Homework 2 Writeup

Instructions

- Describe any interesting decisions you made to write your algorithm.
- Show and discuss the results of your algorithm.
- Feel free to include code snippets, images, and equations.
- Use as many pages as you need, but err on the short side If you feel you only need to write a short amount to meet the brief, th
- **Please make this document anonymous.**

Implementation of image filter function

First Implementation - Basic Approach

My initial implementation is to do the convolution calculation iterating all the entries in the original image.

Initially, I checked if the input filter size is valid (Width, height both odd).

```
if mod(filter_h, 2) ~= 1 || mod(filter_w, 2) ~= 1
    output = "Error Message";
    return
end
```

To pad the input image with reflected image content, I used MATLAB `padarray` with argument `'symmetric'`. The output image doesn't contain padding so that its size is identical to the input image.

```
pad_h = (filter_h - 1) / 2;
pad_w = (filter_w - 1) / 2;
target_img =...
    padarray(img, [pad_h, pad_w], 'both', 'symmetric');
```

I pre-allocated the output matrix to the size of the input image for optimization. Then I iterated through the entries of the input image.

```
output = zeros(img_h, img_w, img_c);
% for conv
filter = rot90(filter, 2);

for i = pad_h + 1 : pad_h + img_h
```

```
    for j = pad_w + 1 : pad_w + img_w
        for c = 1 : img_c
            target = target_img(...
                i - pad_h : i + pad_h,...
                j - pad_w : j + pad_w,...
                c);
            output(i - pad_h, j - pad_w, c) = ...
                sum(sum(filter .* double(target)));
        end
    end
end
```

Since the MATLAB `imfilter` function return image with a same sample type with the input image, I also made `my_imfilter` function follow this policy.

```
output = cast(output, 'like', img);
```

Second Implementation - Using FFT

In order to enhance the performance of implementation, I next tried using FFT.

My idea of applying FFT to implementation of image filtering started from the 1D-array case.
For 2 arrays `A` and `B` with same size $1 \times m$, the result of `ifft(fft(A).* fft(B))` is an array `C`, where `C[i]` is $\sum_{a=1}^{m}$ `A[a]` $\times$ `B[mod(i-a, m)+ 1]`.

However, what we need for image filtering is different from this - we want to apply convolution by filter, which is usually smaller than the image, to the each element of the image.

In order to get this desired result from input 1D-arrays `img` and `filter`, I reconstructed new arrays `fft_img` and `fft_filter`.

- `fft_img` is constructed by padding the `img`. Size of the padding follows the same policy of the basic implementation - $(w_{\text{filter}} - 1)/2$. Note that height padding is zero for this 1D case.
- `fft_filter` has a same size with `fft_img`. Its first $(w_{\text{filter}} - 1)/2 + 1$ entries are filled with last $(w_{\text{filter}} - 1)/2 + 1$ entries of the `filter`, and its last $(w_{\text{filter}} - 1)/2$ entries are filled with first $(w_{\text{filter}} - 1)/2$ entries of the `filter`.

Then, `ifft(fft(fft_img).* fft(fft_filter))` will have result which is identical to applying convolution by `filter`, to the each element of the `img` with padding.

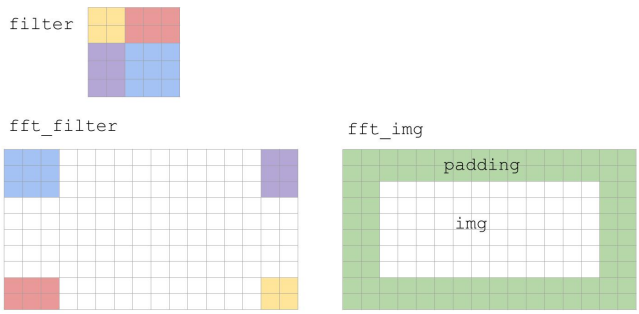

Figure 1: Applying 1D-array FFT based convolution idea to 2D.

I applied the same idea to 2D. Figure 1 illustrates how I applied the idea to my 2D implementation.

Below code is the implementation of the idea in 2D. In order to apply this to multi-channel image, I had to iterate the channels.
Other parts (Initial input size checking, final output casting) are identical to the first version of implementation.

```
[fh, fw] = size(filter);
ph = (fh - 1) / 2;
pw = (fw - 1) / 2;
fft_img = padarray(img, [ph, pw], 'both', 'symmetric');

[h, w, c] = size(fft_img);

fft_filter = zeros(h, w);
fft_filter(1 : ph+1, 1 : pw+1) = ...
    filter(fh-ph : fh, fw-pw : fw);
fft_filter(h+1-ph : h, 1 : pw+1) = ...
    filter(1 : ph, fw-pw : fw);
fft_filter(1 : ph+1, w+1-pw : w) = ...
    filter(fh-ph : fh, 1 : pw);
fft_filter(h+1-ph : h, w+1-pw : w) = ...
    filter(1 : ph, 1 : pw);

output = zeros(h, w, c);
for z = 1 : c
    output(:, :, z) = ...
        ifft2(fft2(fft_img(:, :, z)) .* ...
              fft2(fft_filter));
end

output = output(ph+1 : h-ph, pw+1 : w-pw, :);
```

A Result

Compared to my first approach (iteration on image matrix entries), my second approach can take advantage of both FFT and vectorization.
I measured elapsed time for MATLAB `imfilter`, first version of `my_imfilter`, and the second FFT based version of `my_imfilter`, varying filter size and image size. The results are summarized in Table 1.

		Elapsed time (s)		
Filter size	Img size	MATLAB imfilter	my_imfilter (iteration)	my_imfilter (FFT)
3×3	500×1000	0.0351	3.1161	0.1131
3×3	1000×2000	0.0823	12.2776	0.3344
5×5	500×1000	0.0336	3.1122	0.0842
5×5	1000×2000	0.0818	12.6263	0.4303
7×7	500×1000	0.0403	3.5438	0.0850
7×7	1000×2000	0.0815	14.2838	0.3582

Table 1: Comparison of performance of image filtering functions, varying filter and image sizes.

Though the performance of my FFT based implementation of image filtering wasn't as fast as the built-in MATLAB `imfilter` function, it clearly had a better performance than the first version that does basic iteration.

Implementation of hybrid image

For the first image, which has to be low-pass-filtered, I applied filter generated by `fspecial('Gaussian',cutoff_frequency*4+1,cutoff_frequency)`, where `cutoff_frequency = 7`.
For the second image, which has to be high-pass-filtered, I applied the same filter to the image, and subtracted the result image from the original second image. Since the result image from applying filter is low frequency image, subtracting it from the original image results in high-pass-filtering.
Generating a hybrid image was done by adding the two images each generated from above two steps.