Comp Photography
Final Project

Clement Henghui Li

Spring 2018

cli620@gatech.edu

Food Manga NPR Replication:

Getting Hungry From Cartoons

This project turns a regular food image into a manga style NPR image that emphasizes the gloss of the food to make it more appetizing.

The Goal of Your Project

Original project scope:

The scope of this project is to replicate “Image generation system of delicious food in a manga style” (Fujieda et. al 2017). This system contains two parts: a Boyadzhiev’s Band-Sifting Decomposition method to emphasize the image of its glossiness and a manga-converted image using edge extraction and toon-shading. The results are combined for the final improved product.

What motivated you to do this project?

I wanted a project that uses my food pictures from my Japan trip in April. As I browsed through papers, I found this paper very suitable due to its conversion of food pictures into a Japanese style art.

Scope Changes

- Did you run into issues that required you to change project scope from your proposal?
  - The scope of the project remained the same. However, methods must be changed or omitted to obtain good results. This is because the system uses two algorithms in separate papers with different purposes than when merged for the purpose of the main paper. However, all parts from the paper were implemented, resulting in the general scope of the project unchanged.
- Give a detailed explanation of what changed
  - One method is that the original author first converts the input to a gray image to get a more ‘manga’ feel to the output. However, I would argue that manga drawings also come in color, and the results are much more impressive. The output of the gloss emphasized algorithm is gray and this is added on to a colored manga non-photorealistic rendered output. In the edge detection algorithm, the edge filtering part was omitted as filtering the edge anymore would cause loss in data and produce worst results. A bilateral filter was used instead of a guided filter, because of the need to install a cv2 extension of the imgproc library. The results still turned out well, so there was not an urge to pursuit the guided filter. I believe the results would be less noisy and the gloss output would be smoother if a guided filter was used or if a higher resolution image is used.

Showcase

Input Output



1



2



3

Image1 was taken from: http://nice-japan.com/foods/385/ as a baseline for the results shown in original paper.

Images 2-6 are original and taken in Japan between 03/30/2018 and 04/09/2018. Automatic settings were used because this system should work for any camera settings.

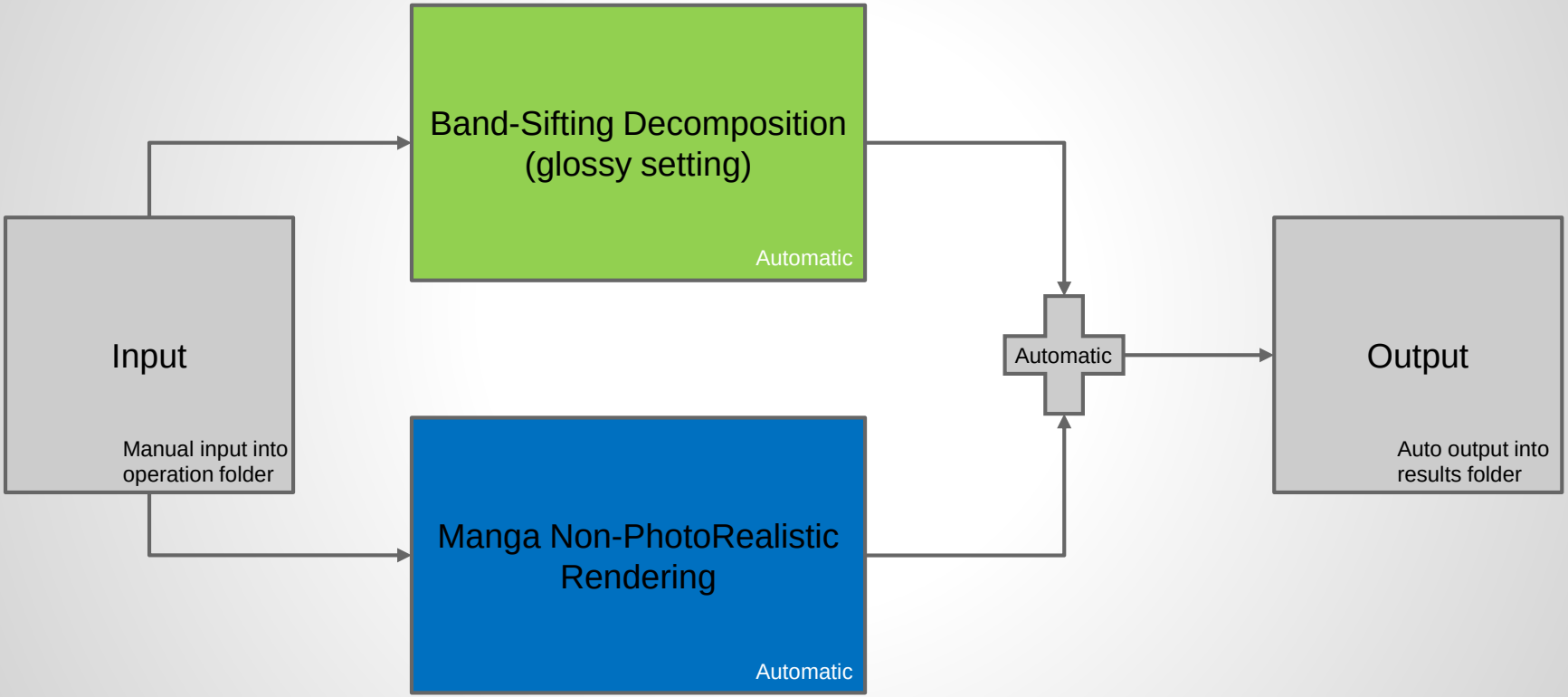
Input Output

4


5

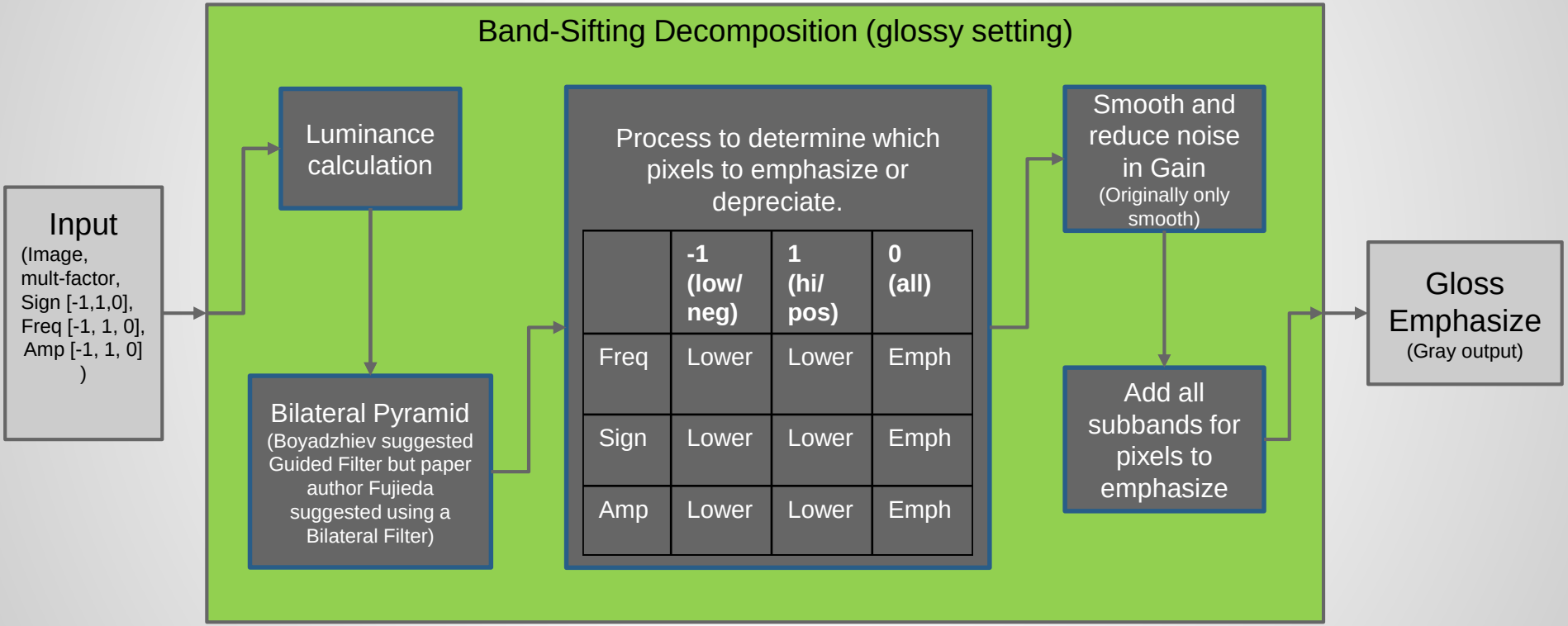

6


Project Pipeline



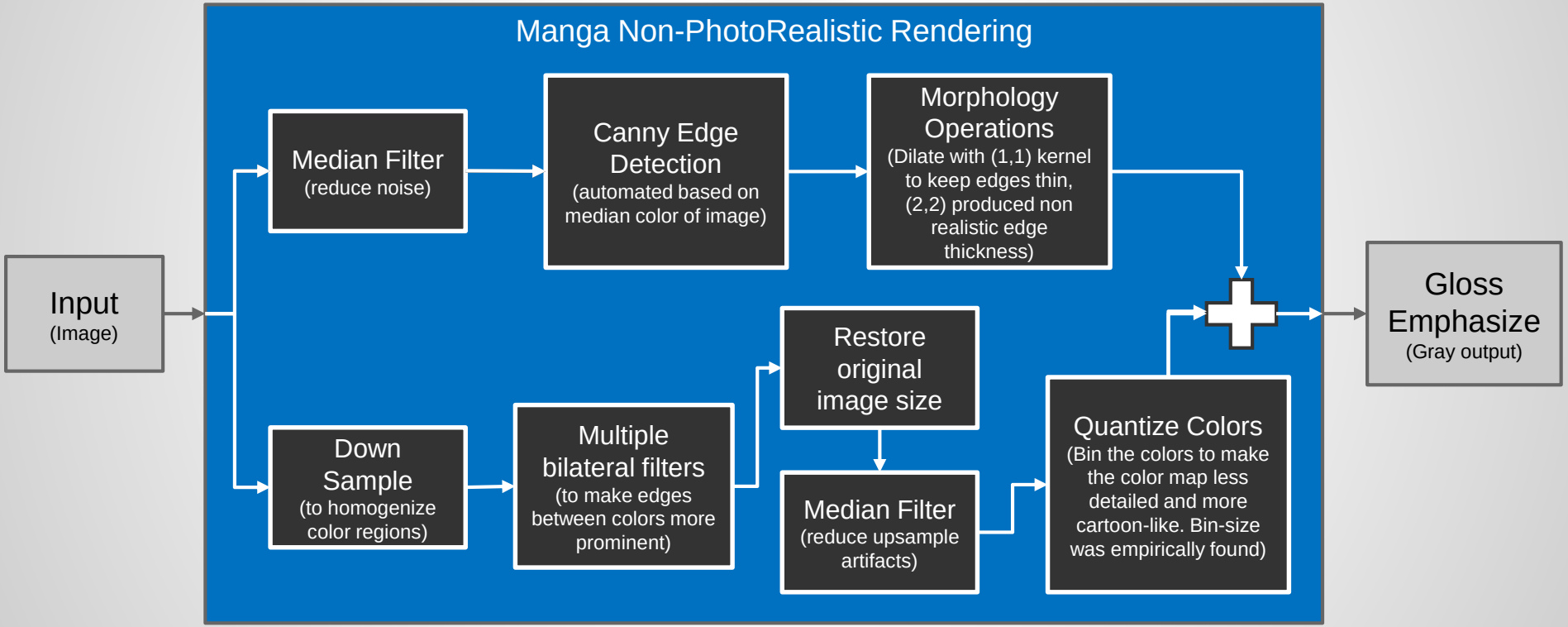
Note: all steps are completed

Project Pipeline (Cont'd)



Note: Pseudocode was provided by Boyadzhiev for this step. My contribution lies with adapting different methods to match Fujieda's purpose of emphasizing glossiness of food images.

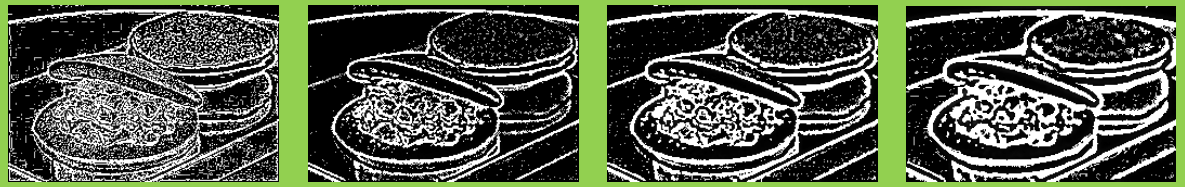
Project Pipeline (Cont'd)



Note: This code was self written
(Dade, Toonify: Cartoon Photo Effect Application.)

Demonstration: Result Set 1

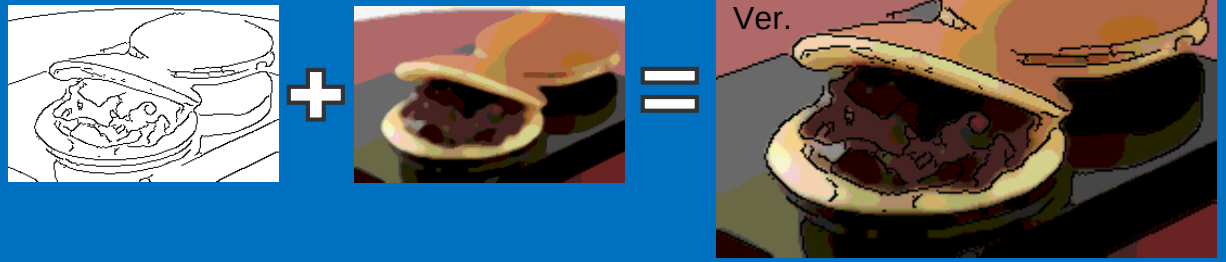
Subband1 Subband2 Subband3 Subband4



(Smooth & Denoised gain) + (Smooth & Denoised gain) + (Smooth & Denoised gain) + (Smooth & Denoised gain) =

Desired emphasize regions

Edge Map + Toon Shaded =

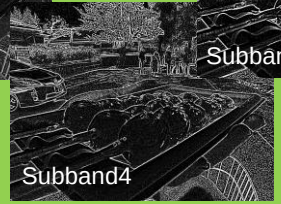


Manga Ver.

Our Results

Fujieda et. al paper results

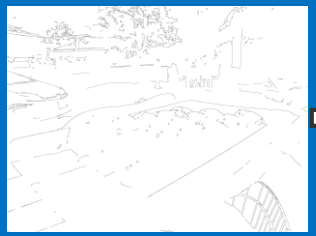
Demonstration: Result Set 2



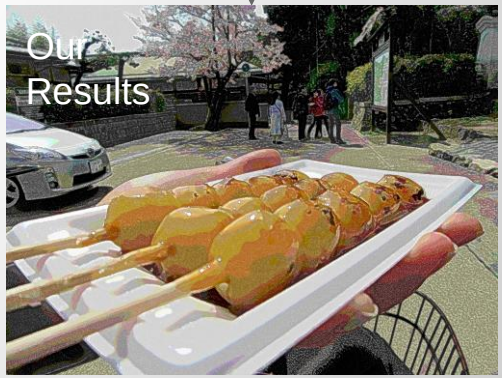
A subband is the difference between adjacent filtered frequency bands. The list of subbands are smoothed and denoised before summation into the desired emphasize regions on the right.

Desired emphasize regions

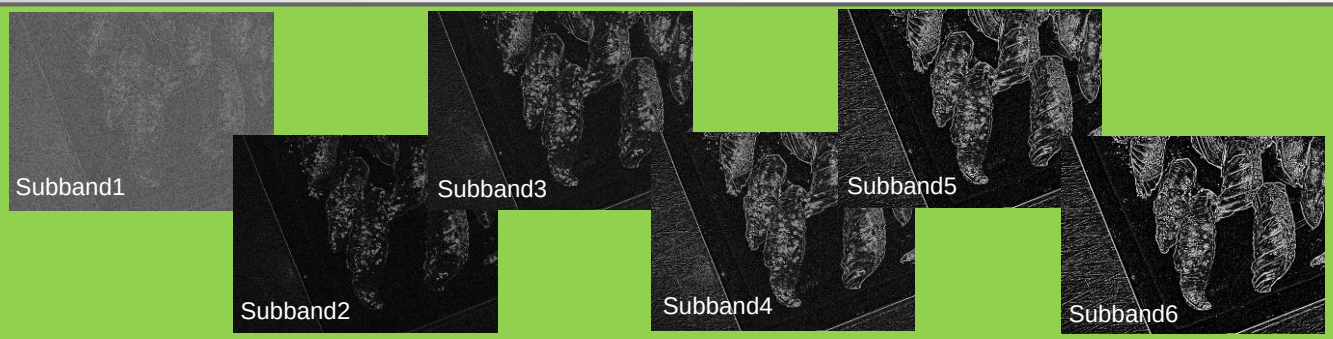
Edge Map



Toon Shaded



Demonstration: Result Set 3



A subband is the difference between adjacent filtered frequency bands. The list of subbands are smoothed and denoised before summation into the desired emphasize regions on the right.

Desired emphasize regions

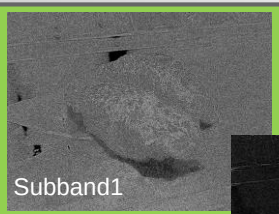
Edge Map

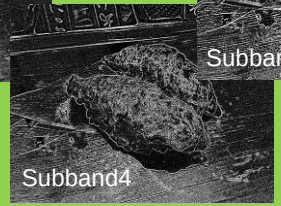
Toon Shaded

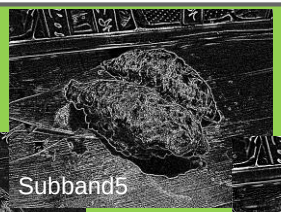
Manga Ver.

Our Results

Demonstration: Result Set 6







A subband is the difference between adjacent filtered frequency bands. The list of subbands are smoothed and denoised before summation into the desired emphasize regions on the right.

Desired emphasize regions

Edge Map

Toon Shaded

Project Development

There were many hurdles as I tackled the Band-sifting Decomposition part first.

The first one is understand which luminance equation was better suited for my application. I set up to switch between the three luminance equations known (relative, color-contrast and HSP), and initially chose the relative color model. Reasoning behind this is, after printing out the Bilateral Pyramid outputs for each luminance equation, the relative model as it was the brightest and had the most information. Next time, I would like to experiment and perform the operation on the luminance map on all three color channels and merge it after. The purpose is to see how the gloss looks in color form.

Another hurdle was the sign_frequency_selected function in gloss.py. This was a python syntax hurdle where if you do not have parentheses around each condition, it will always come back False. This logic skipped over much of this code; after the fix the program ran much slower as expected.

The hardest part was figuring out how to reduce the noise in the mask. If i sum up the subbands without reducing the noise, there will be unexplainable artificial artifacts. To fix this, i must apply a median filter on top of the suggested smoothing filter to get rid of the noise in the gain.

The final emphasized results does not look like the emphasized results of Fujieda's paper because i did not reapply it back to the luminance image. When it was reapplied, averaging the emphasized output and the manga rendered image would get rid of the color quantizing done in the manga rendering portion. Pipelining the process this way also save more time as we do not need to reapply the emphasized part on the luminance image.

Project Development (Cont'd)

The manga non-photorealistic rendering portion consists of 3 main parts: edge detection and boldening, toon-like color rendering and recombination of the two outputs. It was difficult to decide which kernel to use for each filter used in the function. It took a lot of time to empirically find out which setting would produce an image that most represent a piece of Japanese manga art.

One example is that in the paper 'Toonify: Cartoon Photo Effect Application', Dade suggest to use a kernel of size 2 by 2 to bolden the edges enough for the cartoon effect. However, by doing this it became very uncharacteristic of a manga art. The final decision was to omit the edge boldening portion. The line of code was left in the program for reference but using a 1 by 1 kernel, essentially not doing anything.

The second portion that was difficult and I spent much time on was using morphology to filter out small contours. Through many attempts with different type of kernels (described in http://www.imagemagick.org/Usage/morphology/) and playing around with the different morphology methods, I could not rid the small contours in the edge maps. I thought that this might be because the small contours are all clustered together and that caused the kernel to hit multiple small contours and keeping the artifacts. However, this also did not work for pixels that seem to be contour islands. In the end, I skipped that step to move on to the next one. The results without the edge filter is still acceptable. I think it would be less blotchy if it is was implemented, creating even better results.

It was a challenge to decide what bin size to use for the color quantization. This step is the bread and butter to how manga/cartoony the appearance is and required multiple opinions to determine the correct setting to use. I called upon a handful of friends who are manga fanatics to determine which bin size (10, 15, 35, 50) is better represented. The result is to use the 35 uint bin size for the colors in each channel.

Project Development (Cont’d)

The picture of dakoyaki shown in demonstration result set 1 was done specifically as the baseline for testing my code versus the results of the paper. The results of the manga rendered image demonstrate very similar outputs. In fact, including colors in the algorithm might have made my results slightly better. One example can be seen by looking at the details shown by the reflection of my results versus the nearly complete black shadow of their results. My manga rendered image also show more contours inside the red bean paste than theirs does. From these observations, I would say that introducing color into the algorithm would enhance the details in the results.

The edge detection algorithm returns a slightly more spotted image than the output shown in their paper. Despite this, the algorithm seem to have been able to detect glossy areas and tries to amplify those areas. One area to note is commonality in bright, or glossed, regions overlaid on top of the red bean paste between the bread. Similar bright regions are found in the shadow of the dakoyaki on the table. The glossy regions only show in the paper’s results of the emphasized image but not the manga image, whereas both of my outputs demonstrate the reflection off the glass table.

My favorite output is the dango set (Demonstration Result Set 2). Though there are very little emphasis addition onto the dango food itself, the emphasis on the cherry blossom in the back and the shadows on the ground demonstrate very high details and really enhances the beauty and uniqueness of the cherry blossoms. The results from the sushi set (Demonstration Result Set 3) was a bit underwhelming despite adding on the gloss to the manga image created a more appetizing picture. The input image needs to have enough variance in its color for this program to work well.

For a future project, I would like to attempt this on a short video to see the transitions between frames of resulting animated frames.

Computation: Code Functional Description

```
if __name__ == "__main__":
    mainstart = datetime.now()

    use_dir = "test_dir"
    image_dir = os.path.join("pics", "source", use_dir)
    out_dir = os.path.join("pics", "out")
    try:
        _out_dir = os.path.join(out_dir, use_dir)
        not_empty = not all([os.path.isdir(x) for x in
                             glob(os.path.join(_out_dir, "*.*"))])
        # if not_empty:
        #     raise RuntimeError("Output directory is not empty - aborting.")
        os.makedirs(_out_dir)
    except OSError as exception:
        if exception.errno != errno.EEXIST:
            raise
    print "Reading images."
    images = readImages(image_dir)


for i in range(len(images)):
    # inputs: sign --> -1 1 0  ==> neg, pos, all
    #         freq --> -1 1 0  ==> low, high, all
    #         amp  --> -1 1 0  ==> low, high, all

    images[i] = cv2.resize(images[i], (0,0), fx=0.5, fy=0.5)

    imagetime = datetime.now()
    #suggested 010
    freq = 0
    sign = 1
    amp = 0
    ...

    print 'final average of gloss and manga images'
    ...
    manga_img = cv2.add(manga.recombine_edge_color(images[i]).astype(np.uint8),
                        cv2.cvtColor(gloss.band_sift_operator(images[i], 1.5, sign, freq, amp).astype(np.uint8), code=cv2.COLOR_GRAY2BGR))
    ...
    cv2.imwrite(os.path.join(out_dir, use_dir, 'glossy_manga{0:01d}.png'.format(i)), manga_img.astype(np.uint8))

    print '------------ time taken by this image: ', datetime.now() - imagetime

print '------------ TOTAL TIME: ', datetime.now() - imagetime
```

```
import errno
import os
import sys

import numpy as np
import cv2

from glob import glob

import gloss
import manga

from datetime import datetime
```

The main function pulls the source images from ‘./pics/source/test_dir/’ and loops through each picture found. Before processing anything each image is down-sampled by a half to save processing time. The default inputs of the Band-Sifting Decomposition is all frequencies, positive signs and all amplitudes. This is recommended by Fujieda as an empirically found solution that best represent a manga picture. An multiplication factor of 1.5 was selected to show the gloss but not let it stand out too artificially. Here we call the two functions and add them at the same time to reduce allocated memory space used.

Computation: Code Functional Description

```python
# Get Edges of image
def get_edges(I):

    # Median filter
    # Reduce noise --> 7x7 matrix (centroid = median of surrounding neighbors)
    # extreme specks smoothed. needs to be small enough to preserve edges
    # out_I = np.ndarray(shape=I.shape)
    out_I = cv2.medianBlur(src=I, ksize=7)
    # cv2.imwrite('emap_test1.png', out_I)

    # Edge detections
    # Canny edge detector (edges = single pixel edges
    # can change L2 gradient to true later compare results----------------
    # can change 1st and 2nd threshold later to compare results----------
    sigma=0.5
    v=np.median(out_I)
    # lower = int(max(0,(1.0-sigma)*v))
    # upper = int(min(255,(1.0+sigma)*v))
    out_I = cv2.Canny(out_I, int(max(0,(1.0-sigma)*v)), int(min(255,(1.0+sigma)*v)), L2gradient=False)
    # cv2.imwrite('emap_test2.png', out_I)

    # Morphological operations
    # dilation with 2x2 structuring element --> bolden & smooth contours of edges
    # decides how contours look!
    kernel = np.ones((1,1),np.uint8)
    # kernel = np.array([0,1,0], [1,1,1], [0,1,0])
    # out_I = cv2.dilate(out_I, kernel, iterations = 5)
    # out_I = cv2.morphologyEx(out_I, cv2.MORPH_OPEN, kernel)
    out_I = cv2.morphologyEx(out_I, cv2.MORPH_DILATE, kernel)
    # cv2.imwrite('emap_test3.png', out_I)

    # Edge Filter
    # Edge image separate into constituent regents --> concrete edges
    # Get rid of small contours
    # Empirically testing MINIMUM AREA THRESHOLD < 10.
    # kernel_remove = np.ones((3,3),np.uint8) # area 1of 10
    # kernel_remove = np.ones((3,3),np.uint8)
    # for i in range(2):
    #     out_I = cv2.morphologyEx(out_I, cv2.MORPH_OPEN, kernel_remove)
    # uses a 10 area kernel --> decrease noise smaller than 10 pixels_square.
    # then bolden the remaining edges back to normal
    # can use diamond if square isnt good.

    # out_I = cv2.medianBlur(out_I, 3)
    # cv2.imwrite('emap_test4.png', out_I)
    return np.subtract(255, out_I).astype(np.uint8)
    raise NotImplementedError
```

```python
import numpy as np
import scipy as sp
import cv2
import scipy.signal
from datetime import datetime
```

As mentioned in the project development, this manga non-photorealistic rendering portion consists of 3 main parts: edge detection and boldening, toon-like color rendering and recombination of the two outputs. The edge detection part first clean up noise in the picture without losing data in the edges. Then a canny edge detector was used with a variable 1st and 2nd threshold to produce an automated code. The threshold were selected based on the median pixel intensity of the image. I choose sigma of 0.5 based on analyzing outputs with different sigmas. As you deviate from 0.5 the edge detected would be noisier. This may be due to aliasing in the color channel if the range allowed is too high. The morphology operation part is only used as a placeholder. The reason is that any bolding of the edges would create a non-manga style output; this is explained in higher detail in the project development slides. The mask is then negated because the outlines of a manga art is in black. If the edge filter was implemented, it would take out small counters within a 3x3 square kernel.

Computation: Code Functional Description

```python
def get_color_map(I):
    # Bilateral Filter
    # homogenize color regions while preserving edges.
    # --> downsampled factor of 4 ( 1/2 col 1/2 row)
    ori_shape = I.shape
    downI = cv2.resize(I, (0,0), fx=0.5, fy=0.5)
    # cv2.imwrite('cmap_test1a.png', downI)

    # use 9x9 filter kernel and do it 14x.
    for i in range(14):
        downI = cv2.bilateralFilter(downI, 9, 18.0, 4.5)
    # cv2.imwrite('cmap_test1b.png', downI)

    # restore to original size with linear interpolation to fill in missing pixels
    out_I = cv2.resize(downI, (ori_shape[1], ori_shape[0]))
    # cv2.imwrite('cmap_test2.png', out_I)

    # Median Filter
    # smooth any artifacts that occurred for upsampling
    # use 7x7 kernel (from edge median filter kernel)
    out_I = cv2.medianBlur(out_I, 7)
    # cv2.imwrite('cmap_test3.png', out_I)

    # Quantize Colors:
    # p = pixel value, a = factor to reduce # of colors in each channel
    # pnew = floor(p / a) * a
    # truncate colors close to maximum
    # e.g.: if a = 24, reduce 256/24 to this how many values to be used
    # a = 10
    # a = 25
    a = 35
    out_I = np.divide(out_I, a)
    out_I = np.floor(out_I)
    out_I = np.multiply(out_I,a)
    # cv2.imwrite('cmap_test4.png', out_I.astype(np.uint8))

    return out_I.astype(np.uint8)
    raise NotImplementedError
```

```python
import numpy as np
import scipy as sp
import cv2
import scipy.signal
from datetime import datetime
```

This color mapping part first downsample the image, filter with a bilateral filter, upsample the image to its original size and smoothen out the output. All this is done in order to make the borders between colors stand out more. We downsample it so that the colors would be averaged out between a 2 x 2 kernel. The bilateral filter smoothens out the transition between pixel colors. The final step of quantizing colors is crucial in creating a cartoon like image. If thought of in the cartoon artists' point of view, no one can color a picture with such high color variation as the camera can. An artists normally would not be able to use up to 255^3 different colors. Here we tried [10, 25, 35, 50] as bin size, a. I casually asked friends and family to obtain their personal opinion on which bin size would produce a more representable image to a manga picture. The final decision went to using 35 as the bin size for this project.

Computation: Code Functional Description

```
def recombine_edge_color(I):
    # Getting edge map
    print '--- getting edge map '
    startTime = datetime.now()
    emap = get_edges(I)
    cv2.imwrite('edge_map.png', emap)
    print datetime.now() - startTime

    # Getting Color map)
    print '--- getting color map '
    startTime = datetime.now()
    cmap = get_color_map(I)
    cv2.imwrite('color_map.png', cmap)
    print datetime.now() - startTime

    # Two possible methods:
    # average the two maps
    # multiply the two maps.
    print '--- averaging edge and color maps'
    startTime = datetime.now()
    # manga = np.mean(np.array([np.dstack((emap, emap, emap)), np.multiply(cmap,2)]), axis =0)
    # manga = np.add(np.dstack((emap, emap, emap)), cmap)
    # manga = cv2.add(cmap, cv2.cvtColor(emap, code = cv2.COLOR_GRAY2BGR))
    manga = cv2.bitwise_and(cmap, cmap, mask=emap)
    cv2.imwrite('manga_test.png', manga.astype(np.uint8))
    print datetime.now() - startTime

    print '------manga done | size = ', manga.shape
    return manga
    raise NotImplementedError
```

```
import numpy as np
import scipy as sp
import cv2
import scipy.signal
from datetime import datetime
```

This portion of the code simply recombines the edge map and the color map. Cv2.bitwise_and was used because I realized that the emap was technically a bitwise mask. However, if there are some non 0 or 1 gray pixels, which I do not believe we have, by doing this we would lose some data.

Computation: Code Functional Description

```
def band_sift_operator(image, mult_factor, sign, freq, amp):
    ...
    luminance = 0.299*image[:,:,2] + 0.587*image[:,:,1] + 0.114*image[:,:,0]
    ...
    err = 0.00001
    subband = np.add(decompose(np.add(luminance, err)), err)
    n = len(subband)
    ...
    for levels in range(n):
        print '---working on gloss', levels, 'th level'
        # print 'max subband = ', np.max(subband[levels]), ' min subband = ', np.min(subband[levels]), ' mean subband = '
        startTime = datetime.now()
        ref = subband[levels]+err # keep copy of the subband as reference
        # ref
        for coefx in range(subband[levels].shape[0]):
            # ref
            for coefy in range(subband[levels].shape[1]):
                coef = subband[levels][coefx, coefy] #ref
                # print 'coef = ', coef, ' | sign = ', sign, ' | freq = ', freq, ' | n = ', n, ' | levels = ', levels
                if sign_freq_selected(coef, sign, freq, n, levels):
                    if amp == 0:
                        # straight up multiply factors
                        subband[levels][coefx, coefy] = coef * mult_factor
                        # print 'amp = all'
                    else:
                        #smoothing transition between high and low amplitudes
                        sigma = np.std(subband[levels])
                        mu = np.mean(subband[levels])
```

```
import numpy as np
import scipy as sp
import cv2
import scipy.signal
from datetime import datetime
```

```
                    if abs(coef) < 0.8*sigma+mu:
                        alpha = 0
                    elif abs(coef) > 1.2*sigma+mu:
                        alpha = 1
                    else:
                        alpha = (coef - 0.8*sigma)/(1.2*sigma - 0.8*sigma)

                    # orient transition depending on amplitude selection
                    if amp == 1:
                        # print 'amp = high'
                        subband[levels][coefx,coefy] = coef*(1 + alpha*(mult_factor-1))

                    elif amp == -1:
                        # print 'amp = low'
                        subband[levels][coefx,coefy] = coef*(1+ (1-alpha)*(mult_factor-1))
                    else:
                        print 'Please insert correct values of amp [0 -1 1]... current value: ', amp
            else:
                print
        ...
        gain = np.divide(subband[levels], ref)
        for i in range(n):
            gain = cv2.medianBlur(gain.astype(np.float32),ksize=5)
        for i in range(n):
            gain = cv2.blur(gain,ksize=(3,3))

        subband[levels] = np.multiply(gain, np.subtract(ref, err))
        cv2.imwrite('subband{0:01d}.png'.format(levels), subband[levels])
        print datetime.now() - startTime
        # print subband[0].shape
    ...
    print '------gloss done | shape = ', sum(subband).shape
    ...
    out = cv2.exp(sum(subband))-err
    cv2.imwrite('gloss.png', out.astype(np.uint8))
    return out
    raise NotImplementedError
```

First, this program converts the RGB image to a luminance map. After it creates subbands for the luminance map. The function decompose will be discussed in the next slide. For each level generated, we will cycle through every pixel. If the pixel position in the frequency subband count and intensity content are met, then the pixel will be multiplied by the multiply factor to emphasize that pixel intensity. This emphasize will only increase areas that predicts gloss in it. The gains of this output will be filtered for noise and smoothened out before putting it back together.

Computation: Code Functional Description

```python
def decompose(C):
    ...
    C = np.uint8(C)
    n = round(scipy.log2(min(C.shape[0], C.shape[1])))
    n = int(n)

    ...
    temp1 = C
    # print 'here2'
    count = 0
    subband = [None]*(int(n/2) + 1)
    # subband[count] = C
    # print len(subband)
    # # print len(subband)
    # print 'n = ', n
    # print range(0, n+1, 2)
    for i in range(0,n+1,2):
        ...
        sigmacolor = np.std(temp1)
        sigmaspace = np.float32(i/2)

        temp2 = temp1
        temp1 = cv2.bilateralFilter(temp1, i, sigmacolor, sigmaspace)
        # try using guided filter
        # temp1 = cv2.guidedFilter(C, 0.1^2, 2^i)
        # cv2.imwrite('luminance_filtered{0:01d}.png'.format(i), temp1)
        subband[count] = temp2-temp1
        # print len(subband), ' | ', subband[count].shape
        cv2.imwrite('pre_subband{0:01d}.png'.format(i), subband[count])
        count = count + 1

    return subband
```

```python
import numpy as np
import scipy as sp
import cv2
import scipy.signal
from datetime import datetime
```

```python
def sign_freq_selected(c, sign, freq, n, levels):
    # print 'sign == 0? = ', (sign == 0), '  (c < 0 & sign == -1) ? = ',  (c < 0 & sign == -1) , '(c >= 0 & sign == 1) ?
    # print 'sign == 1 ==> ', sign == 1, 'c >= 0 --> ', c >= 0
    # print '(sign == 1) & (c >= 0)', (sign == 1) & (c >= 0)
    signSelected = ((sign == 0) | ((sign == -1) & (c < 0)) | ((sign == 1) & (c >= 0)))

    freqSelected = ((freq == 0) | ((levels <= n/2) & (freq == -1)) | ((levels >= n/2) & (freq == 1)))
    # print 'signSelected = ', signSelected, ' | freqSelected = ', freqSelected

    return signSelected & freqSelected
    raise NotImplementedError
```

This decompose script is really a bilateral pyramid. For n, the log2 of the smallest image dimension in the image, you would take the difference between the current filtered image and the next incoming image. This difference is called the subband. This script returns a list of subbands to be used in band-sift operator function. If you add the list together without doing anything, then the original image comes back out. By editing the image in its subbands, it allows us to change pixels in a controlled parameters of the image setting (such as high or low band frequencies or coefficient amplitudes, or negative or positive change in luminance between bands). With this control, we can produce outputs with different surface appearances, oily, wet, etc.

This function determines the input matches frequency and sign settings. For example, if we are on subband 2/5 then we are in the low frequency, the coefficient is negative and the settings are freq and sign are -1 then this will return True.

Any additional details?

- Please see image resources at following link:
  - Image Resources

Resources

- Sakiko Fujieda, Yuki Morimoto and Kazuo Ohzeki. 2017. “An image generation system of delicious food in a manga style.” In *Proceedings of SA ’17 Posters.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3145691
- Boyadzhiev, Ivaylo et al. “Band-Sifting Decomposition for ImageBased Material Editing.” ACM Transactions on Graphics 34, 5 (November 2015): 1–16 © 2015 The Author(s)
- Kevin Dade, “Toonify: Cartoon Photo Effect Application.” Standford University. https://stacks.stanford.edu/file/druid:yt916dh6570/Dade_Toonify.pdf
- https://stackoverflow.com/questions/18427031/median-filter-with-python-and-opencv
- https://www.pyimagesearch.com/2015/04/06/zero-parameter-automatic-canny-edge-detection-with-python-and-opencv/
- https://docs.opencv.org/3.1.0/da/d22/tutorial_py_canny.html
- http://www.imagemagick.org/Usage/morphology/
- https://docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html
- Dorayaki image used as baseline originated from: http://nice-japan.com/foods/385/
- Luminance reference: https://stackoverflow.com/questions/596216/formula-to-determine-brightness-of-rgb-color
- Translation of Guided Filter: The MIT License (MIT) Copyright (c) 2016 pfchai

Appendix: Your Code

Code Language: Python

List of code files:

- **main.py**
- **manga.py**
- **gloss.py**

Credits or Thanks

- I would like to thank all the faculties of this class for executing the class so well! I have learned a lot about computing with photography and am excited to learn more!