COMP 204
Introduction to image analysis with scikit-image
(part two)

Yue Li
based on slides from Mathieu Blanchette, Christopher J.F. Cameron and Carlos G. Oliver
Back to...Flipping the image horizontally

How to turn flip an image horizontally?
Wrong!

```python
import skimage.io as io
import skimage.color as color
import matplotlib.pyplot as plt

# read image into memory
image = io.imread("monkey.jpg")
n_row, n_col, colours = image.shape

# Flip the image horizontally
for i in range(0, n_row):
    for j in range(0, n_col):
        image[i, j] = image[i, n_col-j-1]

plt.imshow(image)
plt.show()
io.imsave("monkey_flipped_wrong.jpg", image)
```
A second attempt

```python
import skimage.io as io
import skimage.color as color
import matplotlib.pyplot as plt

# read image into memory
image = io.imread("monkey.jpg")
n_row, n_col, colours = image.shape

# Flip the image horizontally
for i in range(0, n_row):
    for j in range(0, int(n_col/2)):
        # swap pixels (i, j) and (i, n_col−j−1)
        t = image[i, j, :]
        image[i, j, :] = image[i, n_col−j−1, :]
        image[i, n_col−j−1, :] = t

plt.imshow(image)
plt.show()
io.imsave("monkey_flipped_wrong.jpg", image)
```
Wrong again!
What’s going on?

```python
# swap pixels (i, j) and (i, n_col−j−1)
t = image[i, j, :]
image[i, j, :] = image[i, n_col−j−1, :]
image[i, n_col−j−1, :] = t
```

t refers to the same memory locations (RGB values) as image[i,j].

When we change image[i,j] (on line 3), the values pointed by t is also changed!

So this is not swapping the two pixels: image[i, n_col-j-1] remains unchanged.
import skimage.io as io
import skimage.color as color
import matplotlib.pyplot as plt

# read image into memory
image = io.imread("monkey.jpg")
n_row, n_col, colours = image.shape

# Flip the image horizontally
for i in range(0,n_row):
    for j in range(0,int(n_col/2)):
        t = image[i,j].copy()  # creates new copy of pixel
        image[i,j] = image[i, n_col−j−1]
        image[i, n_col−j−1] = t

        # we could also do:
        #for c in range(3):
        #    t = image[i,j,c]  # this is now a float value
        #    image[i,j,c] = image[i, n_col−j−1,c]
        #    image[i, n_col−j−1,c] = t

plt.imshow(image)
plt.show()
io.imsave("monkey_flipped_right.jpg",image)
Ah, finally!
Grayscaling

Many image processing algorithms assume a 2D matrix
▶ not an image with a third dimension of color

To bring the image into two dimensions
▶ we need to summarize the three colors into a single value
▶ this process is more commonly know as **grayscaling**
▶ where the resulting image only holds intensities of gray
  ▶ with values between 0 and 1

skimage submodule **color** has useful functions for this task
▶ API
  http://scikit-image.org/docs/dev/api/skimage.color.html
```python
import skimage.io as io
import skimage.color as color
import matplotlib.pyplot as plt
from skimage.color import rgb2gray

# read image into memory
image = io.imread("monkey.jpg")

# convert to grayscale
gray_image = rgb2gray(image)

print(image[0, 0])  # prints [255, 255, 255]
print(gray_image[0, 0])  # prints 1.0
plt.imshow(gray_image)
plt.show()
io.imsave("monkey_grayscale.jpg", gray_image)
```
Binary image

```python
import skimage.io as io
import skimage.color as color
import matplotlib.pyplot as plt
from skimage.color import rgb2gray
import numpy as np

image = io.imread("monkey.jpg")
gray_image = rgb2gray(image)

# this creates a new array, # with 1's everywhere gray_image > 0.5, and 0 elsewhere
black_and_white = np.where(gray_image > 0.5, 255, 0)

plt.imshow(black_and_white)
plt.show()
io.imsave("monkey_black_and_white.jpg", black_and_white)
```
Blurring an image

Goal: Reduce the resolution of an image by blurring it, e.g. to reduce fine-level ”noise” (unwanted details).
Blurring an image

Blurring is achieved by replacing each pixel by the average value of the pixels in a small window centered on it. Example, window of size 5:
Blurring an image

def blur(image, filter_size):
    n_row, n_col, colors = image.shape
    blurred_image = np.zeros((n_row, n_col, colors), dtype=np.uint8)
    half_size = int(filter_size / 2)
    for i in range(n_row):
        for j in range(n_col):
            # define the boundaries of window around (i, j)
            left = max(0, i - half_size)
            right = min(i + half_size, n_row)
            bot = max(0, j - half_size)
            top = min(n_col, j + half_size)

            # calculate average of RGB values in window
            blurred_image[i, j] = \n                image[left:right, bot:top, :].mean(axis=(0, 1))

    return blurred_image

means(axis=(0, 1)) states that we want to take an average over
dimension 0 (rows) and dimension 1 (columns) but not dimension
2 (RGB).
This means that we get back a 1d array containing the average
red, green, and blue values in window.
Window size = 5
Window size = 21
Window size = 101
Running time issues

Note: When our window size is large (say 101), blurring the image is slow (> 1 minute). Why?

▶ Our image is 674 × 1200 pixels.
▶ For each pixel in the image, we need to calculate the average of the 101 × 101 pixels around it, and for each of the three colors!
▶ The total number of operations is proportional to 674 × 1200 × 101 × 101 = 25 Billion operations!

SkImage has many built-in blurring functions (called filters) with faster implementations:
http://scikit-image.org/docs/dev/api/skimage.filters.html
Edge detection

Goal: Identify regions of the image that contain sharp changes in colors/intensities.
Why? Useful for
- delineating objects (image segmentation)
- recognizing them (object recognition)
- etc.
Edge detection
Edge detection
Edge detection

What’s an edge in an image?

Horizontal edge at row $i$: \( \text{image}(i - 1, j) \) is very different from \( \text{image}(i + 1, j) \)

Vertical edge at column $j$: \( \text{image}(i, j - 1) \) is very different from \( \text{image}(i, j + 1) \)

Idea:
For each position \((i, j)\) and each color (RGB), calculate
\[
\begin{align*}
\text{change_hor} &= \text{image}(i, j-1, \text{color}) - \text{image}(i, j+1, \text{color}) \\
\text{change_vert} &= \text{image}(i-1, j, \text{color}) - \text{image}(i+1, j, \text{color}) \\
\text{edge_image}(i, j, \text{color}) &= \sqrt{(\text{change_hor}^2 + \text{change_vert}^2)}
\end{align*}
\]
def detect_edges(image):
    n_row, n_col, colors = image.shape
    edge_image = np.zeros((n_row, n_col, 3), dtype=np.uint8)
    for i in range(1, n_row - 1):
        for j in range(1, n_col - 1):
            for c in range(3):

                # conversion to int needed to accommodate
                # for potentially negative values
                d_r = int(image[i-1, j, c]) - int(image[i+1, j, c])
                d_c = int(image[i, j-1, c]) - int(image[i, j+1, c])
                gradient = math.sqrt(d_r**2 + d_c**2)

                # limit value to 255
                edge_image[i, j, c] = np.uint8(min(255, gradient))

    return edge_image
Edge detection on monkey image

Not so great if our goal is to find the monkey in the image!
Blurring + Edge detection

To smooth out fine details like leaves:
Start by blurring the image, then apply edge detection.
Analysis of microscopy images
Edge detection
Edge detection

Skimage has many edge detection algorithms:
http://scikit-image.org/docs/0.5/auto_examples/plot_canny.html