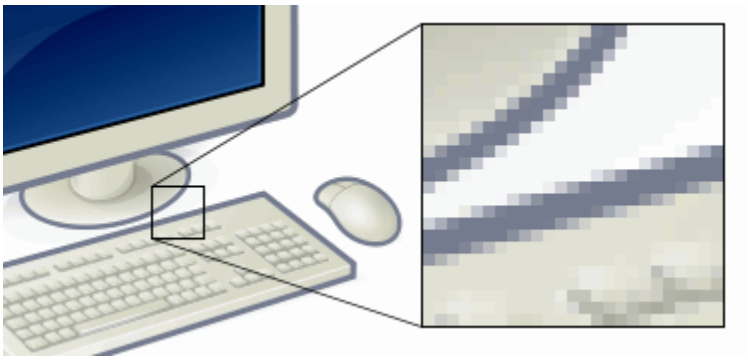
Project 1: Image Processing

Introduction

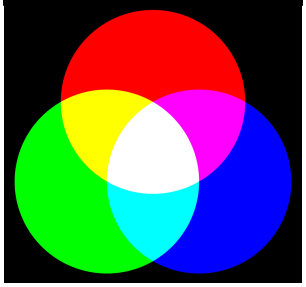
The goal of this assignment is to introduce you to the basic features of JavaScript, such as functions, variables, conditionals, and loops. You should already be familiar with these concepts from other programming languages (e.g., Java). You will use these features to write several *image processing* functions (e.g., blur, sharpen, and adjust brightness). In addition to the programming component, you will also complete and submit a written quiz portion separately on Gradescope.

Colors, Pixels, and Images

Any image you see on your computer screen consists of tiny dots known as *pixels*. On many screens, individual pixels are too small to see with the naked eye, but if you look very closely, you may be able to discern the pattern in which pixels are arranged, as illustrated in the figure to the right.



(0,0)	(1,0)	(2,0)
(0,1)	(1,1)	(2,2)
(0,2)	(1,2)	(2,2)



Pixels are arranged in a grid and each pixel has *x- and y- coordinates* that identify its position in the grid. All coordinates are non-negative integers and the top-left corner has the coordinates (0, 0). Therefore, the x-coordinate increases as you move right and the y-coordinate increases as you move down.[1] For example, if we had an image with just nine pixels arranged in a 3-by-3 grid, their coordinates would appear as shown on the left.

Finally, every pixel has a color that is specified by adding together the three *primary colors* (i.e., red, green, and blue). Therefore, to set the color of a pixel, you have to specify how much red, green, and blue to use. Each of these primary colors has an intensity between 0.0 and 1.0. For example, to get a black pixel, we can set the intensity of the three primary colors to 0.0 and to get a white pixel we can set the intensity of the three primary colors to 1.0.

In this project, you will learn to use a very simple library of image manipulation functions that let you load images from the web, set the color of each pixel and read the color of each pixel. These are the only primitive functions that you need to build sophisticated image processing functions yourself.

[1] Note that this is not the same as coordinates in a graph from math classes, where the y-coordinate *decreases* as we move down. In contrast, on the screen, the y-coordinate *increases* as we move down.

Image Manipulation

Our JavaScript programming environment includes a simple image manipulation library that has just a handful of functions. You can load an image from a URL using the `lib220.loadImageFromURL` function. For example, the following statement loads an image:

```
let robot = lib220.loadImageFromURL(
    'https://people.cs.umass.edu/~joydeepb/robot.jpg');
```

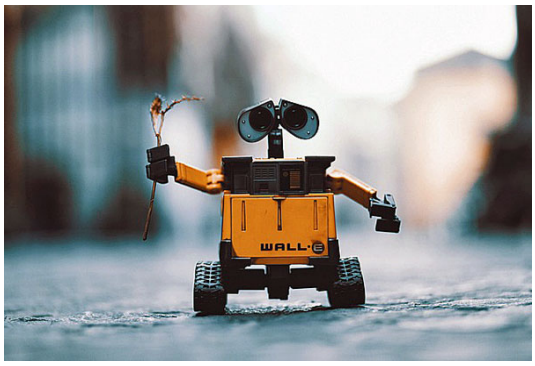
Note that loading an image does not show the image. If you enter robot in the REPL, you'll see that the image is an object with several fields and methods:

```
>> robot
<< {width: 600, height: 400, show: function show(), setPixel: function setPixel(),
 getPixel: function getPixel()}
```

We can use the method `robot.show()` to actually see the image on screen:

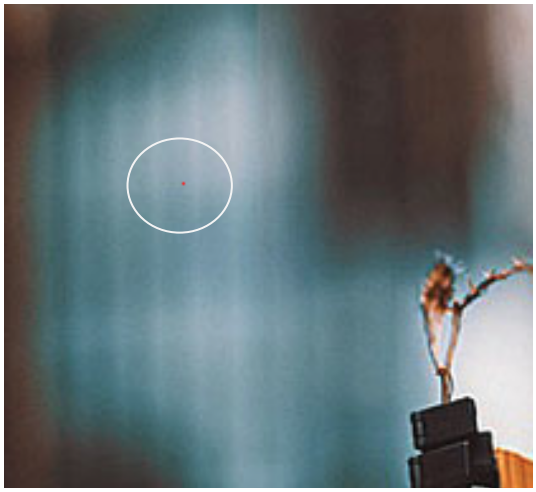
```
robot.show()
```

Images have only two other method that allow you to get and set the color of individual pixels. The method `.getPixel(x, y)` takes the *x*- and *y*- coordinates of a pixel as an argument and returns an array of three elements, which contain the intensity of red, green, and blue in that pixel in order. For example, in this image, the pixel at coordinate (72, 72) has the following color:




```
>> robot.getPixel(72, 72)
<< [0.5529411764705883,
    0.6431372549019608,
    0.6980392156862745]
```

We can set the color of the pixel with the `.setPixel(x, y, color)` method:



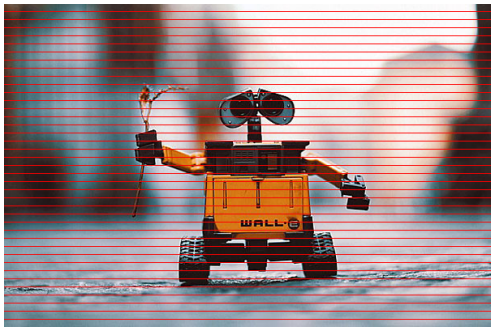
```
>> robot.setPixel(72, 72, [1.0, 0.0, 0.0])
```

However, note that this does not update the visible image. To see a image after a change, invoke the `.show()` method again. This will show the image again and a small red dot will be visible next to the robot's arm. (You may need to zoom in considerably.)

By repeatedly using `.setPixel` and .`getPixel`, we can manipulate the image in a variety of ways. For example, the following program draws a series of red, horizontal lines that are ten pixels apart:

```
let robot =
lib220.loadImageFromURL('https://people.cs.umass.edu/~joydeepb/robot.jpg');
for (let i = 0; i < robot.width; ++i) {
  for (let j = 0; j < robot.height; j = j + 10) {
```

```
    robot.setPixel(i, j, [1.0, 0.0, 0.0]);
  }
}
robot.show();
```

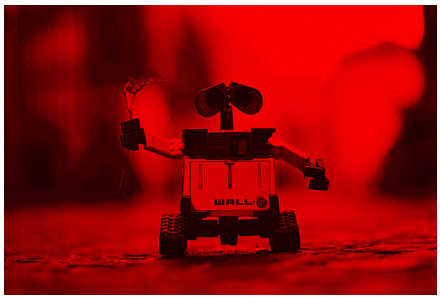


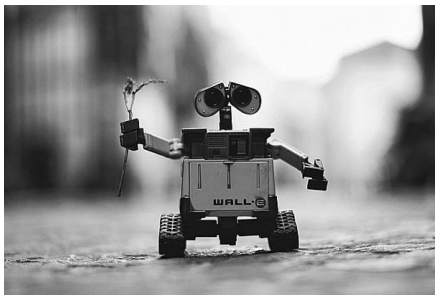
Finally, images have a `.copy()` method that creates a new copy of the image. Without creating a copy of an image, there is no way to write a test case to verify that the output image is different from the input image or to apply several effects to the same image without loading it repeatedly.

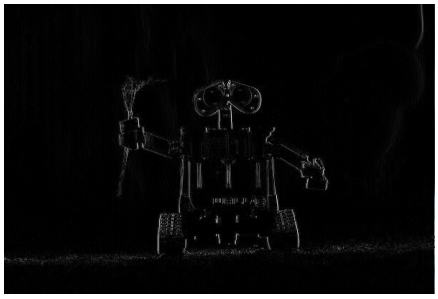
Programming Task

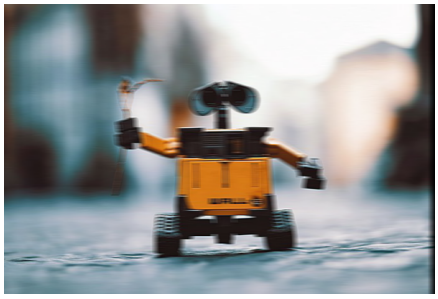
In this assignment, you will write functions for the following four effects applied to an image:

1. Make the image red by removing blue and green,
2. Make the image grayscale by setting all colors to the same intensity,
3. Highlight the edges in the image by comparing the intensity of adjacent pixels, and
4. Blur the image by averaging the colors of adjacent pixels.









You can see examples of these effects in the grid of examples above. Specifically, write the following functions:

1. Write a function called `removeBlueAndGreen` that takes an image as an argument and returns a red version of the input image. To do so, create a copy of the input and iterate over each pixel. If the color of a pixel is (r, g, b) in the input image, its color in the output must be $(r, 0.0, 0.0)$.
2. Write a function called `makeGrayscale` that takes an image as an argument and returns a grayscale version of the input image. To do so, create a copy of the input and iterate over each pixel. If the color of a pixel is (r, g, b) in the input image, its color in the output must be (m, m, m) where m is the mean (average) of r, g, and b.

If you have solved these two tasks, you might notice that the structure of the two functions is very similar, the difference is only in the actual processing applied. We can avoid duplication by defining a function, similar to `map`, that applies the same transformation to all pixels of an image.

3. Write a function called `imageMap` with the following type:
   ```
   imageMap(img: Image, func: (p: Pixel) => Pixel): Image
   ```
   The result must be a new image with the same dimensions as `img`. The value of each pixel in the new image should be the result of applying `func` to each pixel of `img`. You should use either `lib220.createImage` or `image.copy` to create a new image, or make a copy of the original. Write two functions `mapToRed` and `mapToGrayscale` that are equivalent to `removeBlueAndGreen` and `makeGrayscale` but use `imageMap`.

4. Write a function called `highlightEdges` that takes an image as an argument and returns a black-and-white version of the input image where edges are highlighted. To do so, iterate over the (x, y)- coordinates of the image and calculate the mean values of pixel (x, y) (call it $m1$) and the pixel $(x + 1, y)$ (call it $m2$). Set the color of the pixel (x, y) in the output image to $(|m1 - m2|, |m1 - m2|, |m1 - m2|)$.

5. Write a function called `blur` that takes an image as an argument and returns a blurred copy of the argument image. To do so, iterate over the (x, y)- coordinates and then independently consider the red, green, and blue components of each pixel's color. The red component of the pixel at (x, y) should be the mean of the red components of the eight pixels neighboring (x, y), and the (x, y) pixel itself (i.e., the mean of 9 values). Update the blue and green components in the same manner. Note that to perform this operation correctly, you will need to make a copy of your image: writing to the input image directly while performing the blur operation will result in incorrect results (think about why, and write a unit test to check your code for this case!). You can use the .copy() function to make a copy of an input image as so:

   ```
   let imageCopy = inputImage.copy();
   ```

Note that you will have to deal with edge cases (literally!) in this assignment. Do something reasonable (not nothing!) to ensure that your code does not reference pixels outside the image, while producing reasonable values at the edges.

Testing Your Code

An important part of this project is testing your code thoroughly. Without appropriate unit tests, you may not catch bugs in your code, and hence will score poorly. To help you get started, we have provided a few test cases here. It is up to you to define additional tests to check your solution for correctness.

- Check to ensure that the function definition is correct. If it is correct, then this test will run without producing any errors. For example, there are no assertions in the following test since it does not test the output -- rather, it just confirms that the code has no compilation or run-time errors.

```
test('removeBlueAndGreen function definition is correct', function() {
  const white = lib220.createImage(10, 10, [1,1,1]);
  removeBlueAndGreen(white).getPixel(0,0);
  // Need to use assert
});
```

- Check to ensure that the center pixel in the image returned by `removeBlueAndGreen` has no blue or green channel values.

```
test('No blue or green in removeBlueAndGreen result', function() {
  // Create a test image, of size 10 pixels x 10 pixels, and set it to all white.
  const white = lib220.createImage(10, 10, [1,1,1]);
  // Get the result of the function.
  const shouldBeRed = removeBlueAndGreen(white);
  // Read the center pixel.
  const pixelValue = shouldBeRed.getPixel(5, 5);
  // The red channel should be unchanged.
  assert(pixelValue[0] === 1);
  // The green channel should be 0.
  assert(pixelValue[1] === 0);
  // The blue channel should be 0.
  assert(pixelValue[2] === 0);
});
```

- Your unit tests will need to check pixel values. However, in general, when dealing with fractional values, direct comparisons will not work, due to the limited precision of the internal representation of image pixel values. To overcome this, you should check whether the values are within a small threshold (epsilon) of the expected values. The following unit test demonstrates how to do this.

```
function pixelEq (p1, p2) {
  const epsilon = 0.002;
  for (let i = 0; i < 3; ++i) {
    if (Math.abs(p1[i] - p2[i]) > epsilon) {
      return false;
    }
  }
  return true;
};

test('Check pixel equality', function() {
  const inputPixel = [0.5, 0.5, 0.5]
  // Create a test image, of size 10 pixels x 10 pixels, and set it to the inputPixel
  const image = lib220.createImage(10, 10, inputPixel);

  // Process the image.
  const outputImage = removeBlueAndGreen(image);

  // Check the center pixel.
  const centerPixel = outputImage.getPixel(5, 5);
  assert(pixelEq(centerPixel, [0.5, 0, 0]));

  // Check the top-left corner pixel.
  const cornerPixel = outputImage.getPixel(0, 0);
  assert(pixelEq(cornerPixel, [0.5, 0, 0]));
});
```