CSE 559A: Computer Vision

Fall 2020: T-R: 11:30-12:50pm @ Wrighton 300 / Zoom

Instructor: Ayan Chakrabarti (ayan@wustl.edu).
Course Staff: Adith Boloor, Patrick Williams

http://www.cse.wustl.edu/~ayan/courses/cse559a/

Sep 22, 2020
• Problem Set 0 Due Today! Ungraded, but mandatory.
• PSET 1 is now out
  - We haven’t covered all the material yet.
    - You should be able to do the first two problems after today’s class though.
  - Released early so that you have a sense of the problem sets, and to get an early start.
• Office Hours
  - Patrick: Tuesdays: 8am - 9am US Central Time
  - Adith: Fridays: 9am-10am US Central Time
• Will be on Zoom. Links will be available through Canvas. Not Recorded
• Some of the Friday office hours will be “Recitations”.
  - Announced in advance. Roughly one for each problem set.
  - Those Fridays, 9am-10am will be the recitation, followed by regular office hours from 10am-11am
• We will continue to answer questions asynchronously through Piazza.
  - Can set up office hours with me by appointment.
    - Make private post on Piazza letting me know your availability for the next 2-3 days.
RECAP

- Talked about how light becomes an image
  - How an image is formed on the sensor plane
  - Intensities on sensor plane then measured and turned into intensities
  - Discussed noise and other non-idealities in that process
  - Talked about how color is handled: multiplexed measurements with filters + interpolation
OTHER EFFECTS

Other effects we did not talk about. E.g.,

- Real Lenses not thin lenses and have distortions:
  
  Radial Distortion
  Vignetting
  Chromatic Aberration
OTHER EFFECTS

Other effects we did not talk about. E.g.,

Rolling Shutter: No explicit shutter but when pixels reset electronically (along scanlines)
Place array of micro-lenses in front of sensor
Different pixels observe the scene "from a different angle"
Let you estimate depth, shift view point post-capture,
refocus post-capture
16 Different Camera Units
Fuse images to get lower noise, higher dynamic range, synthetically control focal distance and aperture size post-capture
M.S. Asif, A. Ayremolu, A. Sankaranarayanan, A. Veeraraghavan, R. Baranuik, 
"FlatCam: Thin, Bare-Sensor Cameras using Coded Aperture and Computation", 2015.

No lens: lets the camera be much smaller, sensor can be placed on curved surfaces. 
Put a mask in the aperture: causes 'patterns' of confusion instead of circles of confusion. 
Create focused image computationally.
Coded Aperture Photography

NON-STANDARD CAMERAS

Synthetic Focus Post-capture

Coded Aperture Photography

NON-STANDARD CAMERAS

Synthetic Focus Post-capture

Coded Aperture Photography

Unbounded High Dynamic Range Photography using a Modulo Camera

Hang Zhao¹, Boxin Shi¹, ³ Christy Fernandez-Cull², Sai-Kit Yeung³, Ramesh Raskar¹

¹ MIT Media Lab ² MIT Lincoln Lab ³ SUTD

ICCP 2015, Houston, TX [Best Paper runner-up]
IMAGES

- Exist as 2-D (grayscale) or 3-D (color image) arrays

- Precision: uint8 (0-255), uint16(0-65535), Floating point (0-1)
  - We will often treat them as (positive) real numbers.

- Conventions:
  - $I_{n_x, n_y} \in \mathbb{R}$ (for a grayscale image), where $n_x, n_y \in \mathbb{Z}$
  - $I_{n_x, n_y, c} \in \mathbb{R}$ (for a color image)
  - $I_{n_x, n_y} \in \mathbb{R}^3$ (for a color image)
  - $I[n] \in \mathbb{R}$ or $\in \mathbb{R}^3$, where $n \in \mathbb{Z}^2$

[numpy order: (H, W) or (H,W,3)]
NUMPY CONVENTIONS

- An image will exist as a numpy array when you load it, with dimensions (H,W) or (H,W,C)
- In numpy, you can recover sub-arrays by indexing into the main array.

```python
# Assume img is a 3D array of shape (H,W,3)
img2 = img[:,:,0]  # Returns a 2D array of shape (H,W) of red intensities.

img2 = img[:,:,0:1]  # Same, but as a 3D array of shape (H,W,1).

img2 = img[0:10,0:10,:]  # Returns array of shape (10,10,3). The top 10x10
# patch of the image (going from pixels 0 to **9**)

img2 = img[0:10:2,0:10:2,:]  # Returns array of shape (5,5,3). Every alternate
# pixel in top 10x10 patch starting from 0,0.
```
When a number is omitted in the \([a:b:c]\) convention, \(a\) and \(b\) are assumed to be start or end of the image along that axis (depending on whether \(c\) is positive or negative), and \(c\) is assumed to be 1.

```
img2 = img[::2,::2,:) # Returns array of shape (5,5,3). Every alternate pixel in top 10x10 patch starting from 0,0.
```

```
img2 = img[1:10:2,1:10:2,:] # (5,5,3), but pixels that were 'dropped' above.
```

```
img2 = img[::2,::2,:] # Every alternate pixel for the whole image.
```

```
img2 = img[:::-1,:::-1,:] # Skips can be negative, this 'flips' the image.
```

```
img[:,::2,::2] = img2
```

Indexing can also be used for assignment. `img2` should be the right shape of the “slice” corresponding to `img[:,::2,::2,:]`. 

Note that array variables are “pointers” to data. Assignment (without indexing) will not create a new copy of the data automatically.

```python
ing2 = img  # img2 points to the same data as img
img2[0, 0, 0] = 0.5  # This will also change img
img3 = img.copy()  # This creates a copy of the data
img3[0, 0, 0] = 0.5  # This will only change img3
```
NUMPY CONVENTIONS

- Reshaping: All of your data in arrays is stored sequentially, with the last dimension changing fastest. For an array of size (H,W,3), \( \text{img}[a,b,c+1] \) is stored right after \( \text{img}[a,b,c] \), while \( \text{img}[a,b+1,c] \) is stored 3 locations later.

```python
>>> print(v)  # is a (3,3) array
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```python
>>> s = v.reshape((9))
>>> print(s)
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
```

```python
>>> v2 = s.reshape((3,3))
>>> print(v2)
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```python
>>> s = v.reshape((-1))
```

- \text{array.reshape()} \ takes a numpy tuple of the shape. Its product should match the total number of elements in the array. Up to one dimension can be -1, and numpy will automatically calculate its value to match.
NUMPY CONVENTIONS

- Simple shortcut: \( X[:, \text{np.newaxis}, :] \) is equivalent to \( X.\text{reshape}((H, 1, W)) \) if \( X \) had shape \((H, W)\).

IMPORTANT: Array reshares is not the same as taking “transpose”.

```python
>>> print(v)  # is a (3,2) array
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> v2 = np.reshape((2, 3))
>>> print(v2)
[[0 1 2]
 [3 4 5]]
```

- Also, doing a reshape doesn’t always create a copy of the data (just changes the headers). So again, you might want to use `.copy()`.

- Ok, so now, what are some of the operations we can perform on images?
POINT-WISE OPERATIONS

- $Y[n] = h(X[n])$
- $Y[n] = h(X_1[n], X_2[n], \ldots)$
- $Y[n] = h_n(X[n])$ - Might vary based on location.
- $h(\cdot)$ itself might be based on ‘global statistics’
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]

Linear Color Transforms

\[
Y[n] = \begin{bmatrix}
0.7 \\
1.05 \\
0.7
\end{bmatrix} \begin{bmatrix}
X[n]
\end{bmatrix}
\]
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]

\[
Y[n] = \begin{bmatrix}
0.7 & 0.7 & 1.05
\end{bmatrix} X[n]
\]

Linear Color Transforms
POINT-WISE OPERATIONS

$X[n] \in \mathbb{R}^3$

$Y[n] = \begin{bmatrix} 1.05 & 0.7 \\ 0.7 & 0.7 \end{bmatrix} X[n]$

Linear Color Transforms
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]

\[ Y[n] = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix} X[n] \]

Linear Color Transforms
POINT-WISE OPERATIONS

$X[n] \in \mathbb{R}^3$

$Y[n] = \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix} X[n]$

Linear Color Transforms
$X[n] \in \mathbb{R}^3$

$Y[n] = 0.9 \begin{bmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix} X[n] + 0.1 \begin{bmatrix} X[n] \end{bmatrix}$

Linear Color Transforms
**POINT-WISE OPERATIONS**

\[ X[n] \in \mathbb{R}^3 \]

Linear Color Transforms

\[
Y[n] = 0.9 \begin{bmatrix}
1/3 & 1/3 & 1/3 \\
1/3 & 1/3 & 1/3 \\
1/3 & 1/3 & 1/3 \\
\end{bmatrix} X[n] + 0.1 \begin{bmatrix}
X[n] \\
\end{bmatrix}
\]
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]

Linear Color Transforms

\[
Y[n] = 0.5 \begin{bmatrix}
1/3 & 1/3 & 1/3 \\
1/3 & 1/3 & 1/3 \\
1/3 & 1/3 & 1/3
\end{bmatrix} X[n] + 0.5 X[n]
\]
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]

\[ Y[n] = \begin{bmatrix} -1.0 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \\ 1/3 & 1/3 & 1/3 \end{bmatrix} X[n] + 2.0 X[n] \]

Linear Color Transforms
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R}^3 \]

Linear Color Transforms

\[ Y[n] = \begin{bmatrix} -1.0 \\ 1/3 \\ 1/3 \\ 1/3 \\ 1/3 \\ 1/3 \end{bmatrix} X[n] + 2.0 \]
POINT-WISE OPERATIONS

\[ X[n] \in \mathbb{R} \]

\[ Y[n] = X[n]^{0.5} \]

Non-linear Transforms
POINT-WISE OPERATIONS

$X[n] \in \mathbb{R}$

$Y[n] = X[n]^{2.0}$

Non-linear Transforms
POINT-WISE OPERATIONS

$X[n] \in \mathbb{R}$

$Y[n] = X[n]^{2.0}$

Non-linear Transforms  Tone-maps / Change contrast
POINT-WISE OPERATIONS

\[ Y[n] \]

\[ X[n] \]
POINT-WISE OPERATIONS

\[ Y[n] \]

\[ X[n] \]
Can be arbitrary: but usually monotonic
POINT-WISE OPERATIONS

Can be arbitrary: but usually monotonic

People play around with this a lot manually: but also can be done automatically for some objective
Uneven distribution of intensities (many too dark or too bright)
Uneven distribution of intensities (many too dark or too bright)

Find a monotonic function $h(\cdot)$ such that the intensities of $h(X[n])$ are distributed uniformly in the range $[0, 255]$. 

Histogram Equalization
Consider the Cumulative Distribution (think of intensity as a continuous r.v.)
Consider the Cumulative Distribution (think of intensity as a continuous r.v.)

For a uniform distribution
Consider the Cumulative Distribution (think of intensity as a continuous r.v.)

For a uniform distribution
Consider the Cumulative Distribution (think of intensity as a continuous r.v.)

Use the CDF as the tone map: \( h(x) = P(X[n] < x) \times 255.0 \)
POINT-WISE OPERATIONS

Not perfect as dealing with quantized values
POINT-WISE OPERATIONS

Image Matting (combine multiple images with alpha matte)

From Szeliski 3.1
Let’s say you wanted to implement \( Y[n] = X[n]^{0.5} \).

\[
Y = np.zeros_like(X)
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        for k in range(X.shape[2]):
            Y[i,j,k] = X[i,j,k]**0.5
\]

NEVER DO THIS ! Never loop through pixels. This is very inefficient.

\[
Y = X**0.5
\]

This will do the same thing. Not only will you not be going through loops in the interpreter, numpy will internally use available parallelism on your CPU to do this much faster.

In general, numpy provides a large number of point-wise functions for array. It also overloads many common arithmetic operators to these functions.

\[
Y = X1+X2
\]

\[
Y = np.add(X1,X2) \ # Equivalent
\]
If $X_1$ and $X_2$ are the same shape, then $Y$ will be the same shape and $Y[i,j,k] = X_1[i,j,k] + X_2[i,j,k]$

If they are not the same shape, numpy applies what is called “broadcasting”. This means that you can replace the size of any dimension with 1.

$X_1$ is $(H,W,3)$, and $X_2$ is $(H,1,3)$. Then $Y$ has shape $(H,W,3)$:

$Y[i,j,k] = X_1[i,j,k] + X_2[i,0,k]$

When two arrays have different numbers of dimensions, numpy assumes that any missing **leading** dimensions are 1.

- $X_1$ is $(H,W,3)$ and $X_2$ is $(3)$. Then $X_2$ is “reshaped” automatically to $(1,1,3)$.

- If $X_1$ is $(H,W,3)$ and $X_2$ is $(H,W)$, then you will need to reshape $X_2$ to size $(H,W,1)$:

  $Y = X_1 + X_2[:, :, np.newaxis]$

If $X_2$ is a scalar or constant, it is assumed to have shape (). It is broadcast to all dimensions, which is why $X^{**0.5}$ works!
POINTWISE OPERATIONS IN NUMPY

• Your goal, try to express pointwise operations in a way that expresses it as a combination of parallel numpy operations.

• For example, for alpha matting, let’s say you have two images $X_1$ and $X_2$ both of size (H,W,3), and a mask of size (H,W) that goes from 0-1 (1 indicating keep $X_1$, and 0 indicating keep $X_2$):

$$Y = X_1 \cdot \text{mask}[:,:,\text{np.newaxis}] + X_2 \cdot (1. - \text{mask}[:,:,\text{np.newaxis}])$$

• But what about the linear matrix multiplies we talked about?

• Not that the * operator, which corresponds to np.mul, does element-wise multiply.

• np.matmul does matrix multiplies, but what we want are matrix multiplies applied to each pixel’s color vector.