# Image Processing - 67829

# Exercise 5: Super Resolution

Due date: 24/01/2016

## 1 Overview

In this exercise you will be guided through the steps we discussed in class to perform Super Resolution from a Single Image. This exercise is based on the example-based part from "Super-Resolution from a Single Image" (Glasner et al.) and on experiments that the student Danny Barash performed using this method. In this exercise, you will implement a super resolution of an image by a factor of two. Another goal of this exercise (except of implementing the super resolution method) is to get familiar with techniques for accelerating your matlab code.

## 2 Background and Dependencies

You may find it helpful to review the material that was presented in Tirgul 10.

## 3 Image Pyramid Creation

As you learned in class, the first step will be to create a pyramid from the input image. Unlike the pyramid you implemented in `Exercise 3` where at each level we downsampled the image by a factor of 2, here we are going to downsample the image at each level by a factor of $2^{\frac{1}{3}}$ such that in case of 7 levels pyramid, the ratio between the biggest image to the smallest level is 4.

You are required to implement a function that gets an image and returns a pyramid with 7 levels, where the $4^{th}$ level is the input image, and the ratio between every two adjacent levels of the pyramid is $2^{\frac{1}{3}}$. `pyr{1}` is the smallest level and `pyr{7}` is the biggest one - see Fig. 1.

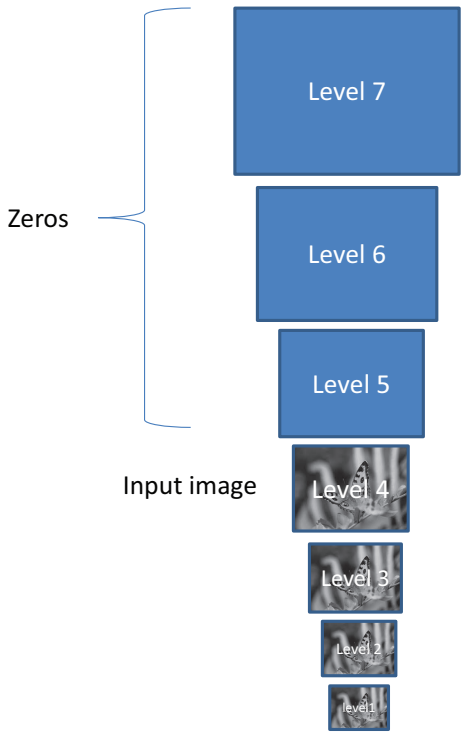



Figure 1: The 7-levels pyramid

Our goal is to find `pyr{7}` which its size is two times bigger than the input image. In order to get levels 3,2 and 1 you should use Matlab's function `imresize`.

Pyramid creation should be implemented in the `createPyramid` function having the following interface.

```
function pyr = createPyramid( im )
% CREATEPYRAMID Create a pyramid from the input image, where pyr{1} is the smallest level,
% pyr{4} is the input image, and pyr{5},pyr{6},pyr{7} are zeros.
% The ratio between two adjacent levels in the pyramid is 2^(1/3).
% Arguments:
% im - a grayscale image
%
% outputs:
% pyr - A 7 × 1 cell of matrices.
```

# 4 Patches sampling

In order to increase the image size we need to search for each patch in the input image (the $4^{th}$ level of the pyramid) similar patches in the lower levels of the pyramid. In other words, find correspondence between low and high resolution image patches. In this exercise we use $5 \times 5$ patches and the patches should overlap other patches as demonstrate in Fig. 2.

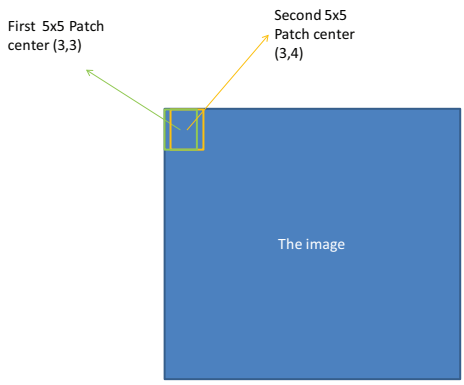



Figure 2: Overlapping between sampled patches

Patches sampling should be implemented in the `samplePatches` function having the following interface.

```
function [p_x, p_y, patches] = samplePatches( im , border )
% SAMPLEPATCHES Sample 5 × 5 patches from the input image. You are allowed to use 2D loops here.
% Arguments:
% im − a grayscale image of size m × n
% border − An integer that determines how much border we want to leave in the image. For example: if border=0 the
    center of the first patch will be at (3,3), and the last one will be at (end−2,end−2), so the number of patches in this
    case is (m − 4) × (n − 4). But if border=1 the center of the first patch will be at (4,4) and the last one will be at (
    end−3,end−3). So in general, the number of patches is (m − 2 · (2 + border)) × (n − 2 · (2 + border)).
%
% outputs:
% p_x − (m − 2 · (2 + border)) × (n − 2 · (2 + border)) matrix with the x indices of the centers of the patches
% p_y − (m − 2 · (2 + border)) × (n − 2 · (2 + border)) matrix with the y indices of the centers of the patches
% patches− (m − 2 · (2 + border)) × (n − 2 · (2 + border)) × 5 × 5 the patches
```

# 5 Create patches DB

We want to create a database of patches so we will be able to search for patches that are similar to a patch in the original image. To do that, we need to transform the pyramid levels 1,2 and 3 into patches. When we sample the patches for the DB we need to avoid taking patches from the border area otherwise we will get into troubles during the process. Using `border=2` should be enough. In this part you are encouraged to use matlab's function `reshape`.

DB creation should be implemented in the `createDB` function having the following interface.

```
function [p_x, p_y,levels, patches] = createDB( pyr )
% CREATEDB Sample 5 × 5 patches from levels 1,2,3 of the input pyramid.
% N represents the number of patches that are found in the three images.
```

```
% Arguments:
% pyr − 7 × 1 cell created using createPyramid
%
% Outputs:
% p_x − N × 1 vector with the x coordinates of the centers of the patches in the DB
% p_y − N × 1 vector with the y coordinates of the centers of the patches in the DB
% levels − N × 1 vector with the pyramid levels where each patch was sampled
% patches− N × 5 × 5 the patches
%
```

# 6 Find Nearest Neighbors

Given the sampled patches from the input image (`pyr{4}`) and the DB patches, we can find for each patch from the input image, `k` candidates that are most similar to the patch in lower resolution images. Let's call these patches *child patches*. Here we are going to use Matlab's function `knnsearch` with `k=3`. Some comments:

- For using the function you need to convert each patch to a vector in $\mathbb{R}^{25}$ using the function `reshape`.
- `knnsearch` returns two parameters: `[IDX,D]` where `IDX` are the coordinates of the nearest neighbors in the DB of each input window. '`D`' are L2 norm between the input patches to their child patches (Euclidean distance over the intensities).

Finding child patches should be implemented in the `findNearestNeighbors` function having the following interface:

```
function [idx,Dist] = findNearestNeighbors( imPatches, dbPatches )
% FINDNEARESTNEIGHBORS Find the 3 nearest neighbors for each patch in the input images from the patches in the
    DB
% N represents the number of patches in the DB, and M represents the number of patches in the input image.
% Arguments:
% imPatches − M × 5 × 5 matrix with M patches that were sampled from the input image (pyr{4})
% dbPatches − N × 5 × 5 matrix with N db patches
%
% Outputs:
% idx − M × 3 matrix where the i^th row has 3 indices of the 3 patches in the db that are most similar to the i^th patch
    from the input image
% Dist− M × 3 matrix where the i^th row contains the euclidean distances between the best 3 patches that has been
    found for the i^th patch
%
```

# 7 Thresholds and Weights

Given child patches for each input patch we want to give them weight.

## 7.1 Threshold selection

Each patch from the input image has $k = 3$ child patches. However, some of the $k = 3$ child patches may be quite different from the input patch. To avoid using such patches we set a threshold for the acceptable difference between a patch and a child patch. This threshold is the difference between a patch and its translation by 0.5 pixel.

For simplicity we supplied a function that calculates the translated versions of an input image by 0.5 pixel. The function `translateImageHalfPixel` gets an image and returns two images: `translatedX,translatedY`. The threshold for each input patch will be the average between the Euclidean distance (L2 norm) of the current patch to the same patch in `translatedX` and `translatedY`. Therefore, in order to calculate the threshold for each input patch, the following steps should be done (without loops at all):

- Get the two translated images from the input image using the supplied `translateImageHalfPixel`
- Sample patches from each image using yours `samplePatches` with `border=0`
- Calculate the euclidean distance between each input patch to its two translated versions.
- Set threshold to be the average between the two calculated distances.

## 7.2 Weighting matches

For each child patch we give a weight. The weights should be high for child patches that are more similar to the input patch, and low for child patches that are less similar. In addition, the weight of the third child patches will be set to zero if its distance is above the threshold computed before.

Given a distance $D_{i,j}$ (the Euclidean distance between patch i and the child patch j), the weight is defined as follows:

$$W_{i,j} = \begin{cases} e^{-\frac{D_{i,j}^2}{\sigma_j}}, & \text{if } D_{i,j} \leq T_i \vee j = 1 \vee j = 2 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

Where $T_i$ is the threshold mentioned above for the $i^{th}$ input patch and $\sigma_j$ is the standard deviation of child patch j (we increase the weight of child patches that have high variance). Please notice that we always take the two most similar child patches, since we want at least 2 candidates for each input patch. The third patch will be taken into account only when its distance is below the threshold.

Setting weights should be implemented in the `weightsSetting` function having the following interface:

```
function [weights] = weightsSetting( imPatches, Dists, pyr ,dbPatchesStd )
% WEIGHTSSETTING Given 3 nearest neighbors for each patch of the input image
% Find a threshold (maximum distance) for each input patch.
% Next, give a weight for each candidate based on its distance from the input patch.
% denote m,n such that [m,n]=size(pyr{4})
% Arguments:
% imPatches - (m-4) × (n-4) × 5 × 5 matrix with the patches that were sampled from the input image (pyr{4})
% Dists - (m-4) × (n-4) × 3 matrix with the distances returned from findNearestNeighbors.
% pyr - 7 × 1 cell created using createPyramid
% dbPatchesStd - (m-4) × (n-4) × 3 matrix with the STDs of the neighbors patches returned from
    findNearestNeighbors.
%
% Outputs:
% weights - (m-4) × (n-4) × 3 matrix with the weights for each DB candidates
%
```

# 8 Sampling centers

In the previous steps, we found for each input patch similar child patches in lower resolution patches. We should like to use the higher resolution sources of these similar patches (parent patches). This process is displayed in Fig. 3.

We denote as a *child patch* the nearest neighbor patch in lower resolution that the yellow arrow is pointed at, and a *parent patch* as the higher resolution version of the child patch, that the red arrow is pointed at (see Fig. 3 for more details).

Please notice that the centers of the *child patches* were found in `findNearestNeighbors` function. Each center is represented by 3 parameters: `xCoord,yCoord,level`. In order to find their *parent patches* centers, you should use the supplied function `transformPointsLevelsUp` which takes 3 matrices that represent the 3 parameters of the *child patches* center, and transform their locations to an upper level in the pyramid, i.e returns the *parent patches* location.

Note that the *parent patches* can be located in various pyramid levels. In order to improve the running time we need to sample all windows from the same image. To do so, you are supplied with a function `renderPyramidEx5` that renders the entire pyramid in a single large image. The result is similar to the version you implemented in ex3, but can work with the current type of pyramid. Now we can sample all the required patches from one image.

Your job in this part of the exercise is:

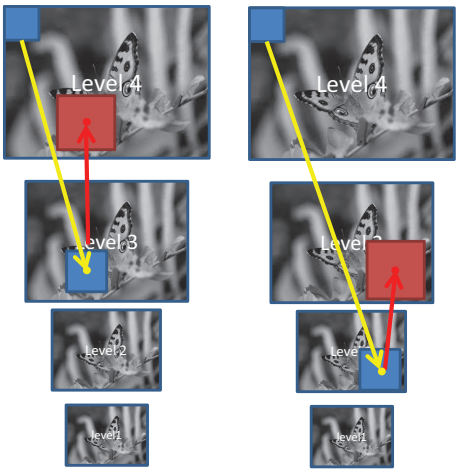



Figure 3: The yellow arrows point from an input patch to a location in the pyramid where we found a lower resolution patch that is similar to the input one (*child patches* center). This patch can be found either one level down like in the left example, or two levels down like in the right example (and can be found three levels down). In all cases we need to sample a window that is located one or more levels up - the points where the red arrows are pointed to (*parent patches* center). In this example we sample a window from one level up, but it can be more than one - see section 11.1.

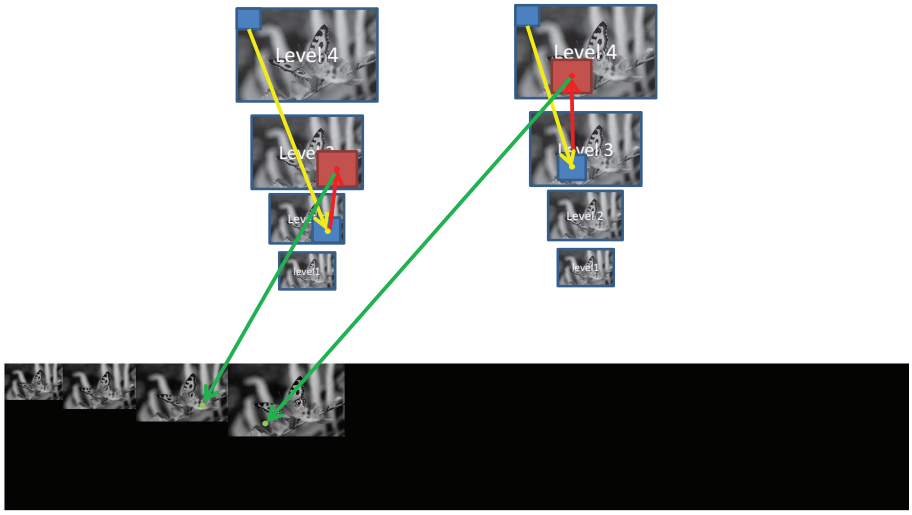



Figure 4: Visualizing the process of the function `getSamplingCenters`

- Render the pyramid to one large image using `renderPyramidEx5`.
- For each patch in the input image, you are getting the centers of its child patches (`xCenters, yCenters, centersPyrLevel`) - resulted from `findNearestNeighbors` function.
- Find the locations of the centers of the corresponding *parent patches* in the higher resolution image using the supplied function `transformPointsLevelsUp`. Note that the centers of the parent patches do not have integer coordinates.
- Use the centers of the *parent patches* to get their locations in the rendered pyramid (the "green points" in Fig. 4.). Note we do not need a pyramid level after this step. We call these points *sampling centers*.

The whole process is shown in Fig. 4.

Computing the sampling centers should be implemented in the `getSamplingCenters` function having the following interface:

```
function [sampleCentersX,sampleCentersY,renderedPyramid] = getSamplingCenters( xCenters, yCenters, centersPyrLevel
    , pyr, levelsUp )
% GETSAMPLINGCENTERS Given 3 nearest neighbors for each patch of the input image, from the patches DB,
% find the location of parent patch in the rendered pyramid image
```

```
% Arguments:
% xCenters - (m-4) × (n-4) × 3 matrix with the x coordinates of the closest patches (child patches) to each sampled
    patch from the image
% yCenters - (m-4) × (n-4) × 3 matrix with the y coordinates of the closest patches (child patches) to each sampled
    patch from the image
% centersPyrLevel - (m-4) × (n-4) × 3 matrix with the levels of the closest patches to each sampled patch from the
    image
% pyr - 7 × 1 cell created using createPyramid
% levelsUp - integer which tells how much levels up we need to sample the parent patch, from the found patch. In the
    figure the case is levelsUp=1.
%
% Outputs:
% sampleCentersX - (m-4) × (n-4) × 3 matrix with the x coordinates of the center of the patches in the rendered
    image (the green points in the figure)
% sampleCentersY - (m-4) × (n-4) × 3 matrix with the y coordinates of the center of the patches in the rendered
    image (the green points in the figure)
% renderedPyramid - a single image containing all levels of the pyramid
%
```

# 9 Patch sampling coordinates

Now we can build the high resolution image using patches from (`renderedPyramid`). For simplicity we will sample only $5 \times 5$ patches from the high resolution images (*parent patches*). The procedure is shown in Fig 5.

For each center of an input patch (in the input image) we have to compute the location of its center in the newly computed higher resolution image using `transformPointsLevelsUp`. Since the pixel coordinates at the higher resolution image `upPixelX,upPixelY`, are not integers, they should be rounded. Next, we want to bring the *parent patches* into this $5 \times 5$ higher resolution patch using the sampling centers from the previous part. As both the coordinates of the sampling centers and the original coordinates of the higher resolution patch are not integers, we need to perform backwarping from the parent patch. This requires the following steps:

- Create two coordinate matrices: a $5 \times 5$ matrix with x coordinates, and a $5 \times 5$ matrix with y coordinates that correspond to the higher resolution patch (patch around the rounded values of `upPixelX,upPixelY`, the orange patch in (c) in figure 6). Lets denote these two matrices as: `assignmentPositionsX` and `assignmentPositionsY` respectively. You should use the function `meshgrid` for that.

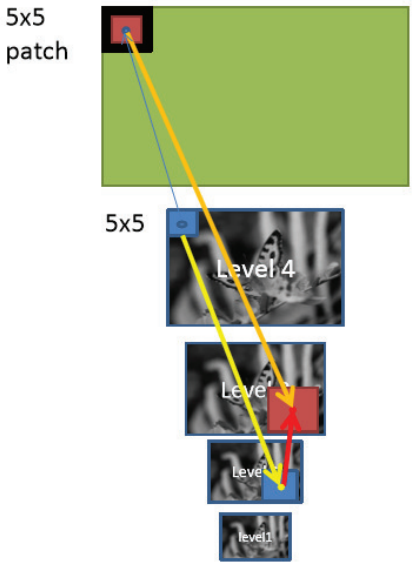



Figure 5: Building high-resolution image

- Subtract `upPixelX` from the rounded values in `assignmentPositionsX` and `upPixelY` from `assignmentPositionsY` (demonsrate in (d) in figure 6). This is the shift of the high resolution patch with rounded coordinates from the original non-integer coordinates.
- Add the `sampleCentersX` to the x locations, and the `sampleCentersX` to the y locations - these are the green centers from the previous step. Let's call the result `samplingPositionsX` and `samplingPositionsY`
- in order to compute the high resolution patch, We got now $5 \times 5$ x-y coordinates into the rendered pyramid image. These are non-integers coordinates.

The procedure is described in Fig 6, .

At the end of this function we have the information for creating a high resolution image: we have pairs of (i) x-y coordinates in the high resolution image (integers) (*assignment positions*), and (ii) x-y coordinates (non-integer) in the rendered pyramid image (*sampling positions*).

Note: this function should be implemented without loops. The above procedure should be implemented in the `getSamplingInformation` function having the following interface:

```
function [assignmentPositionsX,assignmentPositionsY,samplingPositionsX,samplingPositionsY] = getSamplingInformation
    (sampleCentersX,sampleCentersY,pyr,inputPatchesCentersX,inputPatchesCentersY,levelsUp)
% GETSAMPLINGINFORMATION
```

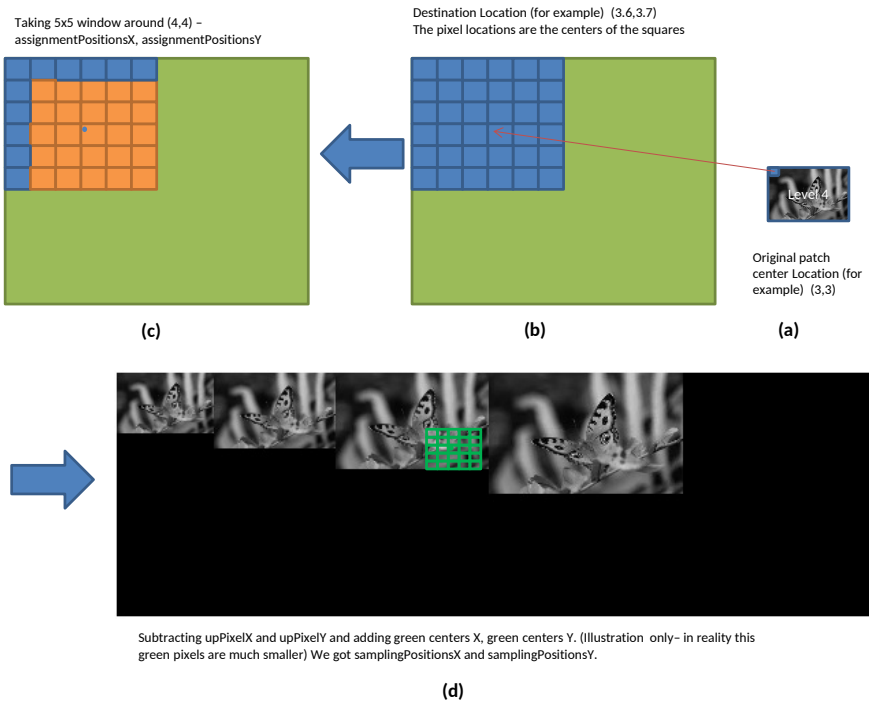



Figure 6: Find (i) x-y coordinates in the high resolution image (integers) (*assignment positions*), and (ii) x-y coordinates (non-integer) in the rendered pyramid image (*sampling positions*).

% Get the information for sampling a high resolution image. Pairs of: assignment positions in the high resolution image, and sampling positions from the rendered pyramid image

% Arguments:

% sampleCentersX $-$ $(m-4) \times (n-4) \times 3$ matrix with the x coordinates of the center of the high resolution patches in the rendered image. This variable should be returned from getSamplingCenters function. (green x locations)

% sampleCentersY $-$ $(m-4) \times (n-4) \times 3$ matrix with the y coordinates of the center of the high resolution patches in the rendered image. his variable should be returned from getSamplingCenters function. (green y locations).

% pyr $-$ $7 \times 1$ cell created using createPyramid

% inputPatchesCentersX $-$ $(m-4) \times (n-4)$ input patches center x coordinates

% inputPatchesCentersY $-$ $(m-4) \times (n-4)$ input patches center y coordinates

% levelsUp $-$ integer which tells how much levels up we need to sample the window, from the found patch. In the figure the case is levelsUp=1

%

% Outputs:

% assignmentPositionsX $-$ $(m-4) \times (n-4) \times 3 \times 5 \times 5$ x assignment coordinates in the high resolution image (see figure)

% assignmentPositionsY $-$ $(m-4) \times (n-4) \times 3 \times 5 \times 5$ y assignment coordinates in the high resolution image (see

figure)
% samplingPositionsX − $(m-4) \times (n-4) \times 3 \times 5 \times 5$ x sampling coordinates in the rendered pyramid image (see figure)
% samplingPositionsY − $(m-4) \times (n-4) \times 3 \times 5 \times 5$ y sampling coordinates in the rendered pyramid image (see figure)
%

# 10 Drawing the high resolution image

From the previous parts we have:

- `assignmentPositionsX,assignmentPositionsY:` $(m-4) \times (n-4) \times 3 \times 5 \times 5$ *assignment positions* matrices which are the integer locations in the high resolution image where the high resolution patches should be placed.
- `samplingPositionsX,samplingPositionsY:` $(m-4) \times (n-4) \times 3 \times 5 \times 5$ *sampling positions* matrices which are the non-integer locations in the rendered pyramid image, where each high-resolution patch should be sampled from.
- `weights - A` $(m-4) \times (n-4) \times 3$ matrix that holds the weight of each high resolution patch based on the similarity of each child patch to the input patch.

We have a black high resolution image that we want to fill with the values. Note that there is an overlap between the $5 \times 5$ patches. Therefore, each pixel in the high resolution image is related to 25 patches, where every patch has 3 child candidates. Hence, for a pixel in the high resolution image, there are up to 75 pixels that give information for that pixel. To solve this part with minimum loops, we are going to calculate 75 high-resolution images and average them according to the weights we calculated in Sec. 7.2. Each of the 75 images contains non-overlapping patches (as shown in Fig. 7). The way to divide the patches to non overlapped windows is a little bit tricky. The simple part is the 3 similar patches which represent the same locations so we can loop over the 3 candidates and each time calculate 25 images with no overlapping patches. So we need to create 25 images from the $(m-4) \times (n-4) \times 5 \times 5$ matrix. Note that if we sub-sample every fifth window in each direction (from the $(m-4) \times (n-4)$ windows) we get non overlapping windows. You can see an example in the Fig. 7 where the sampling process is illustrated for 4 of the 25 options. .

## 10.1 Applying the weights

Each patch has its own weight that should be considered in the process. For each one of the 75 images we build a corresponding weights image, where every square in the image that represent a patch has a

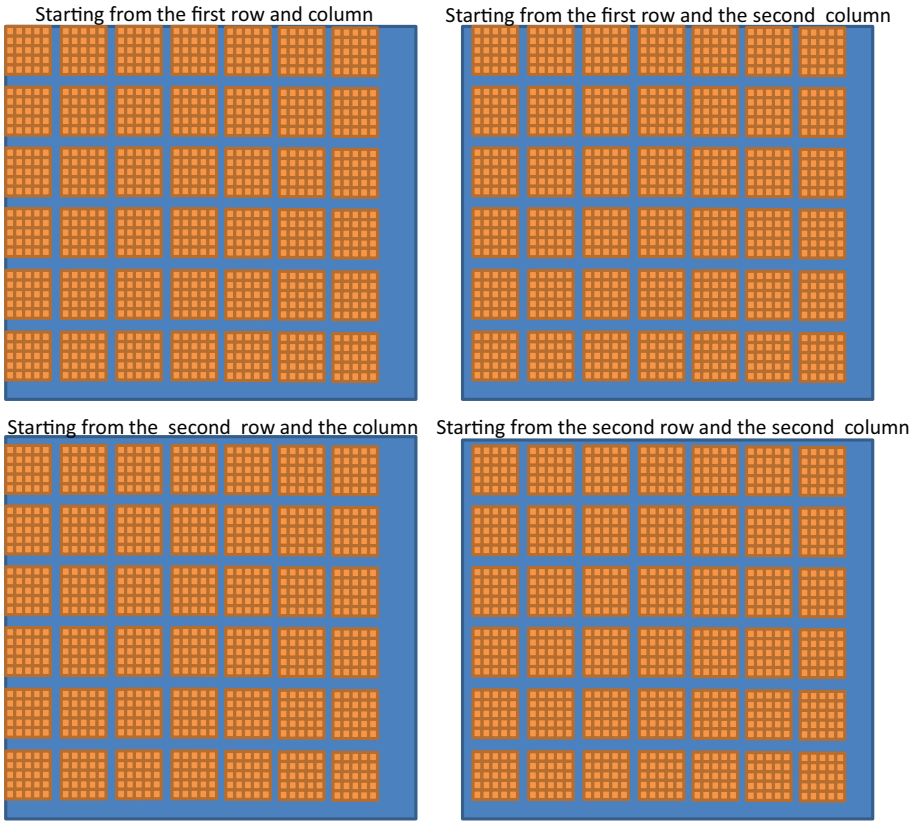



Figure 7: Create 25 images with non-overlapping patches. Here we show 4 options out of 25.

constant value. The value is the weight of the patch calculated in 7.2.

At the end of the process, we need to create one image from the 75 images and 75 weights images, such that each pixel will get the following value:

$$p_{i,j} = \frac{\sum_{t=1}^{75} m_{t,i,j} \cdot w_{t,i,j}}{\sum_{t=1}^{75} w_{t,i,j}}$$

where $m_{t,i,j}$ is the pixel located at $(i, j)$ in the $t^{th}$ image and $w_{t,i,j}$ is the same for the weight image.

In order to calculate the above, we should keep two images: the numerator and the denominator. We will go over the 75 pairs of images ($m_t$ and $w_t$) and add $m_t. \cdot w_t$ to the numerator and $w_t$ to the denominator. Finally, we divide the numerator image by the denominator (point wise division).

The above procedure should be implemented in the `getImage` function having the following interface:

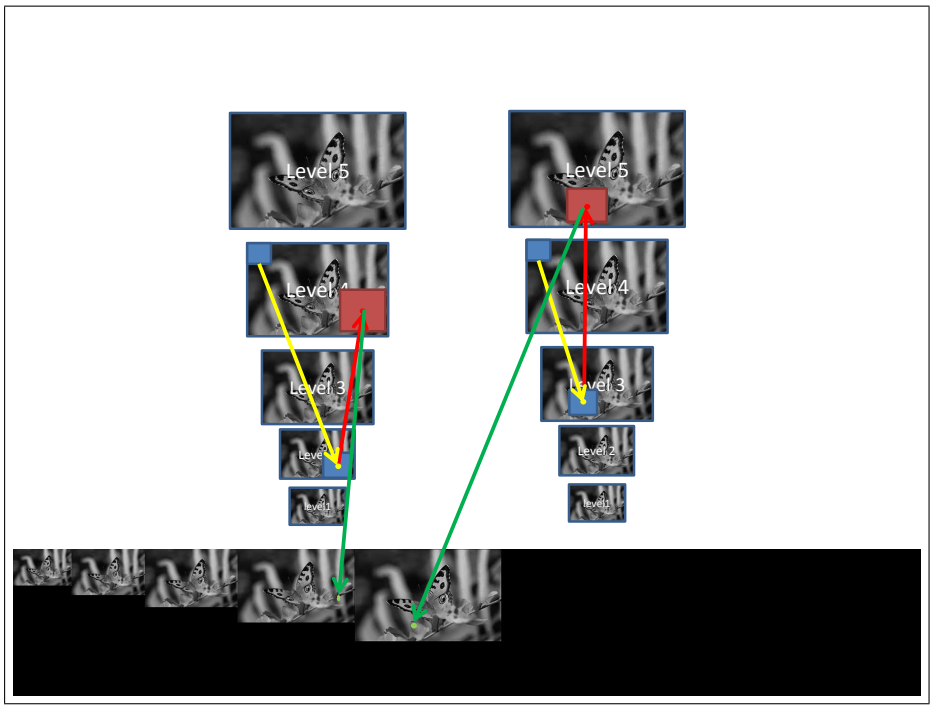
```
function [image] = getImage(assignmentPositionsX,assignmentPositionsY,samplingPositionsX,samplingPositionsY,weights
    ,emptyHighResImage,renderedPyramid)
% GETIMAGE given an image of the rendered pyrmamid, sampling indices from the rendered pyrmamid, and
```

```
    assignment indices in the highres image return a high resolution image
% Arguments:
% assignmentPositionsX - (m-4) × (n-4) × 3 × 5 × 5 x assignment coordinates in the high resolution image (
    getSamplingInformation output)
% assignmentPositionsY - (m-4) × (n-4) × 3 × 5 × 5 y assignment coordinates in the high resolution image (
    getSamplingInformation output)
% samplingPositionsX - (m-4) × (n-4) × 3 × 5 × 5 x sampling coordinates in the rendered pyramid image (
    getSamplingInformation output)
% samplingPositionsY - (m-4) × (n-4) × 3 × 5 × 5 y sampling coordinates in the rendered pyramid image (
    getSamplingInformation output)
% weights - (m-4) × (n-4) × 3 matrix with the weights for each DB candidate
% emptyHighResImage - M × N zeros image, where M and N are the dimensions of a level in the pyramid that should
    be reconstructed in this function
% renderedPyramid - a single image containing all levels of the pyramid
%
% Outputs:
% image - M × N high resolution image
%
```

# 11 Integrate all steps and get all pyramid levels

## 11.1 Building all pyramid levels

As you learned in class once we built the first high resolution image (which is pyr{5}), we can keep going and building the next levels (pyr{6} and pyr{7}). Starting again with the input image pyr{4}, we can use the nearest neighbors we have found before, but this time, instead of going one level up, we go 2 or 3 levels up in the pyramid. Since we built pyr{5}, a high resolution version of nearest neighbors in level 3 can be sampled from level 5. This is illustrated in the following figure (compare it to Fig 4):




## 11.2 Integrate all steps

Once we have all the above we can integrate all:

1. Getting an image
2. Creating a DB of patches using `createDB`
3. For each patch in the image, find its nearest neighbor from the DB using `findNearestNeighbors`
4. Set a weight for each matching from the DB using `weightsSetting`
5. For each nearest neighbor get the patch center location where the high resolution patch should be sampled (from the rendered pyramid) using `getSamplingCenters`
6. For each patch in the input image, find pairs of assignment positions in the high resolution image, and sampling positions from the rendered pyramid. This is done using `getSamplingInformation`
7. Build the high resolution image using `getImage`
8. Looping over steps 7-9 to get pyr{5},pyr{6},pyr{7}, each time with a higher value of `levelsUp`
9. Return pyr{7}

All the above steps are implemented in the supplied function `superResolution.m`.

# 12 Displaying your result

## 12.1 Working with RGB images

To work with RGB image you need to transform your image to a YIQ colorspace and make the super resolution procedure on the Y channel. Resize I and Q parts using imresize function with a factor of 2. Then return back to the RGB colorspace.

## 12.2 Your examples

You are supplied with the image `butterfly.png`. Make a script `MyResults.m` that runs the procedure on `butterfly.png` and on two more images that applying super resolution on them returns nice results. For each one of the 3 images, display the folowing three figures with suitable titles:

- The original image
- The result of the super resolution procedure on the image
- An upsampled version the input image (created using imresize with a factor if 2)

In your Readme file describe for each image (`butterfly.png` and the 2 you added):

1. Which details in the image the procedure keeps?
2. Which details in the image the procedure loses?

## 12.3 More Comments

- Start early. This a large exercise and it will be weighted accordingly in the final grade.

- Except for the functions `createPyramid ,samplePatches, createDB` (looping over pyramid levels) and `getImage` (looping over $5 \times 5 \times 3$), you are not allowed to use loops in the code. It is one of the challenges in this exercise. Instead, use Matlab's functions `reshape`, `repmat` and `permute`.
- Running the procedure on the supplied image `butterfly.png` should take less than 2 minutes on the cs-lab computers. In case that it exceed this period, your submission will be severely penalized.
- Any time you use interpolation, i.e using one of the functions `interp2` or `imresize` use `'cubic'` interpolation.
- Your submission file should be packed in an archive called ex5.zip.

Good luck and enjoy!