Reverse Engineering Instagram's "Hefe" Filter

Coriander Pines

2014 October 22

Abstract

Instagram is a popular image-sharing platform available for mobile devices that allows users to capture pictures, apply one of several artistic filters, and post the resultant images online for others to view. These filters are presented as black-box systems, providing no user-definable parameters or configurations, save the input image itself. In this exercise, we recreated the Instagram filter "Hefe."

To achieve this, we generated a set of test images, applied the filter to each one in the Instagram app, and compared the pre- and post-filter images quantitatively. As a result of this exercise, we distilled our work down to a simple function that replicates the "Hefe" filter when applied to an image array. We were also able to create an inverse filter which, given an Instagram-processed image, was able to recover the original data. The images produced by these functions were accurate to a high degree of similarity.

1 Introduction

This exercise focused on analyzing and recreating the internals of an Instagram filter as a black-box system. Visually, "Hefe", the filter chosen, increases the contrast, applies a red-orange color cast, and adds a vignette effect to an image.

In order to recreate these effects, we chose to work primarily in MATLAB, though some portions of the code were written in Python. Due to the wide range of methods employed, the analysis was broken down into several experiments. The Test Set Generation phase handled the generation of a set of test images. The Pre-Analysis phase consisted of various graphs of transformations performed by the filter, and provided insight on how to best proceed. Experiment 1 checked for the existence of blurring kernels and verified inter-pixel independence. Experiment 2 confirmed per-channel linear independence and function decomposability. Experiment 3 generated polynomial regressions of channel transforms and built several models of the filter's tone mapping. Experiment 4 selected the best tone mapping model as produced by Experiment 3, and validated the appropriateness of the model used. Experiment 5 reconstructed the vignette effect applied by the filter and integrated it with the tone mapping model, signifying the successful completion of the exercise. Both quantitative and qualitative analysis were made after each experiment to evaluate its success, the results of which are included in the relevant sections. Full code for each experiment is provided in the appendix.

As a result of the experiments and analyses performed, we have determined that Instagram's "Hefe" filter operates by first applying a vignette effect using a multiplicative blend mode with a predefined mask image then tone mapping the resultant composite with three polynomial tone mapping functions, applied per channel in RGB-space. Succinct code to perform this transformation is provided in this paper's conclusion.

2 Procedures and Analysis

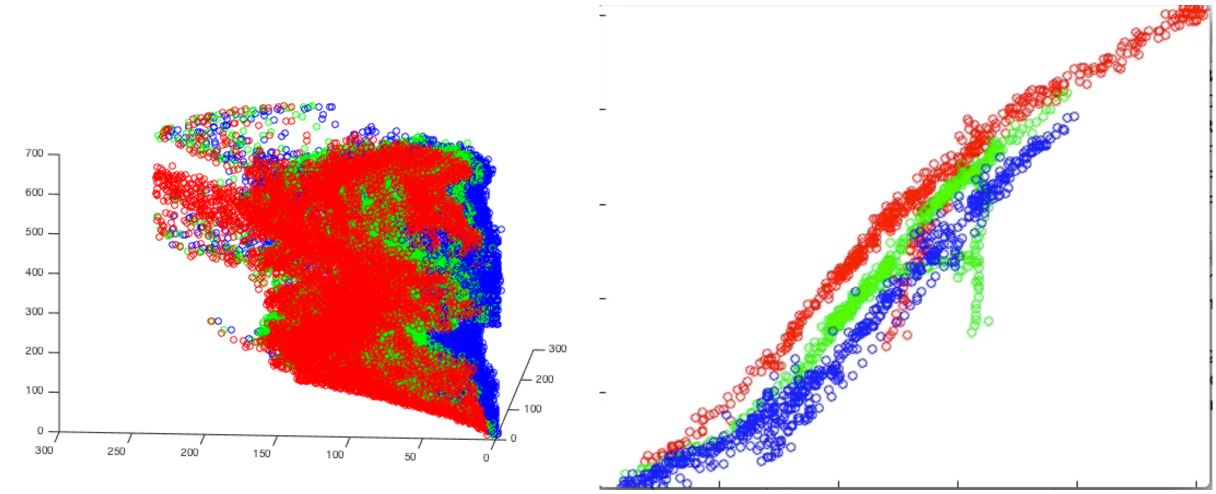
2.1 Test Set Generation

We began the analysis by creating a set of standardized test images on which to apply the "Hefe" filter. Each image is 640px x 640px, the standard dimensions of an Instagram photo. Most images were generated by a Python script, available in Appendix B.1, with the exception of the image `grad_bar` that was created by hand in Photoshop. Descriptions of each test image are available in Appendix A, and the images themselves are in the accompanying archive, 'Data Set.zip'.

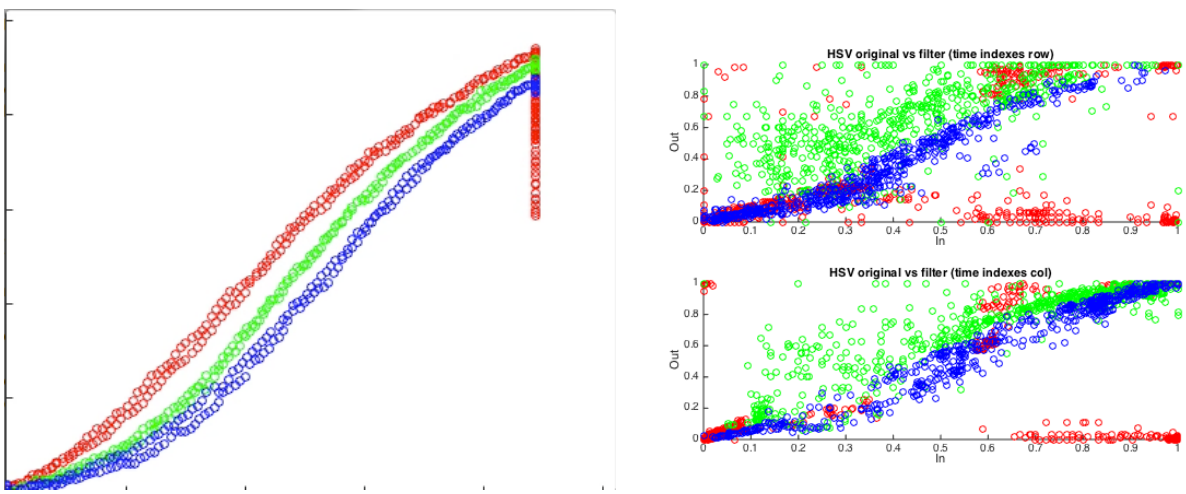
After generation, each image was uploaded to the test device, an Apple iPhone 4S, via Dropbox, then imported into the Instagram app and subjected to the filter. The transformed images were recovered using the online service "instaport.me". Each image was then renamed and converted to the .png format using the OS X terminal command `sips`.

2.2 Pre-Analysis

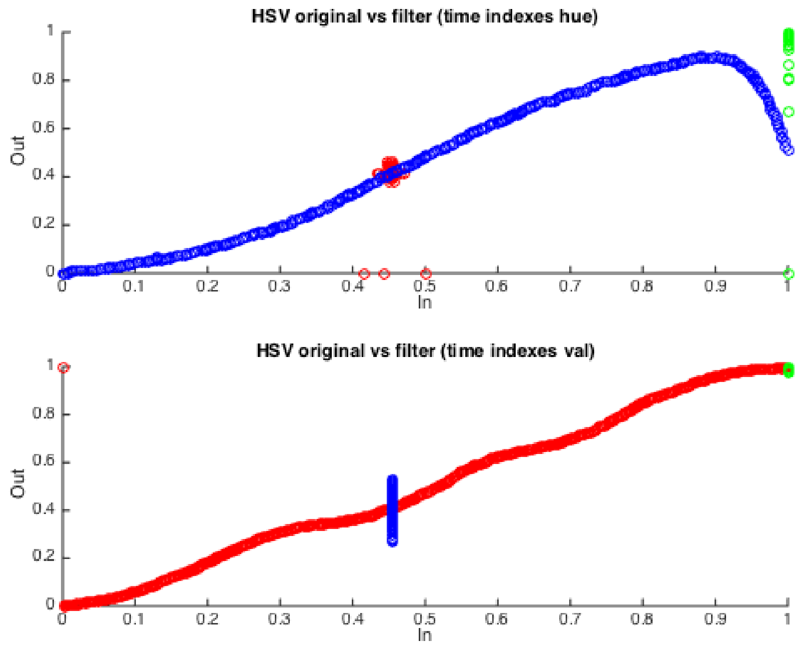
During the pre-analysis phase, we plotted the filter's transfer function by mapping the x-axis to the value of the unfiltered image and the y-axis to that of the filtered image, separated by channel in both RGB and HSV space. This provided us with a general feel for the types of transforms that "Hefe" was performing. Some select examples are reproduced below and a sample of the code used to generate these images can be found in Appendix B.2.



From left to right: RGB transfer function for `bird_0` in three dimensions, and a single row slice.



From left to right: RGB transfer function for a row of `hv_plane`, HSV transfer function for a row of `vegetables_0`.



HSV transfer function for a row and column of `hv_plane`.

From these images, it was apparent to us that the filter was applying some sort of tone mapping curves to the image's channels. The chaotic nature of the HSV-space plots in comparison to the more ordered RGB-space plots suggested that this transform was taking place in RGB-space. The V channel of the HSV plots, however, seemed most co-linear, which makes sense in this model as it is a linear combination of the RGB channels rather than a metric of their differences.

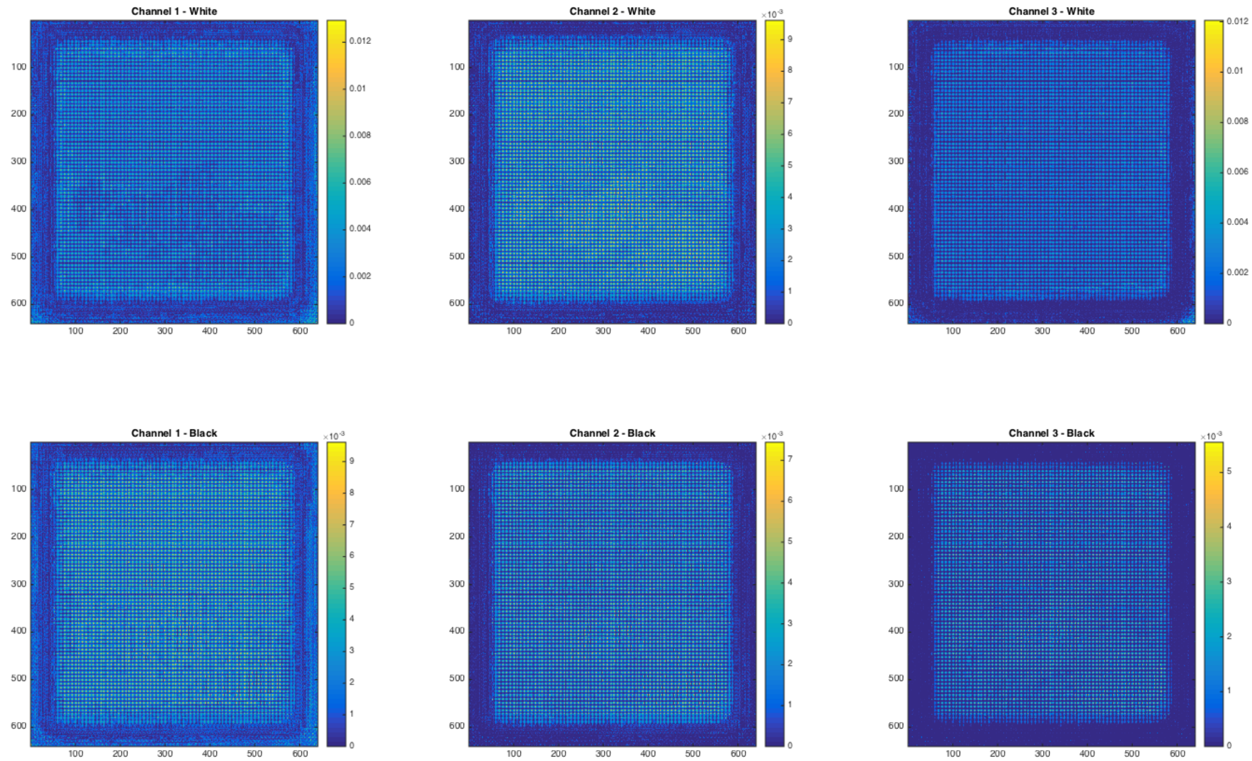
2.3 Experiment 1

2.3.1 Purpose and Procedures

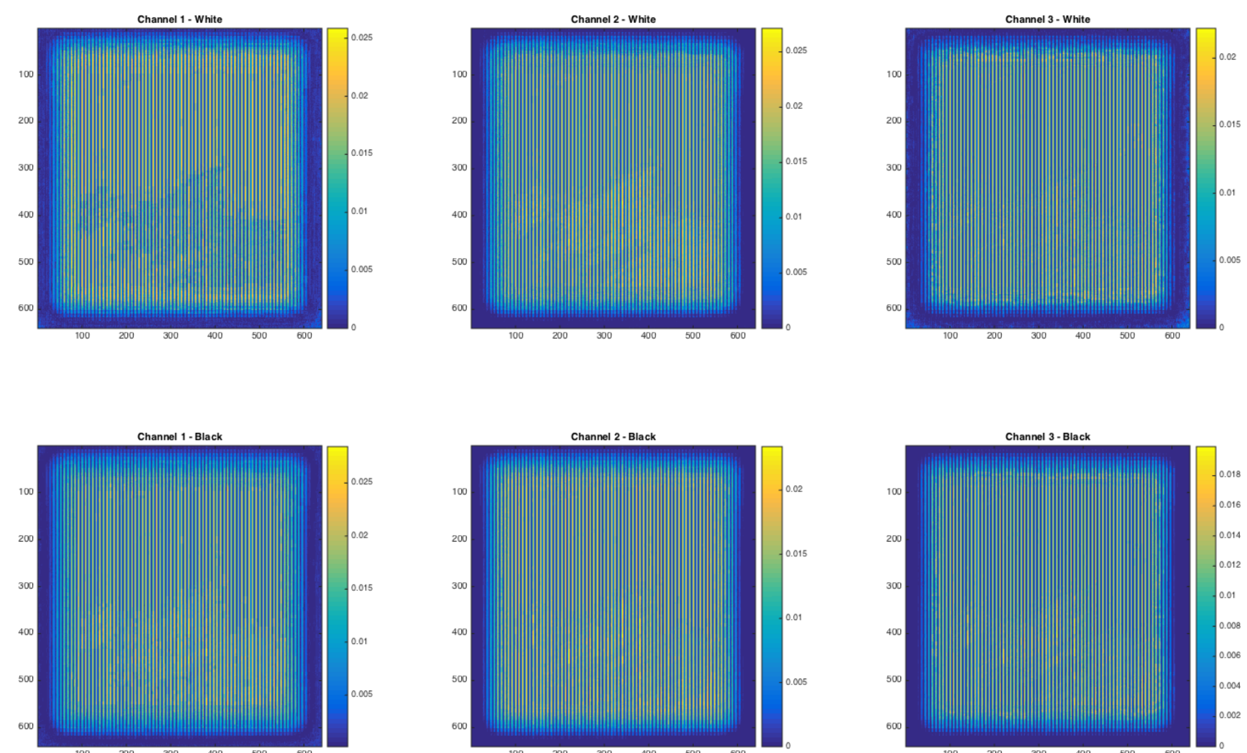
Experiment 1 was designed to test for the absence of blurring kernels, and by extension, inter-pixel independence. Verifying that each pixel in the filtered image had no dependence on the value of its neighbors allowed us to reduce the model's complexity by representing it as a function $f : \mathbb{R}^5 \to \mathbb{R}^3$.

In order to test this, we analyzed the test images `checker_bw_0`, `checker_bw_1`, `horiz_bw_stripe_0`, `horiz_bw_stripe_1`, `vert_bw_stripe_0`, `vert_bw_stripe_1`, `solid_000000`, and `solid_FFFFFF`. Complementary pairs of patterns were interleaved, and each pixel was compared to the solid color plane of the same value. Vis., the black pixels from each pattern were compared to the corresponding location in `solid_000000`, and the white pixels to `solid_FFFFFF`. Pixelwise squared-error terms were calculated and plotted along with the total error for each image set, seen in the next section. The code used to produce this data is available in the appendix.

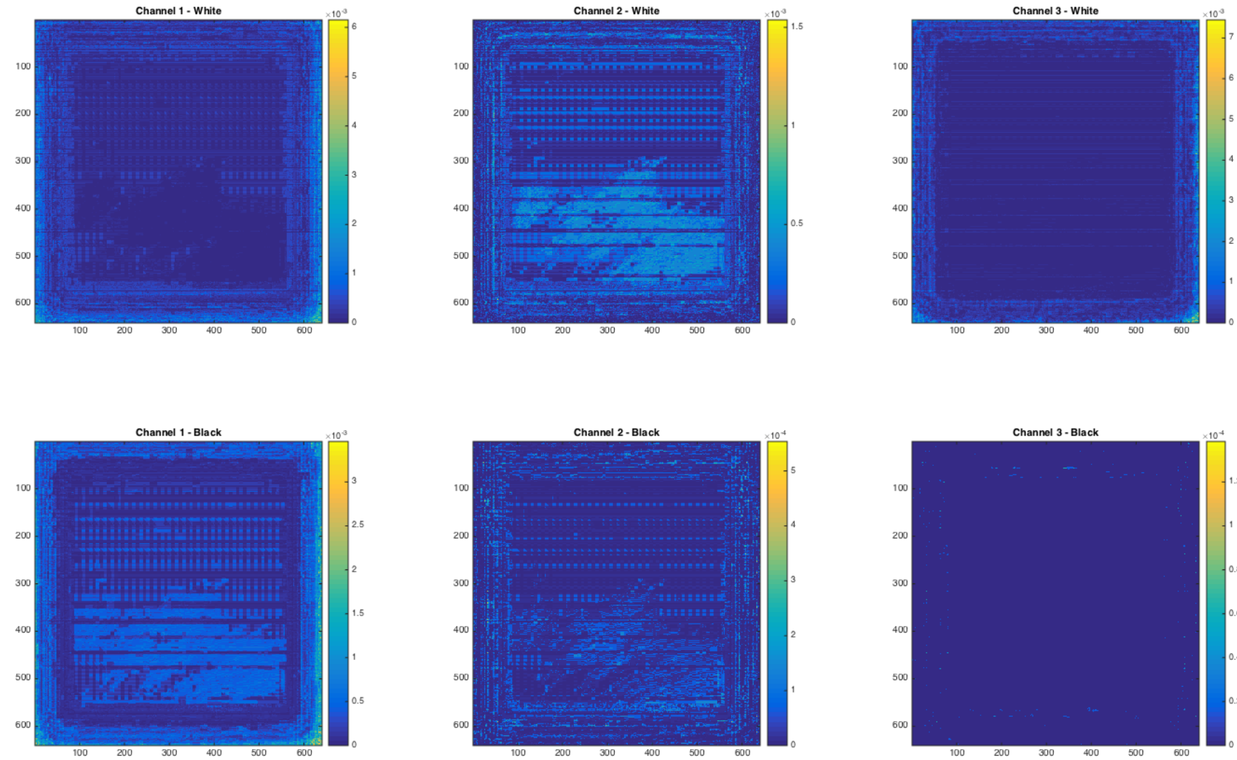
2.3.2 Data



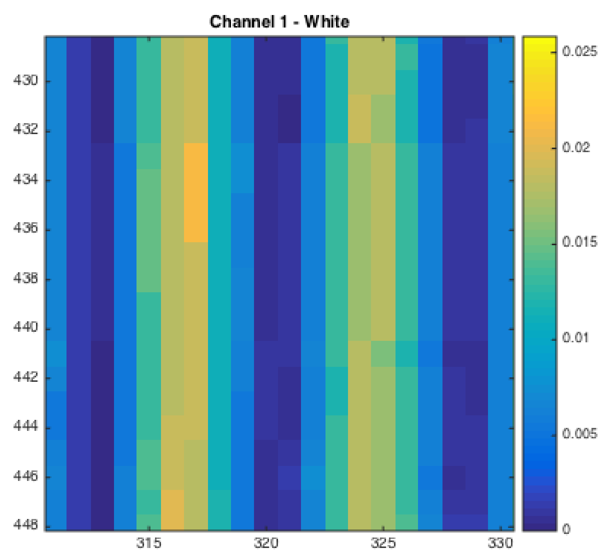
Square error terms for checker pattern. Scale is on the order of 10^{-3}.



Square error terms for vertical line pattern. Scale is on the order of 10^{-2}.



Square error terms for horizontal line pattern. Scale is on the order of 10^{-3}.



Detail of error terms for the vertical line pattern. Note the 8px width of error bands.

Image	Channel	Black		White	
		Mean	Variance	Mean	Maxiumum
Checkerboard	Red	0.001976	0.009612	0.001921	0.012933
	Green	0.001076	0.007443	0.001963	0.009612
	Blue	0.000509	0.005552	0.001132	0.012057
Vertical	Red	0.008586	0.028435	0.008540	0.025852
	Green	0.007130	0.023391	0.008247	0.027128
	Blue	0.005293	0.019931	0.006258	0.022207
Horizontal	Red	0.000253	0.003460	0.000274	0.006151
	Green	0.000020	0.000554	0.000131	0.001538
	Blue	0.000000	0.000138	0.000182	0.007443

Squared error terms for each image and channel (normalized to $[0, 1]$)

2.3.3 Analysis

At first glance, the error plots showed patterned differences that could indicate some sort of blurring kernel. However, on closer inspection, the pattern was found to be regularly spaced on an 8px x 8px grid, the same dimensions as a JPEG macroblock. The sinusoidal nature of the error was also highly reminiscent of the low-order DCT coefficients used by JPEG. This suggested that the error pattern was an artifact of the compression applied by Instagram, rather than the filter itself.

Even with these compression artifacts, the table still shows very low mean squared error, on the order of 10^{-3} for most test cases. With this in mind, we came to the conclusion that each pixel of the filter's output is independent of its neighbors, and can be reduced to a function of only pixel color and position, $(r', g', b') = f(r, g, b, x, y)$. This simplification vastly reduced the analysis needed to recreate the filter effect.

2.4 Experiment 2

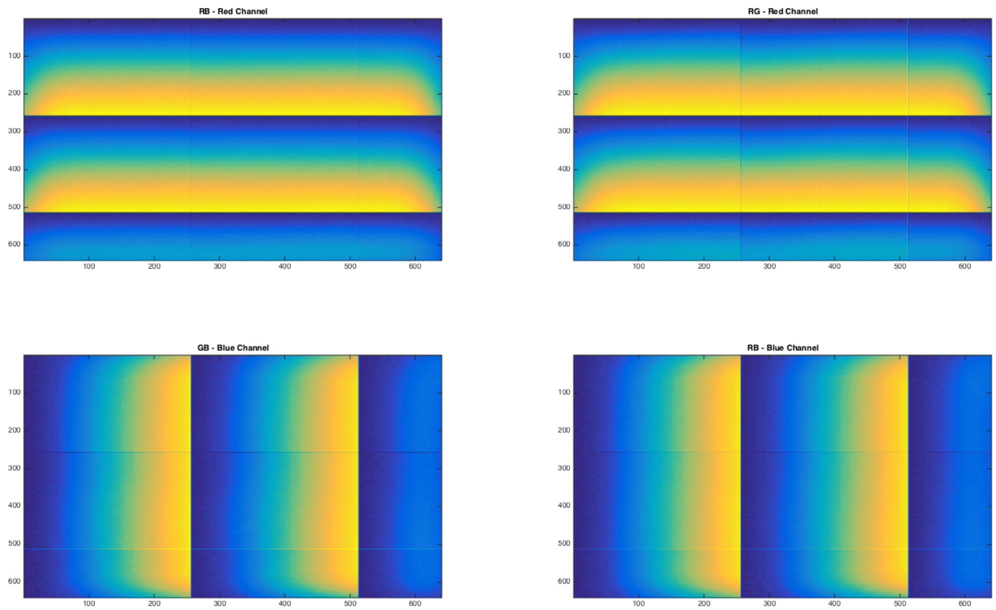
2.4.1 Purpose and Procedures

Based on the graphs produced during the pre-analysis, we hypothesized that the filter operated in *RGB*-space with independent transformations per channel. Experiment 2 was designed to confirm that the filter had per-channel independence rather than operating on a linear combination of channels.

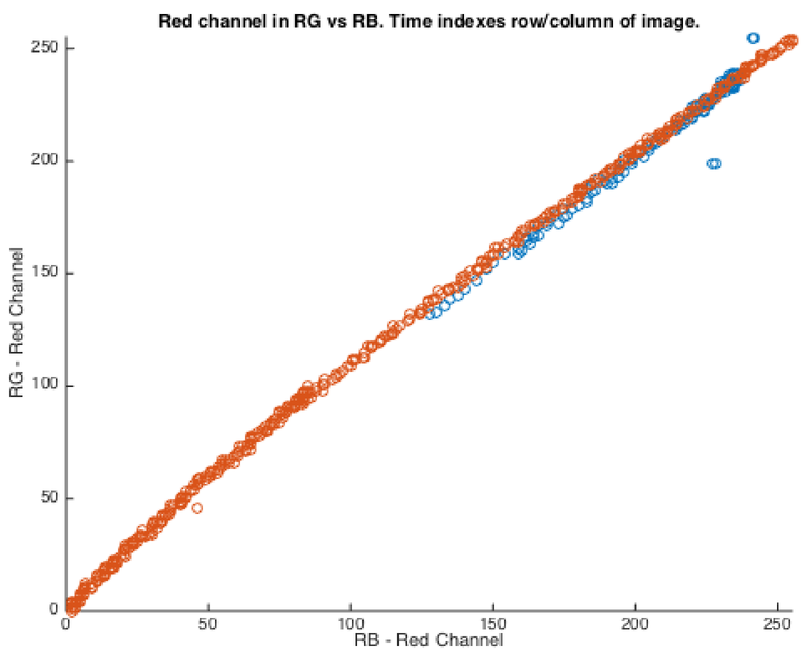
To accomplish this, we plotted the R channels of `rg_plane` and `rb_plane` and the B channels of `gb_plane` and `rb_plane`. These were visually inspected for uniformity across variances in the other channels. In addition, We also used the transfer band plotting code from the Pre-Analysis phase to compare the R channels of `rg_plane` and `rb_plane` as the row and column (and, by extension, the G and B channels) varied. The resultant graph was inspected for linearity.

We also applied the filter to each *RGB* primary, and to solid white and black. For each filtered primary, we verified that each channel was equivalent to the corresponding channel of either the solid white or black filtered image. For example, when testing `solid_00FF00`, we compared the R and B channels to the respective channels in `solid_000000` and the G channel to its respective channel in `solid_FFFFFF`. As in the previous experiment, we plotted the pixelwise squared-error term and outputted the mean of each primary and channel's error, available in the next section.

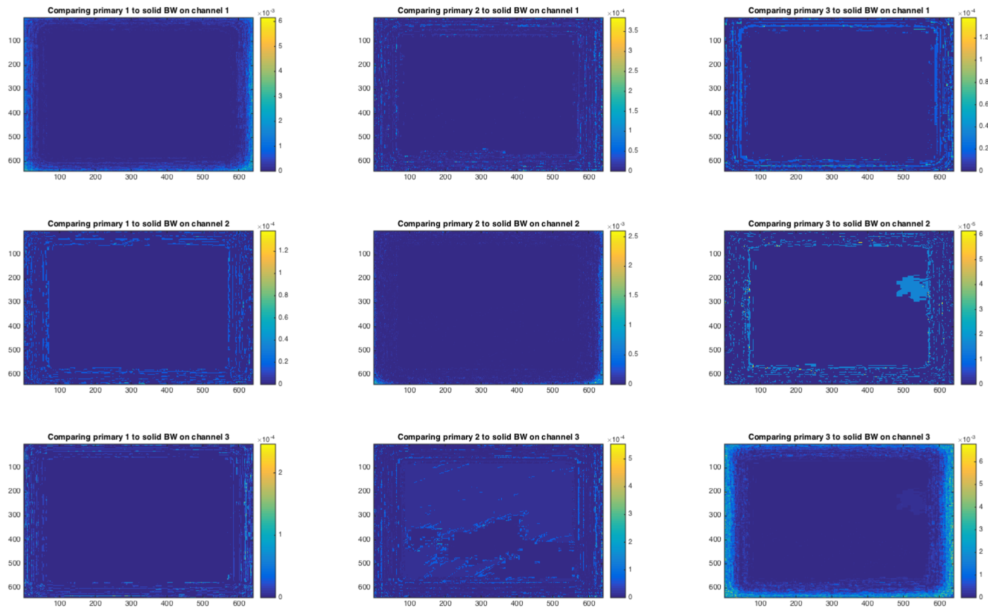
2.4.2 Data



Comparison of the R and B channels along primary gradients.



A representative frame of a video showing the plot of R channels in `rg_plane` and `rb_plane` as the B/G values vary. Orange varies row over time, blue varies column.



Square error terms for per-channel primary comparisons. Scale is on the order of 10^{-4}.

MSE	Primary		
Channel	Red	Green	Blue
Red	0.000118	0.000004	0.000002
Green	0.000001	0.000023	0.000001
Blue	0.000002	0.000013	0.000315

Mean squared-error comparing channels of primary planes

2.4.3 Analysis

The comparison of the R and B channels along primary gradients seemed relatively uniform with the exception of the borders where the secondary channel changed from 1 to 0. At first, we though that this could be evidence of some inter-channel dependence. However, we came to the conclusion that this discontinuity was a simply a ringing effect caused by the JPEG compression applied by Instagram. This is supported by the fact that JPEG encodes overall luminance as a separate channel, which would be affected by changes in all three RGB channels. Our suspicions were confirmed upon noticing that the border on the on the RG plane was about 5 times more pronounced than that of the RB plane; the JPEG algorithm first converts images to $Y'CbCr$ space, in which the luminance channel encodes green with approximately 5 times more weight than blue.

The transfer band plot supported the hypothesis that the filter acts independently on each channel as well. As the green and blue values changed, the red points remained on a uniform line, indicating that no modification to the red value was made beyond the filter's normal transform.

Lastly, the square-error plots for per-channel primary comparisons were remarkably close to zero, confirming that the filter does in fact act separately on each channel. This conclusion allowed us to further simplify the filter model. The previous model, $(r', g', b') = f(r, g, b, x, y)$, could now be reduced to a set of three independent functions: $(r', g', b') = (f_r(r, x, y), f_g(g, x, y), f_b(b, x, y))$. This reduction spared us from computing any linear combinations or inter-channel effects and allowed us to analyze each RGB channel separately.

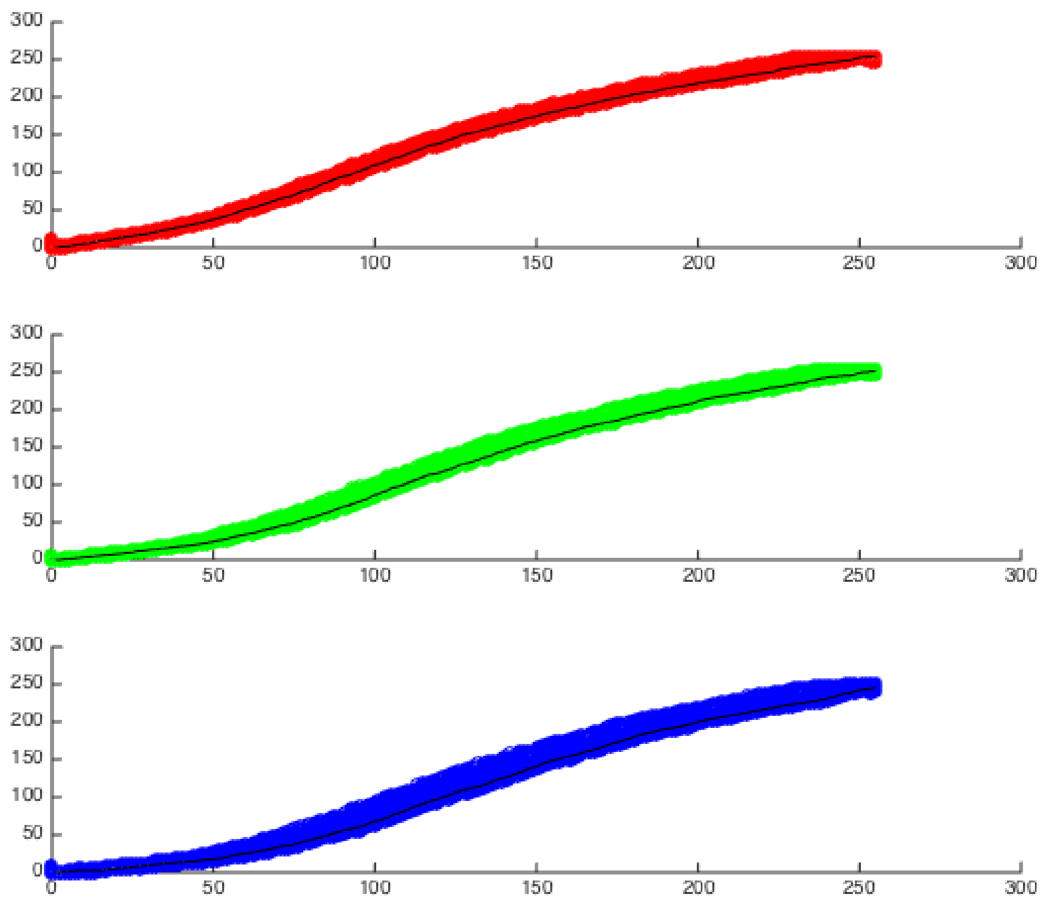
2.5 Experiment 3

2.5.1 Purpose and Procedures

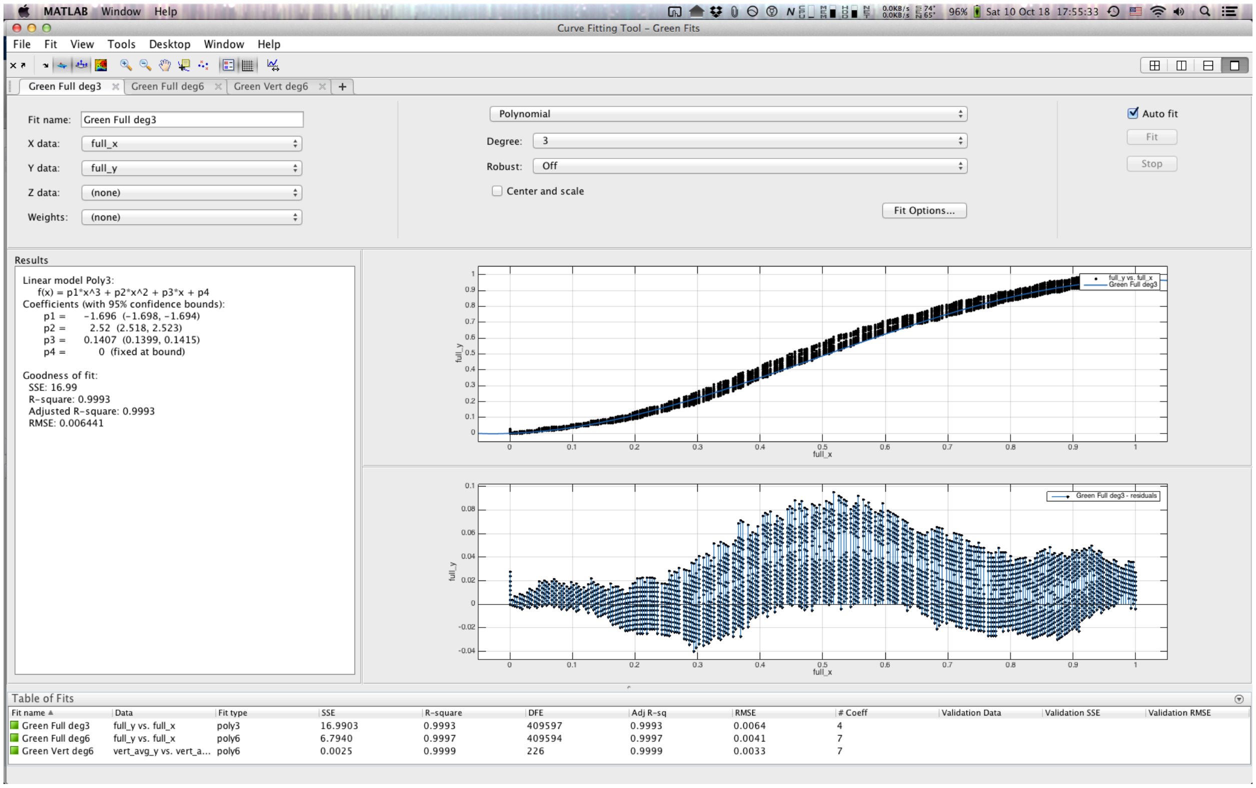
Experiment 3 was designed to use polynomial regression to model each of the functions $f_r(r, x, y)$, $f_g(b, x, y)$, and $f_b(b, x, y)$. Since Experiment 2 confirmed that each channel was processed independently, we were able to perform each regression concurrently using a single data set: `grad_bar`. Because the filter is position-dependent only at the edges (within the vignetted area), the greyscale gradients in this test image were placed only toward the center of the image. This allowed us to form a more generalized model that excluded vignetting and could be applied to each pixel uniformly. The data points were formed by assigning the value of each channel in the unfiltered image to the x-coördinate and that of the filtered image to the y-coördinate. These points were then run through MATLAB's Curve Fitting Toolbox and regression coefficients were recorded.

During this experiment, we calculated three polynomials per channel: one of degree 3, one of degree six, and one of degree six performed on a smoothed data set. The data set was smoothed by taking the mean of observed y-values for each x-value in the data set, accounting for slight spatial variations created by the vignette. This was done to create a smoother and more generalized output polynomial.

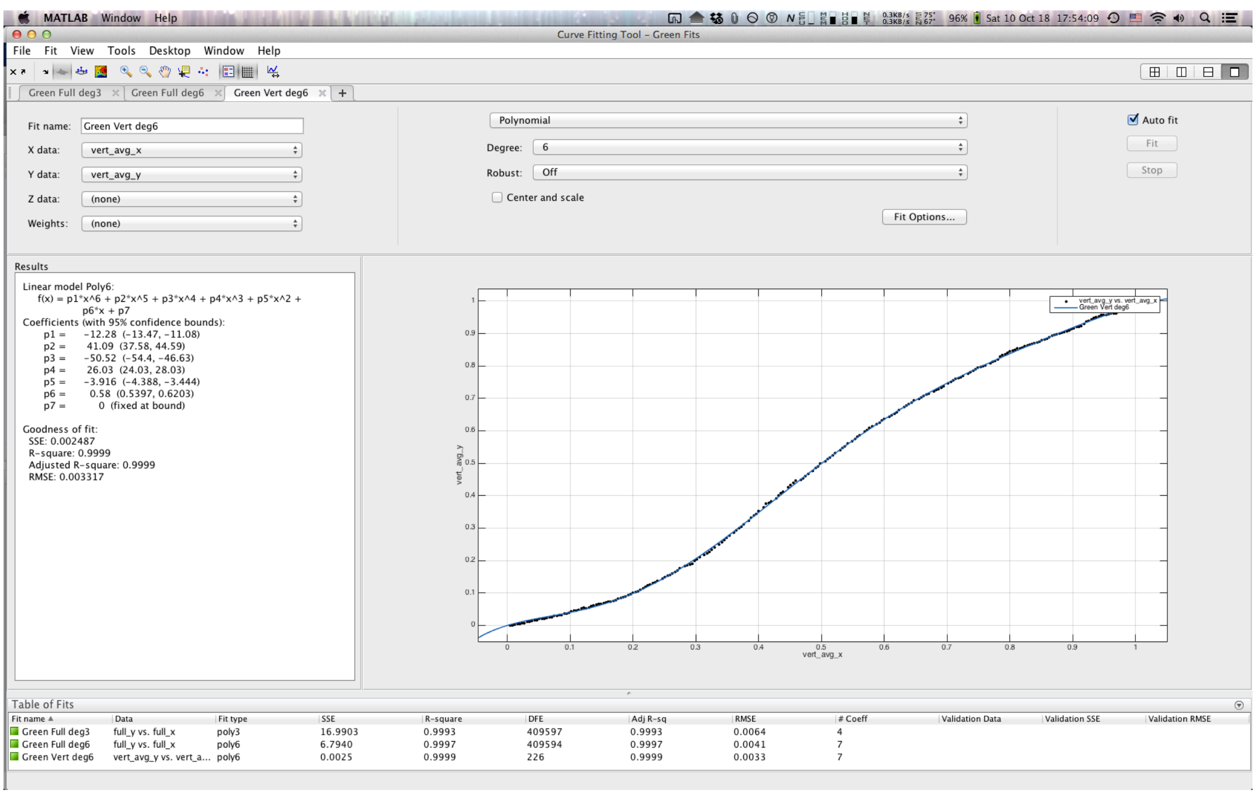
2.5.2 Data



Data points to be fit for each channel. Smoothed version overlaid in black.



Polynomial regression being performed on the green channel of the full data set.



Polynomial regression being performed on the green channel of the smoothed data set.

Model	Channel	Regression Polynomial
Full, deg 3	Red	$-1.389r^3 + 1.768r^2 + 0.5899r$
	Green	$-1.696g^3 + 2.52g^2 + 0.1407g$
	Blue	$-1.698b^3 + 2.731b^2 + 0.09003b$
Full, deg 6	Red	$-13.77r^6 + 42.05r^5 - 45.83r^4 + 19.54r^3 - 1.627r^2 + 0.6362r$
	Green	$-12.6g^6 + 41.85g^5 - 51.04g^4 + 26.07g^3 - 3.878g^2 + 0.5905g$
	Blue	$-1.263b^6 + 10.28b^5 - 19.66b^4 + 13.09b^3 - 1.831b^2 + 0.3575b$
Smoothed, deg 6	Red	$-13.47r^6 + 41.23r^5 - 45.04r^4 + 19.17r^3 - 1.492r^2 + 0.5954r$
	Green	$-12.28g^6 + 41.09g^5 - 50.52g^4 + 26.03g^3 - 3.916g^2 + 0.58g$
	Blue	$-1.066b^6 + 9.679b^5 - 19.09b^4 + 12.92b^3 - 1.835b^2 + 0.3487b$

Polynomial models generated by the Curve Fitting Toolbox. Note that the constant term of each polynomial was artificially restricted to 0, as the filter had already confirmed not to affect black pixels.

2.5.3 Analysis

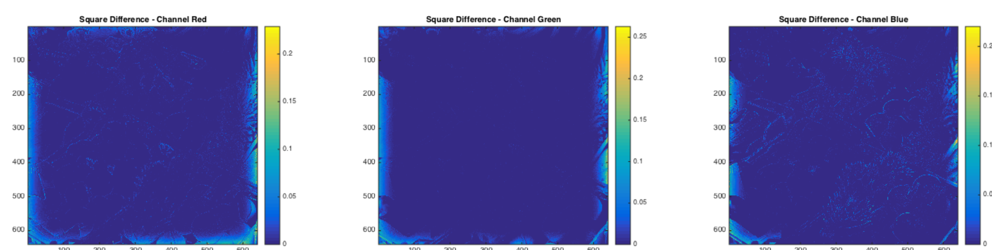
Each polynomial regression, particularly those of degree 6, using both smoothed and full data, showed strong correlation to the test set, with R values in excess of 0.999. Since no extrapolation would be done using these models, we were not concerned with any possible overfitting, and this was not taken into consideration. The high degree of correlation attained leads us to believe that Instagram uses a polynomial tone mapping function internally, though the actual mechanics used have no impact on the filter's recreation, as the effects were replicated almost exactly using the above polynomials.

2.6 Experiment 4

2.6.1 Purpose and Procedures

Experiment 4 tested the accuracy of the three polynomial models by applying them to various test images and comparing the result to the results of the actual filter. For each test image and model, the spatial squared-error was plotted and its mean recorded with the intent of determining the best-fitting polynomial coefficients.

2.6.2 Data



Squared-error terms for the degree 6 smoothed data set polynomial model applied to `vegetables_0`. Scale is from $[0, 0.25]$.

Image	Model	MSE	SSE
`vegetables_0`	Full, deg 3	0.004826	5930.363931
	Full, deg 6	0.004924	6049.998903
	Smoothed, deg 6	0.004693	5766.639317
`landscape_0`	Full, deg 3	0.006739	8281.192691
	Full, deg 6	0.007050	8663.541021
	Smoothed, deg 6	0.006612	8125.435572
`grad_bar`	Full, deg 3	0.000050	61.160565
	Full, deg 6	0.000020	24.781046
	Smoothed, deg 6	0.000025	30.632892

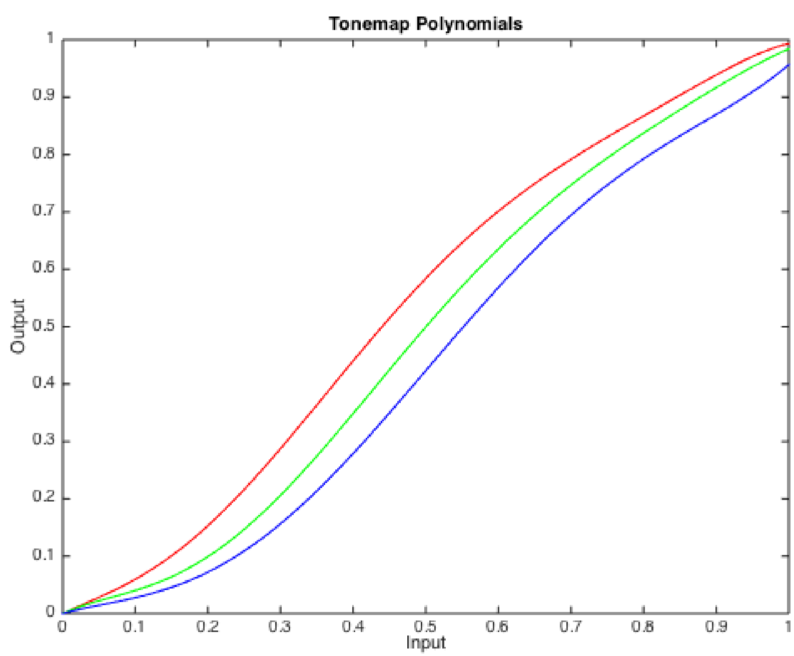
Mean and sum squared-error for various polynomial models applied to three test images.

2.6.3 Analysis

Of the three regressions produced in the previous experiment, the six-degree polynomial generated from the smoothed data set demonstrated the highest accuracy. Looking at the spatial error analysis, the majority of the error was concentrated at the edges of the input image. This was expected and is of no concern, as the models tested made no attempt to take vignetting into account.

After the most accurate polynomial was selected, an inverse function was generated, again running the Curve Fitting Toolbox after interposing the x and y data sets, to produce the following final tone map polynomials:

Forward	Red	$-13.47r^6 + 41.23r^5 - 45.04r^4 + 19.17r^3 - 1.492r^2 + 0.5954r$
	Green	$-12.28g^6 + 41.09g^5 - 50.52g^4 + 26.03g^3 - 3.916g^2 + 0.58g$
	Blue	$-1.066b^6 + 9.679b^5 - 19.09b^4 + 12.92b^3 - 1.835b^2 + 0.3487b$
Inverse	Red	$-3.835r^6 + 12.81r^5 - 17.35r^4 + 13.18r^3 - 5.738r^2 + 1.939r$
	Green	$-8.176g^6 + 28.36g^5 - 39.21g^4 + 28.23g^3 - 11.04g^2 + 2.845g$
	Blue	$-27.59b^6 + 85.64b^5 - 103.7b^4 + 62.86b^3 - 20.07b^2 + 3.905b$



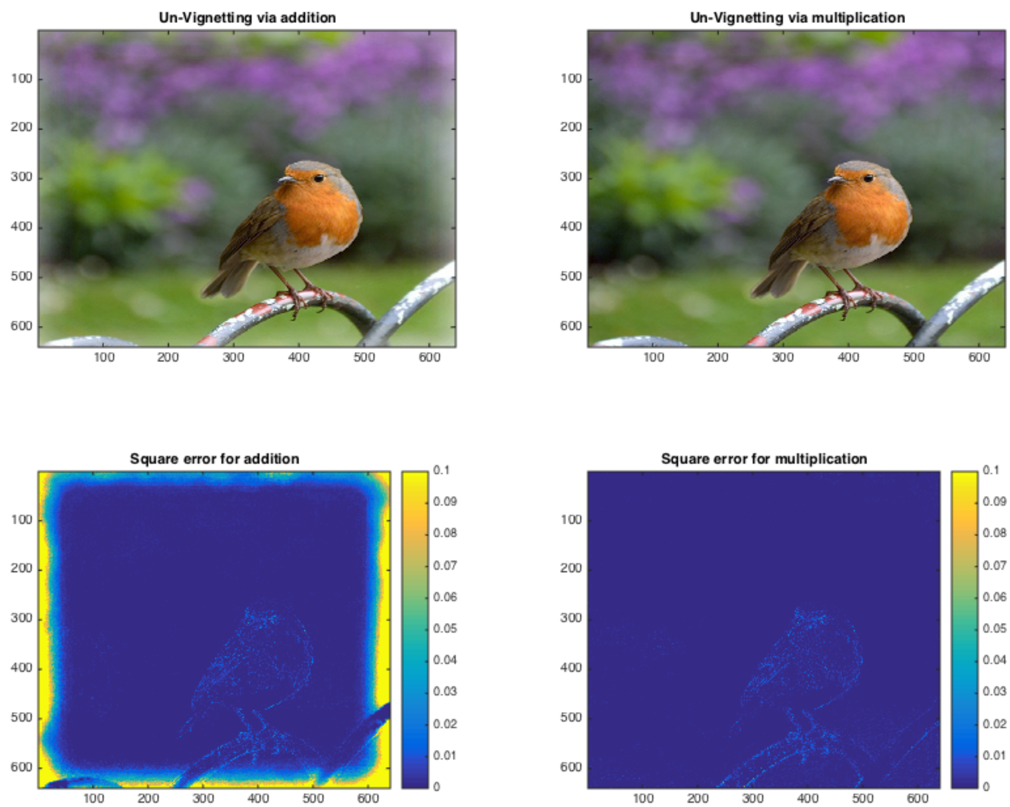
Plot of the forward tone map function for each channel.

2.7 Experiment 5

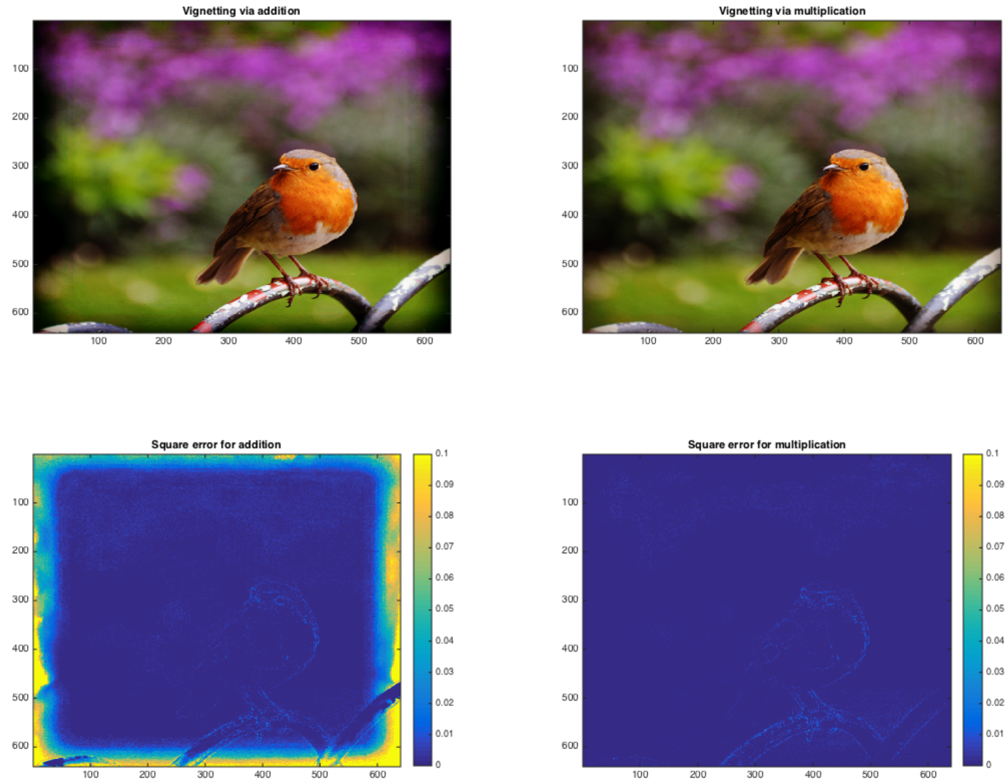
2.7.1 Purpose and Procedures

Experiment 5 proposed two potential models for applying “Hefe”’s vignetting effect: addition of a value from $[-1, 0]$, and multiplication by a value from $[0, 1]$. Each method was tested both forwards and backwards, applying the proposed effect to the original image and testing against the corresponding filtered image, and inverting filtered image for comparison with the original image. The tone mapping polynomials from the previous experiment were used to isolate the vignetting effect of `solid_FFFFFF`, and the resultant data was applied to several test images, once again recording squared-error terms both spatially and cumulatively.

2.7.2 Data



Comparison of backward additive and multiplicative application of vignette data to `bird_0`. Scale is from $[0, 0.1]$.



Comparison of forward additive and multiplicative application of vignette data to `bird_0`. Scale is from $[0, 0.1]$.

Image	Additive MSE	Multiplicative
`bird_0`	0.016177	0.000750
`landscape_0`	0.053105	0.002822
`hv_plane`	0.880683	0.000577

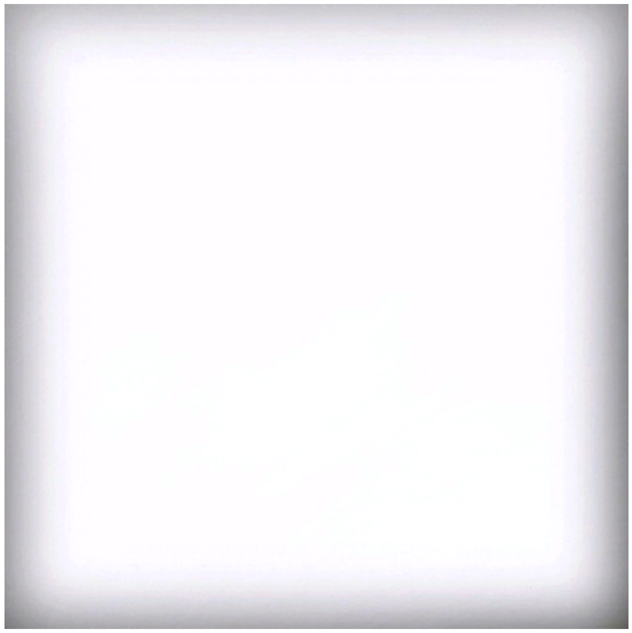
Forward Tests

Image	Additive MSE	Multiplicative MSE
`bird_0`	0.015329	0.000757

Backward Tests

2.7.3 Analysis

Both qualitatively and quantitatively, it was apparent that the vignette has been applied using a multiplicative blending mode. The vignette data, extracted from `solid_FFFFFF` by applying the inverse tone map function, was saved as an image, `vignette.png`, for use in the final filter recreation, and is reproduced below:



Vignette mask, to be applied multiplicatively.

3 Conclusion

Over the course of these five experiments, the Instagram filter "Hefe" was reduced to a set of four functions:

$$\begin{aligned}(r', g', b') &= (f_r(r * v_r(x,y)), f_g(g * v_g(x,y)), f_b(b * v_b(x,y))) \\ f_r(r) &= -13.47r^6 + 41.23r^5 - 45.04r^4 + 19.17r^3 - 1.492r^2 + 0.5954r \\ f_g(g) &= -12.28g^6 + 41.09g^5 - 50.52g^4 + 26.03g^3 - 3.916g^2 + 0.58g \\ f_b(b) &= -1.066b^6 + 9.679b^5 - 19.09b^4 + 12.92b^3 - 1.835b^2 + 0.3487b\end{aligned}$$

where $v : \mathbb{N}^2 \to \mathbb{R}^3$ takes position inputs in $[0, 640]$ corresponding to the position of the pixel and outputs the value of `vignette.png` at that pixel, in the range $[0, 1]$.

3.1 Implementation

MATLAB functions to apply or inverse the "Hefe" filter are as follows:

```
function [ image_out ] = apply_hefe( image_in )
% Applies the Instagram filter 'Hefe' to an image
%   Expects and returns a 640x640x3 double array in the range [0, 1]

tonemap_coeff = ...
    [-13.47 41.23 -45.04 19.17 -1.492 0.5954 0.0; ...
    -12.28 41.09 -50.52 26.03 -3.916 0.58 0.0; ...
    -1.066 9.679 -19.09 12.92 -1.835 0.3487 0.0];

vignette = double(imread('vignette.png'))/255;

result = image_in .* vignette;
for channel = 1:3
    result(:,:,channel) = polyval(tonemap_coeff(channel, :), result(:,:,channel));
end
image_out = result;
end
```

```
function [ image_out ] = reverse_hefe( image_in )
% Inverses the Instagram filter 'Hefe' on an image
%   Expects and returns a 640x640x3 double array in the range [0, 1]

inverse_tonemap_coeff = ...
    [-3.835 12.81 -17.35 13.18 -5.738 1.939 0.0; ...
    -8.176 28.36 -39.21 28.23 -11.04 2.845 0.0; ...
    -27.59 85.64 -103.7 62.86 -20.07 3.905 0.0];

vignette = double(imread('vignette.png'))/255;

result = zeros(size(image_in));
for channel = 1:3
    result(:,:,channel) = polyval(inverse_tonemap_coeff(channel, :), image_in(:,:,channel));
end

image_out = result ./ vignette;

end
```

3.2 Example

The recreated filter applied to `bird_0`. First row, from left to right: original image, filtered image with inverse filter applied, spatial sum-square plot, scale from $[0, 0.07]$. Second row, from left to right: filtered image, original image with recreated filter applied, spatial sum-square plot, scale from $[0, 0.045]$.

From the spatial square-error plots, the majority of error was concentrated around sharp edges. This is most likely due to the JPEG compression that Instagram applied, which was not replicated by this recreation. This difference is not significant nor visible in the final images.

A Test Images and Data Set

All images can be viewed in the accompanying archive, 'Data Set.zip'. Descriptions of the images used during testing are provided below.

Name	Description
bird_0 vegetables_0 landscape_0	Ordinary photographic images of outdoor scenes used to simulate real-world applications of the filter.
checker_bw_0 checker_bw_1	Checkerboard patterns formed by alternating every other pixel white or black. The two images are inverse of each other.
rg_plane gb_plane rb_plane	Color planes made by mapping two RGB channels at a time to spatial position.
horiz_bw_stripe_0 horiz_bw_stripe_1 vert_bw_stripe_0 vert_bw_stripe_1	Parallel stripe patterns made by alternating black and white every other row or column. '0' and '1' pairs are inverses of each other.
horiz_grey vert_grey	Horizontal and vertical greyscale gradients spanning the full width and height.
hs_plane sv_plane hv_plane	Color planes made by mapping two HSV channels at a time to spatial position.
solid_000000 solid_0000FF solid_00FF00 solid_FF0000 solid_7F7F7F solid_FFFFFF	Solid images composed of a single color, specified in hex.
grad_bar	Several greyscale gradients in various orientations, positioned to avoid the majority of vignetting effects.

B Code

B.1 Test Set Generation

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np
from skimage import color
import skimage.filter as filters
from skimage import img_as_float, img_as_ubyte

imgsize = (640, 640)

print 'Generating...'

test_images = dict()
print 'solid colors'
test_images['solid_FFFFFF'] = np.ones((imgsize[0], imgsize[1], 3))
test_images['solid_000000'] = np.zeros((imgsize[0], imgsize[1], 3))
test_images['solid_FF0000'] = np.ones((imgsize[0], imgsize[1], 3)) * [1.0, 0.0, 0.0]
test_images['solid_00FF00'] = np.ones((imgsize[0], imgsize[1], 3)) * [0.0, 1.0, 0.0]
test_images['solid_0000FF'] = np.ones((imgsize[0], imgsize[1], 3)) * [0.0, 0.0, 1.0]
test_images['solid_7F7F7F'] = np.ones((imgsize[0], imgsize[1], 3)) * 0.5
```

```
print 'stripes'
test_images['horiz_bw_stripe_0'] = np.ones((imgsize[0], imgsize[1], 3))
test_images['horiz_bw_stripe_1'] = np.zeros((imgsize[0], imgsize[1], 3))
for x in range(0, imgsize[0], 2):
    test_images['horiz_bw_stripe_0'][x,:,:] = 0.0
    test_images['horiz_bw_stripe_1'][x,:,:] = 1.0
test_images['vert_bw_stripe_0'] = np.ones((imgsize[0], imgsize[1], 3))
test_images['vert_bw_stripe_1'] = np.zeros((imgsize[0], imgsize[1], 3))
for y in range(0,imgsize[1], 2):
    test_images['vert_bw_stripe_0'][:,y,:] = 0.0
    test_images['vert_bw_stripe_1'][:,y,:] = 1.0
print 'grids'
grid = np.ogrid[0:imgsize[0], 0:imgsize[1]]
test_images['checker_bw_0'] = color.gray2rgb(((grid[0] + grid[1]) % 2) == 0)
test_images['checker_bw_1'] = color.gray2rgb(((grid[0] + grid[1]) % 2) != 0)
print 'planes (allocate)'
test_images['rg_plane'] = np.ones((imgsize[0], imgsize[1], 3), dtype=np.uint8)
test_images['gb_plane'] = np.ones((imgsize[0], imgsize[1], 3), dtype=np.uint8)
test_images['rb_plane'] = np.ones((imgsize[0], imgsize[1], 3), dtype=np.uint8)
test_images['hs_plane'] = np.ones((imgsize[0], imgsize[1], 3), dtype=np.float)
test_images['sv_plane'] = np.ones((imgsize[0], imgsize[1], 3), dtype=np.float)
test_images['hv_plane'] = np.ones((imgsize[0], imgsize[1], 3), dtype=np.float)
test_images['horiz_grey'] = np.ones((imgsize[0], imgsize[1]), dtype=np.float)
test_images['vert_grey'] = np.ones((imgsize[0], imgsize[1]), dtype=np.float)
print 'planes (specify)'
for x in range(0, imgsize[0]):
    for y in range(0, imgsize[1]):
        test_images['rg_plane'][x,y,:] *= [x % 256, y % 256, 0]
        test_images['gb_plane'][x,y,:] *= [0, x % 256, y % 256]
        test_images['rb_plane'][x,y,:] *= [x % 256, 0, y % 256]
        test_images['hs_plane'][x,y,:] *= [x / float(imgsize[0]), y / float(imgsize[1]), 1.0]
        test_images['sv_plane'][x,y,:] *= [0.0, x / float(imgsize[0]), y / float(imgsize[1])]
        test_images['hv_plane'][x,y,:] *= [x / float(imgsize[0]), 1.0, y / float(imgsize[1])]
        test_images['vert_grey'][x,y] = x / float(imgsize[0])
        test_images['horiz_grey'][x,y] = y / float(imgsize[1])
print 'planes (convert)'
test_images['hs_plane'] = color.hsv2rgb(test_images['hs_plane'])
test_images['sv_plane'] = color.hsv2rgb(test_images['sv_plane'])
test_images['hv_plane'] = color.hsv2rgb(test_images['hv_plane'])
test_images['vert_grey'] = color.gray2rgb(test_images['vert_grey'])
test_images['horiz_grey'] = color.gray2rgb(test_images['horiz_grey'])

print 'Saving...'
for name, img in test_images.items():
    plt.imsave(name + '.png', img_as_ubyte(img))

print 'Done'
```

B.2 Pre-Analysis

```

% This implementation was manually changed several times to generate the
% video set; below is a typical implementation.

figure(1);

rg_plane_filter = imread('Filter Data Set/hv_plane_filter.png');
rb_plane_filter = imread('Test Images/hv_plane.png');

for i = 1:640

    scatter(rb_plane_filter(i, :, 1)',rg_plane_filter(i, :, 1)','r');
    hold on;
    scatter(rb_plane_filter(i, :, 2)',rg_plane_filter(i, :, 2)','g');
    scatter(rb_plane_filter(i, :, 3)',rg_plane_filter(i, :, 3)','b');
```

```
    axis([0 255 0 255]);
    title('RGB original vs filter (time indexes hue)');
    xlabel('In');
    ylabel('Out');
    hold off;

    M(i) = getframe;
end
```

B.3 Experiment 1

```matlab
% Original checker images
checker_bw_1_rgb = imread('Test Images/checker_bw_1.png');
checker_bw_0_rgb = imread('Test Images/checker_bw_0.png');

% Filtered checkers
checker_bw_1_filter_rgb = imread('Filter Data Set/checker_bw_1_filter.png');
checker_bw_0_filter_rgb = imread('Filter Data Set/checker_bw_0_filter.png');

% Filtered solids
solid_FFFFFF_filter_rgb = imread('Filter Data Set/solid_FFFFFF_filter.png');
solid_000000_filter_rgb = imread('Filter Data Set/solid_000000_filter.png');

% Extract a channel and convert to float
for channel = 1:3
    checker_bw_1 = double(checker_bw_1_rgb(:,:,channel)) / 255;
    checker_bw_0 = double(checker_bw_0_rgb(:,:,channel)) / 255;
    checker_bw_1_filter = double(checker_bw_1_filter_rgb(:,:,channel)) / 255;
    checker_bw_0_filter = double(checker_bw_0_filter_rgb(:,:,channel)) / 255;
    solid_FFFFFF_filter = double(solid_FFFFFF_filter_rgb(:,:,channel)) / 255;
    solid_000000_filter = double(solid_000000_filter_rgb(:,:,channel)) / 255;

    % First split the pixels
    black_index_1 = find(checker_bw_1 < 0.5);
    white_index_1 = find(checker_bw_1 ≥ 0.5);
    black_index_0 = find(checker_bw_0 < 0.5);
    white_index_0 = find(checker_bw_0 ≥ 0.5);

    % interleave the black and white pixels from the two checkerboards
    black_reconstruct = zeros(size(solid_000000_filter));
    white_reconstruct = zeros(size(solid_FFFFFF_filter));
    black_reconstruct(black_index_1) = checker_bw_1_filter(black_index_1);
    black_reconstruct(black_index_0) = checker_bw_0_filter(black_index_0);
    white_reconstruct(white_index_1) = checker_bw_1_filter(white_index_1);
    white_reconstruct(white_index_0) = checker_bw_0_filter(white_index_0);

    % Take the squared difference of the interleaved and solid image
    diff_white = (white_reconstruct - solid_FFFFFF_filter) .^ 2;
    diff_black = (black_reconstruct - solid_000000_filter) .^ 2;

    display(sprintf('Examining channel: %d', channel))

    display(sprintf('Black Pixels - Mean: %f  Var: %f', ...
        mean(black_reconstruct(:)), var(black_reconstruct(:))))
    display(sprintf('White Pixels - Mean: %f  Var: %f', ...
        mean(white_reconstruct(:)), var(white_reconstruct(:))))

    display(sprintf('Square error white - Mean: %f  Max: %f', mean(diff_white(:)), ...
        max(diff_white(:))))
    display(sprintf('Square error black - Mean: %f  Max: %f', mean(diff_black(:)), ...
        max(diff_black(:))))
    display(' ');

    subplot(2, 3, channel);
    imagesc(diff_white);
    title(sprintf('Channel %d - White', channel));
    colorbar;
    subplot(2, 3, channel + 3);
    imagesc(diff_black);
    title(sprintf('Channel %d - Black', channel));
    colorbar;
end
```

B.4 Experiment 2

```
% Filtered solids
solid_FFFFFF_filter = imread('Filter Data Set/solid_FFFFFF_filter.png');
solid_000000_filter = imread('Filter Data Set/solid_000000_filter.png');
solid_0000FF_filter = imread('Filter Data Set/solid_0000FF_filter.png');
solid_00FF00_filter = imread('Filter Data Set/solid_00FF00_filter.png');
solid_FF0000_filter = imread('Filter Data Set/solid_FF0000_filter.png');

% Primary planes, original and filtered
gb_plane = imread('Test Images/gb_plane.png');
rb_plane = imread('Test Images/rb_plane.png');
rg_plane = imread('Test Images/rg_plane.png');
gb_plane_filter = imread('Filter Data Set/gb_plane_filter.png');
rb_plane_filter = imread('Filter Data Set/rb_plane_filter.png');
rg_plane_filter = imread('Filter Data Set/rg_plane_filter.png');

% Visually compare the channels between planes
figure(1);
subplot(2,2,1);
imagesc(rb_plane_filter(:,:,1)); % red
title('RB - Red Channel');
subplot(2,2,2);
imagesc(rg_plane_filter(:,:,1)); % red
title('RG - Red Channel');
subplot(2,2,3);
imagesc(gb_plane_filter(:,:,3)); % blue
title('GB - Blue Channel');
subplot(2,2,4);
imagesc(rb_plane_filter(:,:,3)); % blue
title('RB - Blue Channel');


figure(2);

% Compare solid primaries to corresponding BW planes
primaries = {solid_FF0000_filter, solid_00FF00_filter, solid_0000FF_filter};
for primary = 1:3
    for channel = 1:3
        subplot(3, 3, primary + channel * 3 - 3);
        x = primaries{primary};
        if primary == channel % compare to white
            y = solid_FFFFFF_filter;
        else % compare to black
            y = solid_000000_filter;
        end
        differences = (double(x(:,:,channel))/255 - double(y(:,:,channel))/255) .^ 2;
        display(sprintf('Primary %d channel %d MSE: %f  VSE: %f Max: %f', primary, channel, ...
            mean(differences(:)), var(differences(:)), max(differences(:))))
        imagesc(differences);
        title(sprintf('Comparing primary %d to solid BW on channel %d', primary, channel));
        colorbar;
    end
end
```

B.5 Experiment 3

```
grad_bar_filter = (imread('Filter Data Set/grad_bar_filter.png'));
grad_bar = (imread('Test Images/grad_bar.png'));

colors = {'r','g','b'};

% Select a single channel to collect data about
channel = 1;


x = grad_bar(:, :, channel);
y = grad_bar_filter(:, :, channel);
scatter(x(:), y(:), colors{channel});
hold on;

% Take mean along each input value to average along the output set.
totals = zeros(256,1);
counts = zeros(256, 1);
for i = 1:numel(x)
    totals(x(i)+1) = totals(x(i)+1) + double(y(i));
    counts(x(i)+1) = counts(x(i)+1) + 1;
end
idx = find(counts > 0);
val = totals(idx) ./ counts(idx);

% Plot scatter and vertical average
plot(idx, val, 'k');
hold off;

% Rescale
vert_avg_y = double(val)/255;
vert_avg_x = double(idx)/255;

full_y = double(y(:))/255;
full_x = double(x(:))/255;
```

B.6 Experiment 4

```
vertical_avg_coeff = ...
    [-13.47 41.23 -45.04 19.17 -1.492 0.5954 0.0; ...
    -12.28 41.09 -50.52 26.03 -3.916 0.58 0.0; ...
    -1.066 9.679 -19.09 12.92 -1.835 0.3487 0.0];

full_d3_coeff = ...
    [-1.389 1.768 0.5899 0.0; ...
    -1.696 2.52 0.1407 0.0; ...
    -1.698 2.731 -0.09003 0.0];

full_d6_coeff = ...
    [-13.77 42.05 -45.83 19.54 -1.627 0.6362 0.0; ...
    -12.6 41.85 -51.04 26.07 -3.878 0.5905 0.0; ...
    -1.263 10.28 -19.66 13.09 -1.831 0.3575 0.0];

data_set = double(imread('Filter Data Set/grad_bar_filter.png'))/255;
input_image = double(imread('Test Images/grad_bar.png'))/255;

recreation = zeros(size(input_image));

colors = {'Red','Green','Blue'};
for channel = 1:3
    subplot(1,3,channel);
    recreation(:,:,channel) = polyval(vertical_avg_coeff(channel, :), ...
        input_image(:,:,channel));
    difference = (recreation - data_set) .^ 2;
    imagesc(difference(:,:,channel));
    colorbar;
    title(sprintf('Square Difference - Channel %s', colors{channel}));
end
display(sprintf('Mean: %f  Sum: %f  Var: %f  Max: %f', mean(difference(:)), ...
    sum(difference(:)), var(difference(:)), max(difference(:))));
```

B.7 Experiment 5

```
vertical_avg_coeff = ...
    [-13.47 41.23 -45.04 19.17 -1.492 0.5954 0.0; ...
    -12.28 41.09 -50.52 26.03 -3.916 0.58 0.0; ...
    -1.066 9.679 -19.09 12.92 -1.835 0.3487 0.0];

inverse_color_coeff = ...
    [-3.835 12.81 -17.35 13.18 -5.738 1.939 0.0; ...
    -8.176 28.36 -39.21 28.23 -11.04 2.845 0.0; ...
    -27.59 85.64 -103.7 62.86 -20.07 3.905 0.0];

solid_white_filter = double(imread('Filter Data Set/solid_FFFFFF_filter.png'))/255;
solid_grey_filter = double(imread('Filter Data Set/solid_7F7F7F_filter.png'))/255;
data_set = double(imread('Filter Data Set/vegetables_0_filter.png'))/255;
input_image = double(imread('Test Images/vegetables_0.png'))/255;


% undo the polynomial
vignette = zeros(size(input_image));
for channel = 1:3
    vignette(:,:,channel) = polyval(inverse_color_coeff(channel, :), ...
        solid_white_filter(:,:,channel));
end

% foreward tests
forward_add = input_image + vignette - 1.0;
forward_mult = input_image .* vignette;
for channel = 1:3
    forward_add(:,:,channel) = polyval(vertical_avg_coeff(channel, :), ...
        forward_add(:,:,channel));
    forward_mult(:,:,channel) = polyval(vertical_avg_coeff(channel, :), ...
        forward_mult(:,:,channel));
end

diff_add = sum((forward_add - data_set) .^ 2, 3);
diff_mult = sum((forward_mult - data_set) .^ 2, 3);

figure;

subplot(2,2,1);
image(forward_add);
title('Vignetting via addition');

subplot(2,2,2);
image(forward_mult);
title('Vignetting via multiplication');

subplot(2,2,3);
imagesc(diff_add,[0.0 0.1]);
colorbar;
title('Square error for addition');

subplot(2,2,4);
imagesc(diff_mult, [0.0 0.1]);
colorbar;
title('Square error for multiplication');

display('Foreward Tests:');
display(sprintf('Mean Squared Error for addition: %f', mean(diff_add(:))));
display(sprintf('Mean Squared Error for multiplication: %f', mean(diff_mult(:))));


% backwards tests
untint = zeros(size(input_image));
for channel = 1:3
```

```
    untint(:,:,channel) = polyval(inverse_color_coeff(channel, :), ...
        data_set(:,:,channel));
end

backward_mult = untint ./ vignette;
backward_add = untint - vignette + 1.0;
diff_add = sum((backward_add - input_image) .^ 2, 3);
diff_mult = sum((backward_mult - input_image) .^ 2, 3);

figure;

subplot(2,2,1);
image(backward_add);
title('Un-Vignetting via addition');

subplot(2,2,2);
image(backward_mult);
title('Un-Vignetting via multiplication');

subplot(2,2,3);
imagesc(diff_add,[0.0 0.1]);
colorbar;
title('Square error for addition');

subplot(2,2,4);
imagesc(diff_mult, [0.0 0.1]);
colorbar;
title('Square error for multiplication');

display('Backward Tests:');
display(sprintf('Mean Squared Error for addition: %f', mean(diff_add(:))));
display(sprintf('Mean Squared Error for multiplication: %f', mean(diff_mult(:))));


figure;
subplot(2,1,1);
imagesc(vignette(:,:,1) - vignette(:,:,2));
colorbar;


subplot(2,1,2);
imagesc(vignette(:,:,1) - vignette(:,:,3));
colorbar;

imwrite(vignette, 'vignette.png');
```

C References

"JPEG." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 18 Oct 2014. Web. 18 Oct 2014. https://en.wikipedia.org/wiki/JPEG

"MATLAB Documentation." The MathWorks, Inc. 2014. Web. 18 Oct 2014. http://www.mathworks.com/help/index.html

"YCbCr." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 6 Oct 2014. Web. 18 Oct 2014. https://en.wikipedia.org/wiki/YCbCr