Image Processing
Lab 3

Kevin Gevers (s25595987)
Jeroen Overschie (s2995697)
Group 01

January 20, 2021

Note, that all used source code can be found attached next to this report's pdf file. Its structure should be self-explanatory; all requested functions are named accordingly and any extra functions are explained in the report. Note that (almost) every function has a corresponding test script, which is named just like its function, but with a suffix '`_test`'. Where possible, we followed the terminology from the book [1] for variable naming.

Exercise 1

First some basic theory on the subject. The goal is to work with Laplacian Pyramid decompositions, for which we apply both the decomposition process and the reconstruction process. The decomposition allows one to conveniently store images in multiple formats to use any suited version wherever desirable. By applying some filter, i.e. a Gaussian filter, we aid the subsampling process by making increasing the chance for picking pixels that are good representations of the original image; which the filter does by somewhat 'smoothing' the image. Moreover, by saving the differences between the various decomposition levels, we are able to reconstruct the original image by performing the same process in reverse, using the difference images to again obtain an image of the original dimensions.

Let us explain the notation used in the process. We are given some grey scale image, f. We now denote the Laplacian pyramid of f using:

$$f_1 = f$$
$$f_j = \text{REDUCE}(f_{j-1}) \text{ for } j = 2, ...J$$

Please do denote the subtle differences between the lowercase j and uppercase J. J denotes the number of levels of the pyramid composition, e.g. when $J = 3$, the pyramid will consist out of 3 images: the original, a decomposition of the original, and a decomposition of the decomposition. Furthermore, we define $d_1, d_2, ...d_{J-1}$ as the *detail signals*, or *residuals*:

$$d_j = f_j - \text{EXPAND}(f_{j+1}) \text{ for } j = 1, 2, ...J - 1$$

Which are more simply said the decompositions scaled back to the dimensions of their previous form, to then get the difference between the two images using matrix subtraction. Because we are diffing the images here, our vector d can only be of length $J - 1$. Lastly, we define the REDUCE and EXPAND operators:

$$\text{REDUCE}(f) = \downarrow_2 (h_\sigma * f)$$
$$\text{EXPAND}(f) = h_\sigma * (\uparrow_2 (f)),$$

where h_σ denotes *Gaussian filtering*, which is a filter with a Gaussian response that can be used for performing a sort of image blur. The filter has the parameter σ, which is the standard deviation of the kernel. Furthermore, the operations denoted by the arrows, $\downarrow_2$ and $\uparrow_2$ denote **shrinking** and **zooming** an image by a factor 2, respectively. In the spirit of maximum re-usability and efficiency, we reused some functions from Exercise 1 for this: the 'fundamental' functions IPINTERPOLATE (❖ Listing 2) and IPSCALING_TRANSFORMATION (❖ Listing 1) to do the scaling work, and IPDOWNSAMPLE (❖ Listing 3) and IPZOOM (❖ Listing 4) to apply them. These functions are thoroughly documented in the first report. In this lab, we use IPDOWNSAMPLE for our shrinking functionality ($\downarrow_2$) and IPZOOM for zooming functionality ($\uparrow_2$).

(a)

In this first exercise, we were asked to implement a function IPPYR_DECOMP that builds a Laplacian pyramid decomposition. For the parameters of this function, f is the input image, J the composition level and σ the Gaussian filter kernel parameter. For our implementation, see ❖ Listing 5.

In our implementation, we use matlab *Cell* types to store the pyramid and the detail signals, in variables f and d, respectively. Note that the Matlab cell type indeed makes it possible to store matrices of variable lengths, though we do need the function MAT2CELL and CELL2MAT for storing/retrieving from the cell array. Like announced, we were allowed to use the function IMGAUSSFILT to perform the Gaussian filtering process. We hand it a *double* image (IM2DOUBLE) and the parameter σ. We perform both the Reducing and Expanding operations and store the residuals in d.

Now finally, we can store the result in a matrix g. We can compute the height for this result matrix using:

$$P = M \times (1 + \frac{1}{2} + (\frac{1}{2})^2 + ... + (\frac{1}{2})^{J-1}),$$

which we conveniently implemented using an element-wise squaring operation in Matlab, i.e. using `.^` (see IPPYR_DECOMP code line 35 in Listing 5). Note we add 1 to compensate for the Matlab indexing starting at 1 instead of 0. Next come some basic geometry operations to stack the result images vertically over the space of P and centering each image horizontally over the space of M. This done, we result with a result matrix g, which is also saved to `IPpyr_decomp-g_J=3,sigma=1.0.mat`.

(b)

Next, we visualize the results, using again a `*_test` script. See Listing 6 for the testing code. See Figure 1 for the original image and Figure 2 for its Laplacian pyramid decomposition using $J = 3$ and $\sigma = 1.0$. We can observe that the results look very much alike the example images given in the assignment instructions, despite having tuned the parameters slightly differently (sigma is higher in the assignment instructions, i.e. is 5.0 instead of our 1.0). As can be seen in the bottom most image, a decomposition of a 4x smaller size was made and still looks reasonably alike thanks to the filtering techniques applied at every step. Also the difference images look alike what was given in the assignment, indicating a correct decomposition.

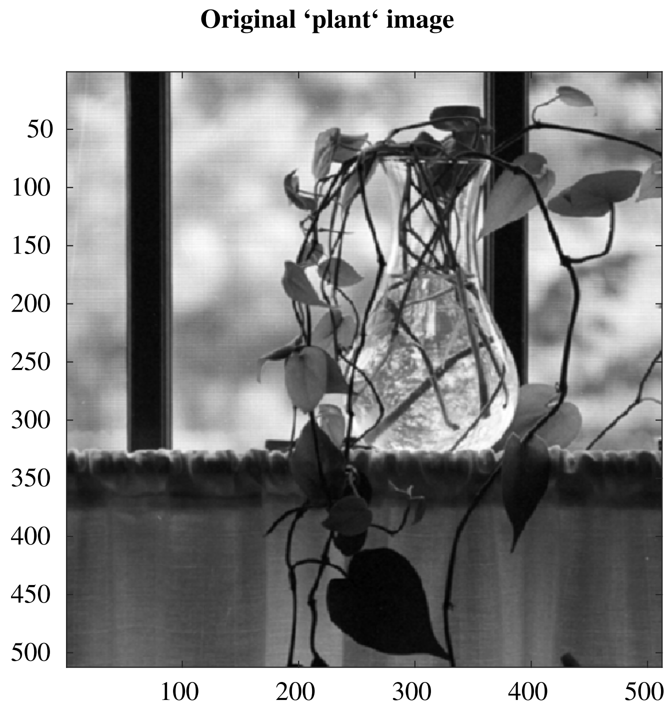

Figure 1: Original plant image for Exercise 1 without any filtering applied.

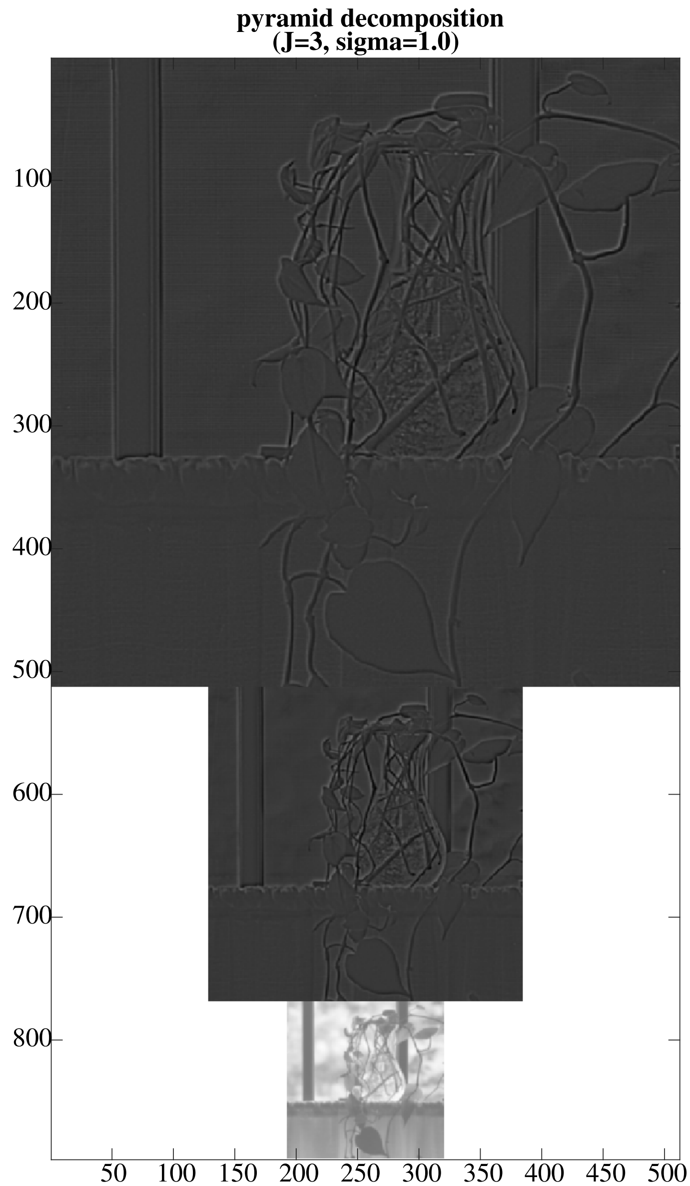

Figure 2: Laplacian pyramid decomposition of 1 using $J = 3$ and $\sigma = 1.0$. Residuals d are vertically stacked, with at the bottom the final decomposition, f_3.

(c)

See `IPpyr_decomp_data-J=3,sigma=1.0.mat` for the result matrix data g.

Exercise 2

(a)

For exercise 2 we are going to reconstruct the decomposed image from exercise 1. The exercise states *"Let f be a grey value image with Laplacian pyramid decomposition of J levels given by detail signals $d_1, d_2, \ldots, d_{J-1}$ and coarsest approximation f_J. As before you may assume that the image f is a square image of size $M \times M$, where M is a power of 2."* To reconstruct the image we need to use Formula 1, which in turn needs Formula 2.

$$f_j = \text{EXPAND}\left(f_{j+1}\right) + d_j, \quad j = J-1, \ldots, 1 \tag{1}$$

$$\text{EXPAND}(f) = h_\sigma * (\uparrow 2(f)) \tag{2}$$

Now we are tasked with writing a Matlab function *IPpyr_recon(g,J,sigma)* that implements the Laplacian pyramid reconstruction. Here g is the decomposed image that was the result of exercise 1, J is the number of decomposition levels, and *sigma* is the standard deviation for the Gaussian filter. Our implementation can be seen in ❖ Listing 7.

(b)

Next we had to apply our function *IPpyr_recon(g,J,sigma)* to the result of exercise 1. We first load the .mat file that was saved in Exercise 1, extract the coarsest level image, f_J and then apply our reconstruction function *IPpyr_recon* to it. See ❖ Listing 8, for the final testing code.

Figure 3 shows the input image and the reconstructed image side by side. Here we used $J = 3$ and $sigma = 1.0$ as stated in the assignment.

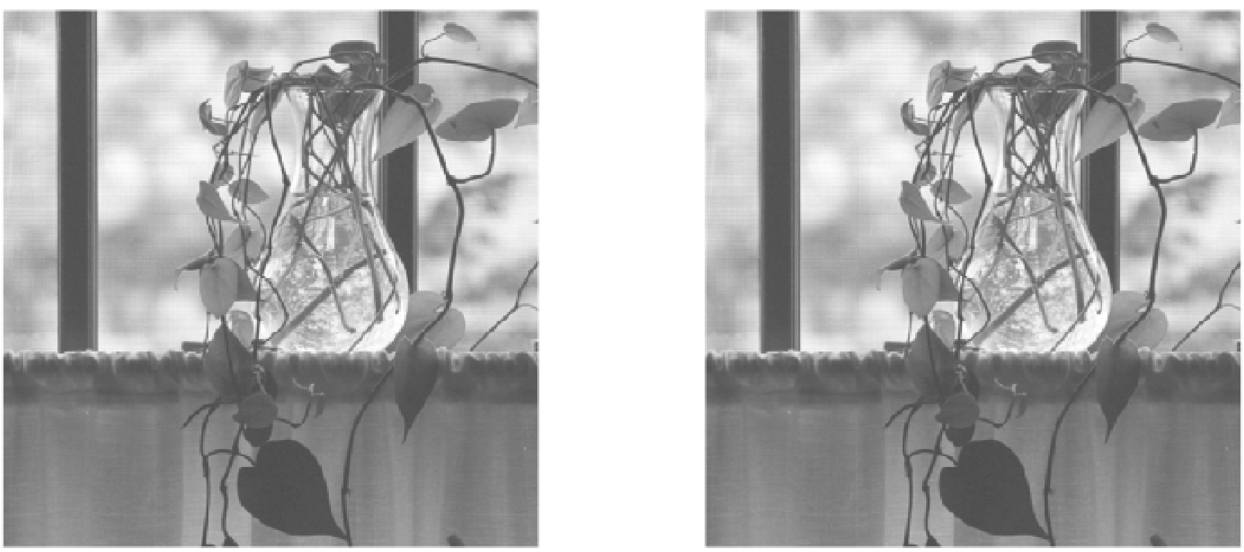

Figure 3: (Left) Input image. (Right) Reconstruction of the image after having decomposed it.

(c)

The next step is to calculate the mean absolute error between the reconstructed image and the input image. For this we need to use Formula 3 and we need to make sure both images are of the same type. We loaded the input file again as *uin8* and converted our reconstructed image to *uint8* as well.

$$\text{error}(f, g) = \frac{1}{M \times N} \sum_{i=1}^{M} \sum_{j=1}^{N} |f(i,j) - g(i,j)| \tag{3}$$

We also need to compute the difference image between the input and reconstructed images. This is done by taking the mean absolute value of each pixel value using both images. The result is presented in Figure 4.

(d)

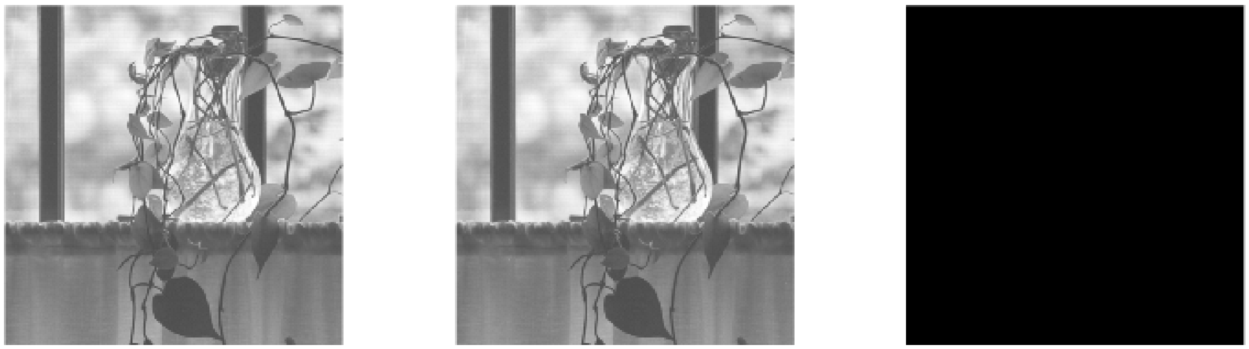

Figure 4: (Left) Input image. (Middle) Reconstruction of the image after having decomposed it. (Right) Difference image between the input image and reconstructed image.

The last part of the assignment is to put the input image, the reconstructed image, and the difference image together in a single figure and save it. We did this as can be seen in ❖ Listing 8 and in Figure 4. The mean absolute error between the two images as implemented in (c) gives us the error 0, which means that the images are identical. This also shows in the completely black difference image in Figure 4. Using the Laplacian pyramid decomposition and reconstruction give the ability to compress the image without losing detail and later reconstruct the exact same image as the original.

Individual contributions

- **Exercise 1**. Contribution to program design, program implementation, answering questions posed and writing the report: *Jeroen* 100%.
- **Exercise 2**. Contribution to program design, program implementation, answering questions posed and writing the report: *Kevin* 100%.

References

[1] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Prentice Hall, Upper Saddle River, N.J., 2008.

A Code

A.1 Exercise 1

A.1.1 Exercise 1 (a)

Listing 1: IPscaling_transformation.m: perform geometric transformations of the *scaling* kind, given an affine transformation matrix or a scaling constant.

```
function It = IPscaling_transformation(I, A, interpolation)
% IPscaling_transformation Computes an image scaling operation
% using an affine transformation matrix.
% Arguments:
%       I: Input image
%       A: Affine transformation matrix [3x3] of form
%           [cx 0 0; 0 cy 0; 0 0 1;] or a scalar value, then cx=cy.
%       interpolation: interpolation method. See IPinterpolate.m
if isnumeric(A)
    A = [A 0 0; 0 A 0; 0 0 1;];
end
if ~exist('interpolation', 'var')
    interpolation = 'none';
end
I = im2double(I);

% Image size
[M, N] = size(I); % height, width
D = [M, N, 1];  % dimensions

% Transformed dimensions
Dt = D * A;
Mt = round(Dt(1));
Nt = round(Dt(2));

% Map coordinates to new values
It = zeros(Mt, Nt);

%% Inverse mapping
% Perform _inverse mapping_ instead of forward mapping. Compute
% the inverse affine transformation matrix $A^{-1}$. See
% section 2.6 of (DIP, 42 - Gonzalez, Woods) book, page 102.
for y = 1:Mt
    for x = 1:Nt
        Pt = [x, y, 1];
        P = Pt / A; % original coordinate (same as Pt * inv(A)).
        offset = diag(0.5 * (1 - inv(A)))'; % inverse mapping centering offset
        It(y, x) = IPinterpolate(I, P, offset, interpolation);
    end
end

end
```

Listing 2: IPinterpolate.m: interpolate an unknown pixel intensity value using one of the supported methods.

```
function v = IPinterpolate(I, P, offset, interpolation)
```

```
% IPinterpolate Interpolate image pixel for image using given method.
% Arguments:
%       I: Input image
%       P: Coordinates of pixel with unknown intensity value; in original
%           (i.e. non-transformed) coordinate space. e.g. coordinates might
%           have decimals.
%       interpolation: one of
%           ('none' | 'nearest' | 'bilinear')
%           meaning no interpolation, nearest neighbor interpolation and
%           bilinear interpolation, respectively.
[M, N] = size(I);
switch interpolation
    %% Nearest neighbor interpolation
    % Finds the nearest pixel in the inverse mapping using `ceil`.
    case 'nearest'
        % Make sure within bounds
        Po = P + offset;
        x = min(max(1, Po(1)), N);
        y = min(max(1, Po(2)), M);
        v = I(round(y), round(x));
    case 'bilinear'
    %% Bilinear interpolation
    % See Bilinear interpolation Wikipedia page for used terminology and
    % also https://bit.ly/3o3vcxD for parts of implementation.
        Po = P + offset;
        x = Po(1);
        y = Po(2);
        % Any values out of acceptable range
        x(x < 1) = 1;
        x(x > N - 0.001) = N - 0.001;
        x1 = floor(x);
        x2 = x1 + 1;
        y(y < 1) = 1;
        y(y > M - 0.001) = M - 0.001;
        y1 = floor(y);
        y2 = y1 + 1;
        % Neighboring Pixels
        Q = [I(y1,x1); I(y1,x2); I(y2,x1); I(y2,x2);];
        % Pixels Weights
        b = [(y2-y)*(x2-x); (y2-y)*(x-x1); (x2-x)*(y-y1); (y-y1)*(x-x1);];
        v = dot(b, Q);
    otherwise % no interpolation, i.e. 'none'
        % if this pixel maps to an original pixel
        if mod(P(1), 1) == 0 && mod(P(2), 1) == 0
            v = I(P(2), P(1));
        else % otherwise pad with zeros; no interpolation
            v = 0;
        end
end
end
```

Listing 3: IPdownsample.m: downsample images using IPSCALING_TRANSFORMATION.

```
function downsampledImage = IPdownsample(I, factor)
% IPdownsample Down-samples an image by a factor of `factor`, which
```

```
% must be a (positive) integer, i.e. factor >= 1.
% Arguments:
%   I: input image to downsample
%   factor: factor to shrink with. Must be an integer.
assert(isinteger(factor), 'factor must be an integer (was %d)', factor);
It = IPscaling_transformation(I, 1/double(factor), 'nearest');
downsampledImage = uint8(It * 2^8); % normalize back to 8-bit int image.
end
```

Listing 4: IPzoom.m: zoom images using IPSCALING_TRANSFORMATION.

```
function zoomedImage = IPzoom(I, factor)
% IPzoom Zooms an image by a factor of `factor`, which
% must be a (positive) integer, i.e. upsamplingFactor >= 1. Function zooms
% image by replicating pixels.
% Arguments:
%   I: input image to zoom
%   factor: factor to zoom with. Must be an integer.
assert(isinteger(factor), 'factor must be an integer (was %d)', factor);
It = IPscaling_transformation(I, double(factor), 'nearest');
zoomedImage = uint8(It * 2^8); % normalize back to 8-bit int image.
end
```

Listing 5: IPPYR_DECOMP function: Laplacian pyramid decomposition.

```
function g = IPpyr_decomp(f, J, sigma)
% IPpyr_decomp Laplacian pyramid decomposition
%   Arguments:
%       f: input image
%       J: the number of decomposition levels
%       sigma: standard deviation of the Gaussian filter
f_1 = im2double(f);
[M, N] = size(f_1); % height, width
assert(M == N);

% Store pyramid and differences in cells
f = cell(1, J);
f(1) = mat2cell(f_1, M);
d = cell(1, J - 1);

% Build image pyramid
for j = 2:J
    f_prev = cell2mat(f(j - 1));

    % REDUCE
    f_j = imgaussfilt(f_prev, sigma);
    f_j = IPdownsample(f_j, uint8(2));

    % EXPAND & difference
    expanded = IPzoom(f_j, uint8(2));
    expanded = imgaussfilt(expanded, sigma);
    d_j = f_prev - expanded;

    % Store in cells
    f(j) = mat2cell(f_j, size(f_j, 1));
```

```
    d(j - 1) = mat2cell(d_j, size(d_j, 1));
end

% Build output g
D = M * (1/2) .^ (0:J-1);        % compute image dimensions D for all levels
P = sum(D) + 1;                  % +1 for Matlab indexing
x = M/2 - D/2 + 1;               % compute `g` x-coordinates
y = cumsum([0, D(1:J-1)]) + 1; % compute `g` y-coordinates

% Insert detail signals `d` and coarsest decomposition level `J` into image
g = ones(P, M);
for j = 1:J
    if (j == J); im = f(j); else; im = d(j); end
    g(y(j):y(j)+D(j)-1, x(j):x(j)+D(j)-1) = cell2mat(im);
end
end
```

A.1.2 Exercise 1 (b)

Listing 6: IPpyr_decomp_test: Testing the Laplacian pyramid decomposition function, IPPYR_DECOMP.

```
clc;                                    % clear the command window
close all;                              % close open figure windows

imname = 'plant';
inputfile = ['input_images/', imname,'.tif'];
f = imread(inputfile);                   % read input image

figure;
% Original image
colormap(gray(256));
imagesc(f);
axis equal;
axis tight;
title({'Original `plant` image', ' '});
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Write current figure to file
all_file = ['output_plots/', imname,'_original','.svg'];
saveas(gcf, all_file);
fprintf('\nComplete image has been saved in file %s\n', all_file);

% Pyramid decomposition
figure;
[M, N] = size(f); % height, width
J = 3;
sigma = 1.0;
g = IPpyr_decomp(f, J, sigma);
colormap(gray(256));
save('IPpyr_decomp_data-J=3,sigma=1.0.mat', 'g')
imagesc(g);
axis equal;
axis tight;
title({'pyramid decomposition', '(J=3, sigma=1.0)'});
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% Write current figure to file
all_file = ['output_plots/', imname,'_all','_pyr-decomp', '.svg'];
set(gcf, 'PaperUnits', 'normalized')
set(gcf, 'PaperPosition', [0 0 0.75 1.00])
saveas(gcf, all_file);
fprintf('\nComplete image has been saved in file %s\n', all_file);
```

A.2 Exercise 2

A.2.1 Exercise 2 (a)

Listing 7: IPpyr_recon function: Laplacian pyramid reconstruction.

```
function g2 = IPpyr_recon(g,J,sigma)
% IPpyr_recon Laplacian pyramid reconstruction
%   Arguments:
%       g: result matrix from IPpyr_decomp
%       J: the number of decomposition levels
%       sigma: standard deviation of the Gaussian filter

[~, M] = size(g);

% calculate the coordinates needed for each level
D = M * (1/2) .^ (0:J-1);       % compute image dimensions D for all levels
x = M/2 - D/2 + 1;              % compute `g` x-coordinates
y = cumsum([0, D(1:J-1)]) + 1; % compute `g` y-coordinates

% extract coarsest level image
f_j = g(y(J):y(J)+D(J)-1, x(J):x(J)+D(J)-1);

for j = (J-1):-1:1
    % EXPAND = upsample & Gaussian filter
    f_j = IPzoom(f_j, uint8(2));
    f_j = imgaussfilt(f_j, sigma);

    % extract previous d
    g_j = g(y(j):y(j)+D(j)-1, x(j):x(j)+D(j)-1);

    % add previous d to expanded image
    f_j = f_j + g_j;
end
g2 = f_j;
end
```

A.2.2 Exercise 2 (b-d)

Listing 8: IPpyr_recon_test test script: executing the various task from exercise 2b-d.

```
clc;                              % clear the command window
close all;                        % close open figure windows

% Define J and sigma
J = 3;
sigma = 1.0;

% Load initial input image
```

```
imname = 'plant';
inputfile = ['input_images/', imname,'.tif'];
f = imread(inputfile);                    % read input image
f = im2double(f);
[M, N] = size(f);
assert(M==N);

% % Decompose the image
% g = IPpyr_decomp(f, J, sigma);

% Load decomposed image from file
mat = load('IPpyr_decomp_data-J=3,sigma=1.0.mat');
g = mat.g;

% Pyramid reconstruction
g2 = IPpyr_recon(g, J, sigma);

% Show results
figure;
subplot(121);
imshow(f)
subplot(122);
imshow(g2)

% Write current figure to file
saveas(gcf, 'output_plots/IPpyr_recon_test_partial.svg');

% Nicer output image but requires Matlab 2020a or above
% exportgraphics(gcf, 'output_plots/IPpyr_recon_test_partial.png');

% turn the next two lines off if you want to compare double images
g2 = im2uint8(g2);
f = imread(inputfile);

% compute absolute error between f and g2 + create difference image
diffImage = abs(f - g2);
error = sum(abs(f - g2), 'all') / (M * N);

% show results
figure;
subplot(131);
imshow(f)
subplot(132);
imshow(g2)
subplot(133);
imshow(diffImage)

% Write current figure to file
saveas(gcf, 'output_plots/IPpyr_recon_test.svg');

% Nicer output image but requires Matlab 2020a or above
% exportgraphics(gcf, 'output_plots/IPpyr_recon_test.png');
```