Lab 15

Numerical Derivatives

Lab Objective: *Understand and implement finite difference approximations of the derivative in single and multiple dimensions. Evaluate the accuracy of these approximations. Then use finite difference quotients to find edges in images via the Sobel filter.*

Derivative Approximations in One Dimension

The derivative of a function f at a point x_0 is

$$f'(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}. \tag{15.1}$$

In this lab, we will investigate one way a computer can calculate $f'(x_0)$.

Forward Difference Quotient

Suppose that in Equation (15.1), instead of taking a limit, we just pick a small value for h. Then we would expect $f'(x_0)$ to be close to the quantity

$$\frac{f(x_0 + h) - f(x_0)}{h}. \tag{15.2}$$

This quotient is called the *first order forward difference approximation* of the derivative. Because $f'(x_0)$ is the limit of such quotients, we expect that when h is small, this quotient is close to $f'(x_0)$. We can use Taylor's formula to find just how close.

By Taylor's formula,

$$f(x_0 + h) = f(x_0) + f'(x_0)h + R_2(h),$$

where $R_2(h) = \left(\int_0^1 (1-t)f''(x_0 + th)dt\right) h^2$. (This is called the *integral form* of the remainder for Taylor's Theorem; see Volume 1 Chapter 6). When we solve this equation for $f'(x_0)$, we get

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} - \frac{R_2(h)}{h}. \tag{15.3}$$

Thus, the error in using the first order forward difference quotient to approximate $f'(x_0)$ is

$$\left|\frac{R_2(h)}{h}\right| \leq |h| \int_0^1 |1-t||f''(x_0 + th)|dt.$$

If we assume f'' is continuous, then for any δ, set $M = \sup_{x\in(x_0-\delta,x_0+\delta)} f''(x)$. Then if $|h| < \delta$, we have

$$\left|\frac{R_2(h)}{h}\right| \leq |h| \int_0^1 M dt = M|h| \in O(h).$$

Therefore, the error in using (15.2) to approximate $f'(x_0)$ grows like h.

Centered Difference Quotient

In fact, we can approximate $f'(x_0)$ to the second order with another difference quotient, called the *centered difference quotient*. We begin by trying to find the *backward difference quotient*. Evaluate Taylor's formula at $x_0 - h$ to derive

$$f'(x_0) = \frac{f(x_0) - f(x_0 - h)}{h} + \frac{R_2(-h)}{h}. \tag{15.4}$$

The first term on the right hand side of (15.4) is called the *backward difference quotient.* This quotient also approximates $f'(x_0)$ to first order, so it is not the quotient we are looking for. When we add (15.3) and (15.4) and solve for $f'(x_0)$ (by dividing by 2), we get

$$f'(x_0) = \frac{\frac{1}{2}f(x_0 + h) - \frac{1}{2}f(x_0 - h)}{h} + \frac{R_2(-h) - R_2(h)}{2h} \tag{15.5}$$

The *centered difference quotient* is the first term of the right hand side of (15.5). Let us investigate the remainder term to see how accurate this approximation is. Recall from the proof of Taylor's theorem that $R_k = \frac{f^{(k)}(x_0)}{k!}h^k + R_{k+1}$. Therefore,

$$\begin{aligned}
\frac{R_2(-h) - R_2(h)}{2h} &= \frac{1}{2h}\left(\frac{f''(x_0)}{2}h^2 + R_3(-h) - \frac{f''(x_0)}{2}h^2 - R_3(h)\right) \\
&= \frac{1}{2h}(R_3(-h) - R_3(h)) \\
&= \frac{1}{2h}\left(\left(\int_0^1 \frac{(1-t)^2}{2}f'''(x_0 + th)dt\right)h^3 - \left(\int_0^1 \frac{(1-t)^2}{2}f'''(x_0 - th)dt\right)h^3\right) \\
&= \left(\int_0^1 \frac{(1-t)^2}{4}(f'''(x_0 + th) - f'''(x_0 - th))\right)h^2 \\
&\in O(h^2)
\end{aligned}$$

once we restrict h to some δ-neighborhood of 0. So the error in using the centered difference quotient to approximate $f'(x_0)$ grows like h^2, which is smaller than h when $|h| < 1$.

Accuracy of Approximations

Let us discuss what step size h we should plug into the difference quotients to get the best approximation to $f'(x_0)$. Since f' is defined as a limit as $h \to 0$, you may think that it is best to choose h as small as possible, but this is not the case. In fact, dividing by very small numbers causes errors in floating point arithmetic. This means that as we decrease $|h|$, the error between $f'(x_0)$ and the difference quotient will first decrease, but then increase when $|h|$ gets too small because of floating point arithmetic.

Here is an example with the function $f(x) = e^x$. A quick way to write f as a function in Python is with the `lambda` keyword.

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> f = lambda x: np.exp(x)
```

h	1e-1	1e-3	1e-5	1e-7	1e-9	1e-11
Error	5e-3	5e-7	6e-11	6e-11	7e-9	1e-5

Table 15.1: This table shows that it is best not to choose h too small when you approximate derivatives with difference quotients. Here, "Error" equals the absolute value of $f'(1) - f_{app}(1)$ where $f(x) = e^x$ and f_{app} is the centered difference approximation to f'.

In general, the line `f = lambda <params> : <expression>` is equivalent to defining a function `f` that accepts the parameters `params` and returns `expression`.

Next we fix a step size `h` and define an approximation to the derivative of `f` using the *centered difference quotient*.

```
>>> h = 1e-1
>>> Df_app = lambda x: .5*(f(x+h)-f(x-h))/h
```

Finally, we check the accuracy of this approximation at $x_0 = 1$ by computing the difference between `Df_app(1)` and the actual derivative evaluated at 1.

```
# Since f(x) = e^x, the derivative of f(x) is f(x)
>>> np.abs( f(1)-Df_app(1) )
0.0045327354883726301
```

We note that our functions `f` and `Df_app` behave as expected when they are passed a NumPy array.

```
>>> h = np.array([1e-1, 1e-3, 1e-5, 1e-7, 1e-9, 1e-11])
>>> np.abs( f(1)-Df_app(1) )
array([  4.53273549e-03,   4.53046679e-07,   5.85869131e-11,
         5.85873572e-11,   6.60275079e-09,   1.04294937e-05])
```

These results are summarized in Table 15.1.

Thus, the optimal value of h is one that is small, but not too small. A good choice is `h = 1e-5`.

Problem 1. Write a function that accepts as input a callable function object `f`, an array of points `pts`, and a keyword argument `h` that defaults to `1e-5`. Return an array of the *centered difference quotients* of `f` at each point in `pts` with the specified value of `h`.

You may wonder if the forward or backward difference quotients are ever used, since the centered difference quotient is a more accurate approximation of the derivative. In fact, there are some functions that in practice do not behave well under centered difference quotients. In these cases, one must use the forward or backward difference quotient.

Finally, we remark that forward, backward, and centered difference quotients can be used to approximate higher-order derivatives of f. However, taking derivatives is an *unstable* operation. This means that taking a derivative can amplify the arithmetic error in your computation. For this reason, difference quotients are not generally used to approximate derivatives higher than second order.

Derivative Approximations in Multiple Dimensions

Finite difference methods can also be used to calculate derivatives in higher dimensions. Recall that the Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ at a point $x_0 \in \mathbb{R}^n$ is the $m \times n$ matrix $J = (J_{ij})$ defined component-wise by

$$J_{ij} = \frac{\partial f_i}{\partial x_j}(x_0).$$

For example, the Jacobian for a function $f : \mathbb{R}^3 \to \mathbb{R}^2$ is defined by

$$J = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \frac{\partial f}{\partial x_3} \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \end{pmatrix}.$$

The Jacobian is useful in many applications. For example, the Jacobian can be used to find zeros of functions in multiple variables.

The forward difference quotient for approximating a partial derivative is

$$\frac{\partial f}{\partial x_j}(x_0) \approx \frac{f(x_0 + he_j) - f(x_0)}{h},$$

where e_j is the j^{th} standard basis vector. Similarly, the centered difference approximation is

$$\frac{\partial f}{\partial x_j}(x_0) \approx \frac{\frac{1}{2}f(x_0 + he_j) - \frac{1}{2}f(x_0 - he_j)}{h}.$$

Problem 2. Write a function that accepts

1. a function handle `f`,
2. an integer `n` that is the dimension of the domain of `f`,
3. an integer `m` that is the dimension of the range of `f`,
4. an `1 x n`-dimensional NumPy array `pt` representing a point in $\mathbb{R}^n$, and
5. a keyword argument `h` that defaults to `1e-5`.

Return the approximate Jacobian matrix of `f` at `pt` using the centered difference quotient.

Problem 3.

Let $f : \mathbb{R}^2 \to \mathbb{R}^2$ be defined by

$$f(x, y) = \begin{pmatrix} e^x \sin(y) + y^3 \\ 3y - \cos(x) \end{pmatrix}$$

Find the error between your Jacobian function and the analytically computed derivative on the square $[-1, 1] \times [-1, 1]$ using ten thousand grid points (100 per side). You may apply your Jacobian function to the points one at a time using a double `for` loop. Once you get the error matrix for a given point, calculate the Frobenius norm of this matrix (`la.norm` defaults to the Frobenius norm). This norm will be your total error for that point. What is the maximum error of your Jacobian function over all points in the square?

Hint: The following code defines the function $f(x, y) = \begin{pmatrix} x^2 \\ x + y \end{pmatrix}$.

```
# f accepts a length-2 NumPy array
>>> f = lambda x: np.array([x[0]**2, x[0]+x[1]])
```

Application to Image Filters

Recall that a computer stores an image as a 2-D array of pixel values (i.e., a matrix of intensities). An image filter is a function that transforms an image by operating on it locally. That is, to compute the ij^{th} pixel value in the new image, an image filter uses only the pixels in a small neighborhood around the ij^{th} pixel in the original image.

In this lab, we will use a filter derived from the gradient of an image to find edges in an image.

Convolutions

One example of an image filter is to *convolve* an image with a filter matrix. A filter matrix is a matrix whose height and width are relatively small odd numbers. If the filter matrix is

$$F = \begin{pmatrix} f_{-1,-1} & f_{-1,0} & f_{-1,1} \\ f_{0,-1} & f_{0,0} & f_{0,1} \\ f_{1,-1} & f_{1,0} & f_{1,1} \end{pmatrix},$$

then the convolution of an image A with F is $A * F = (C_{ij})$ where

$$C_{ij} = \sum_{k=-1}^{1} \sum_{\ell=-1}^{1} f_{k\ell} A_{i+k,j+\ell}. \tag{15.6}$$

Say A is an $m \times n$ matrix. Here, we take $A_{ij} = 0$ when $i \notin \{1, \dots m\}$ or $j \notin \{1, \dots, n\}$. The value of C_{ij} is a linear combination of the nearby pixel values, with coefficients given by F (see Figure 15.1). In fact, C_{ij} equals the Frobenius inner product of F with the 3×3 submatrix of A centered at ij.

Implementation in NumPy

Let us write a function that convolves an image with a filter. You can test this function on the image `cameraman.jpg`, which appears in Figure 15.2a. The following code loads this image and plots it with matplotlib.

```
>>> image = plt.imread('cameraman.jpg')
>>> plt.imshow(image, cmap = 'gray')
>>> plt.show()
```

Here is the function definition and some setup.

```
1. def Filter(image, F):
2.     m, n = image.shape
3.     h, k = F.shape
```

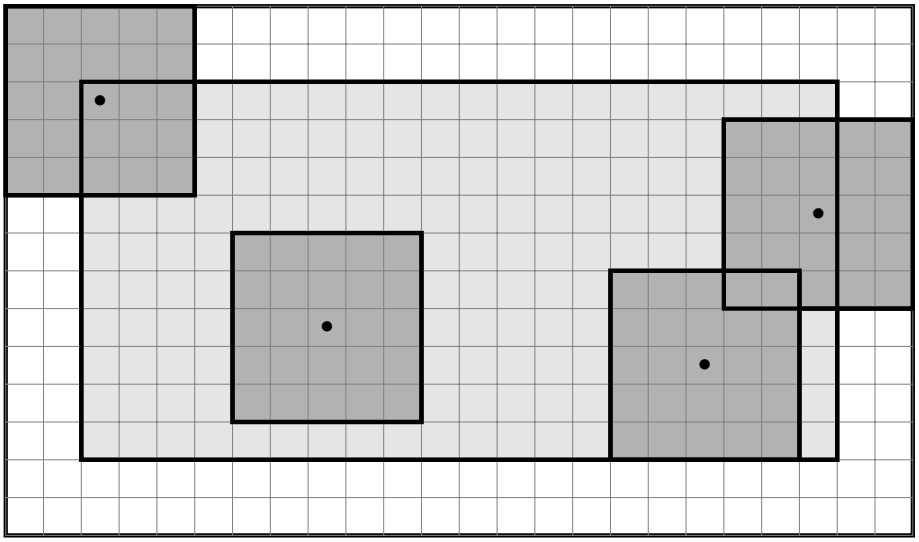


Figure 15.1: This diagram illustrates how to convolve an image with a filter. The light grey rectangle represents the original image A, and the dark grey squares are the filter F. The larger rectangle is the image padded with zeros; i.e., all pixel values in the outer white band are 0. To compute the entry of the convolution matrix C located at a black dot, take the inner product of F with the submatrix of the padded image centered at the dot.

To convolve `image` with the filter `F`, we must first *pad* the array `image` with zeros around the edges. This is because in (15.6), entries A_{ij} are set to zero when i or j is out of bounds. We do this by creating a larger array of zeros, and then making the interior part of the array equal to the original image (see Figure 15.1).

For example, if the filter is a 3×3 matrix, then the following code will pad the matrix with the appropriate number of zeros.

```
 # Create a larger matrix of zeros
image_pad = np.zeros((m+2, n+2))
# Make the interior of image_pad equal to the original image
image_pad[1:1+m, 1:1+n] = image
```

We want to do this in general in our function. Note that the number of zeros we need to pad our array depends on the size of the filter `F`.

```
5.    image_pad = # Create an array of zeros of the appropriate size
6.   # Make the interior of image_pad equal to image
```

Finally, we iterate through the image to compute each entry of the convolution matrix.

```
7.    C = np.zeros(image.shape)
8.    for i in range(m):
9.        for j in range(n):
10.           C[i,j] = # Compute C[i, j]
```

Gaussian Blur

A *Gaussian blur* is an image filter that operates on an image by convolving with the matrix

$$G = \frac{1}{159}\begin{pmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{pmatrix}.$$

Blurring an image can remove "noise", or random variation that is the visual analog of static in a radio signal (and equally undesirable).

Problem 4. Finish writing the function `Filter` by filling in lines 5, 6, and 10. Hint: Note in 15.6, C_{ij} was calculated by summing from -1 to 1. This is only the case if the filter `F` is 3×3. A slight modification is needed in the general case. Test your function on the image `cameraman.jpg` using the Gaussian Blur. The result is in Figure 15.2b.

Edge Detection

Automatic detection of edges in an image can be used to segment or sharpen the image. We will find edges with the Sobel filter, which computes the gradient of the image at each pixel. The magnitude of the gradient tells us the rate of change of the pixel values, and so large magnitudes should correspond to edges within the image. The Sobel filter is not a convolution, although it does use convolutions.

We can think of an image as a function from a 2×2 grid of points to $\mathbb{R}$. The image maps a pixel location to an intensity. It does not make sense to define the derivative of this function as a limit because the domain is discrete—a step size h cannot take on arbitrarily small values. Instead, we *define* the derivative to be the centered difference quotient of the previous section. That is, we define the derivative in the x-direction at the ij^{th} pixel to be

$$\frac{1}{2}A_{i+1,j} - \frac{1}{2}A_{i-1,j}.$$

We can use a convolution to create a matrix A_x whose ij^{th} entry is the derivative of A at the ij^{th} entry, in the x-direction. In fact, $A_x = A * S$, where

$$S = \frac{1}{8}\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}.$$

Note that this convolution takes a weighted average of the x-derivatives at (i, j), $(i, j+1)$, and $(i, j-1)$. The derivative at (i, j) is weighted by 2. Using a weighted average instead of just the derivative at (i, j) makes the derivative less affected by noise.

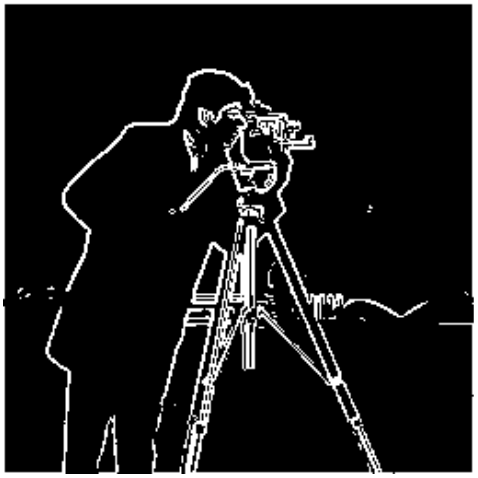
Now we can define the Sobel filter. A Sobel filter applied to an image A results in an array $B = (B_{ij})$ of 0's and 1's, where the 1's trace out the edges in the image. By definition,

$$B_{ij} = \begin{cases} 1 & \text{if } \|\nabla A(ij)\|_2 > M \\ 0 & \text{otherwise.} \end{cases}$$

Here, $\nabla A(ij) = ((A * S)_{ij}, (A * S^T)_{ij})$ is the gradient of A at the ij^{th} pixel. The constant M should be "sufficiently large" enough to pick out those pixels with the largest gradient

(a) Unfiltered image.

(b) Image after Gaussian blur is applied.



(c) Image after the Sobel filter is applied.

Figure 15.2: Here is an example of a Gaussian blur and the Sobel filter applied to an image. This photo, known as "cameraman," is a standard test image in image processing. A database of such images can be downloaded from `http://www.imageprocessingplace.com/root_files_V3/image_databases.htm`.

(i.e., those pixels that are part of an edge). A good choice for M is 4 times the average value of $\|\nabla A(ij)\|_2$ over the whole image A.

When the Sobel filter is applied to `cameraman.jpg`, we get the image in Figure 15.2c. Here, the 1's in B were mapped to "white" and the 0's were mapped to "black."

Problem 5. Write a function that accepts an image as input and applies the Sobel filter to the image. Test your function on `cameraman.jpg`. Hint: If you want to find the average of a matrix `A`, use the function `A.mean()`.