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

Outline

Assignment 5

Image compression by matrix decomposition (very brief)

Brief survey on image denoising and inpainting

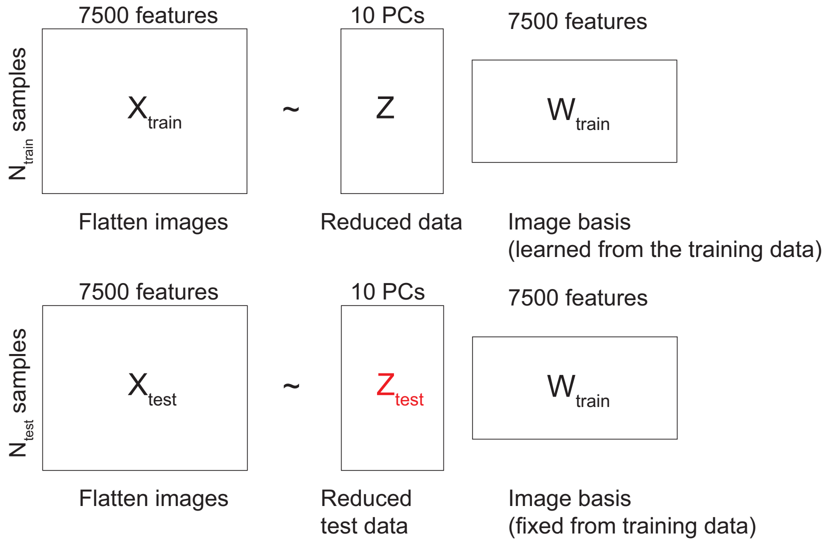
Image processing (caveat & review from last lecture)

Image blurring

Edge detection

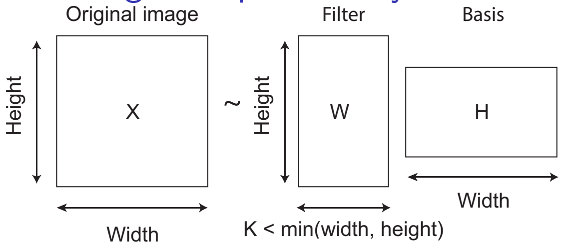
Assignment 5 posted
open Jupyter Notebook

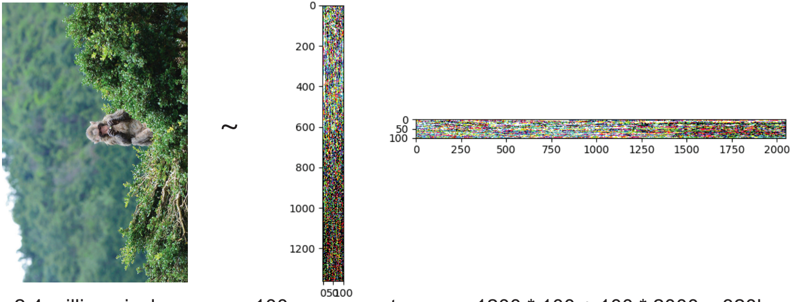
Principal Component Analysis (PCA) (background to A5)



Outline

Image compression by matrix decomposition





2.4 million pixels

100 components ⟶ 1200 * 100 + 100 * 2000 = 320k

(7.5 times smaller than the original image!)

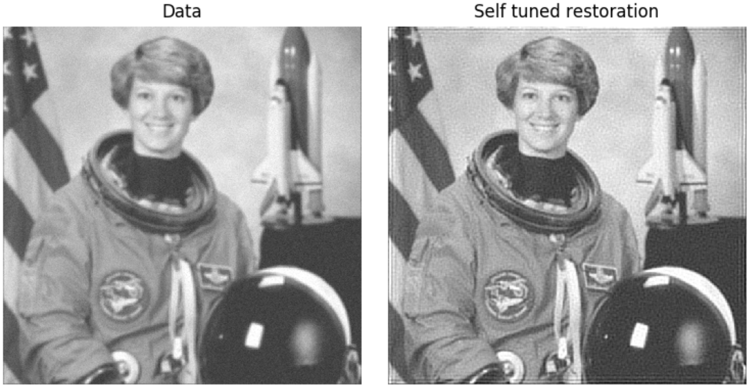
Running non-negative matrix factorization with sklearn

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import NMF
import skimage.io as io

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

image_imputed = image.copy()

k = 100
Ws = np.zeros((image.shape[0], k, 3))
Hs = np.zeros((k, image.shape[1], 3))

for c in range(3):
    print(c)
    model = NMF(n_components=k, init='random',
     ↪  random_state=0)
    image_imputed[:,:,c] = image[:,:,c]
    W = model.fit_transform(image[:,:,c])
    H = model.components_
    image_imputed[:,:,c] = np.dot(W, H)
    Ws[:,:,c] = W
    Hs[:,:,c] = H
```

Reconstructed image (lossy de-compression)

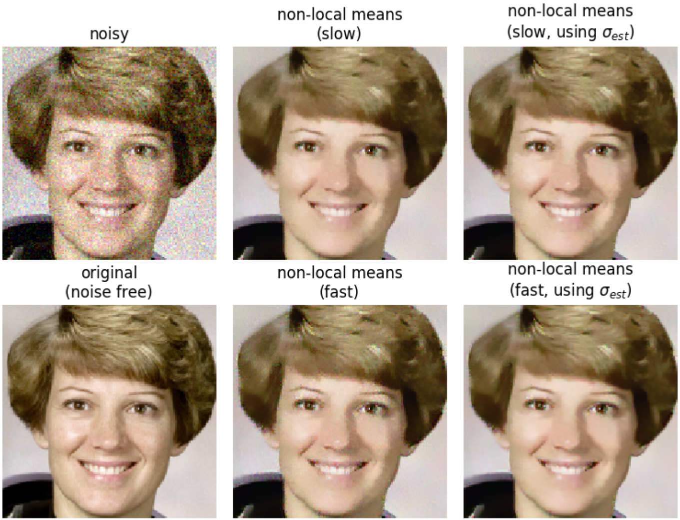
Outline

Image restoration



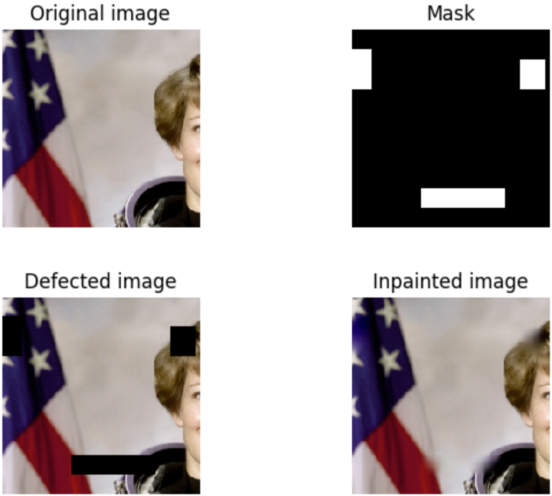
http://scikit-image.org/docs/dev/auto_examples/filters/plot_restoration.html

Image denoising



http://scikit-image.org/docs/dev/auto_examples/filters/plot_nonlocal_means.html

Image inpainting



http://scikit-image.org/docs/dev/auto_examples/filters/plot_inpaint.html

Outline

Assignment 5

Image compression by matrix decomposition (very brief)

Brief survey on image denoising and inpainting

Image processing (caveat & review from last lecture)

Image blurring

Edge detection

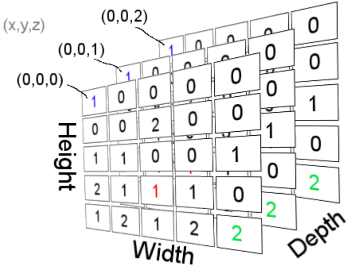
What's an image in Python? (recap)

An image is stored as a NumPy ndarray (n-dimensional array).

- ndarrays are easier and more efficient than using 2-dimensional lists as we've seen before.

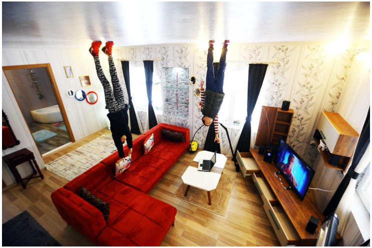
A color image with R rows and C columns is

- represented as a 3-dimensional ndarray of dimensions $R \times C \times 3$
- element at position (i, j) of the array corresponds to the RGB value at row i and column j
- each pixel is represented by 3 numbers, each between 0 and 255: Red, Green, Blue



Flipping the image up side down (recap)

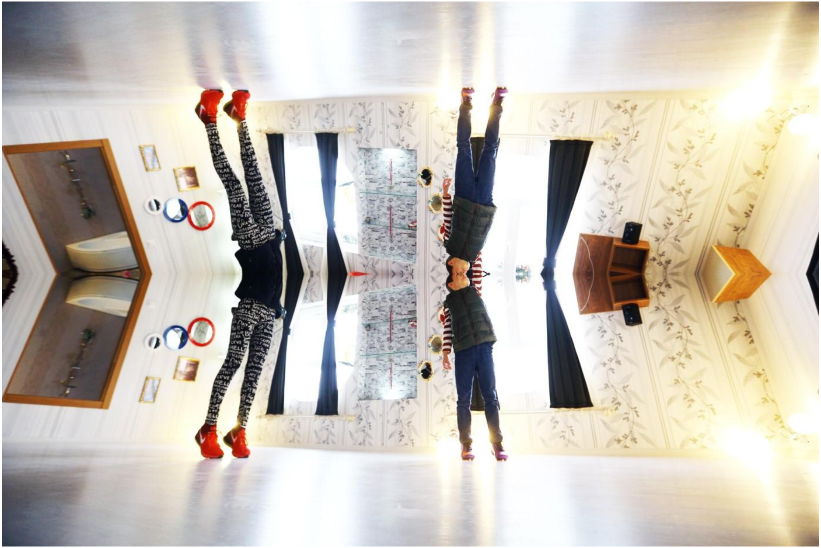
How to turn flip an image up side down?

 to 

Incorrect attempt 1

```python
def upsidedown_wrong1(image):
    n_row, n_col = image.shape[0:2]
    for i in range(0,int(n_row/2)):
        for j in range(0,n_col):
            image[i,j] = image[n_row-i-1,j]
    return image
```

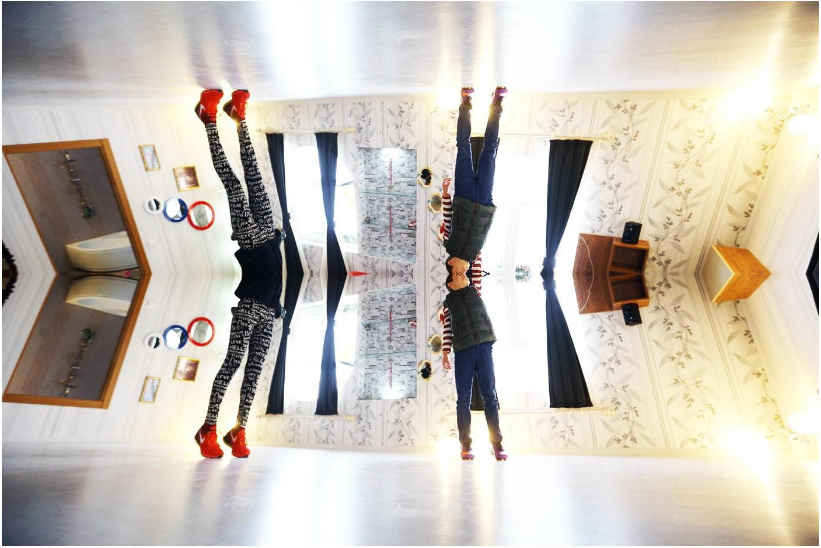
What went wrong?
The top half of the image is replaced by the bottom half of the image



Incorrect attempt 2

```
def upsidedown_wrong2(image):
    n_row, n_col = image.shape[0:2]
    for i in range(0,int(n_row/2)):
        for j in range(0,n_col):
            t = image[i,j]
            image[i,j] = image[n_row-i-1,j]
            image[n_row-i-1, j] = t
    return image
```

Still incorrect



What went wrong in attempt 2?

```
16 t = image[i,j]
17 image[i,j] = image[n_row-i-1,j]
18 image[n_row-i-1, j] = t
```

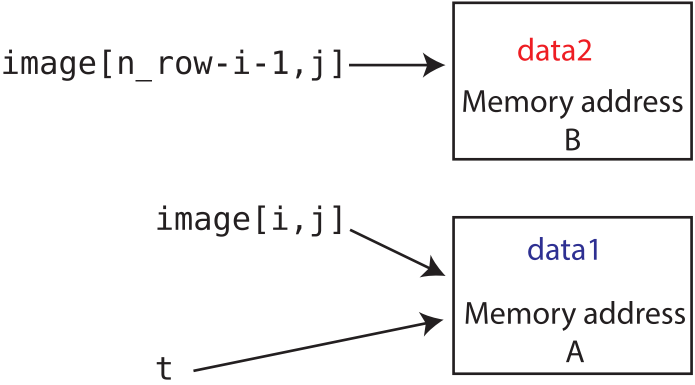
`t` refers to <u>the same memory locations</u> (RGB values) as `image[i,j]`.

When we change `image[i,j]` (on line 20), the values pointed by `t` is also changed!

So this is not swapping the two pixels: `image[n_row-i-1,j]` remains unchanged.

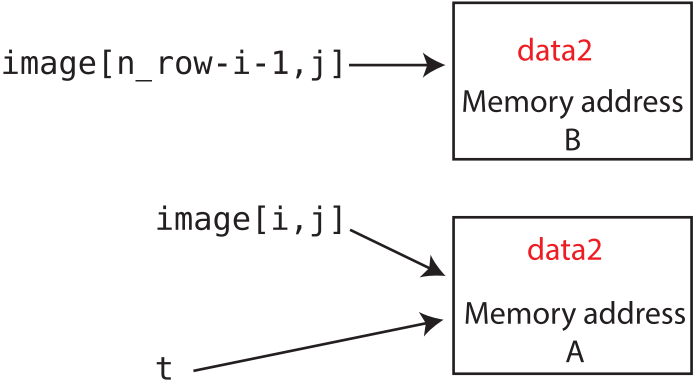
t and image[i,j] refers to the same memory address

```
t = image[i,j]
```



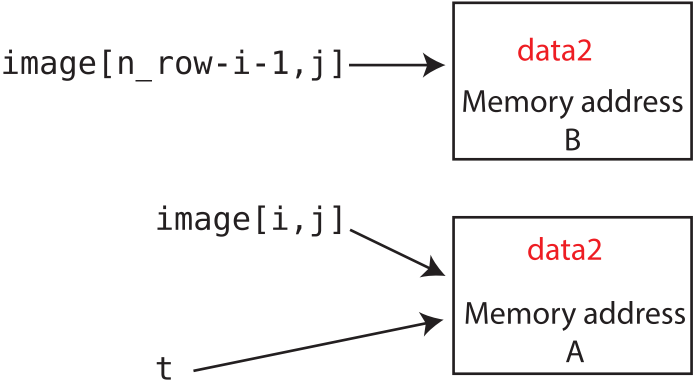
data1 in memory A is replaced by data2 in memory B

```
t = image[i,j]
image[i,j] = image[n_row-i-1,j]
```



Replacing data2 in memory B with data 2 in memory A

```
t = image[i,j]
image[i,j] = image[n_row-i-1,j]
image[n_row-i-1,j] = t
```

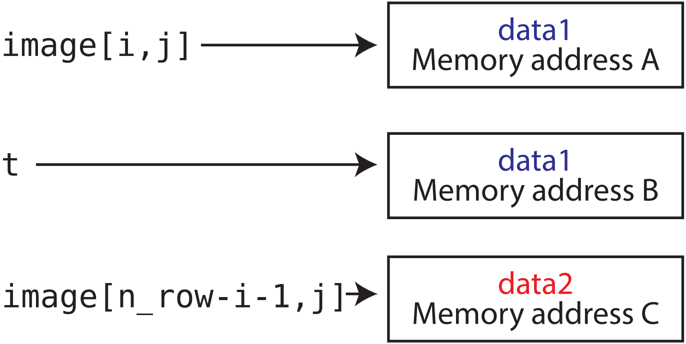


Correct way to do it (pay attention to line 25)

```
21 def upsidedown_correct1(image):
22     n_row, n_col = image.shape[0:2]
23     for i in range(0,int(n_row/2)):
24         for j in range(0,n_col):
25             t = image[i,j].copy()
26             image[i,j] = image[n_row-i-1,j]
27             image[n_row-i-1, j] = t
28     return image
```

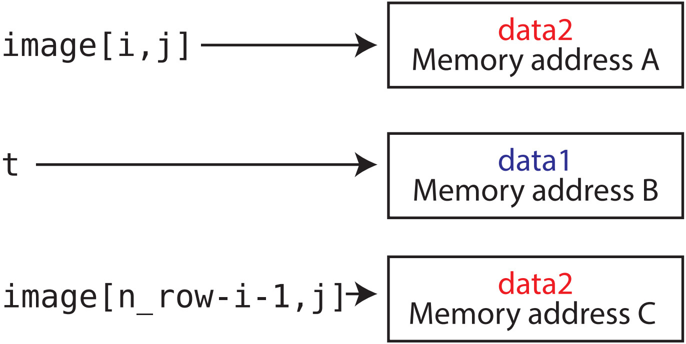
`t` and `image[i,j]` refers to the *different* memory address

```
t = image[i,j].copy
```



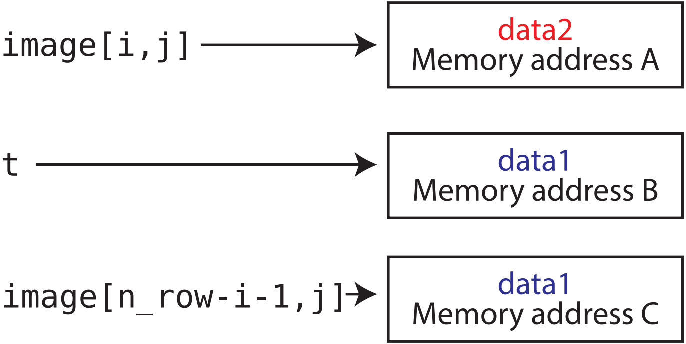
data1 in memory A is replaced by data2 in memory B

```
t = image[i,j].copy
image[i,j] = image[n_row-i-1,j]
```



Replacing data2 in memory B with data 2 in memory A

```
t = image[i,j].copy
image[i,j] = image[n_row-i-1,j]
image[n_row-i-1,j] = t
```



Correct output image

Another correct way to do it (pay attention to line 35)

```
30 def upsidedown_correct2(image):
31     n_row, n_col = image.shape[0:2]
32     for i in range(0,int(n_row/2)):
33         for j in range(0,n_col):
34             for c in range(3):
35                 t = image[i,j,c] # a float value
36                 image[i,j,c] = image[n_row-i-1,j,c]
37                 image[n_row-i-1, j, c] = t
38     return image
```

A new variable with a float value will be stored in a separate memory location. For a simpler example,

```
>>> a = 1
>>> b = a
>>> a = 3
>>> print(b) # 1
```

A couple of more correct ways to do it

```
def upsidedown_correct3(image):
    return image[::-1,:]
    # image[::-1,:] reverse rows
    # image[:,::-1] reverse columns


def upsidedown_correct4(image):
    return np.flip(image, 0)
    # axis=0 flip vertically;
    # axis=1 flip horizontally
```

Outline

Assignment 5

Image compression by matrix decomposition (very brief)

Brief survey on image denoising and inpainting

Image processing (caveat & review from last lecture)

Image blurring

Edge detection

Blurring an image

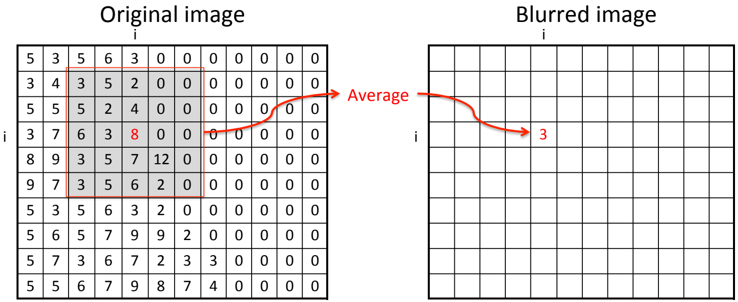
Goal: Reduce the resolution of an image by blurring it, e.g. to reduce fine-level ”noise” (unwanted details).

We may also want to place emphasis on certain area of the image (e.g., “portrait mode” on an iPhone camera)

to

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:



5	3	5	6	3	0	0	0	0	0	0	0
3	4	3	5	2	0	0	0	0	0	0	0
5	5	5	2	4	0	0	0	0	0	0	0
3	7	6	3	8	0	0	0	0	0	0	0
8	9	3	5	7	12	0	0	0	0	0	0
9	7	3	5	6	2	0	0	0	0	0	0
5	3	5	6	3	2	0	0	0	0	0	0
5	6	5	7	9	9	2	0	0	0	0	0
5	7	3	6	7	2	3	3	0	0	0	0
5	5	6	7	9	8	7	4	0	0	0	0

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)
            bot=max(0,i-half_size)
            top=min(i+half_size,n_row)
            left=max(0,j-half_size)
            right=min(n_col,j+half_size)

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

    return blurred_image
```

`means(axis=(0,1))` takes 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.

Original image

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 ($\sim$0.8 million 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 $674 \times 1200 \times 101 \times 101 = 25$ Billion operations!
- It takes $\sim$5 minutes to run

SkImage has many built-in blurring functions (called filters) with faster implementations:
The one equivalent to your purpose is:
https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.uniform_filter.html
More filters are here:
http://scikit-image.org/docs/dev/api/skimage.filters.html

It is much faster than the nested for loop version

This takes less than a second!

```
#from scipy import ndimage
#blurred_image = ndimage.uniform_filter(image,
↪  size=(101, 101, 1))
#plt.imshow(blurred_image)
#plt.show()
#io.imsave("car_blur101_uniform_filter.jpg",blurred_image)
```

A lots of numerical tricks went into the function (beyond the scope of this class)

Outline

Assignment 5

Image compression by matrix decomposition (very brief)

Brief survey on image denoising and inpainting

Image processing (caveat & review from last lecture)

Image blurring

Edge detection

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?

Vertical edge at row i:

- $image[i-1, j]$ is very different from $image[i+1, j]$

Horizontal edge at column j:

- $image[i, j-1]$ is very different from $image[i, j+1]$

Idea: To determine if an RGB pixel (i, j) belongs to an edge:
For each color $\in \{R, G, B\}$:

- $L_x[color] = image[i, j-1, color] - image[i, j+1, color]$
- $L_y[color] = image[i-1, j, color] - image[i+1, j, color]$
- edge_image[i,j,color] $= \sqrt{L_x[color]^2 + L_y[color]^2}$

Edge detection

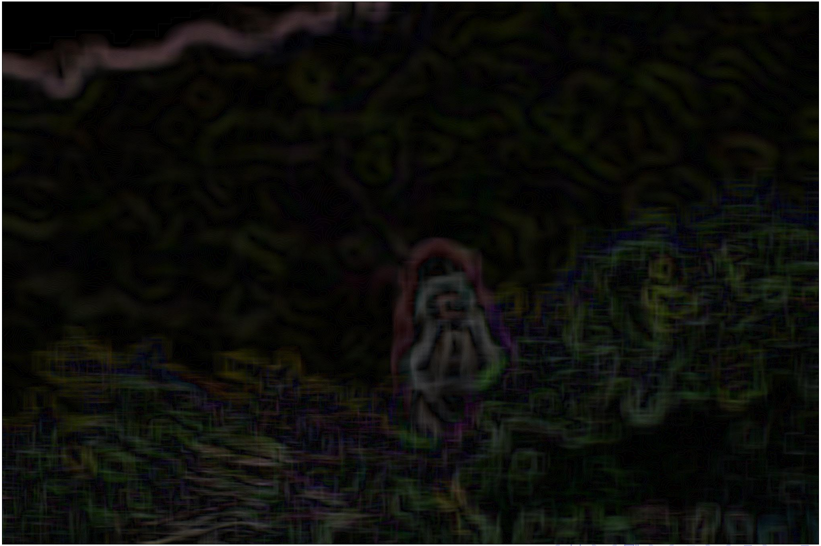
```
def detect_edges(image):
    n_row, n_col, colors = image.shape
    edge_image = np.zeros( (n_row,n_col,3), dtype=np.uint8)
    for i in range(1,n_row-1):
        for j in range(1,n_col-1):
            for c in range(3):

                # 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

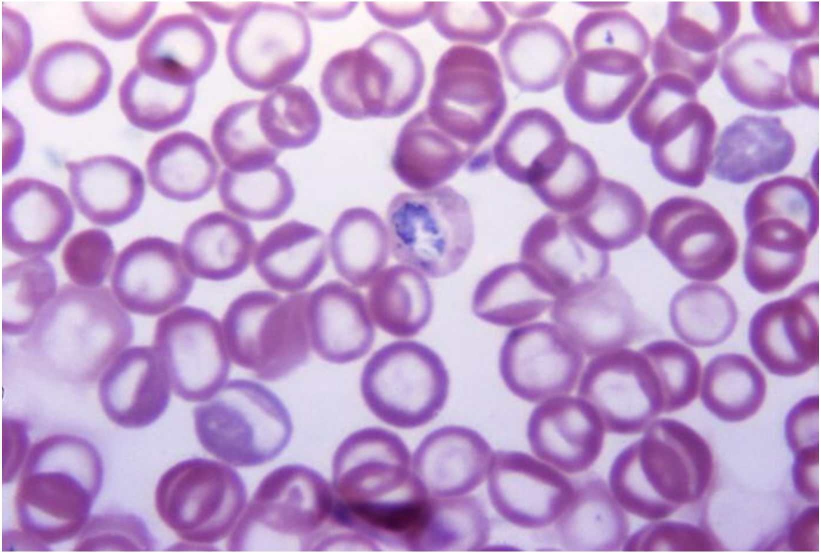
Not so great if our goal is to find the monkey in the image!

Blurring + Edge detection

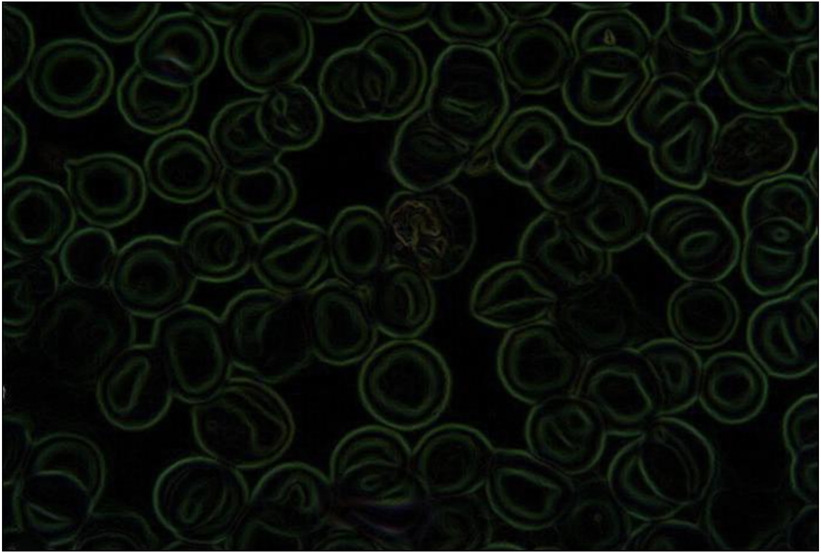
To smooth out fine details like leaves:
Start by blurring the image, then apply edge detection.



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