Lecture 5:
The Calculus of Images

Discrete vs. Continuous

- Images are discrete objects. We are only given data at pixel locations.
- But many important geometry concepts are defined for continuous functions.
  - Derivatives and gradients
  - Area and volume
  - Curvature
  - Arc Length

Images as Functions

- We can think of an image as a function of two variables $f(x,y)$ defined on some rectangular domain Ω.
- We know the value of the function at integer locations, e.g. $f(2,3)$.
- But what is the value at non-integer locations, like

$f(2.2, 3.4)$?

- Our data exists at discrete integer pixel locations. But we can *pretend* that the values exist in between the pixels.
- This allows us to discuss continuous concepts like derivatives and integrals on images.

Discretization

- <u>Discretization</u> is the process of *approximating* a mathematical concept defined for continuous objects (like functions) into an equivalent concept for discrete objects (like images).

Continuous mathematical concept

Discretization

Discrete approximation

<u>*Example*</u>*: Riemann Sum*

$$\int_a^b f(x)\,dx$$

$$\Downarrow$$

$$\sum_{i=1}^{n} f(x_i)\,\Delta x$$

Finite Differences

- Let's start with a 1D signal f(x).
- Recall the definition of the derivative.

$$f_x = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- But we can't let h go to zero. The smallest h can become is 1, because our data points are 1 pixel apart.

- So we *approximate* the derivative with h=1:

$$f_x \approx f(x+1) - f(x)$$

- This type of approximation of the derivative is called a finite difference.

Boundary Conditions

- So for our signal f(x), we can approximate the derivative f_x at each point by looking at the difference with the next point.
- Suppose our vector has length n: n = length(f);
- What happens when we reach the last point?

```
f_x(1) = f(2) - f(1);
f_x(2) = f(3) - f(2);
f_x(3) = f(4) - f(3);
        ....
f_x(n-1) = f(n) - f(n-1)
f_x(n) = ???
```

- Why does this code not work? f_x = f(2:n) - f(1:n);

Boundary Conditions

- Neumann boundary conditions assumes an unknown derivative at the end of the data is zero.

```
f_x(1) = f(2) - f(1);
f_x(2) = f(3) - f(2);
f_x(3) = f(4) - f(3);
        ....
f_x(n) = f(n) - f(n) = 0;
```

- We can code this elegantly in one line of Matlab code using the colon operator. Just repeat the last entry.

```
f_x = f([2:n,n]) - f(1:n);
```

OR

```
f_x = f([2:n,n]) - f;
```

Derivative of a Sine Wave

```
x = 0:0.1:2*pi;
f = sin(x);
n = length(f);
f_x = f([2:n,n]) - f;
subplot(121);  plot(x, f);
subplot(122);  plot(x, f_x);
```

Finite Difference Schemes

- There are several ways we could approximate the derivative f_x. Different approaches are called <u>schemes</u>.
- **<u>Forward</u> <u>Difference</u>**: h=1

  $$f_x \approx D_x^+ f = f(x+1) - f(x)$$

  ```
  f_x = f([2:n,n]) - f;
  ```
- **<u>Backward</u> <u>Difference</u>**: h=-1

  $$f_x \approx D_x^- f = f(x) - f(x-1)$$

  ```
  f_x = f - f([1,1:n-1]);
  ```
- **<u>Central</u> <u>Difference</u>**: h=2

  $$f_x \approx D_x^0 f = \frac{f(x+1) - f(x-1)}{2}$$

  ```
  f_x = ( f([2:n,n]) - f([1,1:n-1]) ) / 2;
  ```

Partial Derivatives

- An image is 2-dimensional, so we have a derivative in the x-direction and a derivative in the y-direction.
- Let f(x,y) be a grayscale image.
- The forward differences would give:

```
[m,n] = size(f);
f_x = f(:,[2:n,n]) - f;
f_y = f([2:m,m],:) - f;
```

$f_x \approx D_x^+ f = f(x+1,y) - f(x,y)$

$f_y \approx D_y^+ f = f(x,y+1) - f(x,y)$

Partial Derivatives

- The derivative in x-direction u_x locates vertical edges.
- The derivative in y-direction u_y locates horizontal edges.

```
A = imread('cameraman.tif');
A = double(A);
[m,n] = size(A);
A_x = A(:,[2:n,n]) - A;
A_y = A([2:m,m],:) - A;
subplot(121);  imagesc(A_x);
subplot(122);  imagesc(A_y);
```

Finite Differences as Filters

- You can think of a finite difference as a 3x3 linear filter applied to an image.

- Forward Difference: h=1

$$f_x \approx D_x^+ f = f(x+1,y) - f(x,y)$$

- Backward Difference: h=-1

$$f_x \approx D_x^- f = f(x,y) - f(x-1,y)$$

- Central Difference: h=2

$$f_x \approx D_x^0 f = \frac{f(x+1,y) - f(x-1,y)}{2}$$

Other Derivative Approximations

- The Prewitt filter uses more pixels so it is less sensitive to noise. But it de-emphasizes values near the center.

$$u_x \approx \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \quad u_y \approx \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

- The Sobel filter gives more emphasis to changes around the center pixel.

$$u_x \approx \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad u_y \approx \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

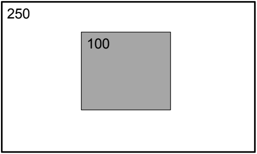
- There are many other finite difference schemes, like upwind and minmod. Each has pros and cons.

The Gradient

- The gradient is a 2D vector listing the values of the partial derivatives at each point:

$$\nabla u = \langle u_x, u_y \rangle$$

- The gradient always points in the direction of maximum positive change (dark to light).

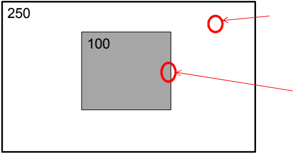


Norm of Gradient

- The norm (magnitude) of the gradient vector tells us the total amount of change at each pixel.

$$\|\nabla u\| = \sqrt{u_x^2 + u_y^2}$$

- The norm of the gradient is large at edges of the image and zero in flat (single color) regions.



Edge Detector

- We use the norm of the gradient to detect edges.

```
P = imread('pout.tif');
P = double(P);
[m,n] = size(P);
Px = P(:,[2:n,n]) - P;
Py = P([2:m,m],:) - P;
N = sqrt(Px.^2 +Py.^2);
imagesc(N);
```

Note the .^ for pointwise exponents.

Second Derivatives

- To approximate a second derivative, we take a finite difference of a finite difference.

$$u_{xx} \approx D_x^-(D_x^+ u)$$

- Note we use one forward and one backward difference.

3 Ways to Code u_{xx}

1.) Forward then Backward Difference.

$$u_{xx} \approx D_x^-(D_x^+ u)$$

```
Dplus = u(:,[2:n,n]) - u;
u_xx = Dplus - Dplus(:,[1,1:n-1]);
```

2.) Backward then Forward Difference.

$$u_{xx} \approx D_x^+(D_x^- u)$$

```
Dminus = u - u(:,[1,1:n-1]);
u_xx = Dminus(:,[2:n,n]) - Dminus;
```

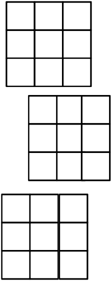
3.) Write out the formula.

$$u_{xx} \approx u(x+1,y) - 2u(x,y) + u(x-1,y)$$

```
u_xx = u(:,[2:n,n]) - 2*u + u(:,[1,1:n-1]);
```

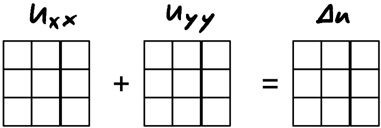
Second Derivatives

```
[m,n] = size(u);
% Second derivative in x:  u_xx
u_xx = u(:,[2:n,n]) - 2*u + u(:,[1,1:n-1]);
% Second derivative in y: u_yy
u_yy = u([2:m,m],:) - 2*u + u([1,1:m-1],:);
% Diagonal derivative u_xy
u_xy = ( u([2:m,m],[2:n,n])
        + u([1,1:m-1],[1,1:n-1])
        - u([1,1:m-1],[2:n,n])
        - u([2:m,m],[1,1:n-1]) ) / 4;
```



The Laplacian

- The Laplacian is the sum of the second derivatives:
$$\Delta u = u_{xx} + u_{yy}$$
- Recall we implemented a Laplacian filter.
- We use the Laplacian to locate edges. Subtracting the Laplacian sharpens the images.



Double Integrals

- The double integral of u(x,y) over the domain Ω is

$$\iint_{\Omega} u(x, y)\, dx\, dy$$

- The discrete approximation is simply a double summation of all values of u(x,y):

  d = sum(sum(u));

- Sometimes we get lazy and vectorize the variables as $\vec{x} = (x, y)$ so we can write a single integral:

$$\int_{\Omega} u(\vec{x})\, d\vec{x}$$

- But don't let this fool you, it's still a double sum!

Measuring Noise

- We measured the noise levels last week using SNR and RMSE.
- But these statistics require an ideal noise-free image, which in general we don't have.
- We would like a way to judge how much noise an image contains without requiring a magical perfect image.

Total Variation

- The Total Variation (TV) energy of an image u(x,y) is found by adding up the norm of the gradient *(Rudin-Osher-Fatemi, 1989)*.

$$TV(u) = \iint_{\Omega} \|\nabla u\| \, dx \, dy$$

- We interpret TV as the total amount of jumps (variation) in the image.
- Or if we vectorize $\vec{x} = (x, y)$, we can write as

$$TV(u) = \int_{\Omega} \|\nabla u\| \, d\vec{x}$$

1D TV

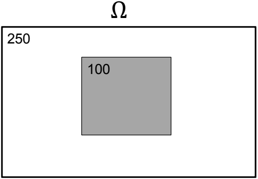
- The 1-dimensional version for a function f(x) on [a,b] is

$$TV = \int_a^b |f'(x)|dx$$

- <u>Ex</u> Calculate the TV value of a sine wave on $[0,2\pi]$.

2D TV

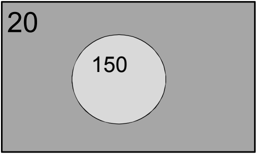
- Calculate TV energy for the image below.
- Assume the image is 100x200 pixels and the gray square is 30x30 pixels.



The Co-Area Formula

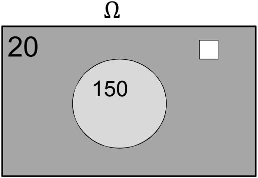
- Co-Area Formula: The TV norm is equal to the perimeter of each shape times the jump at the perimeter.
- Suppose we have a circle of radius 5 pixels on a dark gray background. Calculate the TV energy value.

Ω



Noise on TV

- Again suppose we have a circle of radius 5 pixels on a dark gray background.
- Let's add one noise pixel with a value 250.



Measuring Noise

- TV does not tell us exactly how much noise is in the image, but if we have a version of an image with a high TV value then it is probably noisy.

TV Energy = 11,150,000

TV Energy = 1,830,000

Your Very Own TV

- Ex Write a function that calculates the TV energy value of a grayscale image.

$$TV(u) = \iint_{\Omega} \|\nabla u\| \, dx \, dy$$

- We'll have to discretize this energy, so really we'll be computing an approximation of the TV energy.

Curvature

- The curvature of a surface u(x,y) measures how quickly the unit tangent vector to the surface is changing:

$$\boxed{\kappa = \nabla \cdot \left(\frac{\nabla u}{\|\nabla u\|} \right)}$$

- Ex Write a function that computes the curvature matrix of a grayscale image. Watch out for division by zero!