Discussion of Homework 3 DFT and Block Transform

2020-06-11

- Submission: Upload your code on Cody Coursework^{TM} Mathworks platform https://grader.mathworks.com/courses/12790-dsp-ss20 according to the instructions given there for all problems marked with (Cx.y).
- For all problems marked with (Ax.y) refer to the quiz questions at https://www.moodle.tum.de/course/view.php?id=54118. There you can also find the homework rules and more information.
- Result verification: Check your functions as often as you want with Cody Coursework^{TM}.
- Notation: `variable name`, *`file name`*, `MATLAB_function()`

1 Discrete Fourier Transform

1. (C1.1) In this problem you will perform filtering of the image in frequency domain using bandpass filter with specified kernel parameters. For approximation of the bandpass filter we use the difference of two Gaussian filters with standard deviations σ_1 and σ_2, respectively. Thus, the resulting filter is given as $G = G_1 - G_2$. For this write a function that given input image, size and filter parameters returns a filtered image (spatial and frequency domains). Make sure result is identical to the equivalent output of filtering in spatial domain, i.e. image is filtered using appropriate border options. *(8 points)*

```
function [im_freq_filt_spat, im_freq_filt_freq] = ⤸
    filter_frequency_domain(im, N, sig1,sig2)

% Calculate G in frequency domain
g = fspecial('gaussian', N, sig1)-fspecial('gaussian', N, sig2);
g_pad = padarray(g, ([size(im,1) size(im,2)] - size(g)), 'post');
g_pad = circshift(g_pad, -floor(size(g)/2));
G = fft2(g_pad);
% Filter in frequency domain
imf = fft2(im);
im_freq_filt_freq = imf .* G;
im_freq_filt_spat = ifft2(im_freq_filt_freq);

end
```

2. (A1.1) When performing filtering in frequency domain on the image in step (1), analyze what effects do you observe at the border. How can you reproduce this effect when you perform filtering solely in the spatial domain? *(5 points)*

DFT

Image filtering in the frequency domain implicitly extends the image to a periodic signal. As a result, the border areas contain information from the opposite border after filtering. In the spatial domain, the same effect is achieved using the `'circular'` option of the `imfilter()` function. Note the padding and circular shifting operations in the code, which are required to obtain an unshifted version of the filtered image.

Image filtered in the frequency domain (left) and by spatial convolution with border repetition (middle). Difference image (scaled by 15 for visual purposes) is shown on the right. Note the small difference along the border between the left and the right image.

3. (A1.2) Observe the effects of bandpass filtering in step (1), when the input image `im` is distorted with `'salt & pepper'` noise with a noise density of 0.05. Is band-pass filtering in principle influenced by salt and pepper noise? *(5 points)*

   The `'salt & pepper'` noise is known to introduce sharp and sudden disturbances in the image signal. In other words, this noise has high frequency components which are already removed when using a bandpass filter.

4. (A1.3) Compare the complexity of convolution in spatial and frequency domain. *(5 points)*

   The direct convolution of two signals of length N and M has a complexity of order $O(NM)$. The process of performing a convolution in frequency domain exhibits a complexity of $O(N \log N)$ for the case $N > M$. Thus, this complexity analysis depends on the signal length. For relatively small filters direct convolution is more efficient, whereas for longer filters there comes a point at which FFT-based convolution is more efficient. The reduction in the complexity when using FFT becomes more important with increasing signal length.

5. (C1.2) The frequency transform of a real-valued signal should be conjugate point-symmetric around the $(0, 0)$ coefficient. Write a program to verify this property on `imf`. It should return true in case the property holds for this 2D signal and false otherwise.

   The function `fftshift()` can be used to shift the quadrants of the frequency space between this convention and the convention of `fft2()`. *(9 points)*

```
function conj_point_symm = is_conj_point_symm(imf)

% Use fftshift to have the zero frequency component at the center of ⤸
    the spectrum
ffts = fftshift(imf);
center = round(size(ffts) / 2 +1); %Point of symmetry

% Check entire matrix:
```

```
% Copy the first row/col for to the end+1^th row/col
ffts(:, end+1) = ffts(:,1);
ffts(end+1, :) = ffts(1,:);

% Get upper & lower matrix, check point-symmetry:
upper = ffts(1:center(1),    1:end);
lower = ffts(center(1):end, 1:end);
is_conj_symm = upper == conj(rot90(lower, 2));
conj_point_symm = all(is_conj_symm(:));

end
```

6. (C1.3) The Discrete Fourier Transform[1] (DFT) was previously used for frequency analysis of filters. Apart from that, it exhibits some other properties which can be used to speed up certain processing tasks considerably. One of these properties we will consider here is the Fourier shift theorem, which relates a shift in the spatial domain of image i_1 by (x_0, y_0) to a phase term in frequency domain:

$$i_2(x,y) = i_1(x - x_0, y - y_0) \circ\!\!-\!\!\bullet \mathcal{I}_2(\xi, \upsilon) = e^{-j2\pi(\xi x_0 + \upsilon y_0)} \cdot \mathcal{I}_1(\xi, \upsilon) \qquad \text{with } \mathcal{I}_n = \mathcal{F}(i_n)$$

The phase term is isolated using the cross-power spectrum:

$$\mathcal{T} = \frac{\mathcal{I}_1(\xi,\upsilon)\mathcal{I}_2^*(\xi,\upsilon)}{|\mathcal{I}_1(\xi,\upsilon)\mathcal{I}_2^*(\xi,\upsilon)|} = e^{j2\pi(\xi x_0 + \upsilon y_0)} \qquad \text{, where } \mathcal{I}^* \text{ denotes complex conjugate.}$$

By back-transforming $\mathcal{T}$ to the spatial domain, the spatial shift is found using:

$$(t_x, t_y) = \arg\max_{t_x, t_y} \mathcal{F}^{-1}(\mathcal{T}^*)$$

The Fourier shift theorem can help us find a spatial shift of the smaller template created from the original image by cutting out a part of it. Thus, such template can be considered a result of multiplication of the larger image with a 2D rect function (see *template1.png)*. To find the spatial shift in the frequency domain you have to use the formulas given above. You need to pad the template to obtain the same size as the original image. The found coordinates of spatial shift should be returned in ☞ `x,y`. *(8 points)*

```
function [x,y] = find_part_location_image(im, ims)

% Pad the ims with zeros to have the same size with the original image
imt = padarray(ims, size(im)-size(ims), 'post');

% Use the given formula to calculate the spatial shift
G1 = fft2(imt);  G2 = fft2(im);
T = G1 .* conj(G2) ./ abs(G1 .* conj(G2));
t = ifft2(conj(T));
[c,i] = max(t(:));
[y,x] = ind2sub(size(im), i);

end
```

[1]The Fast Fourier Transform (FFT) is an efficient implementation of the DFT for certain block sizes

2 Separable Block Transforms

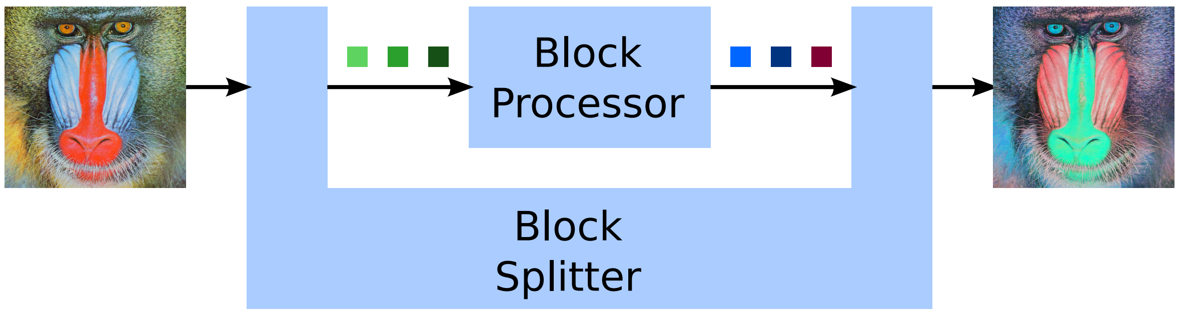


Figure 1: Block-based image processing system

When processing an image with block transformations, you generally need two types of functions: A block splitter, which splits the image into blocks of a predefined size, and a block processor which performs the actual transformation on one block. The block splitter calls the block processor once for each block and stitches the outputs of the block processor back to an image. On multicore processors or graphical processing units (GPU) several block processors can run independently in parallel, providing a simple mean of speeding up processing.
In this problem, separable block transforms of the form $Y = AXA^T$ are considered.

1. (C2.1) Implement a versatile block splitter for any block size with special support for border blocks: If there are not enough pixels at the right or bottom image border to create a full block, pad the block by repeating the last row/column using standard matrix operations. After processing, remove the padding again.
   In the following tasks you will create some block processors. Test your block splitter and each block processor with *lenna_gray.jpg* and different quadratic block sizes. It is recommended to work with function handles (`doc function_handle`) in order to “plug in” any block processor into the block splitter. *(10 points)*

   The block splitter loops over all blocks in the image, addressing the respective top-left element with x_start and y_start. For each block, the processing function is called (using a function handle), and the result is written back to the same position.

```
function image_out = block_splitter_image(image_in, block_size, ⤸
    block_func, block_arg)
% Input arguments:
% image_in - input image
% block_size - scalar size of the 2D block (then the resulting block ⤸
    is of dimensions block_sizexblock_size)
% block_func - function handle to apply on each block, takes as input ⤸
    a block and block_arg and returns processed block of the same size
% block_arg - argument provided to block_func (to support notation AXA⤸
    ^t)
% Output arguments:
% image_out - processed image (put together from processed blocks)

% 1. Pad the input image based on the block size
sz = size(image_in);
pad = block_size - mod(sz, block_size);
pad(pad == block_size) = 0;
image_in = padarray(image_in, pad, 'replicate', 'post');
sz = sz + pad;
```

```
% 2. Allocate output image variable
image_out = zeros(size(image_in));

% 3. Loop over blocks over x,y with block_size and apply block_func on
     each block
for x_start = 1:block_size:(sz(2)-block_size+1)
    for y_start = 1:block_size:(sz(1)-block_size+1)
       block_in  = image_in(y_start:y_start+block_size-1,  x_start:
           x_start+block_size-1);
       block_out = block_func(block_in, block_arg);
       % 4. Save processed block onto output image into appropriate
           part
       image_out(y_start:y_start+block_size-1,  x_start:x_start+
           block_size-1) = block_out;
    end
end
% 5. Cut away padded areas
image_out(end-pad(1)+1:end, :) = [];
image_out(:, end-pad(2)+1:end) = [];

end
```

2. (C2.2) Write a simple block processor function that takes an image block as input and flips the block left to right and decreases the brightness by subtracting 0.3 from each pixel. Make sure to stay in the correct intensity range using clipping. *(5 points)*

```
function block_out = block_processor(block_in)
block_out = fliplr(block_in)-0.3;
block_out(block_out<0) = 0;
end
```

3. (A2.1) Is the operation described in step (2) linear? *(3 points)*

Please note that there are a number of definitions of linearity. In this course we operate the definition of a linear system that is linear if and only if $y_1 = L[x_1]$; $y_2 = L[x_2]$ and $ay_1 + by_2 = L[ax_1 + bx_2]$ holds for all input signals and constants. According to this definition, the clipping operation, which ensures that the image brightness does not go below 0, is non-linear.

4. (C2.3) In this problem, you need to apply the given Haar transform matrix to the image. For this you will implement a function that takes as input an image and Haar matrix and outputs the transformed image. You are given the function handle to the block splitter, so that you do not need to implement it again. Function arguments for the block splitter are the same as in problem (1). *(7 points)*

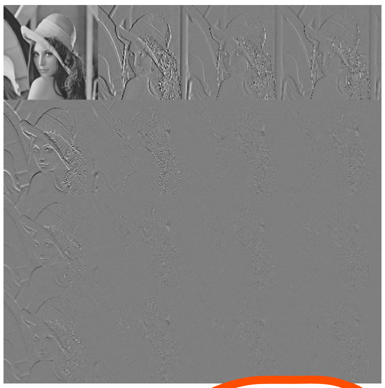
```
function haar_im = filter_image_haar(im, block_size, haar_4,
    fcnHandleBlockSplitter)
```

```
% Describe the block processor
% Haar transform on a block:  A * block_in * A' where A is the
    transform matrix
block_processor_linear = @(block_in, A) A * block_in * A';

% Apply block splitter and block processor with the given
    fcnHandleBlockSplitter
haar_im = fcnHandleBlockSplitter(im, block_size,
    block_processor_linear, haar_4);

end
```

5. (C2.4) The resulting image of a block-based transform is rather meaningless for a human observer. However, when reordering the pixels such that all coefficients from the same basis function are grouped together, we can grasp the properties of the transform much better. Write a reordering routine that groups together the pixels of a Haar-transformed image in this way. For an $N \times N$ transform, you will obtain $N \times N$ sub-images of size $\frac{w}{N} \times \frac{h}{N}$ as illustrated in the lecture (with image dimensions w, h). The order of the sub-images should be the same as in the problem above – hence, the top-left sub-image should be a smaller version of the original image. *(7 points)*

Reordering collects the $(0, 0)$ (usually DC) coefficient from all image blocks and creates a new block out of them, which has $\frac{\textit{image size}}{\textit{block size}}$ elements. The same is done for all other coefficients. The generated image allows an intuitive interpretation of what the different coefficients mean.

```
function im_reorder = reorder_blocks(im, block_size)

sz = size(im);   im_reorder = zeros(sz);
rows = 1:block_size:sz(1);   cols = 1:block_size:sz(2);
lr = length(rows);   lc = length(cols);

for y = 0:block_size-1
    for x = 0:block_size-1
        block = im(rows+y, cols+x); matrix index
        % Block-based scaling
        % block = block - min(block(:));
        % block = block / max(block(:));
        im_reorder(y*lr+1:(y+1)*lr, x*lc+1:(x+1)*lc) = block;
    end
end

end
```



Transformed image after reordering for block size 4

6. (C2.5) When transforming the image using Haar transform, it is useful to analyze statistics for each coefficient over all blocks. Implement a function to calculate mean ☞ `im_mean` and variance ☞ `im_var` for each coefficient over all blocks. The resulting variables should have the same dimensions as a single block. *(9 points)*

   There are multiple possibilities for block-based calculation of means and variances. In the code below, blocks are collected in a 3D matrix, allowing to address each individual block using the third dimension. Like that, it is possible to use MATLAB's `mean` and `var` functions. You need to specify that they operate along dimension 3.

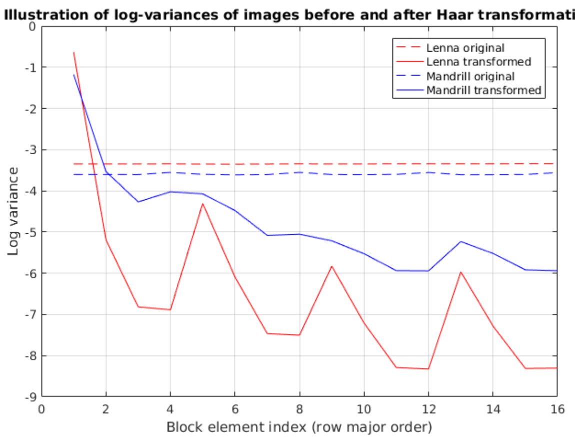
```
function [im_mean, im_var] = calculate_statistics(haar_im, block_size)

    sz = size(haar_im);
    % Pad input image
    pad = block_size - mod(sz, block_size);
    pad(pad == block_size) = 0;
    haar_im = padarray(haar_im, pad, 'replicate', 'post');
    sz = sz + pad;
    image_out = zeros(size(haar_im));
    all_blocks = zeros(block_size, block_size, sz(1)*sz(2)/block_size⤸
        ^2);
    ibl = 0;
    % Loop over blocks
    for x_start = 1:block_size:(sz(2)-block_size+1)
        for y_start = 1:block_size:(sz(1)-block_size+1)
            ibl = ibl + 1;
            block  = haar_im(y_start:y_start+block_size-1, x_start:⤸
                x_start+block_size-1);
            all_blocks(1:block_size, 1:block_size, ibl) = block;
        end
    end
    % Calc stats
    im_mean = mean(all_blocks, 3);
    im_var  = var(all_blocks, [], 3);

end
```

7. (A2.2) What is your general observation upon analyzing `im_mean` and `im_var` from step (6) for lenna image? Verify the energy-preserving property of the Haar transform based on these statistics (see Parseval's theorem). How and why are your results different if you use the Mandrill image (*mandrill_gray.png*)? *(6 points)*

   Transformed Lenna image has more energy concentrated in low-order coefficients (compared to Mandrill).

   For a natural image, the means and variances of the block elements are similar to the global image statistics – all pixels in the block show equal statistics. This may change once a (linear) transformation is applied: Many image transformations, such as the Haar transformation, pack most energy (or variance) into low-order coefficients. The considerable decay in variance from the $(0,0)$ coefficient to higher ones is shown in the plot below.

   As the Haar transform is unitary, signal energy is preserved. In the code below this is verified using the block-based variances.



8. (C2.6) In this problem, implement an image blur as a separable block transform of general block size `n`. The blur applies a symmetric impulse response of $[0.25, 0.5, 0.25]$ to the image horizontally and vertically. Create a sketch of how to express the FIR filter in the AXA^T notion and find the corresponding matrix A (☞ `blur_A`). Hint: remember that the filtered pixel value is a linear combination of itself and its neighbors. *(7 points)*

```
function blur_A = return_blur_A_matrix(n)
  blur_A = 0.5*diag(ones(1,n), 0) + 0.25*diag(ones(1,n-1), -1) + 0.25*
      diag(ones(1,n-1), 1);
end

% now apply blur matrix to the image
n = 16;
blur_A = return_blur_A_matrix(n);
blur_lenna = block_splitter(im, n, @block_processor_linear, blur_A);
imshow(blur_lenna);
```

9. (A2.3) Apply the previously computed filter `blur_A` to lenna image. What do you observe? Discuss the fundamental limitations of AXA^T notation used in the previous problem along the block borders. What is your suggestion to address the observed problems? *(6 points)*

We observe that the image has been blurred, i.e. low-pass filtered so that fine details in the image cannot be observed anymore. Smoothing is a filtering operation with a FIR-filter which requires pixels in a local neighborhood. However, the block processor does not have access to neighbor pixels located outside the borders of a block. This is why border effects (e.g. dark frame) are observed around blocks. Similar effects can be observed with block-based compression algorithms (JPEG, MPEG). In order to mitigate the border problems, one could use overlapping blocks, at the cost of performance.