Comp Photography
Final Project

Ram Subramanian
Fall 2016
ram.s@gatech.edu

Multi-focus image fusion

In this project I attempt to capture multiple images of a scene differing only by the focal plane and combine them into a single image with all planes in simultaneously in focus.

The Goal of Your Project

I was inspired to take on this project after using a commercial image stacker (ZereneStacker) to make a multi-focus image for my assignment on epsilon photography. In that assignment, my variable was the focal plane in focus. However, the variable could just as easily be different points in the scene in focus (like different objects, not necessarily planes).

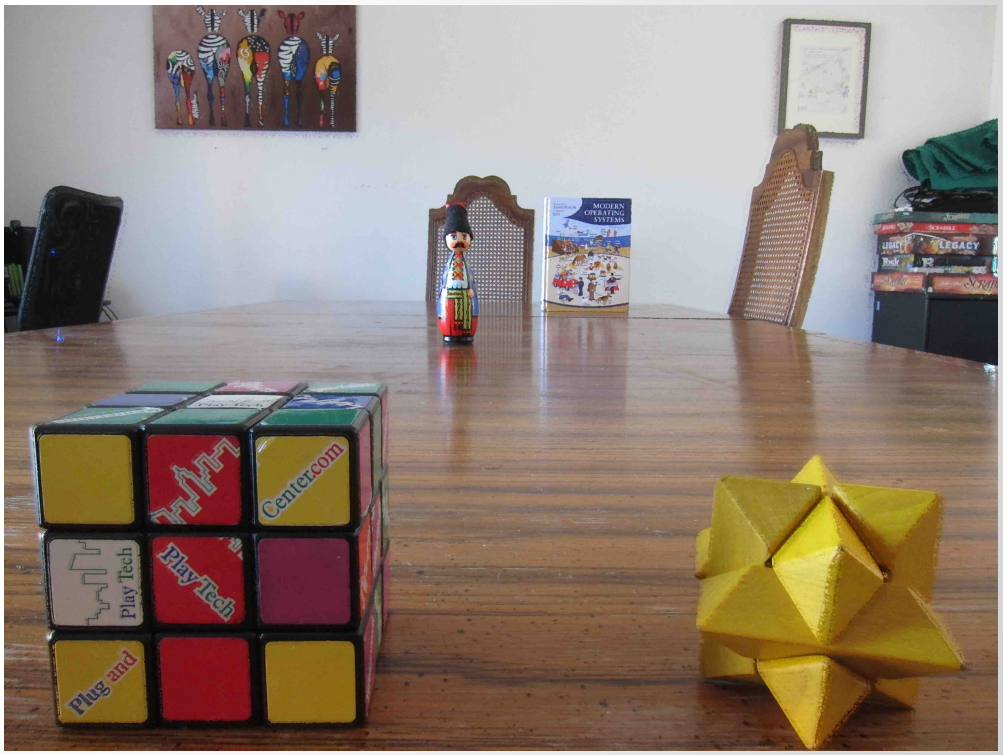
I attempt to use two common methods of achieving this goal in this project.

Showcase what you did on **This One Single Slide**. That might be challenging. You may use several images and format how you wish; but this single slide should be a good pictorial representation of your work. Be creative.

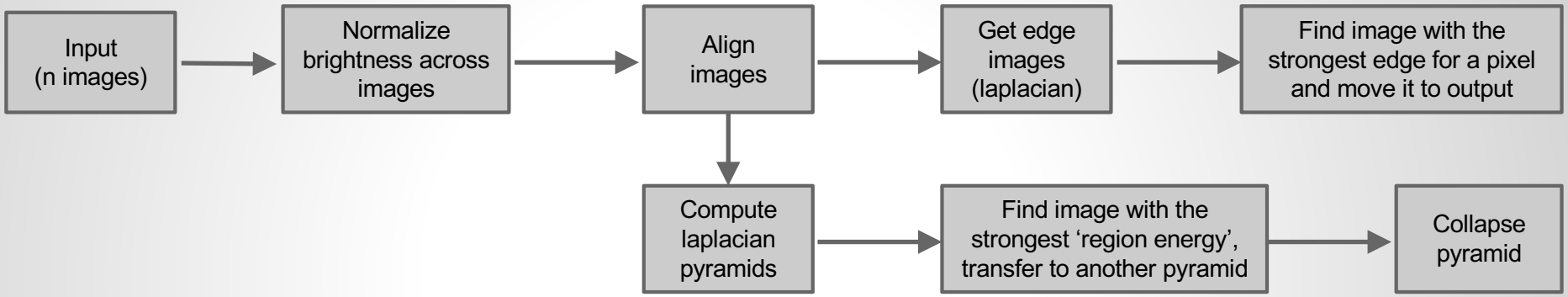
Input

a: f/4.0
e:1/30
iso: 800

Output

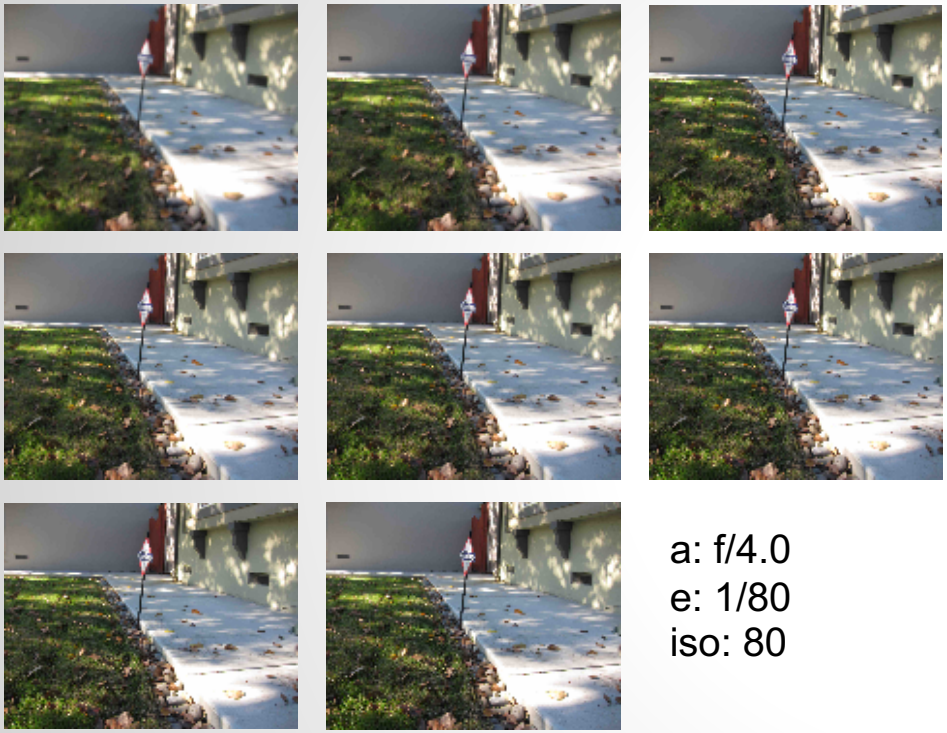


Your Pipeline



1. Start with n images of a scene varying in the focal composition of any sort.
2. Normalize the brightness across all the images to eliminate any lighting or exposure differences thereby increase chances of a good result in the end.
3. Align the images together. They could be misaligned due to many factors including slight-zoom when changing focus, environmental factors, etc. However, I've commented the call to this function because of some difficulties I will get into later.
4. Now I tried using two separate methods to the end goal – one is simply computing the edge images and copying over pixels with strong edges to the final image, the other is using laplacian pyramids where 'good' regions are copied over as a whole.

Demonstration: Show complete Input/Output results

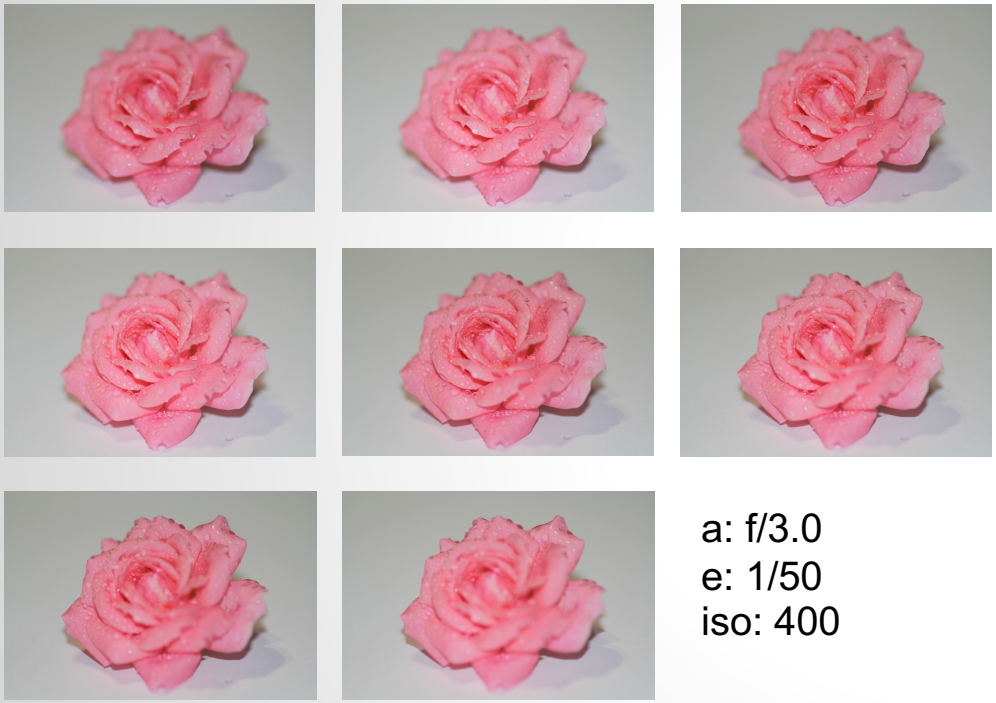


a: f/4.0
e: 1/80
iso: 80

Output
“Protection One”

The shadow of the leaves and branches of the tree were moving significantly when the input images were taken. This affects the blending of those areas in the final image.

Demonstration: Show complete Input/Output results



a: f/3.0
e: 1/50
iso: 400

Output

I suspected that my camera has a big role to play in quality of the outcome and so I got these set of pictures from a thread in Magic Lantern forum (link in Resource section). The outcome was almost flawless!

Computation: Project Development

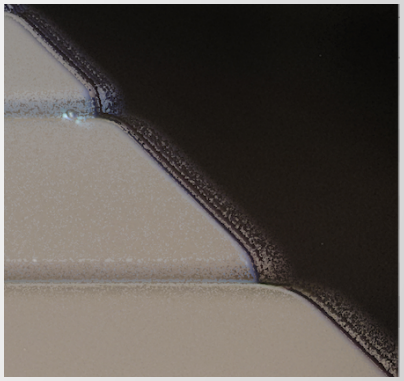
Before conducting any official research, I thought about possible ways I could go about solving this problem. You could do a Fourier Transform to detect if an image is sharp or blurry, but since the images will have both areas in them, you will have to do a transform on a window and analyze the high frequency distribution. But having never done a Fourier Transform ever (either theoretical or practical), I decided to read through some papers on the subject.

Upon conducting some research into this topic, it soon emerged that there are two main way to tackle this problem: using pixel-based methods or using multi-scale transforms (Laplacian pyramids, wavelet transforms). The former had the advantage that it was relatively simple to implement and much quicker than the latter, which produce better results.

The first pixel-based method I tried was simply averaging the input images and using an un-sharp mask. This predictably didn't not work very well: the output was too blurry, was of low contrast and was too 'jumpy'. This is primarily because some my individual images had a very low depth of field, i.e. all planes other than the one in focus were super blurry. The jumpiness came from the fact that, despite having the save EV, ISO and aperture settings some pictures differed in the intensity, perhaps due to the fluorescent lighting. So I had to normalize the brightness across images first.

Computation: Project Development

The next method I tried was to compute the edges in each image – the areas in focus will have strong edges while the areas not in-focus will have weak edges. To do this, I applied the Laplacian operator to each of the input images, blurred it just a bit and then for every pixel I computed the image in which the edge at that pixel was strongest and copied the corresponding intensity to the output image. This worked fairly well, except around strong edges there would be a jot of jumpyness (shown below). Dilating the Laplacians worked quite a bit in reducing this.



Upon analyzing the input images further I noticed that there was a slight zooming effect on the captured picture depending on the focus setting (https://dl.dropboxusercontent.com/u/14887563/animated.gif) . Since my camera isn't one with a particularly good lens, I had to deal with this in software. I tried to use ORB and Brute Force matcher from my previous assignments to find homography between the images and warp them to align them all together.

This worked very well for images that were close in the focus setting, but produced really warped output for images whose focal planes were far apart (for example: 2 cm vs 5 m, shown next slide). This was because the a lot of the strongest matches were either from or to a blurry part of an image. To look at this another way, between the images whose focal planes were farthest apart (in a low depth of field scenario), it will be unlikely to get a successful warp.

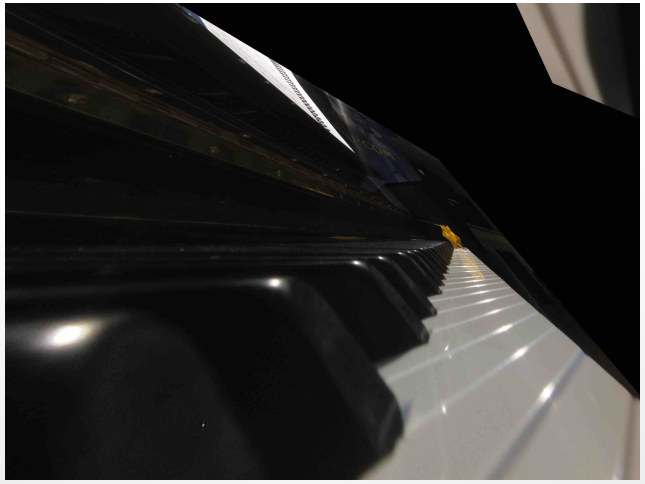
Computation: Project Development

Because I couldn't get the alignment function to work reasonably well, I resorted to blurring the chosen image index matrix for each pixel in the output, so pixels adjacent to a strong edge are more likely to be chosen from the same image as the strong edge. This method yielded better results, especially for the objects and the flower.

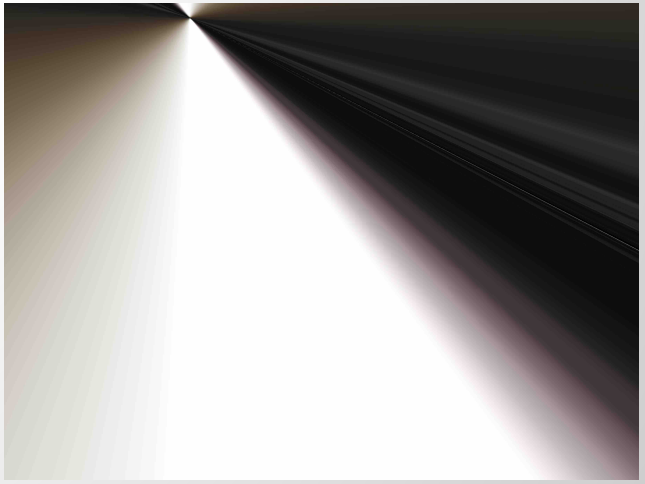
Some examples of warping by the alignment function:

Warp Base

Greatly warped

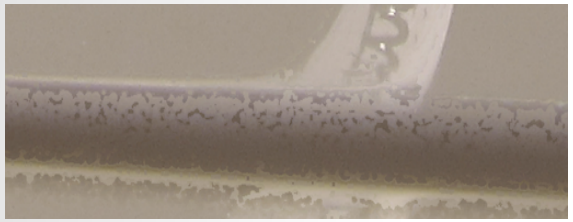


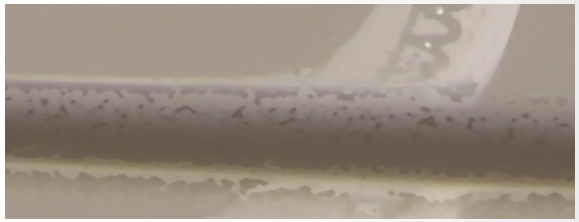
!!???



Computation: Project Development

If you examine the final artifacts closely, you you see a bits of pixels that don't quite fit in (see below). Using a median filter here worked to erase some of them, but obviously, the rest of the image gets quite blurry and I didn't think the trade-off was quite worth it.

Before 

After 

Finally, I implemented a modified version of the multi-scale transform algorithm, using Laplacian pyramids (from the paper by W. Wang and F. Chang). This algorithm computes the deviation (from mean) and the entropy of each window in each of the pyramid levels of each of the image. The deviation along with the entropy give that particular window (or 'region') a score with which selections for the final output can be made.

This was a hard one to get results on simply because of the complexity and execution times, even for small images with small pyramid sizes. Since the paper was quite sparse on the implementation details and since I was out of ideas at this point, I couldn't make further progress to objectively compare the results.

Details: What did not work? Why?

- The image alignment did not work very well. Again this was due to poor quality of keypoints and matches due to variable blurriness between the images (explained in the previous slides).
- Computation time was a big problem! Especially converting python lists to numpy arrays – for a list of “dimensions” (15, 640, 480, 3), this conversion took over a minute! I ended up keeping the lists as lists due to this.
- The region-energy based algorithm was simply too complex to be coded efficiently in python (also in part due to the above point). I had to strip away certain elements from it to make it somewhat tolerable. Perhaps it will perform better when using the C++ libraries.

Computation: Code Functional Description

```
def norm_brightness(imstacks):
        imstack, imstackBW = imstacks
        out = [imstack[0]]
        m1 = np.mean(imstackBW[0])
        for i, img in enumerate(imstack[1:]):
                m2 = np.mean(imstackBW[i+1])
                out.append(img + (m1 - m2))
        return [out, imstackBW]
```

This function takes the set of images and normalizes the brightness across them by adding any difference in the mean intensity compared to a base image. Since we don’t expect a wide variance of brightness across the images, this method performs sufficiently.

Computation: Code Functional Description

```
def find_homography(kp1, kp2, matches):
    pt1 = np.array([[kp1[match.queryIdx].pt] for match in matches],
dtype=np.float_)
    pt2 = np.array([[kp2[match.trainIdx].pt] for match in matches],
dtype=np.float_)
        H, _ = cv2.findHomography(pt1, pt2, method=cv2.RANSAC,
ransacReprojThreshold=4.0)
        return H
```

This function computes the homography between two images given the keypoints of key features and the matches (computed by bfmatcher).

Computation: Code Functional Description

```
def align_images(stacks):
    imstack, imstackBW = stacks
    out = [imstack[0]]
    orb = cv2.ORB()
    kp1, des1 = orb.detectAndCompute(imstackBW[0], None)
        bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    for i, img in enumerate(imstack[1:]):
        kp2, des2 = orb.detectAndCompute(imstackBW[i], None)
        matches = bf.match(des1, des2)
        matches = sorted(matches, key=lambda x: x.distance)
        H = find_homography(kp1, kp2, matches[:30])
        wimg = cv2.warpPerspective(img, H, (img.shape[1], img.shape[0]),
flags=cv2.INTER_LINEAR)       ## Do only linear interpolation
        out.append(wimg)
    return [out, imstackBW]
```

This function, given a set of images, should align them all together such that any minor differences (such as zoom, translation, rotation, etc) can be eliminated before further analysis.

Quite simple in terms of how: computes feature points using orb and matches them using bfmatcher. Finally the warping is done through two library calls to find homography and to do the actual warping.

Computation: Code Functional Description

```
def genKernel(parameter):
    kernel = np.array([0.25 - parameter / 2.0, 0.25, parameter, 0.25, 0.25 -
parameter /2.0])
    return np.outer(kernel, kernel)
def reduce_layer(image):
    kernel = genKernel(0.4)
    return cv2.filter2D(image, -1, kernel)[::2, ::2]
def expand_layer(image):
    kernel = genKernel(0.4)
    out = np.zeros((image.shape[0]*2, image.shape[1]*2, 3))
    out[::2, ::2, :] = image
    return cv2.filter2D(out, -1, kernel, borderType=cv2.BORDER_CONSTANT) * 4
```

Functions are similar to the ones I submitted for assignment #6.

genKernel – generates a normalized gaussian kernel according to parameter
reduce_layer – given an image, blurs and downsamples it (for gaussian pyramid)
expand_layer – opposite of reduce_layer, i.e. given a image, expands it by x4

Computation: Code Functional Description

```
def gaussian_pyramid(image, levels):
    gpyr = [image]
    for i in range(levels):
        bimg = cv2.GaussianBlur(gpyr[-1], (5, 5), 0.05)
        gpyr.append(reduce_layer(bimg))
    return gpyr
def crop(image, size):
    imh, imw = image.shape[:2]
    if not imh == size[0]:
        image = image[:size[0]-imh, :]
    if not imw == size[1]:
        image = image[:, :size[1]-imw]
    return image
```

gaussian_pyramid - Given an image, constructs it's gaussian pyramid to n levels, by successively blurring and reducing the image n-times.
crop – Given an image and a shape (smaller than the image), crops the image to the shape.

Computation: Code Functional Description

```
def laplacian_pyramid(gpyr):
    gpyr = gpyr[::-1]
    out = [gpyr[0]]
    for i, img in enumerate(gpyr[:-1]):
        img = expand_layer(img)
        img = crop(img, gpyr[i+1].shape)
        out.append(gpyr[i+1] - img)
    return out[::-1]
def collapse_pyramid(lpyr):
    s = lpyr[-1]
    for i in range(len(lpyr)-1, 0, -1):
        s = expand_layer(s)
        s = crop(s, lpyr[i-1].shape)
        s = s + lpyr[i-1]
    return s
```

laplacian_pyramid – Given a gaussian pyramid of n levels, constructs the laplacian pyramid by successively subtracting the expanded N-th level gaussian with the N-1-th level gaussian.
Collapse_pyramid – Given a laplacian pyramid, collapses it to down to a single image, by succesilve adding the N-th level gaussian to the N-1…1 level laplacians.

Computation: Code Functional Description

```
def make_single_image(stack):
    out = stack[0]
    (h, w) = out.shape[:2]
    for img in stack[1:]:
        (nh, nw) = img.shape[:2]
        img = np.concatenate((img, np.zeros((h-nh, nw, 3))), axis=0)
        out = np.concatenate((out, img), axis=1)
    return out
def laplacian(img, ksizeL=5, ksizeM=5, ksizeG=5, sigma=10):
         img = cv2.GaussianBlur(img, (ksizeG, ksizeG), sigma)
         img = cv2.Laplacian(img, cv2.CV_32F, ksize=ksizeL)
         return cv2.medianBlur(img, ksizeM)
```

make_single_image – given an image stack, concatenates them all together side-by-side into a single image
laplacian – given an image, blurs it, filters it through a laplacian kernel (for getting an edge image), and again through a median filter (to preserve the edges).

Computation: Code Functional Description

```
def mfocus_laplacian_of_gaussian(imstacks):
        imstack, imstackBW = imstacks
        # laplsum = np.zeros(imstackBW[0].shape, dtype=np.int64)
        laplstack = np.zeros((len(imstackBW),) + imstackBW[0].shape, dtype=np.float32)
        for i, img in enumerate(imstackBW):
                img = laplacian(cv2.dilate(img, np.ones((5, 5))))
                # laplsum = laplsum + img
                laplstack[i] = img
        out = np.zeros(imstack[0].shape, dtype=np.float32)
        idm = np.zeros(imstackBW[0].shape, dtype=np.uint8)
        for i in range(out.shape[0]):
                for j in range(out.shape[1]):
                        p = np.abs(laplstack[:, i, j])
                        idx = np.where(p == max(p))[0][0]
                        # out[i, j] = imstack[idx][i, j]
                        idm[i, j] = idx
        # st = zip(*np.where(laplsum == max(laplsum)))[0]
        idm = np.around(cv2.GaussianBlur(idm, (5, 5), 10)).astype(np.uint8)
        for i in range(out.shape[0]):
                for j in range(out.shape[1]):
                        out[i, j] = imstack[idm[i, j]][i, j]
        # out = cv2.medianBlur(out, 5)
        return out
```

Computation: Code Functional Description

mfocus_laplacian_of_gaussian:

The main premise of this algorithm is that areas in focus in an image will show strong edges in the edge image. I create the edge image for each image in the image stack using a laplacian filter.

Then, for each pixel in the output image, I find the index of the image (in the image stack) which shows the strongest gradient at that pixel. Since this is very crude and likely to cause very ‘jumpy’ image, I blur the matrix of indexes to get a closer selection.

Computation: Code Functional Description

```
def mfocus_multi_scale_lapl(lpyr_stack):
    bpx = 2
    wsize = bpx * 2 + 1
    lstack = []
    nlevels = len(lpyr_stack[0])
    for i in range(nlevels):
        level = []
        for lpyr in lpyr_stack:
            lpyr = cv2.copyMakeBorder(lpyr[nlevels-i-1], bpx, bpx, bpx, bpx, cv2.BORDER_REFLECT_101)
            level.append(lpyr)
        lstack.append(level)
    new_lpyr = []
    D = -np.inf
    idx = 0
    pyr = np.zeros(lstack[0][0].shape)
    for h in range(len(lstack)):
        for i in range(pyr.shape[h]-2*bpx):
            for j in range(pyr.shape[1]-2*bpx):
                for k, l in enumerate(lstack[h]):
                    patch = l[i:i+wsize, j:j+wsize].astype(np.float32)
                    mean = np.mean(patch)
                    patch = (patch - mean) ** 2 / 25.0
                    d = np.sum(patch)
                    if d > D:
                        D = d
                        idx = k
                pyr[i, j] = lstack[0][idx][i, j]
        new_lpyr.append(pyr)
        return new_lpyr
```

Computation: Code Functional Description

In this multi-scale implementation, the basic premise is that instead of selecting individual pixels (like in the previous algorithm), I choose regions from one image so as to produce a cleaner looking output.

The regions/windows are chosen according to their deviation from their mean intensity (D), the amount of entropy (E) and the regional energy, given my the formula:

$$D = \sum_{m=1}^{J} \sum_{n=1}^{K} (X(m,n) - \bar{X})^2 / (J \times K) \qquad (10)$$

$$E = -\sum_{i=0}^{L-1} P_i \log P_i \qquad (11)$$

$$RE_l = \sum_{m' \in J, n' \in K} \omega^l(m', n')[LP_l(m+m', n+n')]^2 \qquad (13)$$

What would you do differently?

I would definitely pick C++ to redo this project as I think that will decrease the runtime for the various algorithms by a few factors. I'm not sure how Matlab will compare though, that might be another option.

I would also experiment with more methods to align images and expand on the laplacian of gaussian algorithm, since it was quite (and I say the work 'quite' liberally) close to a good stack. Perhaps, instead of getting the intensities from the image with the greatest gradient, get a small region and blend them together. But my feeling is that pixel-based methods will only get you so far.

Try more ways to approach this problem. For example, create masks for each image corresponding to their strongest edges and blend them all together. This will probably make for very clean and smooth results.

I would also try to implement an algorithm involving Fourier Transforms, but I have yet to find an academic study on this online. So this will very much go into the experimental realm.

Since almost all pictures had some artifacts, I'd like to employ the median filter after the final image is computed and blend the two together to keep the sharpness, while removing the salt and pepper kind of noise.

Resources

“Pyramid based computer graphics” http://web.mit.edu/persci/people/adelson/pub_pdfs/RCA85.pdf

“A Multi-focus Image Fusion Method Based on Laplacian Pyramid“ http://www.ece.drexel.edu/courses/ECE-C662/notes/LaplacianPyramid/laplacian2011.pdf

Pixel-based implementation - http://www.roman10.net/2011/04/12/all-focused-image-by-focal-stacking/

Brightness adjustment - http://www.mathworks.com/matlabcentral/answers/41169-adjust-brightness-of-an-image-with-respect-to-another-image

Flower images - http://www.magiclantern.fm/forum/index.php?topic=11886.0

http://stackoverflow.com/questions/17973507/why-is-converting-a-long-2d-list-to-numpy-array-so-slow

http://stackoverflow.com/questions/20371053/finding-entropy-in-opencv

http://docs.opencv.org/trunk/d5/daf/tutorial_py_histogram_equalization.html

http://www.graficaobscura.com/depth/index.html

http://web.engr.illinois.edu/~goodsit2/cs498dwh/final/

https://en.wikipedia.org/wiki/Depth_of_field

Numpy and OpenCV docs

Appendix: Your Code

https://dl.dropboxusercontent.com/u/14887563/mfocus.py

Credits or Thanks

Thank you to Dr. Irfan Essa and all the TAs and my peer for this learning experience and making this class a fun one! :)