Discussion of Homework 1
Image Manipulation and Periodic Sequences

2020-14-5

- Submission: Upload your code on Cody Coursework™ 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™.
- Notation: `variable name`, *`file name`*, `MATLAB_function()`

1 Image manipulation

1. (A1.1) Load image *`pears.jpg`* to variable `im` and analyze it: Determine data-type, dimension and plot the color histogram of the image. Calculate its minimum, maximum, mean and standard deviation. How do you interpret these values? *(4 points)*

```
im = imread('data/pears.jpg');
datatype = class(im);     sz = size(im);
im_min = min(im(:));      im_max = max(im(:));
im_mean = mean(im(:));    im_stddev = std(double(im(:)));
imhist(im);
```

Variable `sz` indicates width and height of the image in pixels (or rows / columns of the matrix `im`). If the .jpg-file was a color image, there would be a third element `sz(3)==3`, indicating 3 color channels (red, green, blue). The image is returned as uint8 datatype, e.g "unsigned integer of 8 bit length". An intensity value of 0 corresponds to black, $2^8-1=255$ corresponds to white. What conclusion can you make from observing the histogram? The grayscale histogram is not completely balanced and shows that the image has been slightly underexposed, therefore more pixel intensities are concentrated in low pixel intensity region. This is also confirmed by the mean value and standard deviation. There are some clipping effects at the lower (black) range and no clipping effects on the upper (white) one. See `http://www.cambridgeincolour.com/tutorials/histograms1.htm` for a discussion of image histograms. Interpretation of the mean value: it corresponds the average intensity of the image for the respective color channel.

2. (A1.2) Now copy `im` to `imd`, and convert the data-type of `imd` to *double*, which is MATLAB's default type required by many built-in functions. Also, by definition, image data of this type must lie in the range $[0;1]$. Use this convention as the default for all your homework submissions. Display `imd` using at least 3 different MATLAB functions. What is the difference if you display `im` instead of `imd` using `imshow()`? *(3 points)*

```
imd_255 = double(im);
imd = (1/255) * imd_255;
figure(1);  imshow(imd);          % Works; range 0-1, type double
figure(2);  imshow(im);           % Works: range 0-255, type uint8
figure(3);  imshow(imd_255);      % Fails; range 0-255, type double
figure(4);  colormap(gray(256));  imagesc(im/16);
figure(5);  colormap(jet(256));   imagesc(imd);
figure(6);  colormap(gray(256));  image(imd_255);
% Image as 3D surface
```

```
figure(7);  colormap(gray(256));
surface([0 1], [0 1], 0.5*ones(2), im, 'FaceColor','texturemap', '
    CDataMapping','direct');
% Image viewer app
imtool(imd);
```

What is the difference displaying `im` *and* `imd` *using* `imshow()`*? There is no difference. This is due to the fact that function* `imshow()` *automatically recognizes the type of the image and based on this devises the correct range for display (either 0..1 for float or 0.255 for uint8).*

3. (C1.1) Implement a function to draw a 2 pixel wide black border within the *double*-typed image, using sub-matrix addressing. Do not change its size. Verify your implementation using hidden and visible tests at Cody Coursework platform. *(5 points)*

```
function imd_frame = im_add_frame(im)
frame_color = 0;
imd_frame = im;
imd_frame([1:2, end-1:end], :) = frame_color;
imd_frame(:, [1:2, end-1:end])  = frame_color;
end
```

4. (A1.3) Cody Coursework platform accepts a tolerance of 0.01 for the values of `imd_frame`. Add Gaussian noise with $\sigma = 0.005$ to that image and retry the tests. How many pixels (matrix elements) do you expect to lie outside the tolerance range? *(5 points)*

The probability of noise values outside the tolerance ± 0.01 *can be calculated using the complementary error function* $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, dt$*. In this case, working with the normal distribution, we have* $x = \frac{0.01}{\sigma\sqrt{2}}$*. The experiment exhibits results close to the statistical expectation.*

5. (C1.2) Implement a function to copy each second row and each third column (both starting at index 1) of the (noiseless) framed image and output it. *(4 points)*

```
function im_part = im_cut_part(im)
im_part = im(1:2:end, 1:3:end);
end
```

6. (A1.4) Draw a red frame around the image you just displayed with the `plot()` function. How is this approach different to changing pixel values in `im`? Try out different options of the `plot()` function and add some text to the image with the `text()` function. *(4 points)*

`plot()` *does not change the image data. It adds vector graphics to the figure, which can be rendered at any resolution and modified with the* `set()` *function.* `plot()` *is not constrained to integer pixel positions.*

```
figure;
imshow(imd);
hold on;    % Allows to add more elements to figure
e = 30;
X = [1+e sz(1)-e sz(1)-e 1+e 1+e];        % quadrangle
Y = [1+e 1+e sz(2)-e sz(2)-e 1+e];
handle = plot(X, Y, '--ro', 'LineWidth', 4);
```

```
pause(3);        % Change plot object after 3 seconds:
set(handle, 'Color', 'green');
set(handle, 'XData', X+5);
set(handle, 'YData', 0.8*Y-15);
```

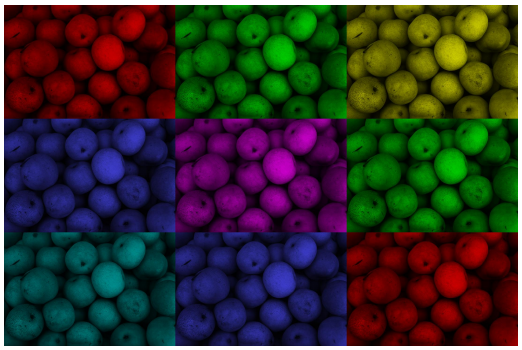
7. (C1.3) In the following, you will practice sub-matrix addressing and use the Kronecker product to create colorized copies of one grayscale image, similar to "Warhol-style" images.
Use the grayscale image `imd` from above as the input image. The output will be a color image of larger size. Calculate each color channel separately, using:

$$imk_i = c_i \otimes im \qquad \forall i \in \{R, G, B\} \text{ with}$$

$$c_R = \begin{bmatrix} 1 & 0 & 1 \\ 0.3 & 1 & 0 \\ 0 & 0.3 & 1 \end{bmatrix}, \qquad c_G = \begin{bmatrix} 0 & 1 & 1 \\ 0.3 & 0 & 1 \\ 0.7 & 0.3 & 0 \end{bmatrix}, \qquad c_B = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 0 \\ 0.7 & 1 & 0 \end{bmatrix}$$

Merge these channels to a color image and analyze it by showing on the screen. Do not use the `kron()` function – rather use sub-matrix addressing and for-loops stepping through all elements of $c_{R,G,B}$. (Usually, you should avoid for-loops in MATLAB for better performance.) *(7 points)*

```
function im_kron_rgb = merge_channels_kronecker(im)
cR = [ 1 0 1 ;   0.3 1 0;   0   0.3 1 ];
cG = [ 0 1 1 ;   0.3 0 1;   0.7 0.3 0 ];
cB = [ 0 0 0 ;   1   1 0;   0.7 1   0 ];
sz = size(im);
im_k_R = zeros(3*sz);  im_k_G = zeros(3*sz);  im_k_B = zeros(3*sz);
for r = 1:3
    for c = 1:3
        range_r = (r-1)*sz(1)+1 : r*sz(1);
        range_c = (c-1)*sz(2)+1 : c*sz(2);
        im_k_R(range_r, range_c) = cR(r,c) * im;
        im_k_G(range_r, range_c) = cG(r,c) * im;
        im_k_B(range_r, range_c) = cB(r,c) * im;
    end
end

im_kron_rgb = zeros([3*sz 3]);
im_kron_rgb(:,:,1) = im_k_R;
im_kron_rgb(:,:,2) = im_k_G;
im_kron_rgb(:,:,3) = im_k_B;
end
```



2 Periodic Sequences

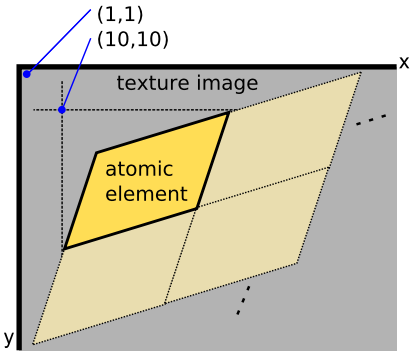


Figure 1: Placing and repetition of the atomic element in the original texture image

1. (C2.1) In this problem you will need to write a function to compute atomic image elements from the texture image `texture1.jpg`. The atomic element is fully specified by the periodicity matrix N, the texture image, and its location (x, y). In (x, y), x is the location of the atomic element's leftmost pixel within the texture image, and y is the location of the topmost pixel, see Figure 1. Make sure the matrices for your atomic elements are of the minimal required size (with the given periodicity matrix). Verify that you create correct atomic image elements for $(x, y) = (1, 1)$ (MATLAB coordinates), $(x, y) = (10, 10)$ and $(x, y) = (50, 20)$ with following periodicity matrices:

$$N_A = \begin{bmatrix} 200 & 0 \\ 0 & 200 \end{bmatrix}, \qquad N_B = \begin{bmatrix} 200 & 200 \\ 0 & 200 \end{bmatrix}, \qquad N_C = \begin{bmatrix} 200 & 200 \\ -200 & 200 \end{bmatrix}.$$

The function should return atomic image elements and masks (if used) for the given pair (x, y) as `double` images[1].
Hints: You should work with binary image masks for non-rectangular elements. Create the required parallelograms using MATLAB's `poly2mask()` function (see `http://de.mathworks.com/help/images/ref/poly2mask.html`). When using `poly2mask()` pay attention that it uses a sub-pixel grid to decide which pixels are inside the specified periodicity matrix-based polygon. See Figure 2 for illustration. *(18 points)*

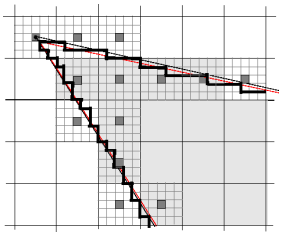


Figure 2: Illustration how `poly2mask()` determines which pixels are inside the region (taken from `de.mathworks.com`)

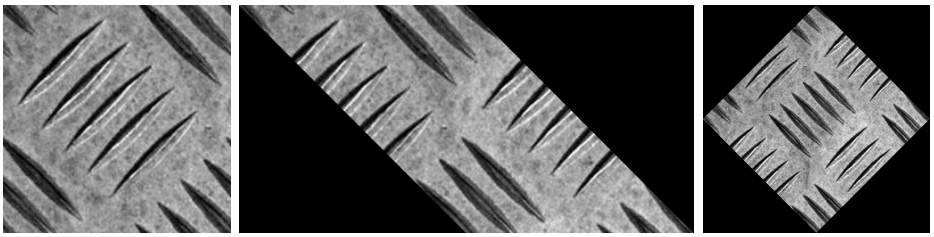
> *The 2 columns $n_{1,2}$ of $N_\star$ define the periodicity of the signal. That means the generated periodic signal must be equal at positions $(x, y) + k \cdot n_1 + l \cdot n_2 \quad \forall x, y, k, l$. For instance, for N_B, the top-left element (pixel) of the atomic element must be repeated at*

[1] When saving a `double` image, make sure to use the correct scaling!

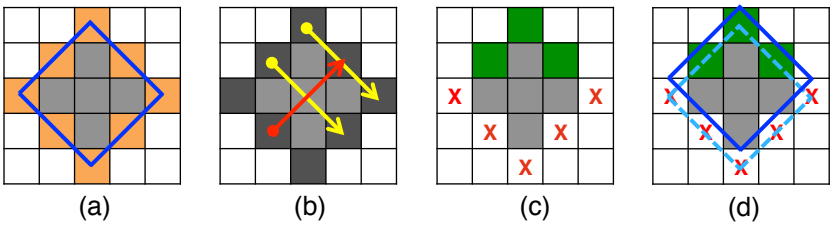
$(10, 10), (210, 10), (210, 210), (410, 210), (410, 10), \ldots$ *The atomic element is derived from the parallelogram spanned by* $n_{1,2}$. *Its size (= number of active pixels in the mask) must be equal to* $det(N_\star)$, *and its shape must ensure that no self-overlapping occurs. When discretizing the parallelogram to a matrix, there are several, slightly different solutions at the border.*
The atomic elements can be defined as follows: First, as shown in the code below, a rectangle enclosing the parallelogram can be used. The valid pixels are defined by a mask of equal size. Second, all the pixels within the parallelogram could be written into a 1D vector, using additional index vectors which keep the spatial information.

```
imd = im2double(imread('data/texture1.jpg'));
loc = [ 10 10 ]; % loc = [ 1 1 ]; loc = [ 50 20 ];
N_A = [ 200 0;      0 200 ];
N_B = [ 200 200;    0 200 ];
N_C = [ 200 200; -200 200 ];
[atom_A, atom_mask_A] = get_atomic(imd, N_A, loc');
[atom_B, atom_mask_B] = get_atomic(imd, N_B, loc');
[atom_C, atom_mask_C] = get_atomic(imd, N_C, loc');
```

```
function [atomic, mask] = get_atomic(im_ref, pm, loc_base)
% function to get atomic element for a given image im_ref
% Input arguments:
% loc_base - starting location
% pm - periodic matrix
% Output arguments:
% atomic - atomic element of the image
% mask - corresponding mask for the atomic element

% Define polygon:
poly = zeros(2,5);                  poly(:,1) = loc_base;
poly(:,2) = poly(:,1) + pm(:,1);    poly(:,3) = poly(:,2) + pm(:,2);
poly(:,4) = poly(:,3) - pm(:,1);    poly(:,5) = poly(:,4) - pm(:,2);

% Create mask for atomic element (parallelogram)
poly = poly + repmat([0;0.4] - min(poly,[], 2), 1, 5);
sz_mask = round(max(poly,[],2));
mask = poly2mask(poly(1,:), poly(2,:), sz_mask(2), sz_mask(1));

% Create atomic element
atomic = im_ref(loc_base(2):loc_base(2)+sz_mask(2)-1, loc_base(1):↩
    loc_base(1)+sz_mask(1)-1);
atomic = atomic .* mask;
end
```



Atomic elements for N_A, N_B, N_C *with masks applied*



*(a) The shape of the atomic element (which is defined by the periodicity vectors) needs to be discretized. Along its borders, pixels overlap only partly with the shape. (b) If all pixels touched by the shape are added to the mask, the resulting mask will be too large, and for the active area (= number of 1-pixels) $a > det(N_\star)$. Also, when applied to any element within the mask, the periodicity vectors must point outside the mask. (c) One possible solution for a valid discretized mask. When this mask is periodically repeated, it fills the entire space without gaps. (d) MATLAB's `poly2mask()` can create the depicted mask if **all** corner points are shifted by a small amount.*

2. (C2.2) Here, you would need to implement an inverse operation: generate a periodic image for the given atomic element. For this you are given as input arguments an atomic element, atomic mask, periodicity matrix, base location and size of the image. The size of the periodic image should be at least the same size as the original texture image. *(18 points)*

Periodic repetition is performed by `create_periodic()`.

```
function [im_out] = create_periodic(im_atomic, im_mask, pm, loc_base, ...
    size_output)
% Create periodic image
%% Input arguments:
% im_atomic - atomic element of the image
% im_mask - mask corresponding to the atomic element
% pm - periodicity matrix
% loc_base - base location (x,y)
% size_output - needed size of the output
%% Output arguments
% im_out - output periodic image
% im_mask_full - full mask of the image

im_out = zeros(size_output);
im_mask_full = zeros(size_output);
sz = fliplr(size(im_out))';

% Calculate required range for nx, ny
p_ex_ = [ 1 1 ; sz(1) sz(2) ; sz(1) 1 ; 1 sz(2) ]' - repmat(loc_base, ...
    1, 4);
p_ex = inv(pm) * p_ex_;
nx_range = floor(min(p_ex(1,:))-1):ceil(max(p_ex(1,:))+1);
ny_range = floor(min(p_ex(2,:))-1):ceil(max(p_ex(2,:))+1);

% "Copy-Paste" the atomic element multiple times, add to output image
for nx = nx_range
    for ny = ny_range

        % Calculate pixel positions top-left and bottom-right:
        loc = loc_base + nx * pm(:,1) + ny * pm(:,2);
        loc2 = loc + flipud(size(im_atomic)') - 1;
```

```
        % All positions this repetition addresses:
        all_x = loc(1):loc2(1);
        all_y = loc(2):loc2(2);
        atomic_x = 1:size(im_atomic,2);
        atomic_y = 1:size(im_atomic,1);

        % Remove cols/rows outside matrix:
        valid_x = and(all_x ≥ 1, all_x ≤ sz(1));
        valid_y = and(all_y ≥ 1, all_y ≤ sz(2));

        %if isempty(valid_x) || isempty(valid_y); continue; end
        if all(¬valid_x) || all(¬valid_y); continue; end

        % Copy the atomic element with transparency mask
        at_tran = im_atomic(atomic_y(valid_y), atomic_x(valid_x));
        im_out(all_y(valid_y), all_x(valid_x)) = im_out(all_y(valid_y)↩
            , all_x(valid_x)) + at_tran;

        % And create sum of masks (should be all 1)
        im_mask_full(all_y(valid_y), all_x(valid_x)) = im_mask_full(↩
            all_y(valid_y), all_x(valid_x)) + ...
             im_mask(atomic_y(valid_y), atomic_x(valid_x));
    end
end

assert(all(im_mask_full(:)==1), 'Your resulting mask is not all ones -↩
    you incorrectly sum something');
end
```

In the following and future problems, the normalized cross correlation (NCC) between two matrices i_1,i_2 is used to measure the similarity between two images. The coefficient is a scalar and generally calculated as follows:

$$\mathrm{ncc}(i_1, i_2) = \frac{1}{N} \sum_{x,y} \frac{(i_1(x,y) - \overline{i_1})(i_2(x,y) - \overline{i_2})}{\sigma_{i_1} \sigma_{i_2}} \tag{1}$$

With N — number of elements, $\overline{i}$ — mean, σ — standard deviation. Note that some MATLAB functions evaluate the NCC over a shift parameter by default.

3. (C2.3) Write a function to calculate NCC between two equal-sized periodic sequences (images in this case). Use as little for-loops as possible. Refrain from using the built-in MATLAB functions for computation of cross-correlation. *(8 points)*

```
function ncc = get_ncc(t1, t2)
sz = size(t1);
t1_norm = (t1 - mean(t1(:))) / std(t1(:));
t2_norm = (t2 - mean(t2(:))) / std(t2(:));
ncc = 1/(numel(t1)) * (t1_norm .* t2_norm);
ncc = sum(ncc(:));
end
```

4. (C2.4) In this task you will write a function to find the atomic element of a given naturally periodic texture image using exhaustive search and correlation. Work with image ***texture2.jpg*** with $(x, y) = (1, 1)$ and search for quadratic atomic elements of edge length $s \in [1, 300]$ pixels,

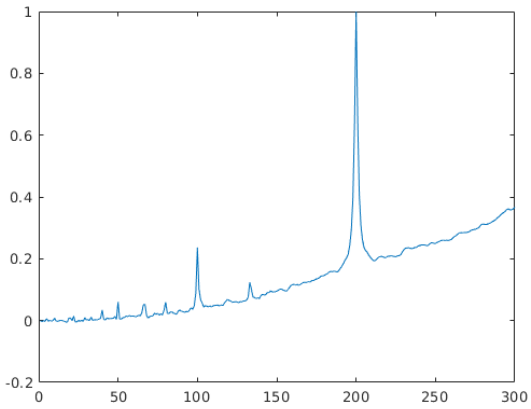
using the following steps:

- Create a quadratic atomic element of edge length s
- Generate a periodic image from the atomic element
- Compare the generated periodic image to the original image using NCC
- Repeat until all edge lengths are tested

Write a function that computes NCCs and returns them in ☞ `ncc_range` (length: 300 elements). Also, the function has to search for (the correct) local maxima in `atom_ncc` and save the corresponding edge lengths to ☞ `ncc_max_loc`. Check the correctness of your results by inspection. *(19 points)*

```
function [ncc_range, ncc_max_loc] = compute_ncc_find_max(im, range, ↩
    fcnHandleNCC)
% Input arguments:
% im - given image (of type double in range 0..1)
% range - vector containing elements from 1 till 300
% fcnHandleNCC - function handle to "get_ncc(i1,i2)" -> see C2.3
% Output arguments:
% ncc_range - vector of NCC values for the edge lengths from "range"
% ncc_max - location of maximum in NCC

% Compute ncc in the given range
sz = size(im);
bx = 1;  by = 1;
ncc_range = zeros(1, max(range));
for si = 1:length(range)
    s = range(si);
    if (by+s > sz(1) || bx+s > sz(2)); continue; end

    % Cut atomic element
    atomic = im(by:by+s-1, bx:bx+s-1);

    % Create periodic image
    nr = ceil(max(sz) / s);
    im_per = repmat(atomic, nr, nr);
    im_per = im_per(1:sz(1), 1:sz(2));

    % Calc NCC   - for full image:
    i1 = im_per; i2 = im;
    ncc_range(s) = fcnHandleNCC(i1,i2);
end

% Find maximum and save the numbers you get
[¬, ncc_max_loc] = findpeaks(ncc_range), 'MINPEAKHEIGHT', 0.8);
end
```

The code creates quadratic atomic elements for size $s \in [1, 300]$ and generates periodic images from them. The similarity between the generated image and the original texture is measured by NCC, which this time depends on an additional parameter, the size of the atomic element s. First, the NCC is measured using the entire generated image. This introduces a bias, as the first atomic element – which is of course equal to the original texture – is also considered. This is why in the second variant, the NCC is only based on the newly generated part of the image. In order to find maxima in $NCC(s)$, a fixed threshold and a local non-maxima suppression is used. The code finds a clear maximum with a magnitude of ≈ 0.8 for edge length 200. If the NCC is solely based on the generated parts of the image, the maxima come out much better.



5. (A2.1) Why do you observe several maxima in the NCC values of the whole image? How can this be explained by the image properties? *(5 points)*

   Because the considered texture has partial repetitive parts, thus part of it can be matched to the whole. Most importantly, to avoid bias in NCC calculation it is preferrable to use only the newly generated part of the image for NCC computation.