

Zack Owens

Project 1

9/30/2023

## Problem 1.1

The functions I created to handle the filtering of an RGB pixels are the following:

```
function filter_RGB(rgb::RGB{N0f8}, filter::RGB)
    return RGB{N0f8}(rgb.r * filter.r, rgb.g * filter.g, rgb.b*filter.b)
end

function filter_RGB(rgb::RGB{N0f8}, hsv_value_spec::HSV)
    rgb_v = convert(RGB{N0f8},hsv_value_spec)
    return filter_RGB(rgb, rgb_v)
end

function filter_RGB(rgb::RGB{N0f8},color_s::String)
    if(haskey(Colors.color_names ,color_s))
        r,g,b = Colors.color_names[color_s]
        rw = r / 255.0
        bw = b / 255.0
        gw = g / 255.0
        return RGB{N0f8}(rgb.r * rw, rgb.g *gw, rgb.b*bw)
    end
end
```

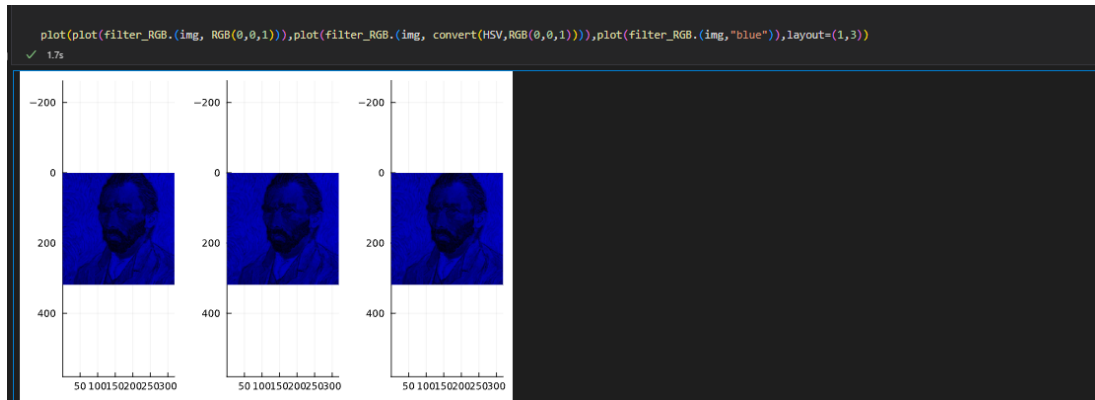
This function takes in a RGB, HSV, or a string that in the Colors library. These functions take advantage of Julia's multiple dispatch [[https://en.wikipedia.org/wiki/Multiple\\_dispatch](https://en.wikipedia.org/wiki/Multiple_dispatch)]. This means at run time Julia's internal language looks up the correct method definition to call based on the parameters. This can even use type tree lookups to find the correct parent type. The function for RGB works by multiplying the individual red, green, and blue of the passed in RGB pixel by the filter RGB pixel. Since both are floats and should not be over the value of one the multiplication will lessen the values creating a filter. If a white pixel is multiplied by a green pixel, then resulting pixel will be green. The same principle can be applied to HSV once a conversion from HSV to RGB occurs. The final conversion of Color.color\_names is different as it returns a tuple of the red, green, and blue values [<http://juliagraphics.github.io/Colors.jl/stable/>].

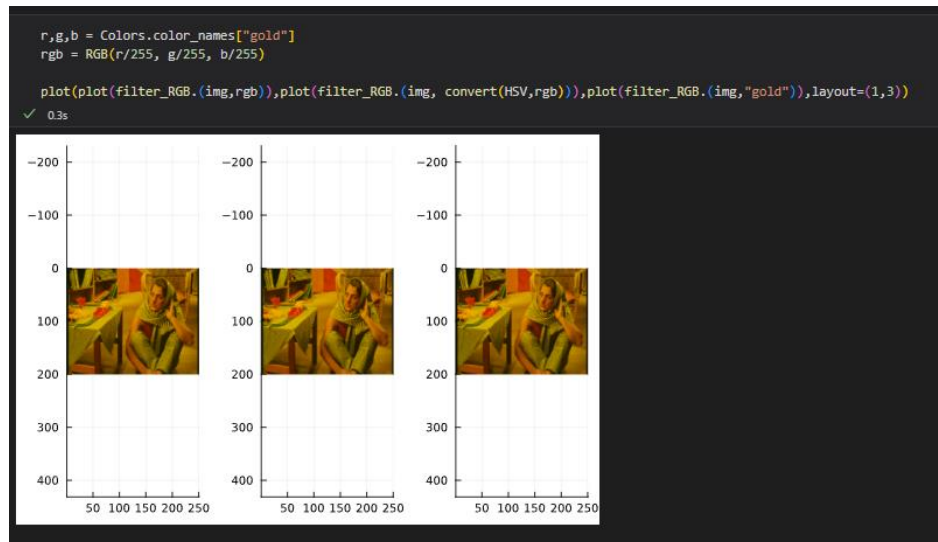
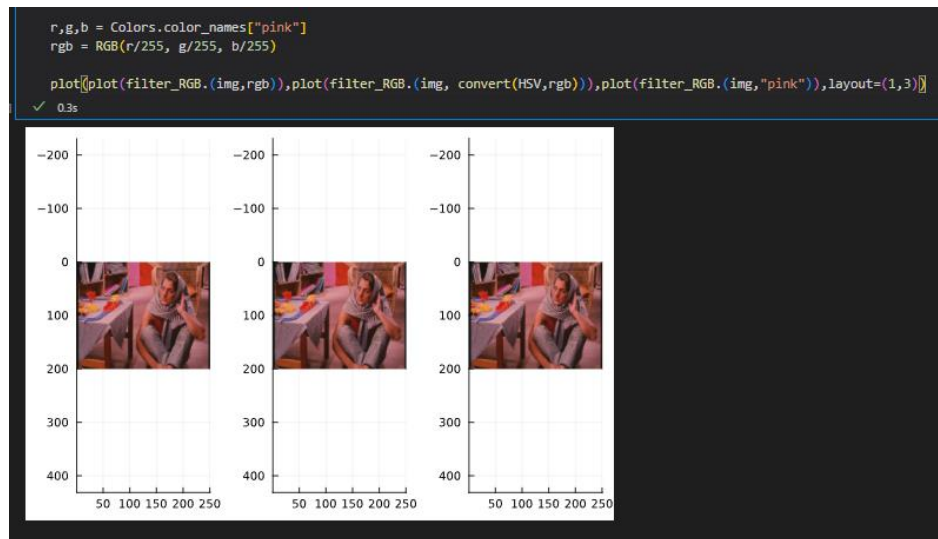
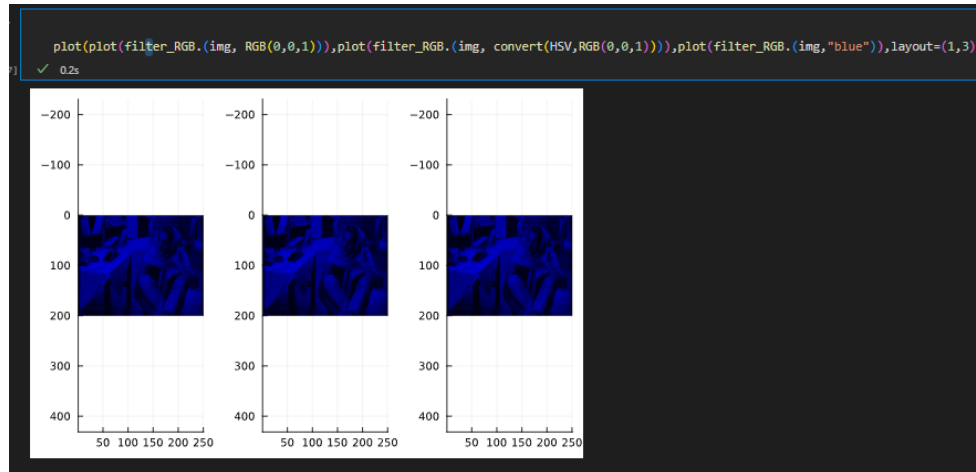


```
[148] Colors.color_names["red"]
✓ 0.3s
... (255, 0, 0)
```

To convert the integer tuple to RGB, a division by 255 as each value is a one-byte integer in this scenario. The values after this can be multiplied by the pixel value and create a valid filtered pixel.

Example of each section running:






## 1.2 Gray image to pseudo color

To create an algorithm that creates color from a grey image, the first step is to understand the data given from a gray pixel.

```
[23] ✓ 0.0s
gray_image = load("./cameraman_gray.png");

[107] ✓ 0.2s
gray_image[1,1]

...


▷
[112] ✓ 0.0s
typeof(gray_image[1,1])

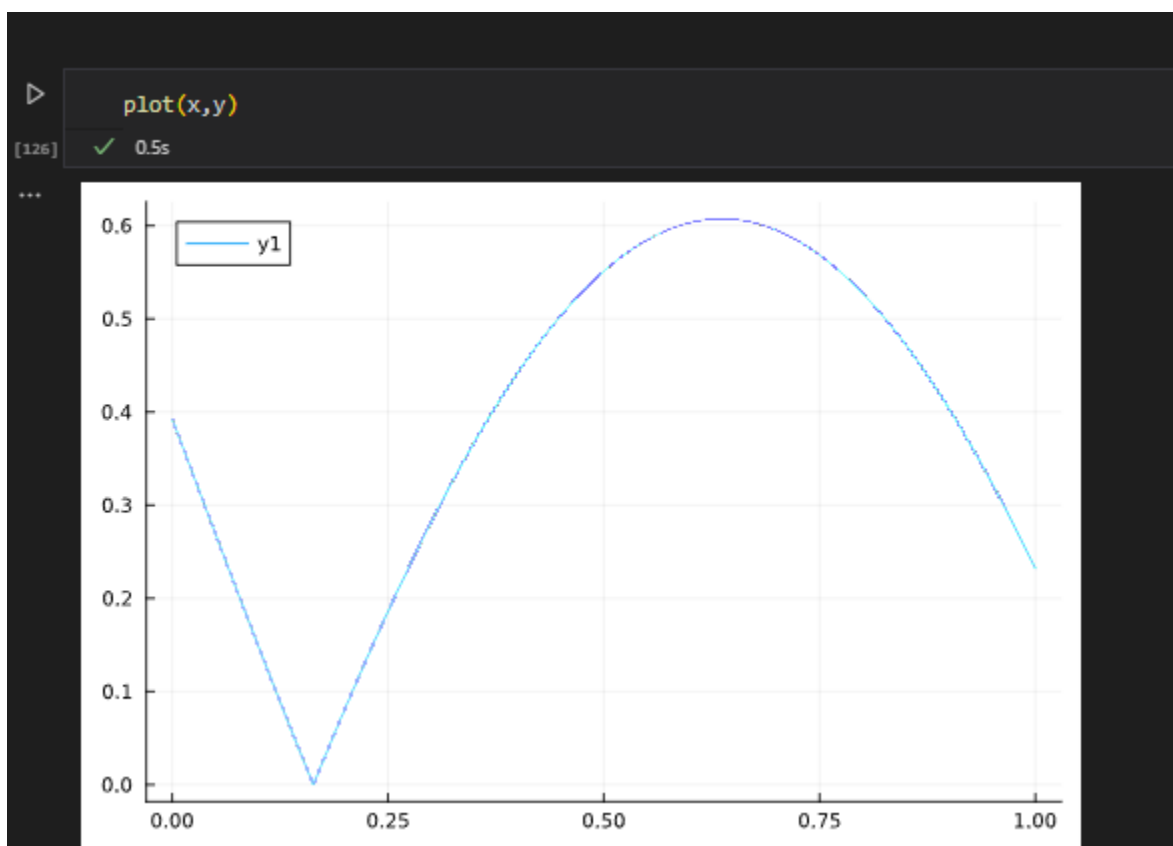
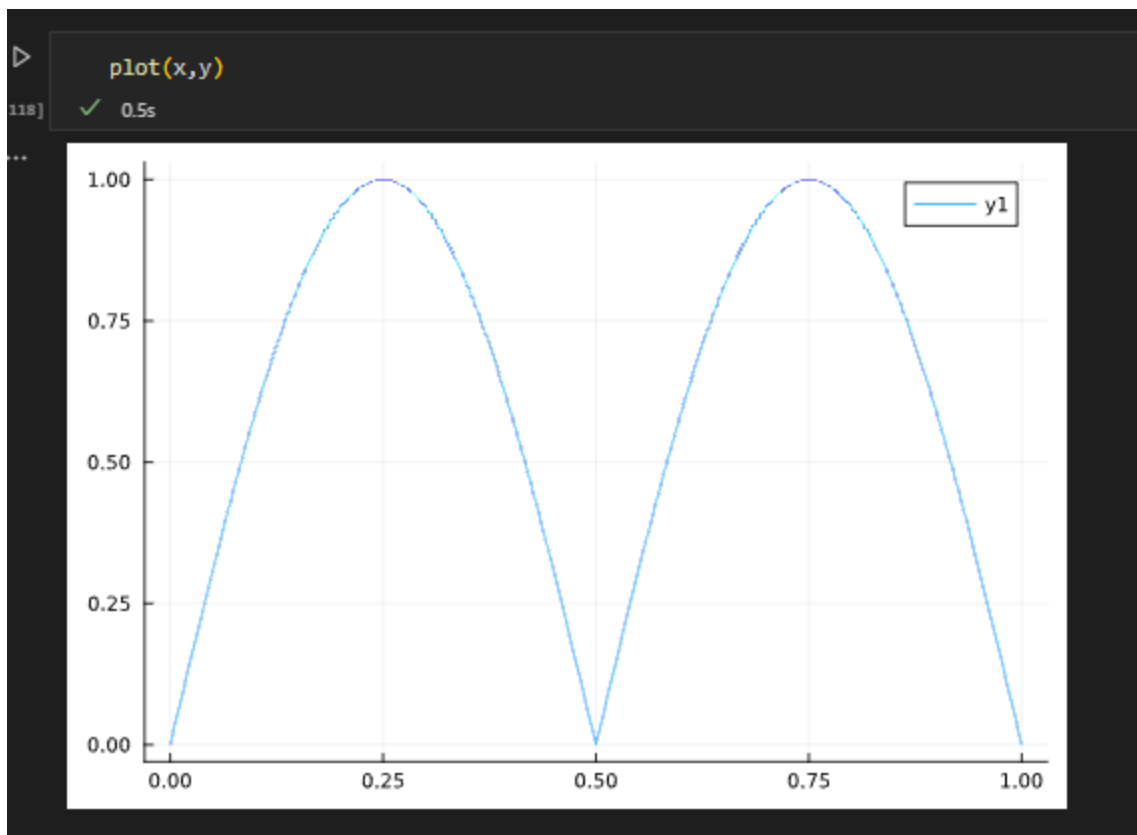
...
Gray{N0f8}
```

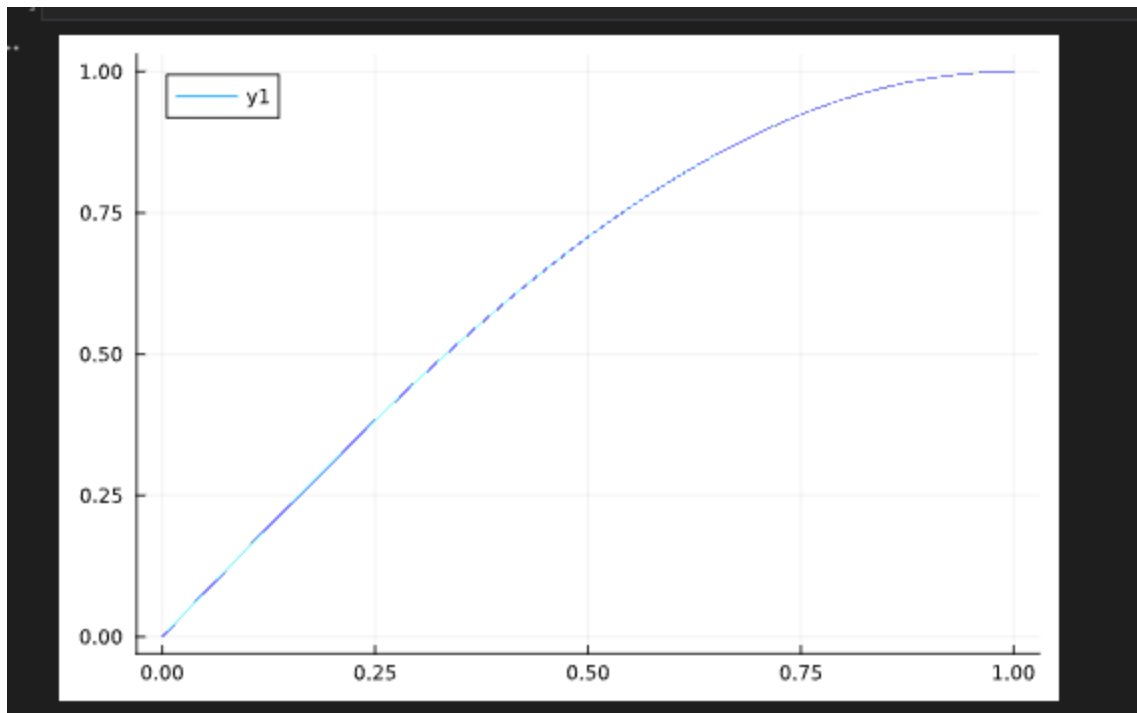
This portion of code shows that the value of a grey pixel is Float8 and used as such directly.

The next step is to explore different ways that pixel values can be represented. One such way is a curve. The curve that was used to create color curves was a sin wave.

```
function generateColorCurve(interval::Float64,phase_expand::Float64 = 1.0,
phase_shift::Float64 = 0.0)
    values = [abs(sin(angle*2pi*phase_expand)-phase_shift) for angle in
0:interval:1]
    return values
end
```

This portion of code creates an array of values from an absolute valued sin wave where the user specifies the interval step of a list, the phase expand, and the phase shift. This creates curves like the following:





By creating unique curves for red, blue, and blue and then using the curve as a lookup value an image can be converted to color.


```
function grayToRgb(grayPixel,rArray,gArray,bArray)
    digits = count(x->x=='0',string(length(rArray)-1))
    a = Gray.(convert.(Normed{UInt8,8}, grayPixel));
    rounded_grayValue = round(Float32.(a), digits=digits)
    index = convert(UInt32,round(rounded_grayValue*(length(rArray))-1))
    rValue = rArray[index]
    gValue = gArray[index]
    bValue = bArray[index]
    #print("$rValue $gValue $bValue")
    return RGB{N0f8}(rValue,gValue,bValue)
end
```

This code takes in the 3 curves y values as 3 arrays. The code then will generate the rounding accuracy by looking at the size of the arrays. The values are then converted from gray values to x values by rounding then converting to integer. The next step y values from the array index (the x values) and gets the red, green, and blue values in each array. This is then applied using a dot operator to apply over an image. Here is the results.

```
▷ gray_image = load("./cameraman_gray.png");
red_curve = generateColorCurve(interval,0.25,0.0)
blue_curve = generateColorCurve(interval,0.15,-0.1)
green_curve = generateColorCurve(interval,0.3,0.3)
141] ✓ 0.0s

... 100001-element Vector{Float64}:
 0.3
 0.2999811504440796
 0.29996230088816583
 0.2999434513322655
 0.29992460177638525
 0.29990575222053184
 0.29988690266471185
 0.2998680531089321
 0.29984920355319916
 0.2998303539975199
 ⋮
 0.6511031041465991
 0.6510972818479102
 0.6510914592112906
 0.6510856362367432
 0.6510798129242694
 0.6510739892738715
 0.6510681652855517
 0.6510623409593115
 0.6510565162951536

▷ #color_image = grayscale_to_color(gray_image,color_mapping)
grayToRgb.(gray_image, Ref(red_curve),Ref(blue_curve),Ref(green_curve))
143] ✓ 0.0s

... 
```

This is an example using very simple straighter curves.



```
gray_image = load("./cameraman_gray.png");
red_curve = generateColorCurve(interval,.8,0.0)
blue_curve = generateColorCurve(interval,0.5,-0.1)
green_curve = generateColorCurve(interval,0.2,0.3)
```

✓ 0.0s

100001-element Vector{Float64}:

```
0.3
0.29998743362938596
0.2999748672587739
0.29996230088816583
0.29994973451756374
0.2999371681469695
0.29992460177638525
0.2999120354058129
0.29989946903525444
0.29988690266471185
⋮
0.6510254457126718
0.6510293300611183
0.6510332142593842
0.6510370983074689
0.6510409822053718
0.6510448659530923
0.6510487495506299
0.6510526329979838
0.6510565162951536
```

```
#color_image = grayscale_to_color(gray_image,color_mapping)
grayToRgb.(gray_image, Ref{red_curve},Ref{blue_curve},Ref{green_curve})
```

✓ 0.0s

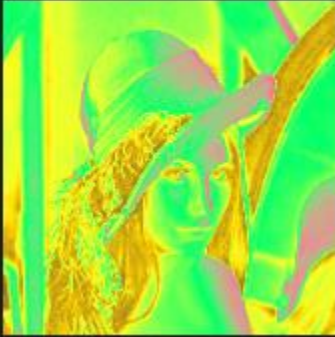


An Example of a more extreme curve.

Here is some examples on the lena image.

```
grayToRgb.(gray_image, Ref(red_curve),Ref(blue_curve),Ref(green_curve))
```


✓ 0.0s



```
gray_image = load("./lena_gray.png")
red_curve = generateColorCurve(interval,.3,0.0)
blue_curve = generateColorCurve(interval,0.5,0.0)
green_curve = generateColorCurve(interval,0.6,0.3)
[157] ✓ 0.0s
```

```
... 100001-element Vector{Float64}:
 0.3
 0.29996230088816583
 0.29992460177638525
 0.29988690266471185
 0.29984920355319916
 0.2998115044419008
 0.29977380533087034
 0.2997361062201614
 0.2996984071098274
 0.2996607079999221
 ⋮
 0.8875412317869749
 0.8875717372732121
 0.8876022419243792
 0.8876327457404323
 0.8876632487213281
 0.8876937508670233
 0.8877242521774746
 0.8877547526526395
 0.887785252292473
```

```
grayToRgb.(gray_image, Ref(red_curve),Ref(blue_curve),Ref(green_curve))
[158] ✓ 0.0s
```

```
... 
```

An example of a slightly better version. Overall to tweak the curve to be perfect one would need to create an average value of grey to RGB curve over a series of images then apply that to images to get a realistic looking-colored images.

## 2.1 Color schemes with Vangogh


Using the library ColorSchemes [<https://juliagraphics.github.io/ColorSchemes.jl/stable/basics/#Pre-defined-schemes-1>], there is a set number of preset color schemes. One of the being the vangogh starry night. The following shows what how the color scheme was made.

```
colorschemes[:vangogh].notes
✓ 0.4s
"from artist Vincent Van Gogh's painting The Starry Night, painted in June, 1889, depicting the view from the east-facing window of his asylum room at Saint-Rémy-de-Provence."
```

## 2.2 Comparing with Extract

The Color schemes library gives an example of example of a color scheme of the starry night. The following is the color scheme from the ColorSchemes library:

```
starry_night_scheme = colorschemes[:vangogh]
✓ 0.6s
```



The ColorSchemes library uses a method called extract to pull the color schemes from a image based on a series of parameters [<http://juliagraphics.github.io/ColorSchemes.jl/v1.5/basics.html#ColorSchemes.extract>]. This method creates a color scheme based on an input image and the parameters. A color scheme is 30 colors that make up a full image.

To compare the extracted version of starry night and the ColorScheme generated version, the first step is to download the starry night image [[https://en.wikipedia.org/wiki/File:Van\\_Gogh\\_-\\_Starry\\_Night\\_-\\_Google\\_Art\\_Project.jpg](https://en.wikipedia.org/wiki/File:Van_Gogh_-_Starry_Night_-_Google_Art_Project.jpg)] and run the follow code to generate a custom color scheme.

```
starry_night_extract = extract("starry_night_full.jpg", 30, 50, 0.01)
[66] ✓ 3.0s
```



The next step is to generate a way to compare the colors. To begin this process it would be beneficial to sort the colors. The following function is an example of how to create a color comparison based on brightness of a color:

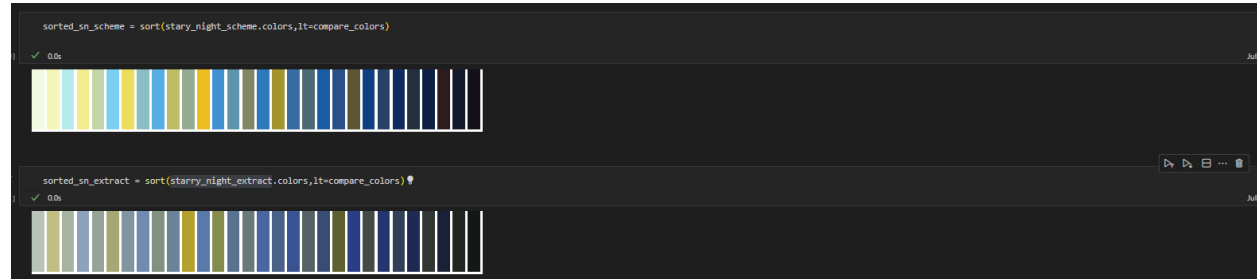
```
function compare_colors(color1, color2)
```

```

r1, g1, b1 = red(color1), green(color1), blue(color1)
r2, g2, b2 = red(color2), green(color2), blue(color2)
if(r1+g1+b1 > r2+g2+b2)
    return true
end
return false
end

```

Using this function and the base Julia sort function [<https://docs.julialang.org/en/v1/base/sort/>] creates the following result:



By first glance, the schemes are very similar and there are clearly some values that are very similar.

Overall, the extracted values are much darker than the colors from the ColorScheme library.

To further compare the two sets of colors two functions can be used. The first function take the most similar color in both list and compares the absolute values between RGB values. The larger the difference the less similar colors are. The other part of the output is the individual red, green, and blue differences that create the overall difference value between two pixels. Attached is text file `closest_color.txt` with the results.

```

for color in sorted_sn_scheme
    best = 999.99
    closest_color = nothing
    values = (0,0,0)
    r1, g1, b1 = red(color), green(color), blue(color)
    for color2 in sorted_sn_extract
        r2, g2, b2 = red(color2), green(color2), blue(color2)
        rd = abs(r1-r2)
        gd = abs(g1-g2)
        bd = abs(b1-b2)
        best = rd+gd+bd
        closest_color = color2
        values = (rd,gd,bd)
    end
    println("$color : $closest_color : diff $best : rgb $values")
end
end

```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

The overall results are interesting as the largest values is 0.608 which is the first values of the extracted value. This is the first value on both lists compared the darkness of the extracted values is much darker creating these larger values. The smallest value was 0.06809 which is the 22<sup>nd</sup> color on the extracted value compared to the 24<sup>th</sup> values on the ColorSchemes. This shows that the sorting does not care about how similar values only about the brightness of an image.

The next comparison is the in-place comparison. This means that index 1 of the first list will be compared to the first of the second color list. This gives an overall comparison of the two list by brightness. The results are stored in index\_color.txt. The results overall are worse as many of the values are over 0.9. This gives an overall realistic view on how different the two list are. To understand how large the values of 0.9 are the max difference of values can be 3.0. A value of 1 would mean that a color is 30% different from its compared pixel. Adding all the differences together gives 14.636207964922842485 and then dividing by number of colors in the list gives the value 0.487873598830761. This means that on average each color is less than 16% different.

## 2.3 Mandelbrot Set

The Mandelbrot sets is a great example of using a heatmap  
[[https://en.wikipedia.org/wiki/Mandelbrot\\_set](https://en.wikipedia.org/wiki/Mandelbrot_set)] to explore the difference in values. The madndelbrot set can be defined as the following equation:

$$z_{n+1} = z_n^2 + c$$

The Mandelbrot set uses the complex plain to create values to create some of the values and so the z is actually the complex unit.

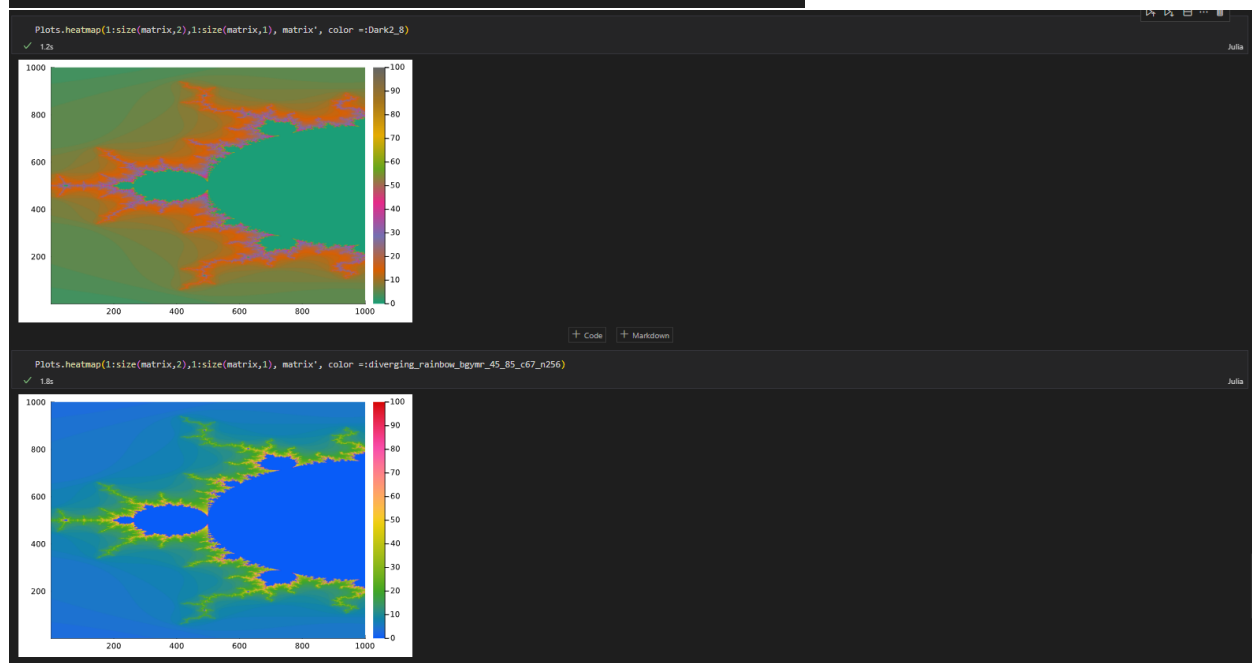
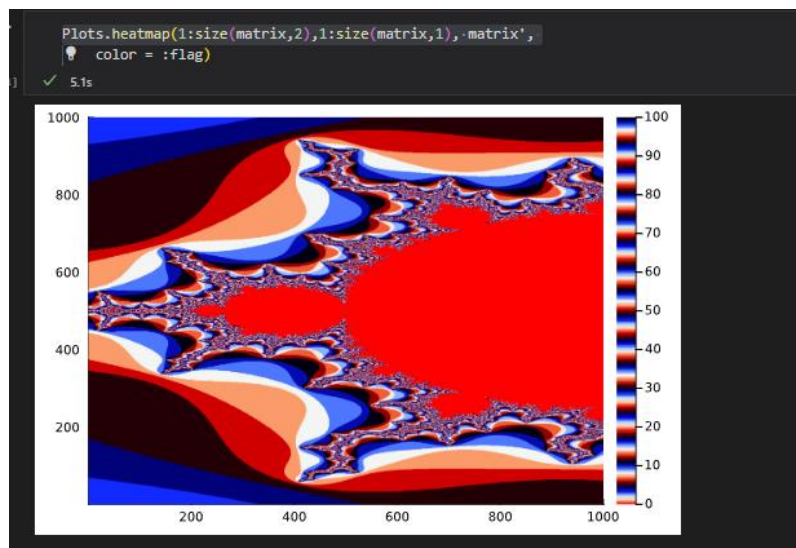
The values can then be fed into a heatmap in with a color scheme to create unique maps. Color schemes can be found at <https://docs.juliaplots.org/latest/generated/colorschemes/> . The script is the following:

```
function mandelbrot(x, y)
    z = c = x + y*im
    for i in 1:100.0
        abs(z) > 2 && return i
        z = z^2 + c;
    end;
end
```

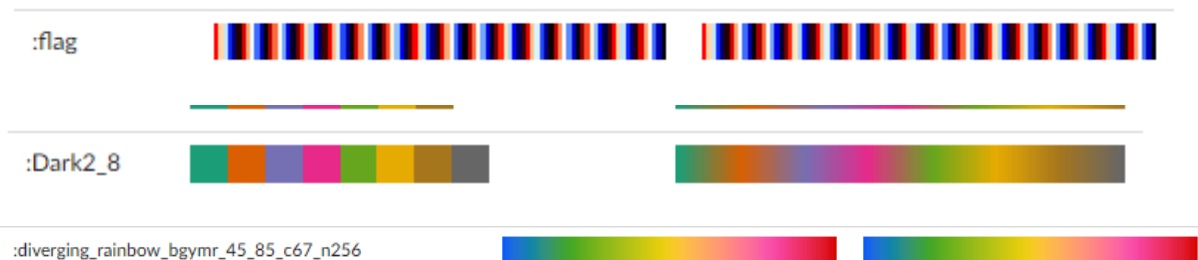
The range used is the following:

```
x = LinRange(-1.5, -1, 1000)
y = LinRange(-.5, .5, 1000)
matrix = mandelbrot.(x, y');
```

This range was picked due to its unique values and many tips which show full range of each scheme.



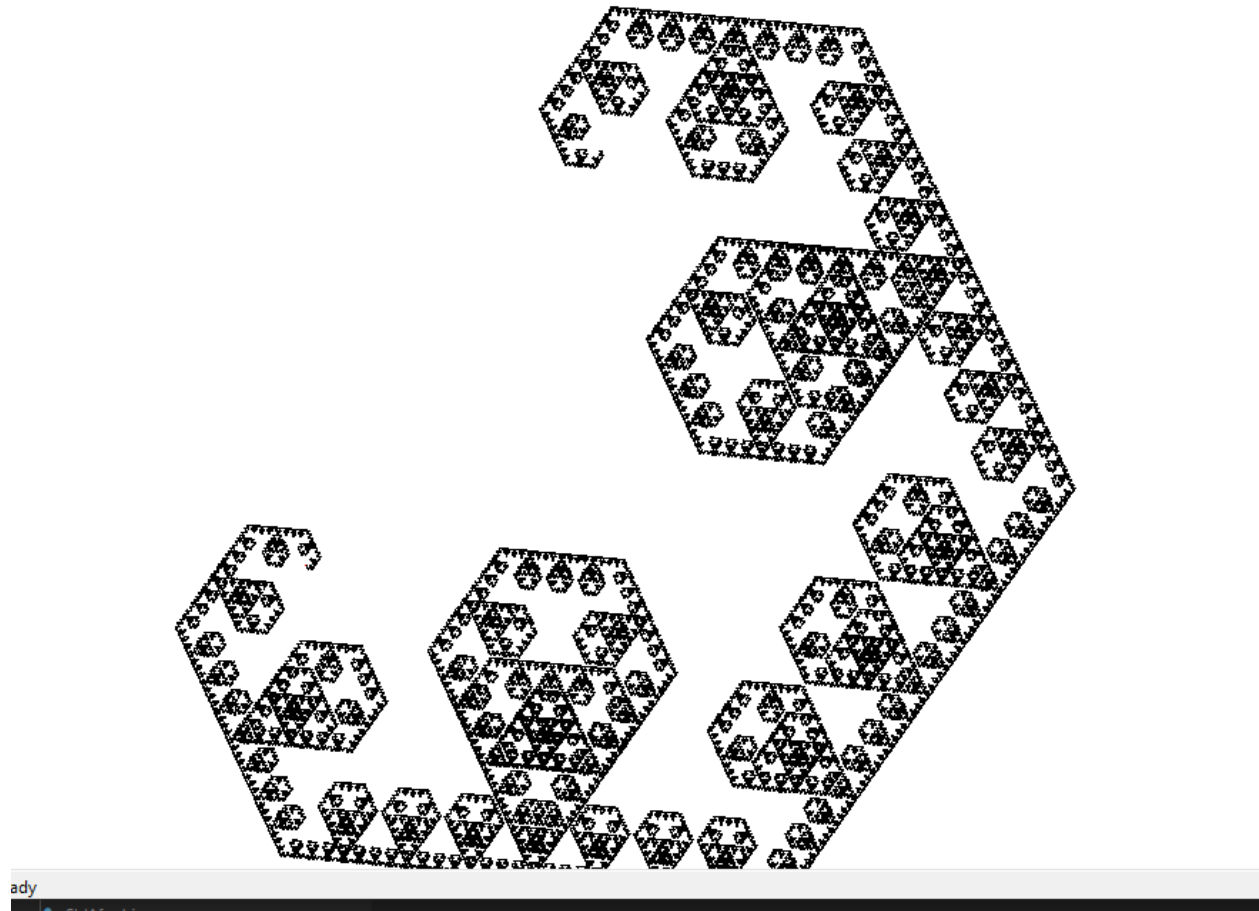
Examples of the color schemes used:





### 3.1 Fractal

The fractal curve that was created is the following. This is a unique curve created by L-System program [<https://en.wikipedia.org/wiki/L-system>]. This curve is clearly a fractal as you can see the repetitions inside itself in the larger diamond shaped portions.



The following is the system that creates the L-System. The starting axiom is FA. Note that all F are the turtle command for drawing. The  $A \rightarrow A+BF+BF+A$  creates a recursive loop as  $B \rightarrow F-B-F$  is called within A. Although this loop is unwrapped by the recursion depth of 18 so this does not infinitely create the values and the angle of each iteration being 30 degrees.

Model Parameters	
Axiom	FA
Production 1	A A+BF+BF+A
Production 2	B F-B-F
Production 3	
Production 4	
Production 5	



## 3.2 Explaining Turtle Graphics

Turtle graphics is a simple graphics technique where main components are stored the location, direction, color, and width [[https://en.wikipedia.org/wiki/Turtle\\_graphics](https://en.wikipedia.org/wiki/Turtle_graphics)]. These attributes can then be applied over a series of commands to create images like how drawing on paper is.

For the L-System, there commands that are uses are move, turn, rotate, pen up and pen down. The L-System takes the Axioms and product equations and parses them into Turtle commands. The F tells the turtle to go forward and + and – is turn left or right based on inputted angle. The next step is to iterate o the initial axiom then replaces the initial axiom with the equation. The level of recursion is shored with the state of the turtle so that when the next level is added the turtle can revert to the pervious location. This is repeated until the depth of the recursion is reached.

Three features that could be added to this application to make it easier to use. The the ability to take a sub formula and display it live to help the user create the inner workings of each formula. Another feature would be command would be the ability to create full shapes without the need for axom. Example would be to add a box command so A->++F+Box which would just draw a square when parsing the box. This feature could also be applied to circle but with a radius. This would work like A->++Circ(3) where the radius is 3 Fs. These commands could make this application much more user friendly and allow for better fractals to be created with the limited number of products that this application allows.