Introduction

All the files necessary to the perceptual gamma correction task are located in my desktop. Within it, there is a folder with all the scripts I created, another folder with all the produced outputs, and yet another folder with "AssociatedFiles" that were produced by Dr. Hannah Smithson, a professor in Oxford's Department of Experimental Psychology. . This document will provide a written description of how all of these files work together, in case that is needed.

The code is broken down into different "Questions" or individual steps that are solved to conduct the whole procedure Every 'question' is answered in a separate file, some of which are functions. Function inputs and outputs are explained at the beginning of every file. Some questions necessitate multiple functions, so multiple scripts were produced. All file names begin with the Question Number they are answering (e.g. Q1 for "Question 1") and continue with a descriptive name (e.g. Q8_SetUp"). Unless specified otherwise at the beginning of the script, each script can be run independently, with the workspace empty, and without any additional input.

I have tried to give comprehensive descriptions of what the code does in the scripts, but there are general decisions and assumptions that for readability purposes were not included there. These decisions and assumptions are the focus of this document. To avoid redundancy, and to dedicate as much space to these explanations as possible, here I have obviated descriptions of simple code and focused solely on high-level descriptions, decisions and assumptions. For a more direct explanation of the superficial working of the code, please refer to the script files.

Q5_PowerLawFit

Here I fit a power-law curve to the data provided in GammaData.csv. This question requires I use one dataset, but I need the same code in Q16. To avoid code redundancy, I created an optional input argument. If satisfied, it fits and plots the data provided in the argument, and if the argument is not provided (i.e. nargin <1), it defaults to using the dataset required by this question.

Because here we are estimating the line that best fits, this graph plots datapoints with red crosses, and draws the bestfitting line with a green line (as opposed to plotting the datapoints with a line). I also print the best fitting exponent calculated by the function (represented by "m") in the graph. I used "%3.2f" to force the format to 3 figures and 2 decimal places. I chose not to use the "round" function instead to avoid having a function within a function (less readable).

Q6_GeneratePatterns

This script generates patterns with varying proportions of black and white. I want 8 arrays for 8 different test patterns, so I determine the variable numArrays to be 8. Because each pattern requires its own matrix, I create an empty cell array called "binaryElements" capable of holding all of them independently. This way, I can create a for-loop that allows me to cycle through the length of numArrays, such that each number from 1:8 creates a new cell in binaryElements with the matrix-pattern I want, and saves it in the desired directory with the appropriate name.

To create the patterns inside the for-loop, I use the GenerateThreeByThreeElement function provided in the AssociatedFiles folder. To use a function from a different directory, I first add the path to that folder. This function takes a number between 1:8, and 'colors' that number of pixels in a binary 3-by-3 matrix. To generate a 600x600 pixel-pattern with varying proportions of black and white, I use Matlab's "repmat" function to stitch 200 repetitions of the pattern provided in a matrix.

Before closing the for-loop, I take advantage of the current value of "i" to also export the just-made binaryElement to its appropriate file. I use the value of "i" to select the matrix within the i'th binaryElement cell, and to name the output file. After running the script, binaryElements should have 8 cells with 8 different halftone patterns. To visualize them, you can use the code in line 12.

Q8_SetUp and Q8_Update

To create the trial, I need to set up a screen in PsychToolBox (PTB), superpose an oval to the background halftone pattern, and update the color of the oval. This is accomplished in two custom functions: Q8_SetUp, and Q8_Update. They can be run using the following code*^{note}:

```
[windowPointer, centeredRect] = Q8_SetUp(1, 0.2);
Q8_Update(windowPointer, centeredRect, 0.5)
```

Q8_SetUp requires two input arguments: numWhite and the startingGreyLevel. The former needs to be a number between 1:8 and be used to pull specific images we created in Q6, and the latter requires a number between 0:1 to specify how dark the oval will be.

First, I do some error management to ensure these conditions are met. Both if-statements could have followed both formats ("~ismember" and "x<0||x >1"), but I kept them different for the sake of practicing both.

Second, I set up PTB. After setting visual debug preferences and sync-tests to avoid PTB bugs, I open a black [0 0 0] window and hide the cursor, since it will not be needed during this experiment. The window opened is outputted as ‘Screen’, on which I will draw the relevant figures. The dimensions of this screen are outputted as ‘WindowRect’, which will become important when centering the oval.

Third, I create the background image. I use Matlab’s “imread” function to read the halftone pattern, and then make a texture out of it using PTB commands. I do this instead of integrating the image directly with “putimage” because drawing textures in PTB is quicker.

The next step is centering the oval. The texture was drawn in the center of the screen by default, but to draw the oval in the center with the we need to get the coordinates of the center of the window. This is accomplished by feeding windowRect to PTB’s function “RectCenter”. The outputted coordinates are used alongside the size of the oval by the “centerRectOnPointd” function to provide a central location for the oval. The output from this last function is used by the “FillOval” to place the oval in the center of the screen.

To determine the color of the oval, we must first transform the startingGreyLevel input, which ranges between 0:1, to the 256 RGB color system used by “FillOval”. Because grey is an even combination of each color (RGB), we can simply repeat the input 3 times to get a vector of length 3, and multiply by 256 to resize the shade given on the 0:1 scale, to a number of equal proportions in the 256 RGB scale.

At this point I have everything that I will be exporting to the Q8_Update function: a screen with the halftonePattern, and the coordinates to place the oval in the center. To store the former, I make an empty cell array, and save the ‘screen’ in one cell, and the image texture (which I call ‘pointer’) in another. These will be unpacked to recreate the same image in the following function without having to set-up PTB again, and re-create the texture. The coordinates are stored in a variable called centeredRect. Finally, we flip the screen to make all changes visible and await user input via KbWait to finalize this first function. This last step (KbWait; line 51) must be muted when running all the experiment (as opposed to just Q8).

Q8_Update requires three input arguments: windowPointer, the centeredRect, and the updated GreyLevel. The last is entered manually, and the former two are exported directly from the previous function. After implementing the same error checking I used for the startingGreyLevel in Q8_SetUp, I unpack the items in my windowPointer and load them both at once using PTB's "PutImage" function. Then I simply draw a circle with PTB's "FillOval" function using the imported centeredRect, flip the screen, and demand a wait of at least 0.25 seconds before accepting a new keystroke and closing all screens. This wait prevents the keystroke entered to finish Q8_SetUp from affecting this one as well. When running the whole experiment, lines 30-32 must be muted.

* $^{\text{note}}$: These functions cannot be called with the sample code provided in the assignment. I could have forced the code to be called using that exact same format, but I thought this to be more efficient. In the original code, "hPattern" and "wPattern" are drawn from the halftonePattern and used to draw the circle with adequate proportions. In my code, I use dimensions from the PTB window to draw the circle with adequate proportions. Although not the same, I thought my code was more akin to the workings of PTB's functions. Indeed, it is structured this way because PTB's "FillOval" function demanded this structure rather than the width and height dimensions of a disc that was drawn **not** using a PTB function. Therefore, I consider my code to be adequately analogous to the original sample format, which was used to run non-PTB code.

Q9_GetKeyPress

This function takes no arguments, and outputs the first key pressed (out of a series of allowable keys) after running it. To ensure that pressing the keyboard to run the function does not trigger a registration, I force a wait of 0.25 seconds before the function begins looking for a response. Then I set a global variable called “press” to equal zero, create an empty cell array to store the key pressed, and specify a range of allowable keys.

Next comes a while-loop stating that for as long as “press” is not changed from its original value, the computer must wait for a keystroke. Once any key is pressed, a for-loop checks if the key pressed is the same (via Matlab’s function “strcmp”) as one of the target keys. It accomplishes this by cycling through the list and comparing them to the key pressed one by one. If one of the logical comparisons is successful, it displays which key was pressed (translating the 256 bit code from KbCheck’s output “keycode” with “KbName”), stores it in the global “keypress” variable, and updates “press”—breaking the while-loop.

Q10_RunTrial

The function Q10_RunTrial runs a complete “method of adjustment” trial. It intakes a number representing the halftonePattern it will be using (i.e. numWhite) and randomizes the starting level of the startingGreyLevel. Then it uses these values to run an experimental procedure that presents this randomized startingGreyLevel color using the custom Q8_SetUp function, and dynamically updates it using Q9_GetKeyPress, and Q8_Update functions.

When the participant presses a key, the script enters into a while loop that collects user input from the range of allowable keypresses specified in Q9, and performs different functions depending on which key was pressed. Namely, it either increases the whiteness of the oval, decreases it, or if the participant indicates a match between the disc hue and the halftone Pattern hue, it exits while-loop and returns the last updatedGreyLevel value.

In Q6, I created 8 halftonePatterns ranging from 1 to 8 pixels of white per 9-pixel matrix (3-by-3). This means that including pure black and pure white, there are 10 possible, evenly spaced patterns. These values range from $\left(\frac{0}{9}\right)$ to $\left(\frac{9}{9}\right)$ in whiteness, passing through $\left(\frac{1}{9}\right)$, $\left(\frac{2}{9}\right)$, etc. However, because we were only asked to create halftonePatterns ranging from 1:8 white pixels per 3-by-3 matrix, the perfect matches should range from $\left(\frac{1}{9}\right)$ to $\left(\frac{8}{9}\right)$. Valid startingGreyLevel values range from 0 (absolute black) to 1 (absolute white), so hypothetical perfect matches in this scale should be represented by $(1*\left(\frac{1}{9}\right)) = 0.\overline{1}$ to $(1*\left(\frac{8}{9}\right)) = 0.\overline{8}$. To ensure that perfect matches are possible, I kept these proportions and randomized the startingGreyLevel to have 10 possible starting points for every perfect match, adding up to a total of 90 starting possibilities. Large changes are $\pm\left(\frac{1}{9}\right)$, and small changes $\pm\left(\frac{1}{90}\right)$. This way, no matter where you are randomly assigned to start, you can reach the perfect match using both types of changes.

Participants are only able to make an increase or a decrease if the change does not exceed the available range (0:1) in steps of $\pm\left(\frac{1}{9}\right)$ or $\pm\left(\frac{1}{90}\right)$. If they press a key that is not correct, or exceed this range, the script issues a beep. Incidentally, this method of updating the color of the disc does not create a whole-number translation into the 256 shades of grey accepted by "FillOval" (e.g. if participant indicates a match at $\left(\frac{3}{9}\right)$, then $\left(\frac{3}{9}\right) * 256 = 85.\overline{3}$). However, even if these decimals are not appreciable to the human eye, they are accepted by "FillOval", and in any case, they correctly represent the exact shade the halftonePattern is supposed to be.

Line 46 (closing the windows) needs to be muted when running the whole experiment.

Note: By keeping the steps additive instead of multiplicative, we keep the proportions constant, making sure we do not miss the exact match. To illustrate, a 10% increase from 50 is 55, but adding another 10% of the original value (i.e. additive; equals 60) is not the same as multiplying the current value times a 10% increase (i.e. multiplicative; equals 60.5). However, it is important to notice that the issue does not arise from the multiplication itself, but from multiplying the X% increase from the *updated* value instead of the original value.

Q11_Random

Q11_Random is a function with three output variables that randomizes a complete experiment presenting each halftonePattern three times. Every pattern is repeated once before moving on to the second 'wave' of repeats. The grey-level matches and the halftonePattern numbers are recorded and later organized into a table.

To randomize the trials, I started by creating empty cell arrays to store the randomized numWhite values and the user input. This allows me to later match each response to each pattern. The actual randomization takes place in two for-loops. The first creates three global repeats and randomizes a list of numbers between 1:8 without replacement to be stored in each version of the numWhite value. The second for-loop is nested, such that before every cycle of the primary for-loop is completed, the second for-loop cycles until completion (i.e. 8 times). With every iteration, the nested for-loop runs the custom function Q10_RunTrial and uses the point in which the first and second for-loops are in their cycling (using "i" and "j") to correctly categorize trial responses into the different cycles (1:3) and trials (1:8), respectively. In this way, we obtain a total of 24 entries (3x8) before finalizing the experiment.

Next, I organize the data. Upon transforming the updatedGreyLevel (user input) and numWhite (halftonePattern number) to a matrix, each cycle of 8 trials is placed below the previous. Therefore, by joining both matrixes, I have a 2-column table of all the trials with the randomized numWhite and their respective updatedGreyLevel in order. To format the table like the sample table, I create one variable for each cycle, and sort the rows of each table in ascending order based on the elements in the first column (i.e. numWhite). I also round the values to 2 decimal points to increase readability. Next, I create a table selecting both numWhite and updatedGreyLevel from the first variable (T1), and append the already-ordered updatedGreyLevel values from the other two variables. This way, I end up with a table containing a column with numWhite values, and three columns with participant input. Finally, I give columns a descriptive header.

Q12_Export

In this script, I collect the participant name and export the table created in Q11. We were tasked with creating a 3-letter participant identifier for the name of the table, so here I ask participants to provide their first, middle and last name (or two last names in its default), and created a while loop that is triggered if the user inputs any number of words different than three. The only way to exit this loop is providing 3 identifying words.

To select the first letter from each identifying word, I used a function (cellfun) that applies an anonymous function taking the first item of each string (i.e. the first letter of each name) in pName. I use the variable created by this function to name the outputted table, which I saved in the “Output” folder.

Q14_Plot

There are 3 key components to plotting the data of 20 participants at once: loading all participant files, separating the file created in Q12 (my data), and stitching all of the sample data in one visual.

First, I get the directory of all the files in the Output folder. Before starting a for-loop that will cycle through all of the ones whose name starts with “ PerceptualGamma_”, I create a 5x4 tiled layout that will serve to stitch all the graphs together. Then, for every file in the directory I take its name, compare it to the name of my own file, and if it is not the same, I continue (if it is the same, I skip it). I extract the 3-letter participant identifier, read the table, and plot the data in a tile. I plot one line for each cycle of the participant, and with every iteration, I plot a new graph in a new ‘tile’. Finally, I use the current figure handle (which contains the 5x4 tiled layout) and export it to the Output folder.

Q15_Summary

As in Q14, I begin by extracting the names of all the PerceptualGamma files from the Output directory. To calculate the within-participant means, I create an empty dataframe and iteratively read each participant’s file (skipping mine), calculate the means, and append it to the empty dataframe I created. Next, I erase the empty column (pertaining to my skipped file), and I create a new Summary variable with 4 columns (halftonePatterns, means, SD & SE). Finally, I make a summary plot of across-participant data, and save it in the Output folder.

Q16_Estimate

Finally, I use the data I produced in Q12 and the custom PowerLawFit function created in Q5 to estimate the gamma of my display. First, I load the data, calculate the inter-cycle means, and transform the number patterns (e.g. halftonePattern1) into what they represent (e.g. 1/9th of whiteness). Then, I run the Q5_PowerLawFit with my data passed as the argument, which overrides the default of using the dataset used in Q5.

With the sample data from the GeneratePerceptualGamma function I obtained a TargetGamma of 2.19, and using my data I obtained a value of 2.026 (2.03 to 2 d.p).