# COMP 204

## Introduction to image analysis with scikit-image (part three)

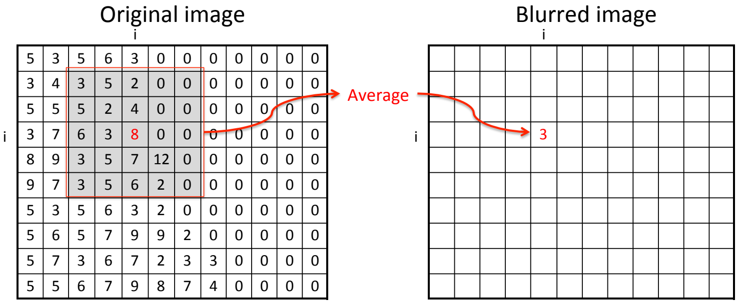
Yue Li
based on slides from Mathieu Blanchette, Christopher J.F. Cameron and Carlos G. Oliver

# Outline

# Blurring an image (recap)

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:



# Different window sizes led to different blurring effects

Original

Window size = 5

Window size = 21

Window size = 101

# Blurring an image (version with mixed up names)

index `i` indicates rows, therefore bottom (bot) and top

index `j` indicates columns, therefore left and right

**lines 13-20** mix up the names. We still get correct output, why?

```
6   def blur(image, filter_size):
7       n_row, n_col, colors = image.shape
8       blurred_image = np.zeros( (n_row, n_col, colors),
        ↪ dtype=np.uint8)
9       half_size=int(filter_size/2)
10      for i in range(n_row):
11          for j in range(n_col):
12              # define the boundaries of window around (i,j)
13              left=max(0,i-half_size)
14              right=min(i+half_size,n_row)
15              bot=max(0,j-half_size)
16              top=min(n_col,j+half_size)
17
18              # calculate average of RGB values in window
19              blurred_image[i,j] = \
20
                    ↪ image[left:right,bot:top,:].max(axis=(0,1))
21
22      return blurred_image
```

# Blurring an image (correct version)

index `i` indicates rows, therefore bottom (bot) and top

index `j` indicates columns, therefore left and right

pay attention to **lines 13-20**

```
6  def blur(image, filter_size):
7      n_row, n_col, colors = image.shape
8      blurred_image = np.zeros( (n_row, n_col, colors), dtype=np.uint8)
9      half_size=int(filter_size/2)
10     for i in range(n_row):
11         for j in range(n_col):
12             # define the boundaries of window around (i,j)
13             bot=max(0,i-half_size)
14             top=min(i+half_size,n_row)
15             left=max(0,j-half_size)
16             right=min(n_col,j+half_size)
17
18             # calculate average of RGB values in window
19             blurred_image[i,j] = \
20                 image[bot:top,left:right,:].max(axis=(0,1))
21
22     return blurred_image
```

# Outline

Correction from blur function in last lecture

Edge detection

Edge detection using thresholding approach

Counting cells by seed-filling algorithm

Final review

# 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)
- automatically counting cells from an imaging scan (discussed in Section 4)

# Edge detection

# Color edge detection

# Edge detection

What's an edge in an image? sharp transition between color tones

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] =** $\sqrt{L_x[color]^2 + L_y[color]^2}$

# Color edge detection

```python
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

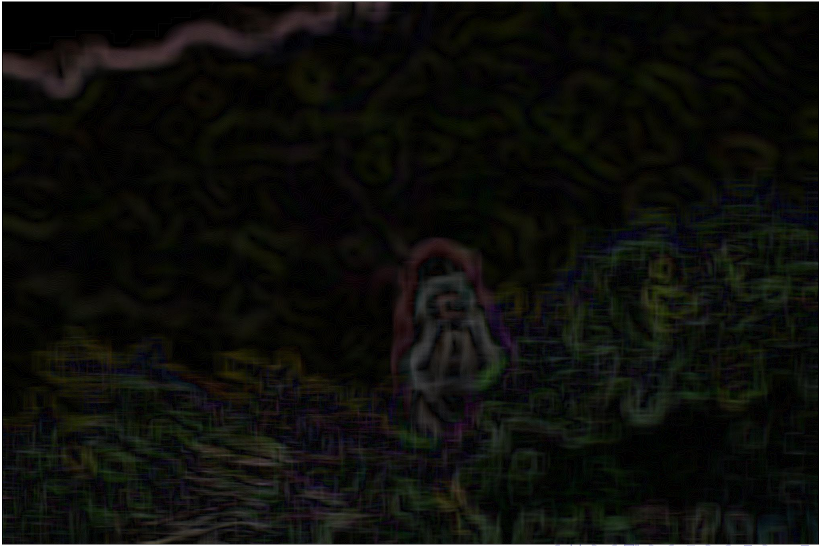
# Edge detection on monkey image

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

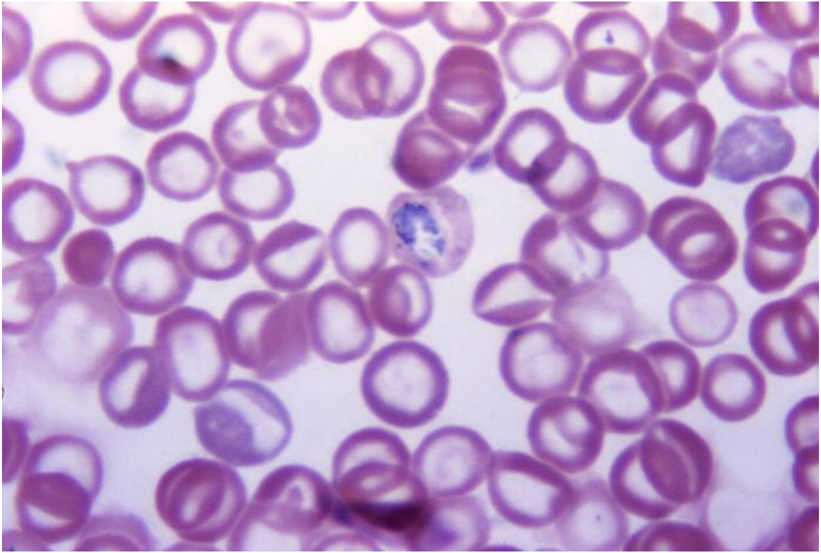
# Blurring (Lecture 33) + Edge detection

To smooth out fine details like leaves:

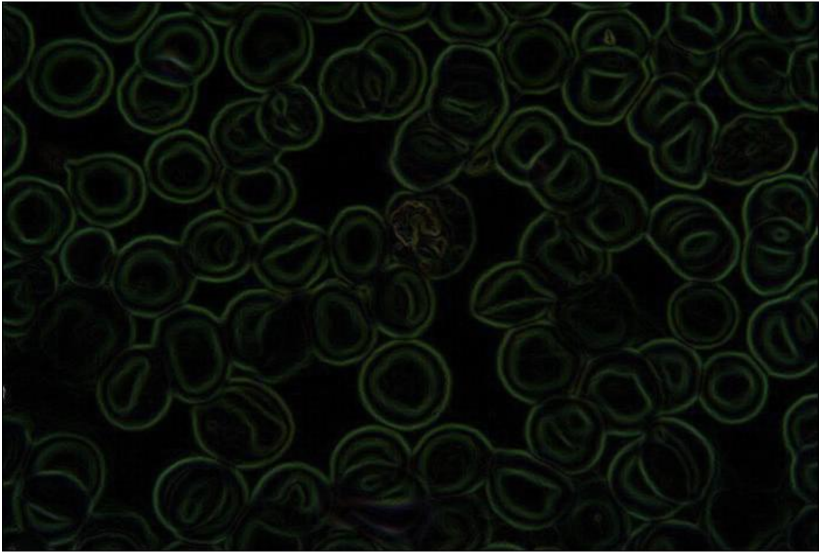
Start by blurring the image, then apply edge detection.



# Analysis of microscopy images



# Analysis of microscopy images



# Outline

Correction from blur function in last lecture

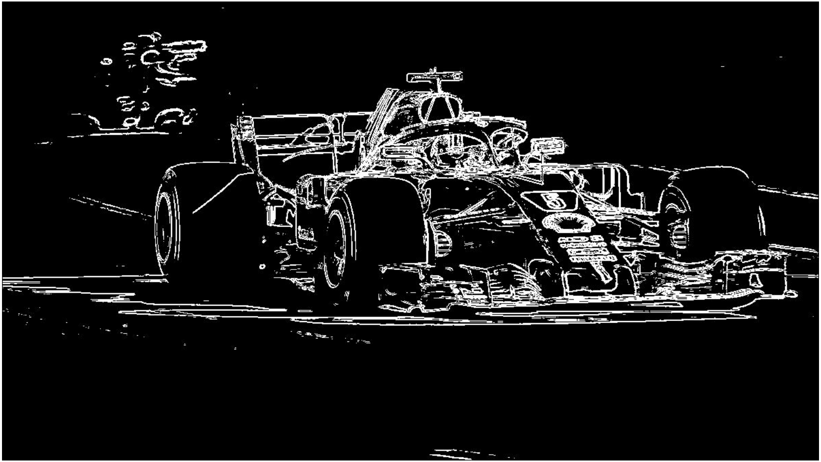
Edge detection

Edge detection using thresholding approach

Counting cells by seed-filling algorithm

Final review

# Edge detection using a thresholding approach



# Edge detection with thresholds

Same as the colour edge detections but the two last lines below.
Horizontal edge at row $i$:

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

Vertical 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 $\in \{R, G, B\}$:

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

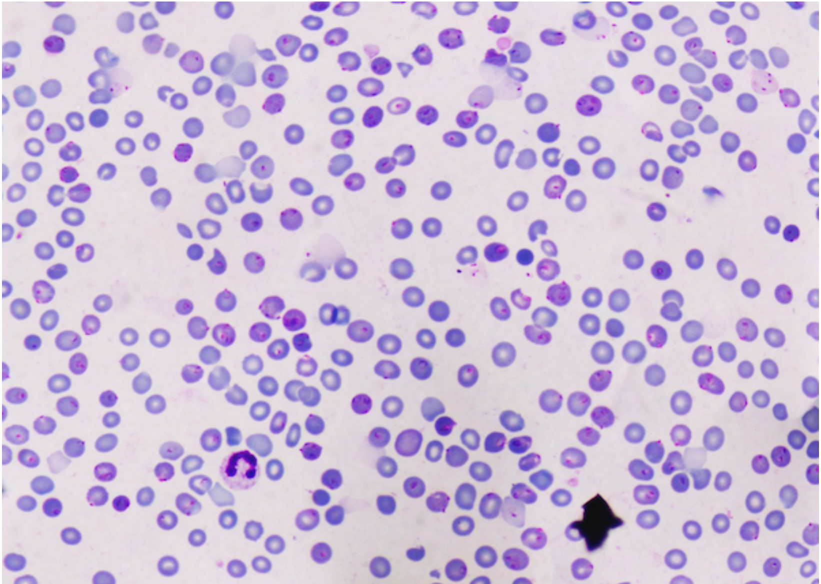
**edginess =** $\sqrt{gradient[R]^2 + gradient[G]^2 + gradient[B]^2}$
**if edginess $>$ some_threshold, then pixel $(i, j)$ belongs to an edge**

# Edge detection

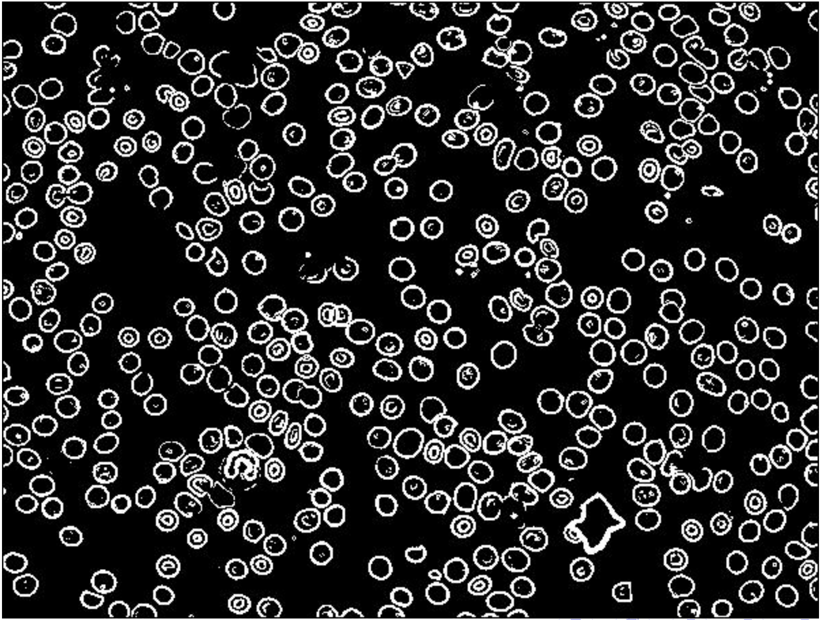
```python
def detect_edges(im, min_gradient=50):
    """
    Args:
    im: The image on which to detect edges
    min_gradient: The minimum gradient value
                  for a pixel to be called an edge
    Returns: An new image with edge pixels set to white,
                     and everything else set to black
    """
    n_row, n_col, colors = image.shape

    # create a empty empty of the same size as the original
    edge_image = np.zeros( (n_row,n_col,3), dtype=np.uint8)

    for i in range(1,n_row-1):  # avoid the first/last row
        for j in range(1,n_col-1):  # and first/last col
            grad=[0,0,0]
            for c in range(3):  # for each color
                Lx = float(im[i-1,j,c])-float(im[i+1,j,c])
                Ly = float(im[i,j-1,c])-float(im[i,j+1,c])
                grad[c] = math.sqrt(Lx**2+Ly**2)
            norm = math.sqrt(grad[0]**2 + grad[1]**2 \
                             + grad[2]**2)
            if (norm > min_gradient):
                edge_image[i,j] = (255,255,255)
    return edge_image
```

# Analysis of microscopy images

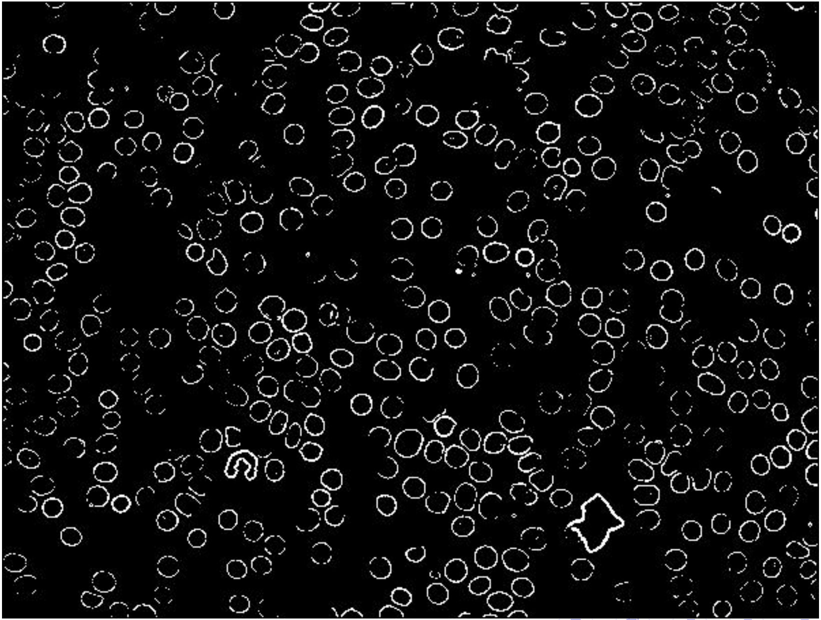
Cells (purple "circles") are infected by Plasmodium falciparum (small red dots).



# Edge detection (threshold = 60)



# Edge detection (threshold = 120)



# Edge detection

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

# Outline

Correction from blur function in last lecture

Edge detection

Edge detection using thresholding approach

Counting cells by seed-filling algorithm

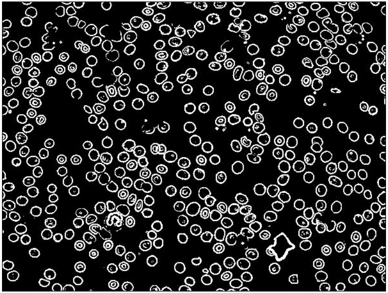
Final review

# Counting/annotating cells

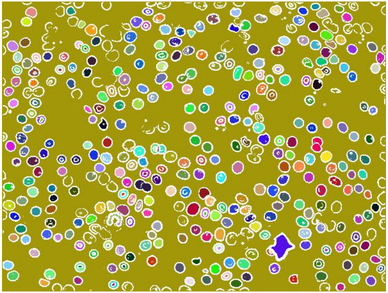
What if we want to automatically identify/count cells in the image?

Idea:

1. Find edges in the image
2. Identify closed (encircled) shapes within the edge image



to



Each closed shape is assigned a different color.
Number of closed shapes ( $=$ approximation to cell count) is calculated.

# Seed filling algorithm

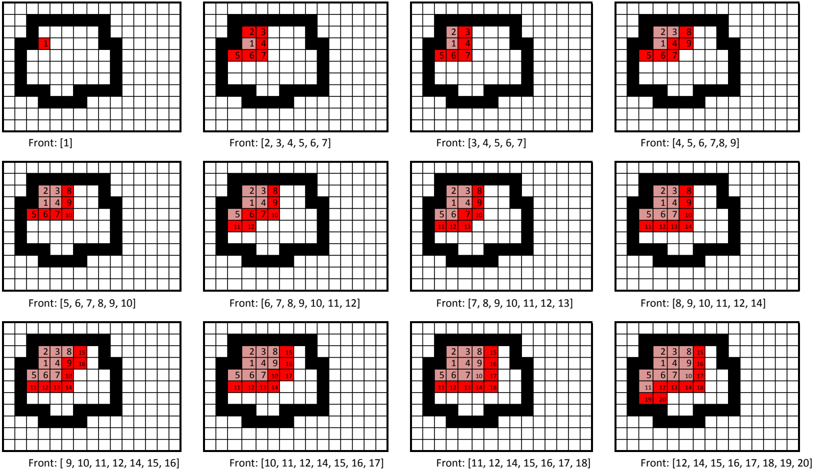
How to take an edge image and fill-in each closed shape?
Seed filling (aka flood filling) algorithm:

- Start from a black pixel.
- Color it and expand to its neighboring pixel, unless neighbor is an edge (white).
- Keep expanding until no more expansion is possible
- Repeat from a new starting point, until no black pixels are left

# Seed filling algorithm

For illustration purpose, we swapped the background and edge colors: Black = edge, white = background

Seed = pixel at position (4,4)




In our real image, Black = background, white = edge

# Seed filling implementation

```python
def seedfill(im, seed_row, seed_col, fill_color,bckg):
    """
    im: The image on which to perform the seedfill algorithm
    seed_row and seed_col: position of the seed pixel
    fill_color: Color for the fill
    bckg: Color of the background, to be filled
    Returns: Nothing
    Behavior: Modifies image by performing seedfill
    """
    size=0  # keep track of patch size
    n_row, n_col, foo = im.shape
    front=[(seed_row,seed_col)]  # initial front

    while len(front)>0:
        r, c = front.pop(0)  # remove 1st element of front
        if np.array_equal(im[r, c,:],bckg):
            im[r, c]=fill_color  # color pixel
            size+=1
            # look at all neighbors
            for i in range(max(0,r-1), min(n_row,r+2)):
                for j in range(max(0,c-1),min(n_col,c+2)):
                    # if background, add to front
                    if np.array_equal(im[i,j,:], bckg) and\
                       (i,j) not in front:
                        front.append((i,j))
    return size
```

# Seeding from all possible starting pixel...

```python
min_cell_size=100  # based on prior knowledge
max_cell_size=300  # based on prior knowledge
n_cells=0
# look for a black pixel to seed the filling
for i in range(image.shape[0]):
    for j in range(image.shape[1]):
        if np.array_equal(edge[i,j,:],(0,0,0)):
            rand_color = (random.randrange(255),
                          random.randrange(255),
                          random.randrange(255))
            size=seedfill_with_animation(edge, i ,j,
                                         rand_color, (0,0,0) )
            if size>= min_cell_size and size<max_cell_size:
                n_cells+=1
print("Number of cells:",n_cells)
```

# Seed filling execution

See live execution of program

# Outline

Correction from blur function in last lecture

Edge detection

Edge detection using thresholding approach

Counting cells by seed-filling algorithm

Final review

# Final exam

**Date and Time**: Tuesday, April 30, 2019 at 6:30:00 PM
**Room**: TBD
In the next 3 lectures, we will review the course materials tested in the final exam

# Materials tested in the final exam

Main materials that are covered in the final exam include:

- Basics: functions, loops, variables, data types (string, list, tuple, dictionary, sets), difference between pass by copy and pass by memory addresses
- Algorithms: Searching (linear and binary search) and sorting (insertion and selection sort)
- Pattern searching by string indexing and regular expression (simple ones)
- Object oriented programming: class, attributes, class inheritance, class methods
- BioPython sequence handling covered in class (I will remind you what the methods are in the exam)
- Machine learning: know what supervised, unsupervised, reinforcement learning are, problems they can solve, TPR, FPR, overfitting, cross-validation, ROC, decision trees
- Image processing: basic understanding of going from a pixel in the image to numpy ndarray