COMP 204 - Assignment 5

Yue Li

Important instructions:

- Release date: March 29, 2022 at 12:00 AM
- Due date: **April 12, 2022 at 23:59**
- Download the following Python files from MyCourses:
    1. cell_counting.py
    2. malaria-1.jpg
    3. malaria-1-small.jpg
- Complete the 6 programming questions
- Submit *ONLY* the completed `cell_counting.py` file to CodePost:
- Write your name and student ID at the top of each file
- Do not use any module or any piece of code you did not write yourself
- For each question, your program will be *automatically* tested on a variety of test cases, which will account for 75% of the mark. To be considered correct, your program should print out *exactly and only* what the question specifies. Do not add extra text, extra spaces or extra punctuation.
- For each question, 25% of the mark will be assigned by the TA based on (i) your appropriate naming of variables; (ii) commenting of your program; (iii) simplicity of your program (simpler = better).

[IMPORTANT] Review code of conduct

Review **Avoid plagiarism and cheating** policy under Content on MyCourses and make sure that you strictly follow the checklist in completing this assignment.

In this assignment, you will continue the work started in class to develop a program that accurately identifies red blood cells in a microscopy image, and then counts the fraction of those cells that are infected by Plasmodium Falciparum (the causative agent of malaria). We will analyze the image below (malaria-1.jpg). Each red blood cell is a pink, roughly circular region, often with a lighter center. Some cells are infected with parasite Plasmodium, which shows as a dark-purple ring due to Giemsa staining. The image also contains some additional stuff, like this very dark blob in the center - ignore them.

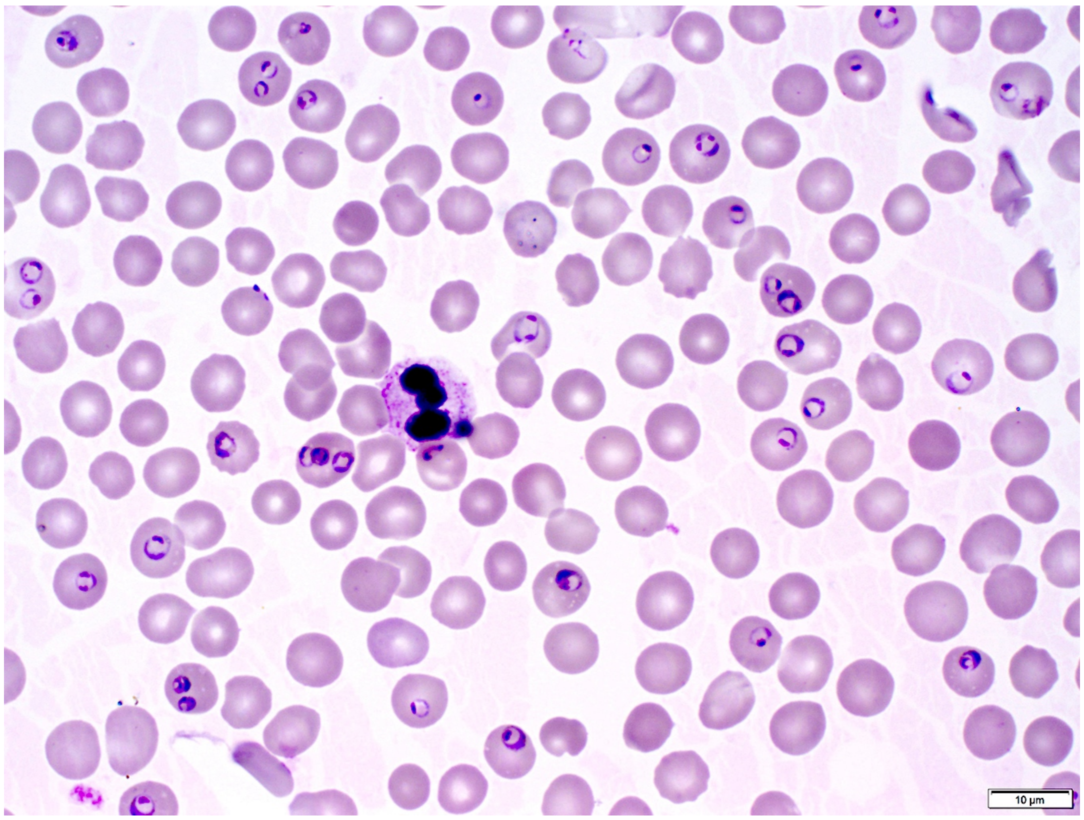

Figure 1: A microscopy image of cells some of which were infected by malaria. Source:http://www.pathologyoutlines.com/topic/parasitologymalariapfalciparum.html

1 Better edge detection [Expected length: 4 lines of code] (10 points)

The edge detection algorithm seen in class is not the most accurate. A more commonly used edge detector is the Sobel algorithm, which is implemented in function `sobel` as a part of the skimage.filters module. See https://scikit-image.org/docs/dev/api/skimage.filters.html

Apply the Sobel algorithm to image malaria-1.jpg. Complete the function `detect_edges` in `cell_counting.py`. Save the resulting image in a file called “Q1_gray_sobel.jpg".

Notes:

- The Sobel algorithm can only be applied to graytone images, so you will first need to convert the original color image to a graytone image as described in class, using the `rgb2gray` function.
- Reminder: Grayscale images have pixel values between 0 and 1, not between 0 and 255.
- The image you produce should look like Figure 2:

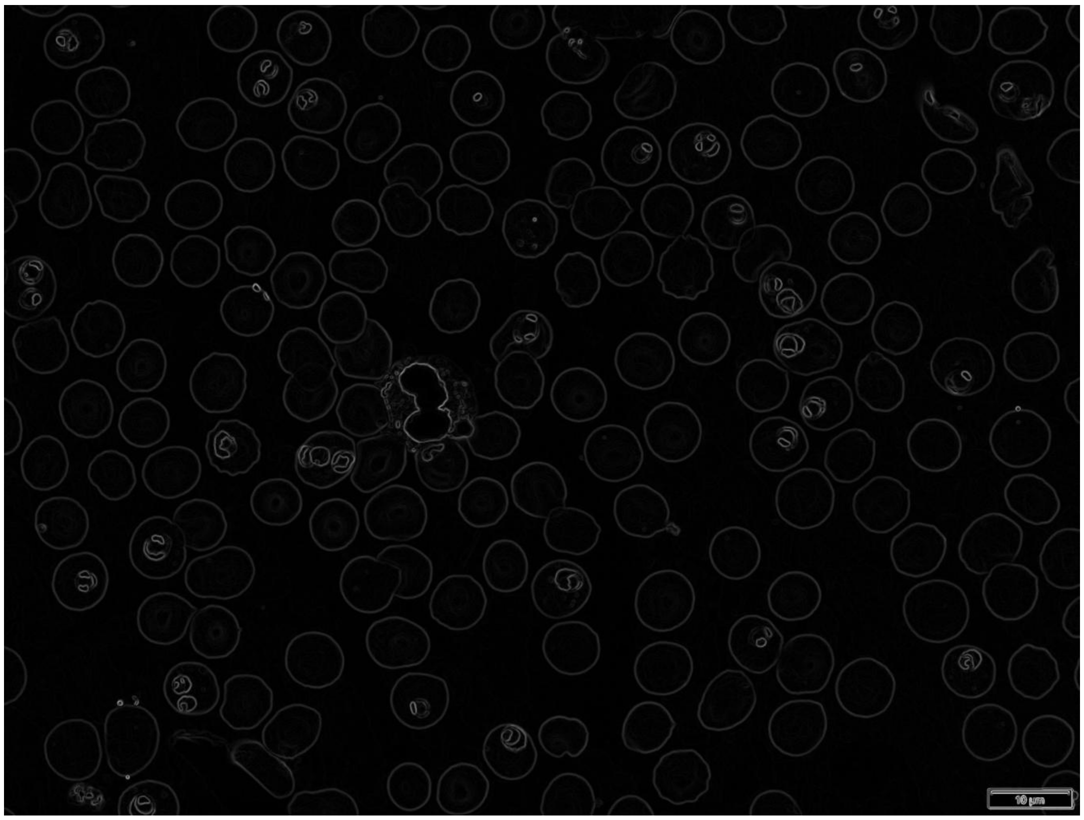

Figure 2: Cell image after the grayscale conversion and the initial edge detection.

2 Binarizing edges [Expected length: 1-5 lines] (10 points)

The Sobel algorithm measures the “edginess" of a pixel with a number between 0 and 1. In order to turn these real-valued images into black-and- white images (white = edges, black = non-edge), we need to threshold the image, i.e. to produce a new image where the pixel value at location (r,c) is set to 0 if `edginess(r,c)<T`, and to 1 if `edginess(r,c)>=T`, where T is a user-defined threshold. Complete function `binarize_edges` with T=0.05. Save the images in files called “Q2_gray_sobel_T005.jpg. Your images should look like this:

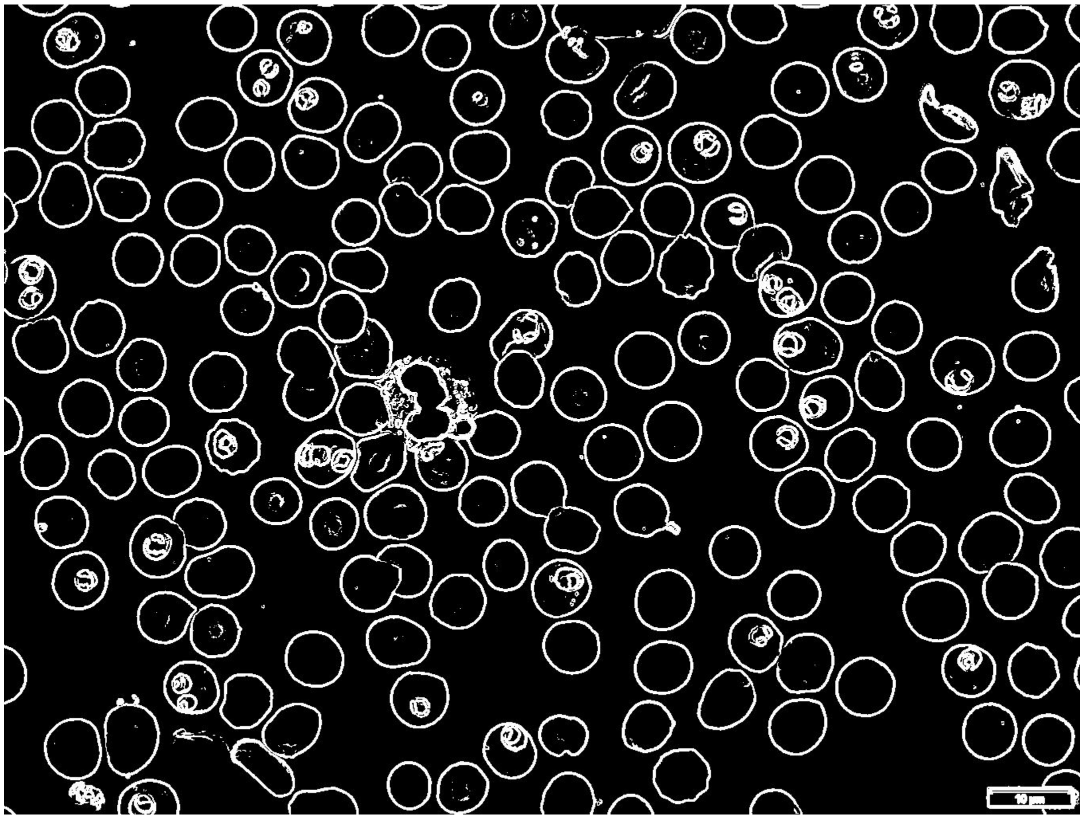

Figure 3: Cell image after the `binarize_edges` detection.

3 Cleaning up. Expected length: [8 lines of code] (10 Points)

The thresholded images obtained in question 2 include edges corresponding to cell perimeters, as well as edges corresponding to the periphery of the darker plasmodium cells, inserted within red blood cells. In order to best delineate each red blood cell, we need to first remove the edge pixels caused by the Plasmodium cells. Pixels corresponding to Plasmodium cells tend to have graytone values less than 0.5, or to be adjacent to a pixel with graytone value less than 0.5.

Write the function `cleanup`. Starting from the image output by `binarize_image` in Q2. Modify `image_sobel2` from `binarize_image` from Q2 so that any white pixel at position (r,c) is replaced by a black pixel if the value of pixel (r,c) or of any of the 8 surrounding pixels **in the original graytone** malaria-1 image is below 0.5.

Save the modified edge image as “Q3_gray_sobel_T005_cleanup.jpg" This should give you the image below. The edge pixels caused by Plasmodium are not completely gone, but this will be sufficient anyway.

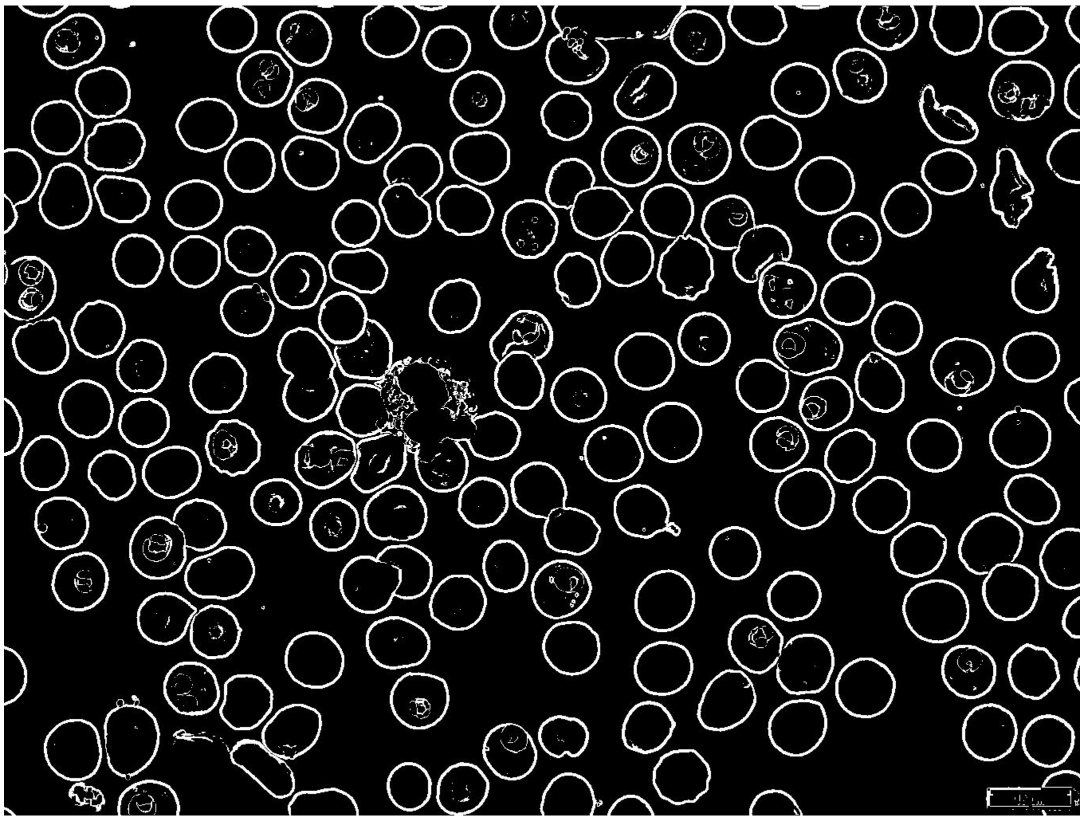
Figure 4: Cell image after the `cleanup` function.

4 Filling cells. [Expected length: 12-25 lines] (20 Points)

Write the `fill_cells` function. The function should take as argument an edge image (e.g. the image obtained in Q3), with black background and white edges. The function creates and returns a new image that is a copy of the edge image provided as argument, but with each closed region filled with a different grayscale value (see result below). This is achieved by repeatedly calling the seedfill function (provided to you), using as seed various pixels determined by your code, as described below.

Start by making a copy of the edge image, which is the one you will then modify and ultimately return. Then call the seedfill from pixel (0,0), with `fill_color=0.1`. This will mark the background of the image with a dark gray. Then, as seen in class, look for pixels that remain black. Whenever you find one, initiate a seedfill from that location, this time choosing `fill_color` as `0.5+0.001*n_regions_found_so_far` (where `n_regions_found_so_far` is the number of closed regions identified to date by your search). So the first closed region will be colored 0.5, the second 0.501, the third 0.502, etc. Once your function is done looking at the entire image, return the resulting image.

From your main program, call your `fill_cells()` function on the image obtained in Q3, and save the resulting image as “Q4_gray_sobel_T005_cleanup_filled.jpg". You should get the image shown below, which identifies **203 enclosed areas**.

Notes:

- On my computer, my code takes about 1 minute to run; it’s normal that it is a bit slow, because the seedfill function is not fast and the image is large.
- The Sobel edge detection algorithm never calls edges in the first and last rows and columns of the image. This is why cells that are at the periphery of the image are not identified. Do not worry about this.

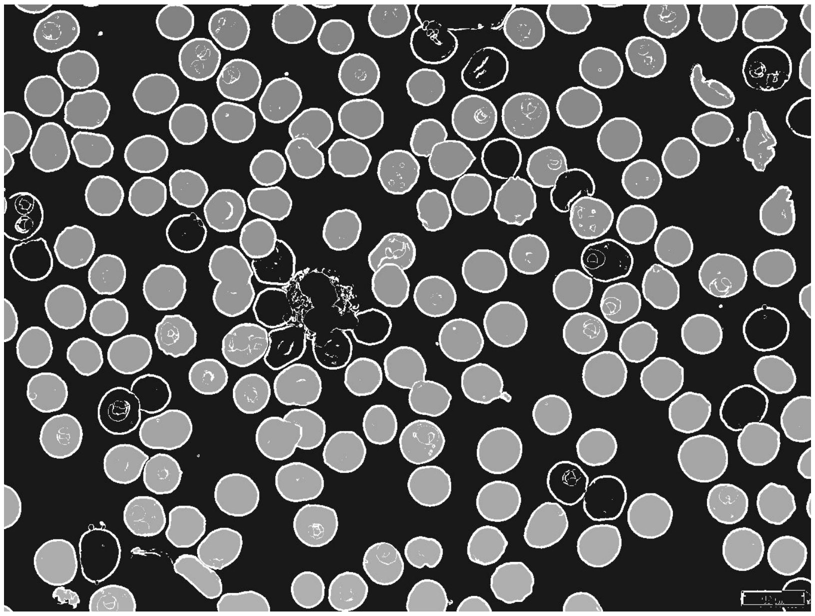

Figure 5: Cell image after the `fill_cells` function.

5 Classifying cells. [25 lines of code] (30 Points)

Each detected closed region is now labeled with a different graytone: 0.5, 0.501, 0.502, ... For each region detected, we now need to determine if (i) it is a red blood cell, and (ii) if the cell is infected or not. We will say that a region with grayscale value g is a valid cell if its size is between 1000 and 5000 pixels (this eliminates tiny closed regions that are not actual cells). We will say that a cell is infected if at least 2% of the pixels it contains (those labeled with grayscale value g) have pixel grayscale value below 0.5 in the original grayscale image.

Write the function `classify_cells`, which takes as argument:

- The original graytone image
- The filled image obtained in Question 4
- Optional keyword arguments: `min_size=1000`, `max_size=5000`,

  `infected_grayscale=0.5`, `min_infected_percentage=0.02`

The function should return a tuple of two Sets. The first set should contain the labels of cells that are infected, while the second set should contain the labels of cells that not infected (see below for the expected output).

There are many ways to achieve this - feel free to use the approach you feel is the best. One approach could go as follows:

1. Build a Set of all grayscale values observed in the labeled image
2. Initialize `infected=set()`, `not_infected=set()`
3. For each grayscale value in the grayscales set:
   (a) Scan the image to identify pixels with that grayscale value in the labeled image, and count separately those that are dark (<=infected_grayscale) and light (>infected_grayscale) in the original image.
   (b) Using the counts of dark and light pixels, determine if the region with that grayscale value is a cell or not, and whether or not it is infected. Add it to the `infected` or `not_infected` sets, as appropriate.
4. Return the pair of `infected` and `not_infected` sets.

Notes: Some cells actually contain pixels that are labeled white, because of left-overs from the Plasmodium edges. To make our life easier, we will not consider those pixels as being part of the cell. With my code, the function returns **29 infected cells** and **107 non-infected cells**:

```
{0.506, 0.6799999999999999, 0.589, 0.54, 0.584, 0.646, 0.669, 0.663, 0.513, 0.545, 0.574, 0.539, 0.616, 0.645, 0.532, 0.53, 0.559, 0.62,
↪  0.569, 0.524, 0.614, 0.531, 0.56, 0.6819999999999999, 0.512, 0.525, 0.602, 0.666, 0.692}
{0.504, 0.51, 0.542, 0.5609999999999999, 0.552, 0.677, 0.593, 0.571, 0.635, 0.696, 0.5640000000000001, 0.546, 0.636, 0.511, 0.665, 0.63,
↪  0.534, 0.659, 0.624, 0.618, 0.647, 0.551, 0.641, 0.565, 0.5680000000000001, 0.629, 0.69, 0.533, 0.658, 0.562, 0.623, 0.527, 0.588,
↪  0.652, 0.556, 0.617, 0.6779999999999999, 0.582, 0.55, 0.675, 0.576, 0.64, 0.544, 0.509, 0.538, 0.599, 0.503, 0.628, 0.59, 0.654,
↪  0.648, 0.642, 0.67, 0.529, 0.622, 0.664, 0.507, 0.555, 0.657, 0.587, 0.581, 0.535, 0.549, 0.674, 0.575, 0.543, 0.604, 0.668,
↪  0.5720000000000001, 0.633, 0.6970000000000001, 0.537, 0.662, 0.563, 0.5660000000000001, 0.627, 0.6910000000000001, 0.6890000000000001,
↪  0.592, 0.685, 0.65, 0.554, 0.615, 0.6759999999999999, 0.679, 0.644, 0.548, 0.609, 0.673, 0.638, 0.632, 0.536, 0.661, 0.501, 0.591,
↪  0.585, 0.649, 0.553, 0.518, 0.579, 0.596, 0.547, 0.608, 0.541, 0.567, 0.631, 0.66}
```

Note your code may labeled your cells in a different order from mine, but it should produce the same number of `infected` and `not_infected` cells.

6 Displaying infected cells [12 lines] (20 Points)

As a last step, we will produce a new color image that is a copy of the original color image, but that highlights in red cells that have been classified as infected, and in green cells that have been labeled as not infected (see result below). This would allow the scientist using our program to verify the calls that are made.

Write the `annotate_image` function, which takes as arguments:

- The original color image
- The labeled grayscale image obtained in question 4
- The Set of grayscale values corresponding to infected cells (obtained from question 5)
- The Set of grayscale values corresponding to not infected cells (obtained from question 5)

The function should return a new color image, corresponding to a copy of the original color image, modified as follows.

The annotated image should have the same pixel values as the original image, except when both (i) the grayscale of pixel (r,c) in the labeled image corresponds to an infected cell or a non infected cell, and (ii) at least one of the eight surrounding pixels in the labeled image is white (grayscale=1). The color at those pixels should be set to red (for infected cells) or green (for non-infected cells). Save the resulting image as “Q6_annotated.jpg". It should look like the image below.

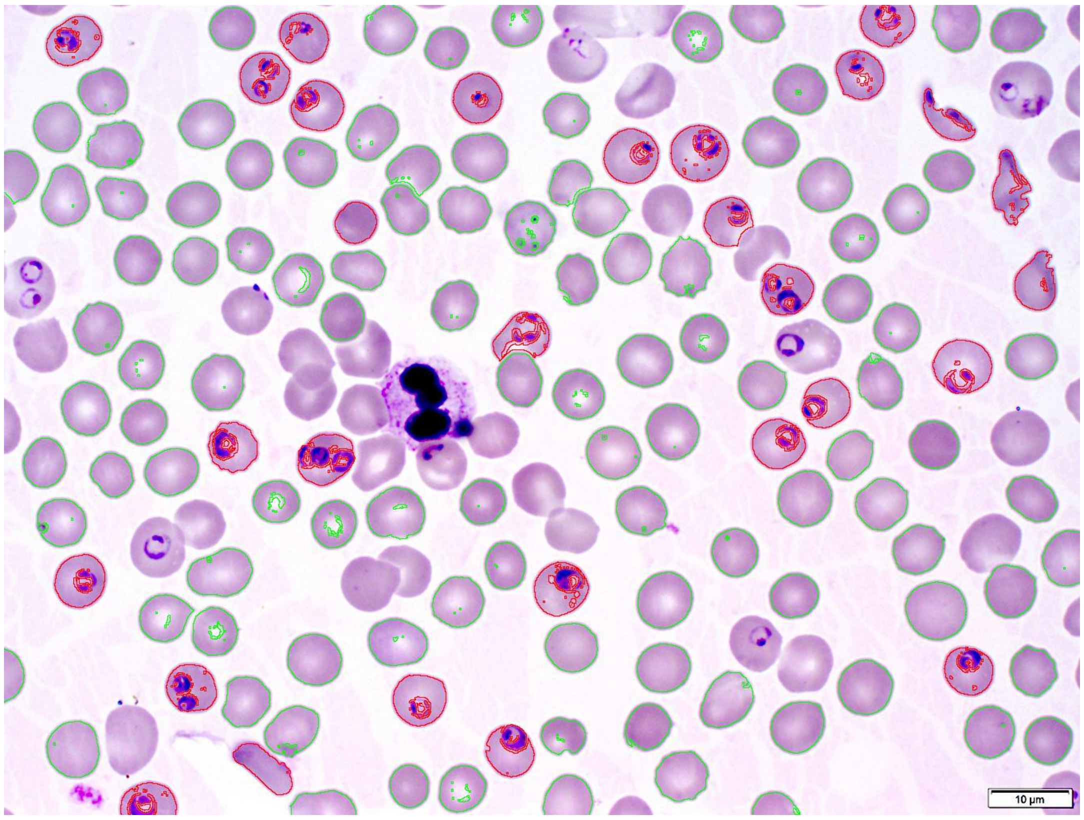

Figure 6: The final cell image produced from the `annotate_image` function.

Tips

- Only use the `imread` function at the very beginning of your program, to read malaria-1.jpg. To proceed from one question to the next, simply continue using the same image (or a copy of it), without re-reading it from the file you've just written.
- Tip2: It is normal that your program takes about 2-3 minutes to execute. During the development of your program, this relatively long running time may make debugging painful. For this reason, temporarily switch to using the smaller image called **malaria-1-small.jpg**. My entire program runs on it in about 5 seconds. The expected outputs when using malaria-1-small.jpg are shown below.
- If you use `np.where` to compute certain things (which would be a good idea), make sure that the values you are using as argument to the function are floating point numbers (e.g. 0.0 and 1.0) rather than integers (e.g. 0 and 1). If you don't, the array you'll get will be an array of integers, which will prevent you from storing values like 0.1 in the image.
- It is normal that you get warnings like the following, when saving figures: "Lossy conversion from float64 to uint8. Range [0, 1]. Convert image to uint8 prior to saving to suppress this warning."
- You'll be dealing with a lot of very similar images/matrices/arrays, so reading the questions very carefully can save you a lot of debugging and keyboard-mashing.

Expected outputs for Q1-6 when using malaria-1-small.jpg

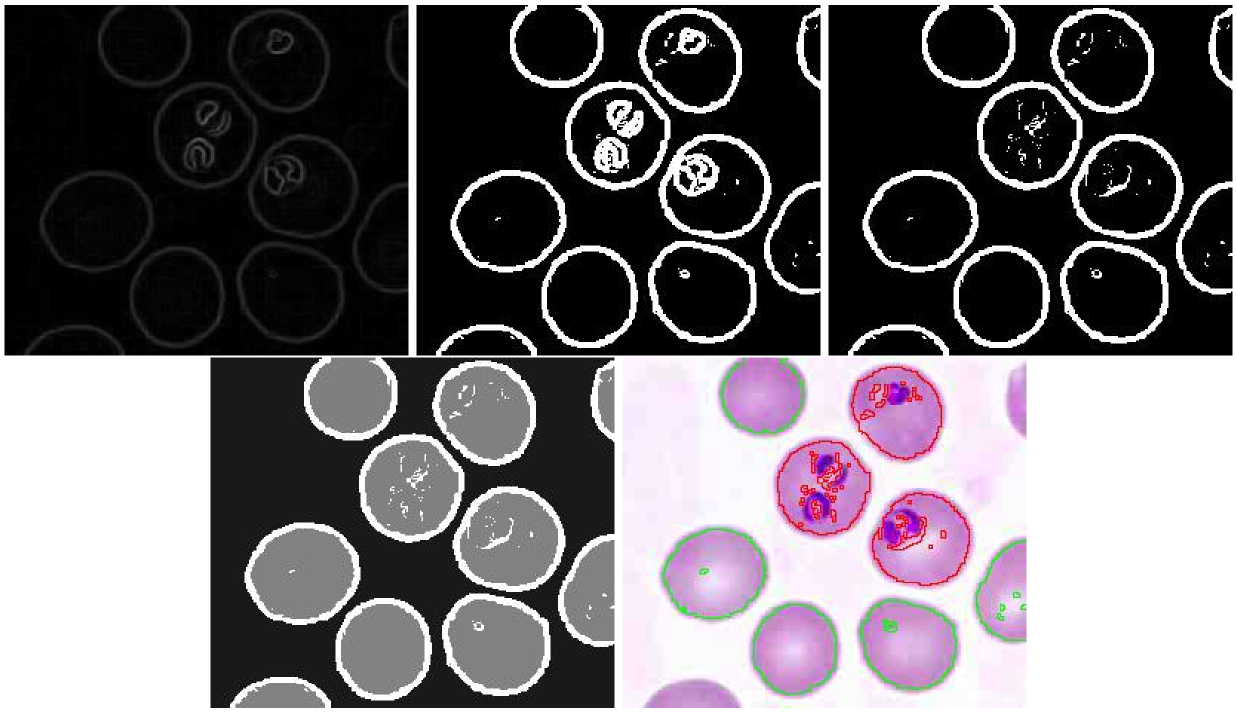

Figure 7: First row, left to right: Q1, Q2, Q3; second row, left to right: Q4, Q6.

For Q5, the function `classify_cells` returns **3 infected cells** and **5 non-infected cells**: `{0.501, 0.503, 0.504}`, `{0.5, 0.505, 0.507, 0.506, 0.508}`