Chris Arancio

Telling Time on a Clock

For the final assignment, I wrote four functions to estimate the time on a black and white clock. The first function is called tellTime(x,y,r) and it takes as parameters the x,y coordinates of the center of the circle and also the best radius of the circle, r. This function is the only one that needs to be called in the test script and returns the final estimated time as a string with the format “xx:xx.” Originally, my tellTime() function had a for loop that reads “for i in range(0,360,6).” The thought was that each minute is 6 degrees apart on a clock so if the algorithm could calculate the x,y coordinates at each of those 6-degree increments using trigonometry, then it would be able to walk from the center to those coordinates and keep track of how many black pixels in a potential minute or hour hand there were. The longest string of black pixels would, in theory, be the minute hand and the second longest would be the hour hand. While others may have had success with a similar loop, I did not.

Instead, I opted for storing each of the x,y coordinates on the edge of the circle first and then calculating the length of the black pixels. I stored these coordinates in the lines[] array where each row is a minute on the clock. The first columns of each row are the calculated x coordinate and the second columns are the calculated y coordinate. Using this method, I could reduce the for loop to only iterate from 6 to 90 with steps of 6. Like with our betterCircle() function, I could calculate the other points on the circle while only iterating through a quarter of the degrees on the circle. To note, the loop starts at 6 because the rightmost, topmost, leftmost, and bottommost coordinates are calculated on their own because of some complications with the trigonometry and thus occupy indexes 0 through 3 in lines[].

Now that we have the points at every minute, I wrote the findHands(x,y,r,lines) function to find the indexes of the lines[] array where the minute and hour hands are likely to be. When these indexes are returned, the structure of lines allows me to calculate which quadrant of the circle that index is in by calculating index % 4, where 0 is the top right, 1 is top left, 2 is bottom left, and 3 is bottom right (this follows the unit circle). I can then calculate the minutes by taking the returned minuteIndex and dividing by 4. This calculates a value from 1-15. Then based on the quadrant of the circle, this value can be adjusted by +15, +30, or +45 to find the actual minute value. For example, the index at 59 is supposed to be 29 minutes. So, int(59/4)=14 and since we know it is in the 3rd/bottom right quadrant, we add 15. Thus 15+14=29, which is the desired and correct minute value. To calculate the hour, I used similar logic based on the known data structure of lines[]. I was able to use knowledge of what quadrant of the clock we are in and what indexes exist in that quadrant to determine the hour with if/else statements.

The findHands() function is pretty simple because mostly calls another helper function called findClockHand(x,y,r,lines). It does hour ensure that the minute hand is not counted twice when finding the longest strings of black pixels. It does this by changing the calculated values at the index of the minute hand in lines to x and y. As you will see, this means that the distance calculated at that degree when findClockHand() is called again will be zero. Essentially, this allows me to reuse the findClockHand() function while forcing it to ignore the longest hand (minute hand) as a possible candidate and use it to instead find the other longest hand, which should be the hour hand.

The findClockHand(x,y,r,lines) gave me the most trouble in this assignment. Originally, I wanted to reuse the algorithm in our line() function to walk from the center point to each calculated coordinate in the lines function. However, for some reason, it was not working as expected. So, I had to write my own function that calculates the slope of the line depending on the quadrant of the circle. To do this, I created a for loop that loops through every row in lines[]. For every iteration, the currX and currY variables are initialized to the given x,y (the center of the circle) and they are incremented by a yInc and xInc, otherwise known as the rise and run of the slope of the line that goes from the center to the point given in line. This function seems to work for a lot of entries in lines[], but there were some complications that I had to account for.

First, there was the problem of vertical lines where calculating their slopes caused a “dividing by zero” error. This was addressed by treating these indexes in lines[] separately. The other problem was with slopes that were less than one. Since we must have discrete values for the pixel coordinates, converting the calculated slope to an int might round it down to zero. This is not accurate, so I accounted for this by forcing the slope to be at least one and tracking this change with a boolean variable called closeToCardinal (meaning that the minute line was close to a cardinal direction of N, E, S, or W and had a small slope). If this was true, then before I returned the index of the longest found line, I checked if the pixel at x+/-abs(int(round((math.cos(degreeInRadians)\*longestLine),0))) and y+/-abs(int(round((math.sin(degreeInRadians)\*longestLine),0))) for every combination of adding or subtracting between the two variables. Basically, I’m trying to determine which degree the longestLine is really at when the slopes were both rounded to 1. I do this by checking the endpoint of the longestLine and seeing if it really is close to the cardinal direction and if it is not, I adjust the longestDegree variable (the variable that stores the index of lines with the longest black line found) by +20. Because of how the data is structured in lines[] this will correct, a false positive at the 40-minute mark and correct it to be at the 35-minute mark. That said, I admit that this is not an ideal solution, but I had to try to make a solution for trying to map continuous slopes in a discrete space. This did work for the several clocks that I tested it on, but there may still be bugs here that I would try to fix if I had more time.

The final function is called checkBlackPixel(x,y) and it is very simple. It just checks if the current x,y are in self.red, self.green, and self.blue are equal to 0. This was made into its own function to avoid repetition of code and to increase readability.

To conclude, I admit that my solution may not be the most robust or efficient. I will say that it did work for several clocks that I found online, but there may still be some bugs. If I had more time, I would explore the possibility of trying to find the hands of the clock in self.convo[] instead of RGB. I would also give a second attempt at trying to calculate more accurate slopes by finding the nearest whole number multiple of lines with decimal slopes. The xInc and yInc could then increase by those whole numbers and the Euclidian distance between the jumps in x and y could be used to calculate the length of the longestLine. Finally, I acknowledge that populating the lines[] array may make the code confusing to read and understand, if the indexing system is not explained. However, its unique data structure also allows me to use some simple modular arithmetic and if statements (as described above) to determine the right minutes and hour. This assignment turned out more complicated than I expected, but I appreciate the challenge and I think that I gave it my best attempt, given the time allotted.