Laplace operator. Unfortunately, this also blurs the boundary lines between distinct elements of the image.

A more general form of the diffusion equation in two dimensions is:

$$u_t = \nabla \cdot (c(x, y, t)\nabla u)$$

where $c$ is a function representing the diffusion coefficient at each given point and time. In this case, $\nabla\cdot$ is the divergence operator and $\nabla$ is the gradient.

To blur a picture uniformly, choose $c$ to be a constant function. Since $c$ controls how much diffusion is allowed at each point, it can be modified so that diffusion is minimized across edges in the image. In this way we attempt to limit diffusion near the boundaries between different features of the image, and allow smaller details of the image (such as static) to blur away. This method for image denoising is especially useful for denoising low quality images, and was first introduced by Pietro Perona and Jitendra Malik in 1987. It is known as Anisotropic Diffusion or Perona-Malik Diffusion.

## A Finite Difference Scheme

Suppose we have some estimate $E$ of the rate of change at a given point in an image. $E$ will be largest at the boundaries in the image. We will then let $c(x, y, t) = g(E(x, y, t))$ where $g$ is some function such that $g(0) = 1$ and $\lim_{x\to\infty} g(x) = 0$. Thus $c$ will be small where $E$ is large, so that little diffusion occurs near the boundaries of different portions of the image.

We will model this system using a finite differencing scheme with an array of values at a 2D grid of points, and iterate through time. Let $U^n_{l,m}$ be the discretized approximation of the function $u$, $n$ be the index in time, $l$ be the index along the $x$-axis, and $m$ be the index along the $y$-axis.

The Laplace operator can be approximated with the finite difference scheme

$$\Delta u = u_{xx} + u_{yy} \approx \frac{U^n_{l-1,m} - 2U^n_{l,m} + U^n_{l+1,m}}{(\Delta x)^2} + \frac{U^n_{l,m-1} - 2U^n_{l,m} + U^n_{l,m+1}}{(\Delta y)^2}.$$

A good metric to use with images is to let the distance between each pixel be equal to one, so $\Delta x = \Delta y = 1$. Rearranging terms, we obtain

$$\Delta u \approx (U^n_{l-1,m} - U^n_{l,m}) + (U^n_{l+1,m} - U^n_{l,m}) + (U^n_{l,m-1} - U^n_{l,m}) + (U^n_{l,m+1} - U^n_{l,m}).$$

Again, since we are working with images and not some time based problem, we can without loss of generality let $\Delta t = 1$, so we obtain the finite difference scheme

$$U^{n+1}_{l,m} = U^n_{l,m} + (U^n_{l-1,m} - U^n_{l,m}) + (U^n_{l+1,m} - U^n_{l,m}) + (U^n_{l,m-1} - U^n_{l,m}) + (U^n_{l,m+1} - U^n_{l,m}).$$

We will now limit the diffusion near the edges of objects by making the modification

$$\begin{aligned} U^{n+1}_{l,m} = U^n_{l,m} + \lambda\Big( & g(|U^n_{l-1,m} - U^n_{l,m}|)(U^n_{l-1,m} - U^n_{l,m}) \\ & + g(|U^n_{l+1,m} - U^n_{l,m}|)(U^n_{l+1,m} - U^n_{l,m}) \\ & + g(|U^n_{l,m-1} - U^n_{l,m}|)(U^n_{l,m-1} - U^n_{l,m}) \\ & + g(|U^n_{l,m+1} - U^n_{l,m}|)(U^n_{l,m+1} - U^n_{l,m})\Big), \end{aligned}$$

where $\lambda \leq \frac{1}{4}$ is the stability condition.

In this difference scheme, each term is affected most by nearby terms that are most similar to it, so less diffusion will happen anywhere there is a sharp difference between pixels. This scheme also has the useful property that it does not increase or decrease the total brightness of the image. Intuitively, this is because the effect of each point on its neighbors is exactly the opposite effect its neighbors have on it.

Two commonly used functions for $g$ are $g(x) = e^{-\left(\frac{x}{\sigma}\right)^2}$ and $g(x) = \frac{1}{1+\left(\frac{x}{\sigma}\right)^2}$. The parameter $\sigma$ allows us to control how much diffusion decreases across boundaries, with larger $\sigma$ values allowing more diffusion. Note that $g(0) = 1$ and $\lim_{x\to\infty} g(x) = 0$ for both functions. In this lab we use $g(x) = e^{-\left(\frac{x}{\sigma}\right)^2}$.

It is worth noting that this particular difference scheme is *not* an accurate finite difference scheme for the version of the diffusion equation we discussed before, but it *does* accomplish the same thing in the same way. As it turns out, this particular scheme is the solution to a slightly different diffusion PDE, but can still be used the same way.

For this lab's examples we read in the image using the `scipy.misc.imread` function, and normalized it so that the colors are represented as floating point values between 0 and 1. An image can converted to black and white when it is read by including the argument `flatten=True`

Our finite difference scheme can be implemented using purely vector operations. First consider the case where the boundaries of the image are considered fixed. The scheme is implemented naively with the following code:

```
import numpy as np
from scipy.misc import imread, imsave
from matplotlib import pyplot as plt
from matplotlib import cm

def anisdiff_bw_noBCs(U, N, lambda_, g):
    """ Run the Anisotropic Diffusion differencing scheme
    on the array A of grayscale values for an image.
    Perform N iterations, use the function g
    to limit diffusion across boundaries in the image.
    Operate on A inplace. """
    for i in xrange(N):
        U[1:-1,1:-1] += lambda_ * \
                        (g(U[:-2,1:-1] - U[1:-1,1:-1]) *
                          (U[:-2,1:-1] - U[1:-1,1:-1]) +
                         g(U[2:,1:-1] - U[1:-1,1:-1]) *
                          (U[2:,1:-1] - U[1:-1,1:-1]) +
                         g(U[1:-1,:-2] - U[1:-1,1:-1]) *
                          (U[1:-1,:-2] - U[1:-1,1:-1]) +
                         g(U[1:-1,2:] - U[1:-1,1:-1]) *
                          (U[1:-1,2:] - U[1:-1,1:-1]))
```

We can implement anisotropic diffusion for black and white images using a different set of boundary conditions. For the top edge let

$$\begin{aligned} U_{l,m}^{n+1} = U_{l,m}^{n} + \lambda ( & g(|U_{l-1,m}^{n} - U_{l,m}^{n}|)(U_{l-1,m}^{n} - U_{l,m}^{n}) \\ & + g(|U_{l+1,m}^{n} - U_{l,m}^{n}|)(U_{l+1,m}^{n} - U_{l,m}^{n}) \\ & + g(|U_{l,m+1}^{n} - U_{l,m}^{n}|)(U_{l,m+1}^{n} - U_{l,m}^{n})); \end{aligned}$$

the other edges are treated similarly. For the top left corner let

$$\begin{aligned} U_{l,m}^{n+1} = U_{l,m}^{n} + \lambda ( & g(|U_{l+1,m}^{n} - U_{l,m}^{n}|)(U_{l+1,m}^{n} - U_{l,m}^{n}) \\ & + g(|U_{l,m+1}^{n} - U_{l,m}^{n}|)(U_{l,m+1}^{n} - U_{l,m}^{n})), \end{aligned}$$

and similarly for the other corners. Essentially we are just using the terms of the difference scheme that are actually defined.

Here is a non-optimized implementation that performs the desired computations for a generic function $g$.

```
def anisdiff_bw_withBCs(U, N, lambda_, g):
    """ Run the Anisotropic Diffusion differencing scheme
    on the array U of grayscale values for an image.
    Perform N iterations, use the function g
    to limit diffusion across boundaries in the image.
    Operate on U inplace. """
    difs = np.empty_like(U)
    for i in xrange(N):
        difs[:-1] = g(U[1:] - U[:-1]) * (U[1:] - U[:-1])
        difs[-1] = 0
        difs[1:] += g(U[:-1] - U[1:]) * (U[:-1] - U[1:])
        difs[:,:-1] += g(U[:,1:] - U[:,:-1]) * (U[:,1:] - U[:,:-1])
        difs[:,1:] += g(U[:,:-1] - U[:,1:]) * (U[:,:-1] - U[:,1:])
        difs *= lambda_
        U += difs
```

We can use this code on an image like this:

```
from scipy.misc import imread, imsave
from matplotlib import pyplot as plt
from matplotlib import cm

# Read the image file 'test.png'.
# Multiply by 1. / 255 to change the values so that they are floating point
# numbers ranging from 0 to 1.
U = imread('test.jpg', flatten=True) * (1. / 255)
# Set inputs for the function.
sigma = .1
g = lambda x: np.exp(x * x * (-1. / sigma**2))
lambda_ = .25
N = 50
anisdiff_bw_noBCs(U, N, lambda_, g)
# Show the image.
plt.imshow(U, cmap=cm.gray)
plt.show()
```

## Speeding up the Implementation

We will use the package numexpr to speed up our computations. Numexpr is a package that can evaluate simple algebraic expressions involving arrays quickly by recognizing and computing common subexpressions and optimizing for fast cache management when accessing memory. You may recall that the primary function in the user interface of numexpr is the `evaluate` function. It evaluates an expression (in some cases, much faster than NumPy can) that has been expressed as a string containing the names of array objects that are in the current scope. Below we provide some example code that uses numexpr:

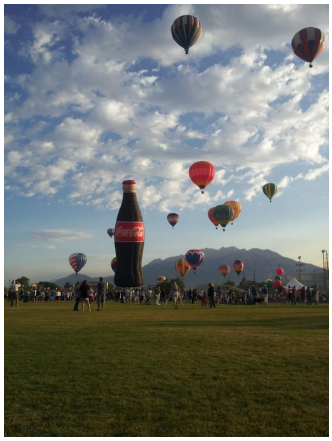
```
import numpy as np
import numexpr as ne
x = np.arange(1000)
y = np.arange(1000,2000)

u = ne.evaluate('x**2. + y**2.')
v = ne.evaluate('sum(x**2. + y**2.)')
```

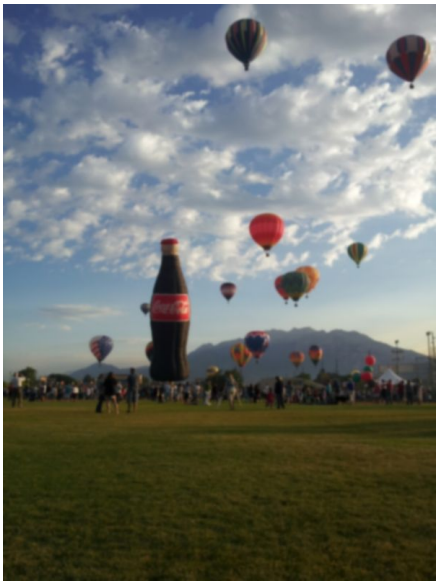
**Problem 1.** Implement anisotropic diffusion (with the correct boundary conditions) using numexpr to optimize the computations. How much speedup can you get over the unoptimized Numpy version? (Note: It is not difficult to speedup the Numpy implementation by reducing the need for Numpy to allocate space for temporary areas. However, we will use numexpr to speed up our implementation.)

In your function, use

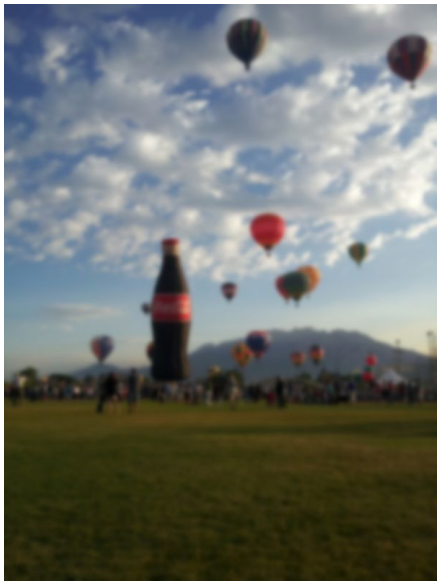
$$g(x) = e^{-\left(\frac{x}{\sigma}\right)^2}$$



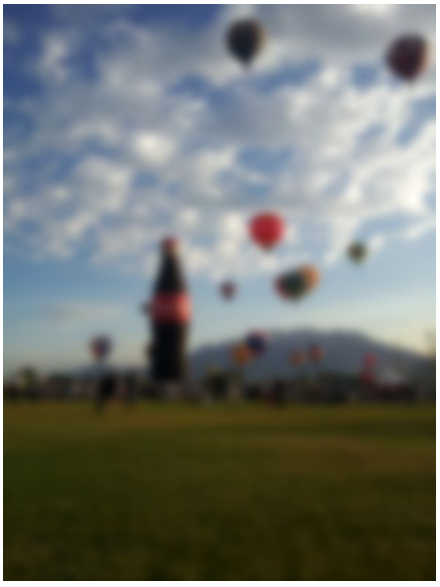
original image



5 iterations with $\sigma = .7$ and $\lambda = .2$



20 iterations



100 iterations

## Working with Color Images

Colored images can be processed in a similar manner. Instead of being represented as a two-dimensional array, colored images are represented as three dimensional arrays. The third dimension is used to store the intensities of each of the standard 3 colors. This diffusion process can be carried out in the exact same way, on each of the arrays of intensities for each color, but instead of detecting edges just in one color, we need to detect edges in any color, so instead of using something

of the form $g(|U^n_{l+1,m} - U^n_{l,m}|)$ as before, we will now use something of the form $g(||U^n_{l+1,m} - U^n_{l,m}||)$, where $U^n_{l+1,m}$ and $U^n_{l,m}$ are vectors now instead of scalars. The difference scheme can be treated as an equation on vectors in 3-space and now reads:

$$\begin{aligned} U^{n+1}_{l,m} = U^n_{l,m} + \lambda(&g(||U^n_{l-1,m} - U^n_{l,m}||)(U^n_{l-1,m} - U^n_{l,m}) \\ &+ g(||U^n_{l+1,m} - U^n_{l,m}||)(U^n_{l+1,m} - U^n_{l,m}) \\ &+ g(||U^n_{l,m-1} - U^n_{l,m}||)(U^n_{l,m-1} - U^n_{l,m}) \\ &+ g(||U^n_{l,m+1} - U^n_{l,m}||)(U^n_{l,m+1} - U^n_{l,m})) \end{aligned}$$

When implementing this scheme for colored images, use the 2-norm on 3-space, i.e $||x|| = \sqrt{x_1^2 + x_2^2 + x_3^2}$ where $x_1$, $x_2$, and $x_3$ are the different coordinates of $x$.

We can use this function like this:

```
A = imread('test.jpg') / np.float32(255.)
Ac = A.copy()
sigma = .1
lambda_ = .2
N = 5
anisdiff_color_noBCs(A, N, lambda_, sigma)
plt.imshow(A)
plt.show()
```

**Problem 2.** Make a new version of the code you wrote for the previous problem which processes a colored image. Measure the difference between pixels using the 2-norm. Use the corresponding vector versions of the boundary conditions given in Problem 1.

To do this, you must break up the computation of the different terms so that you can compute the norm along each triple of color values and then broadcast the g values along the last axis of the array of differences.

**Problem 3.** Use numexpr to speed up your implementation of anisotropic diffusion for color images.