Please refer to homework 1 and the class mechanics web page for homework policies and formatting/submission instructions. The submission checklist for this assignment is as follows:

`hw2.pdf`, `rectangleSums.m`, `boxFilter.m`, `binomialFilter.m`, `pyramid2image.m`.

Keep in mind that your code must also show up in `hw2.pdf`.

The goal of this assignment is to familiarize yourself with convolution, filtering, image pyramids, and the basics of image programming in MATLAB. The discussion of programing idioms below is slightly more general than what would be strictly required for this assignment, as some considerations may be useful in the future.

Programming Idioms

Here are a few things to pay attention to when programming with images in MATLAB.

Image Data Types

Images read from files are typically represented as `uint8` arrays in MATLAB, unless the file contains data that would not fit in that type.

Sometimes as you work on an `uint8` image `I` the intermediate results do not fit in a `uint8` image, but the output does. For instance, when smoothing an image directly (that is, without using the built-in function `conv2` which takes care of conversions for you), you add pixel values, and if you do $200 + 220 = 440$, the result requires more than 8 bits, even if after that you divide by the number of pixels and the final result fits again in 8 bits. Other times, you cannot even predict the type of `I` when you program. In circumstances like these, a typical programming idiom is to keep track of the type of the input image, cast it to a data type appropriate for the intermediate operations, and then cast the output image `J` back to the original type. For example:

```
function J = f(I)

inputClass = class(I);
I = int16(I);

... <intermediate computations on I resulting in image J> ...

J = cast(J, inputClass);
```

If all you do is sums, subtractions, and multiplications, then `int16` may be appropriate, but think whether it is possible to exceed the magnitude of this type. If there are divisions and other numerical operations and you want to preserve precision, then it may be necessary cast to `float` or `double`, and then round the result before casting back.

Furthermore, the final cast of `J` may be unwarranted. For instance, if your function computes image derivatives, then the output may have negative pixel values, so `J` must be in a data type that allows for negative values.

As you display images, keep in mind that `imshow` is designed to work with `uint8`, true color (that is, `double` images with values between 0 and 1), or binary images. Thus, you cannot use this function to display an image with, say, negative values. You would then use `imagesc`, which scales the input image appropriately before displaying it or, in some complex cases, you may have to do the conversion yourself. Some examples are given in the files provided with this assignment. See `flattenBandPass.m` in particular.

Color versus Gray-Level

To determine if an image `img` is color or gray-level, you can say `ndims(img)`. If the result is 2, then `img` is gray-level. If the result is 3, then `img` is color.

The simplest way to write a function that is supposed to work on both color and gray-level images is to program for color. For instance, to access pixel in row `i`, column `j`, you can say

```
img(i, j, :)
```

regardless of whether `img` is color or gray-level. With this syntax, if `img` has only two dimensions, the colon at the end is ignored.

Be careful when working with image sizes. To compute twice the size of image `img` you cannot say

```
2 * size(img)
```

because this would also double the size of the third dimension to 6 if `img` is color. Instead, you say

```
s = size(img);
s(1:2) = 2 * s(1:2);
```

and `s` then contains the desired size. This snippet also works for gray-level.

Image Slicing and Arithmetic

Using MATLAB's array indexing ("slicing") and arithmetic operations rather than `for` loops over images leads to more concise and efficient code. For instance, suppose that you want to detect vertical edges in the gray-level image `img`. You could then create a list of all the pixel values in `img` where the central difference in the horizontal direction has a magnitude greater than 10. The horizontal central difference is the image whose pixel at row `r`, column `c` is

```
(img(r, c+1) - img(r, c-1)) / 2
```

and approximates the local derivative in the horizontal direction. We assume that `img` is stored with a data type that accommodates subtraction and division by 2 appropriately. The following code is then a succinct implementation that avoids looping over `r` and `c`:

```
diff = (img(:, 3:end) - img(:, 1:(end-2))) / 2;
where = false(size(img));
where(:, 2:(end-1)) = abs(diff) > 10;
values = img(where);
```

Explanation of the code: The first line computes the array `diff` that contains half the difference between the values `img` starting in the third column and those starting in the first column. If ellipses were allowed in MATLAB (they are not in this way), this line would be equivalent to

```
diff = [(img(:,3)-img(:,1))/2, (img(:,4)-img(:,2))/2, ..., (img(:,end)-img(:,end-2))/2]
```

so all the required differences are computed in one shot, with no explicit `for` loop (of course, MATLAB needs loops under the hood, but that is compiled code, and runs fast).

To check the absolute value of these differences against the threshold 10 we could then just say

```
where = abs(diff) > 10;
```

However, `diff` has two columns fewer than `img` (and the same number of rows as `img`), because both `3:end` and `1:(end-2)` span that many indices. The first column in `diff` is the central difference for the *second* column of the image, the second for the third, and so forth. So the columns of `img` and `where` would be misaligned.

To fix this misalignment, the second line in the code snippet above first creates a logical image `where` of the same size as `img`, initialized to all `false` values. The line thereafter then checks the magnitudes of `diff` against the constant 10 and places the results into the appropriate columns of `where`. The resulting logical image `where` has the same size as `img` and contains a `true` where the check succeeds and a `false` where it fails.

Finally, the last line picks all the pixels in `img` for which `where` is `true`. This instruction works correctly only if `img` and `where` are the same size. These pixels come from various locations of `img` and no longer form an image, so MATLAB has no choice but to return these values as a column vector. If you also wanted the corresponding pixel locations, you could say

```
[row, col] = find(where);
```

which returns two column vectors the same size as `values`.

If this example looks foreign to you, please spend some time studying it and try out something similar yourself. With some patience, it will become second-nature for you to "think in MATLAB" and write concise and efficient code in this style.

[problems start on the next page]

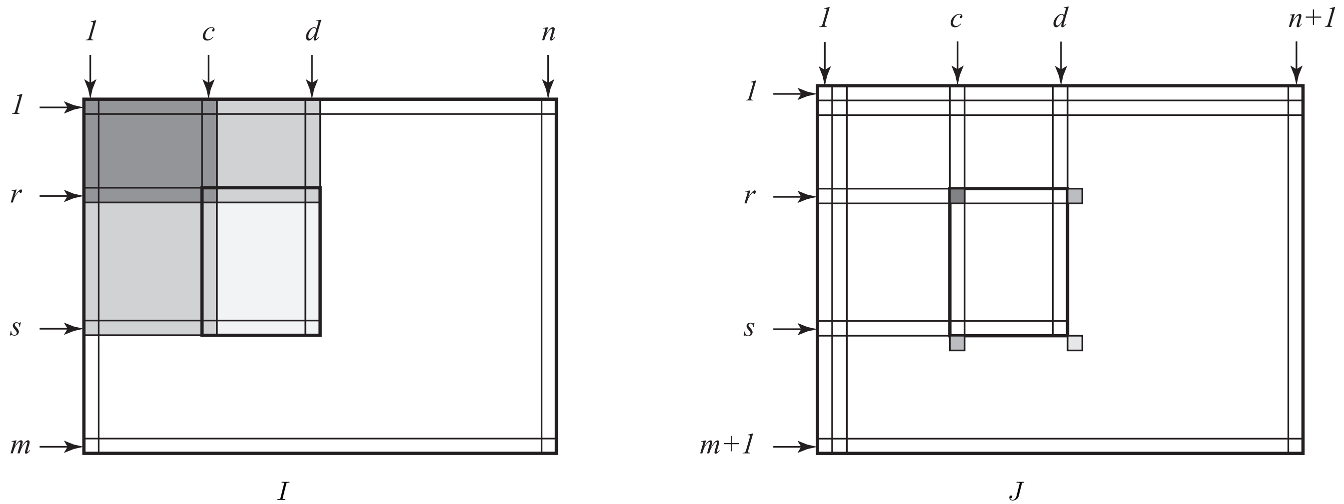

Figure 1: The sum of pixel values in the rectangle between rows r, s and columns c, d in the image I on the left can be written by combining values in the four shaded pixels in the integral image J on the right. Shaded pixels in J contain the integrals over the correspondingly shaded regions in I (shaded regions in I overlap, and start at pixel $(1, 1)$).

Box Filters and the Integral Image

Gaussian convolution kernels are typically best for image smoothing. However, constraints in computation resources sometimes call for simpler filters that can be implemented more efficiently. The kernel B_m of the *box filter* has a square $m \times m$ support and value $1/m^2$ in every pixel. For instance, with $m = 5$, the kernel is

$$B_5 = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} .$$

In this assignment, we assume that m is odd, so B_m is symmetric around a central pixel. Convolving a box filter with an image directly requires $m^2 - 1$ sums and one division per pixel, and just one division.

In addition, the box kernel is separable:

$$B_m = \mathbf{b}_m \mathbf{b}_m^T \quad \text{where} \quad \mathbf{b}_m = \frac{1}{m}\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$$

so convolution with the box kernel can be implemented even more efficiently, as discussed in the class notes.

For relatively large values of m, we can do even better by using the *integral image*. This image, defined next, takes a little time to compute, but then allows computing the sum of pixel values in any image rectangle in constant time, regardless of the size of the rectangle. This advantage is also useful in situations beyond box filtering.

Given an $m \times n$ image I, the integral image J of I is an image of size $(m + 1) \times (n + 1)$. The first row, $J(1, :)$ and the first column, $J(:, 1)$, are all zeros. In the rest,

$$J(i + 1, j + 1) = \sum_{u=1}^{i} \sum_{v=1}^{j} I(u, v) .$$

In words, entry $J(i + 1, j + 1)$ contains the sum of all the pixels in I that are above and to the left of pixel (i, j) inclusive.

The MATLAB image processing toolbox has a function `integralImage` that does this computation. If you do not have access to that code, a simplified version of it is provided with this assignment. This version is named `providedIntegralImage`, so it does not clash with MATLAB's. If you use the provided function, rename it `integralImage`.

Consider now the entries of the original image I in a rectangle of pixels where the top left pixel is at (r, c) and the bottom right pixel is at (s, d). Referring to figure 1, the sum of the pixels in this rectangle can be computed in the following steps:

- Add all the pixel values above and to the left of (s, d) inclusive in I. This is simply $J(s+1, d+1)$.
- Subtract the sum of all the pixel values above and to the left of $(s, c-1)$ inclusive in I. This sum is stored in $J(s+1, c)$.
- Subtract the sum of all the pixel values above and to the left of $(r-1, d)$ inclusive in I. This sum is stored in $J(r, d+1)$.
- The sum of all the pixel values above and to the left of $(r-1, c-1)$ inclusive in I is contained in both the sums we subtracted in the previous two steps, and was therefore subtracted twice. To compensate for this, we add it back in once as $J(r, c)$.

In summary,

$$\sum_{u=r}^{s} \sum_{v=c}^{d} I(u, v) = J(s+1, d+1) - J(s+1, c) - J(r, d+1) + J(r, c) \; .$$

The right-hand side of this equation involves just three sums (once we have J), regardless of how large the rectangle is!

1. Write an efficient MATLAB function

```
function S = rectangleSums(J, p, q)
```

that takes as input the $(m+1) \times (n+1)$ integral image `J` for some $m \times n$ image `I`, as well as two integers p and q with $1 \leq p \leq m$ and $1 \leq q \leq n$ and computes an image `S` of size $(m-p+1) \times (n-q+1)$ whose entry i, j is the sum of pixel values in `I` (not `J`!) in the $p \times q$ rectangle with top-left pixel (i, j) and bottom-right pixel $(i+p-1, j+q-1)$:

$$S(i, j) = \sum_{u=i}^{i+p-1} \sum_{v=j}^{j+q-1} I(u, v) \; .$$

For safety, convert `J` to double before processing, if it is not already:

```
if ~isa(J, 'double')
    J = double(J);
end
```

You get full credit for this question if your code has no explicit loops or convolutions (including calls to `conv2`, `corr2` or their variants). If you use loops, you get a quarter of the credit, and you will waste quite a bit of time waiting for your function to finish its job. If you use convolutions, you will get half the credit.

Hand in your code (in your PDF and as a separate `.m` file) and show in your PDF file the two matrices output by the following code:

```
I = [2 1 3; 0 4 1; 0 3 5; 1 3 2];
J = integralImage(I);
S32 = rectangleSums(J, 3, 2)
S11 = rectangleSums(J, 1, 1)
```

2. Write an efficient MATLAB function

```
function J = boxFilter(I, m)
```

that uses `integralImage` and `rectangleSums` to filter image `I` with an $m \times m$ box filter B_m. There is no need to check for the type of `I` because `integralImage` and `rectangleSums` already do so. It is OK to use floating-point arithmetic to divide by m^2.

The function `boxFilter` should only output the `'valid'` part of the result, as defined on the MATLAB `help` page for `conv2`, and outputs should be of type `double`.

Hand in your code (in your PDF and as a separate `.m` file) and in your PDF show the three matrices `F1`, `F2`, `F3` output by the following code:

```
I = (1:5)' * (1:4);
F1 = boxFilter(I, 1);
F2 = boxFilter(I, 2);
F3 = boxFilter(I, 3);
```

Please format your matrices so they are easily readable.

3. Give a single diagram with three plots on it, showing the running times obtained on the image `DI` that is placed in the MATLAB workspace when running the file `rings.m` provided with this assignment. For each odd box side-length m between 1 and 51 (included), the three plots should show the time *in nanoseconds per pixel* needed to filter the image `DI` with box filter B_m in the following three ways (one per plot):

- Straight convolution,

```
conv2(DI, B, 'valid')
```

  where `B` is B_m.

- Separable convolution,

```
conv2(b, b', DI, 'valid')
```

  where `b` is $\mathbf{b}_m$.

- Box filter,

```
boxFilter(DI, m)
```

To even out fluctuations in running time, the time measurement for each value of m should be the average of `rep = 100` runs on the same image `DI`. To obtain the time in nanoseconds per pixel, use the MATLAB functions `tic` and `toc` to measure elapsed time for a `for` loop that executes the `rep` repetitions, then multiply by 10^9 and divide by `rep` and by the number of pixels in `DI`.

Show only your diagram (not the code you write to make it). Your diagram should have clear and informative labels (including units of measure) on the two axes and a legend (look up the MATLAB help for `legend`) that tells which plot is which. Make sure that your axis labels, tick labels, and legend are readable.

The first time MATLAB runs a function you define it compiles it. Because of this, the first run of your code may take longer than subsequent runs. To prevent this behavior from distorting your plots, run your code twice before saving the picture with your diagram.

4. From your experiments, for what sizes of m is the integral-image box filter preferable to direct convolution? And to separable convolution?

5. Show the image produced by your `boxFilter` for $m = 21$ on the image `DI` produced by `rings.m`. Also report the largest discrepancy between pixel values in the output from `boxFilter` and the corresponding pixels in the image computed by direct convolution. If B and C are the two output images and their domain is $\mathcal{D}$, this discrepancy is defined as follows:

$$\max_{\mathbf{p}\in\mathcal{D}} |B(\mathbf{p}) - C(\mathbf{p})| \; .$$

6. (Optional. No credit.) You may notice some faint gray bands in the filtered image you displayed in response to the previous problem. What causes these bands?

[questions continue on the next page]

Binomial Filters

Gaussian kernels can also be approximated by convolutions that use only sums and bit shifts, for use on computer architectures without a floating-point unit. This is a consequence of the following two facts:

- The repeated convolution of
$$\phi_1 = \frac{1}{2}\left[\begin{array}{cc} 1 & 1 \end{array}\right]^T$$
with itself yields kernels whose coefficients are equal to the binomial probability mass distribution $P_B(k, 1/2)$:
$$\begin{aligned}
\phi_1 &= \frac{1}{2}\left[\begin{array}{cc} 1 & 1 \end{array}\right]^T \\
\phi_2 &= \frac{1}{4}\left[\begin{array}{ccc} 1 & 2 & 1 \end{array}\right]^T \\
\phi_3 &= \frac{1}{8}\left[\begin{array}{cccc} 1 & 3 & 3 & 1 \end{array}\right]^T \\
\phi_4 &= \frac{1}{16}\left[\begin{array}{ccccc} 1 & 4 & 6 & 4 & 1 \end{array}\right]^T \\
&\vdots \\
\phi_k &= P_B\left(k, \frac{1}{2}\right) = \frac{1}{2^k}\left[\begin{array}{cccc} \binom{k}{0} & \binom{k}{1} & \ldots & \binom{k}{k} \end{array}\right]^T .
\end{aligned}$$
In these expressions, the superscript T denotes transposition.
- The binomial distribution for $p = 1/2$ approximates the Gaussian probability density with mean $k/2$ and variance $k/4$ increasingly well as k increases.

Thus, for large enough k, the two-dimensional Gaussian kernel with spread (standard deviation) $\sigma = k/4$, which is separable, can be approximated by the kernel
$$\Phi_k = \phi_k \phi_k^T .$$
While Φ_k is not isotropic[1], it is approximately isotropic for large k, because of the Gaussian approximation.

7. Write a MATLAB function with the following structure (template `binomialFilter.m` provided):

```
function img = binomialFilter(img, reps)

if nargin < 2 || isempty(reps)
    reps = 1;
end

img = uint16(img);

# Your code here

img = uint8(img);

    function img = Phi2(img)
        # Your code here

        function img = Phi1(img)
            # Your code here
        end
    end

end
```

The function `Phi1` convolves the input image with Φ_1 (capital!) without using any of the built-in MATLAB convolution or correlation functions or any multiplication or division operator. Division by 2 can be implemented with the built-in MATLAB function `bitshift`. Your function `Phi1` must take computational advantage of the separability of Φ_1.

[1] "Isotropic" means centrally symmetric.

The function `Phi2` uses `Phi1` to convolve the input image with Φ_2. The function `binomialFilter` uses `Phi2` to convolve the image `img` with Φ_{2r}, where r is the positive integer argument `reps`. For instance, `binomialFilter(img)` uses the default value `reps = 1` to convolve `img` with Φ_2, while `binomialFilter(img, 3)` convolves `img` with Φ_6. In this way, `binomialFilter` always convolves the image with a 2D binomial filter with an even subscript. As a consequence, the kernel has a square support with an odd number of pixels along each side, and is symmetric around a pixel at the center.

All the functions above should output only the `'valid'` part of the convolution, as explained earlier.

Hand in (in your PDF and as a separate `.m` file) both the entire code for `binomialFilter` (including the functions definitions it contains), and show in your PDF file the output `J` of

```
J = binomialFilter(I, 50);
```

where `I` is the image provided in `eye.png` with this assignment. Also state the size of `J` in pixels.

[questions continue on the next page]

Haar Pyramids

The Laplacian pyramid we saw in class is an example of a *bandpass pyramid*, in that every level of the pyramid contains information about image detail in some range of spatial frequency: Fine details are represented at low levels and coarser ones at higher levels. This organization helps in various image analysis tasks, as we will see throughout the course.

The Haar pyramid is a computationally simpler bandpass pyramid where detail at some scale is represented by quantities similar to image derivatives, as explained next. For simplicity, all images are assumed to have size $n \times n$ where $n = 2^k$ for some natural number k. The discussion below holds for color as well as gray-level images.

The input image I is split into the four *quarters*

$$Q_1 = I(a,a) \quad , \quad Q_2 = I(b,a) \quad , \quad Q_3 = I(a,b) \quad , \quad Q_4 = I(b,b)$$

where, using MATLAB notation, we define

$$a = 1:2:n \quad \text{and} \quad b = 2:2:n \; .$$

As an example, an 8×8 image would be split as follows, with each integer specifying which quarter a pixel belongs to:

$$\begin{bmatrix} 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 \\ 2 & 4 & 2 & 4 & 2 & 4 & 2 & 4 \\ 1 & 3 & \mathbf{1} & \mathbf{3} & 1 & 3 & 1 & 3 \\ 2 & 4 & \mathbf{2} & \mathbf{4} & 2 & 4 & 2 & 4 \\ 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 \\ 2 & 4 & 2 & 4 & 2 & 4 & 2 & 4 \\ 1 & 3 & 1 & 3 & 1 & 3 & 1 & 3 \\ 2 & 4 & 2 & 4 & 2 & 4 & 2 & 4 \end{bmatrix} .$$

Consider now one of the 2×2-pixel groups of I for which the top-left pixel is in quarter 1, exemplified by the square in red in the array above, and call its pixel values

$$\begin{bmatrix} q_1 & q_3 \\ q_2 & q_4 \end{bmatrix}$$

(these could be either color or gray-level values). The average of these four pixel values is

$$a = (q_1 + q_2 + q_3 + q_4)/4 \; .$$

An approximation to half of the derivative of image intensity in the vertical direction[2] can be computed as half the difference between the mean of q_1, q_3 and that of q_2, q_4:

$$d_1 = \frac{1}{2}\left(\frac{q_1 + q_3}{2} - \frac{q_2 + q_4}{2}\right) = (q_1 - q_2 + q_3 - q_4)/4 \; .$$

In the horizontal direction, one obtains similarly

$$d_2 = \frac{1}{2}\left(\frac{q_1 + q_2}{2} - \frac{q_3 + q_4}{2}\right) = (q_1 + q_2 - q_3 - q_4)/4 \; .$$

By themselves, the three numbers d_1, d_2, a cannot convey all the information about the four numbers q_1, q_2, q_3, q_4. However, one can add one more piece of detail information:

$$d_3 = (q_1 - q_2 - q_3 + q_4)/4 \; ,$$

which is half the difference between the mean values on the two diagonals. Then, the quadruple $\mathbf{d} = [d_1, d_2, d_3, a]^T$ conveys the same information as $\mathbf{q} = [q_1, q_2, q_3, q_4]^T$. To see this, rewrite the relations above in matrix form:

$$\mathbf{d} = E\mathbf{q} \quad \text{where} \quad E = \frac{1}{4}\begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} . \tag{1}$$

It is easy to check (say, by numerical computation in MATLAB) that E is invertible. Therefore, if we can compute $\mathbf{d}$ from $\mathbf{q}$, we can also compute $\mathbf{q}$ from $\mathbf{d}$: The two sets of numbers are equivalent.

[2]The signs of these derivatives, as well as the division by two, were chosen to make the reasoning for these calculations simpler. There are also deeper reasons, only briefly mentioned in this assignment.

There is much more to it, as the inverse of E is simple and closely related to E itself:

$$E^{-1} = F = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ 1 & -1 & -1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix} = 4E^T \ .$$

[This also implies that $2E$ is an orthogonal matrix, a fact that has important mathematical ramifications but will not be needed here.]

Because of these relations, one can break up the image I into its four quarters Q_1, Q_2, Q_3, Q_4 and compute the four images D_1, D_2, D_3, A by applying transformation (1) above to every quadruple $\mathbf{q}$ of corresponding pixels out of Q_1, Q_2, Q_3, Q_4.

The average image A is just a scaled-down version of I, because each pixel of A is the average of a 2×2 block of pixels of I. This key observation leads to a pyramid: If a superscript (ℓ) denotes level ℓ, just apply the same transformation to $A^{(1)}$ to obtain *its* scaled-down version $A^{(2)}$ and detail images $D_1^{(2)}$, $D_2^{(2)}$, $D_3^{(2)}$, and continue doing so until $A^{(\ell)}$ is a single pixel. The intermediate average images $A^{(\ell)}$ obtained during this process can be discarded, except for the very last one-pixel average, which needs to be stored with the pyramid.

The detail images at all levels, plus the one-pixel average of the input image, represent exactly the same information as the input image, using exactly the same number of values. However, the information is "rearranged by scale," in the sense that the detail images at a given level of the pyramid contain information about edges and image texture at the corresponding scale. For instance, very sharp edges and fine details are encoded by-and-large at low (high-resolution) levels of the pyramid and smoother edges are encoded by-and-large at higher (coarser-resolution) levels.

The computation of a Haar pyramid from an image is implemented in the function `image2pyramid.m` provided with this assignment. You can visualize the result by saying

```
I = imread('eye.png');
B = image2pyramid(I);
figure(1), clf, imshow(flattenBandPass(B))
```

where `eye.png` and `flattenBandPass.m` are provided as well. The latter function takes advantage of the fact that the number of values in `B` is the same as that in `I`. You may want to magnify the resulting picture to see what is going on.

8. After carefully studying the code provided, and in particular how equation (1) is implemented in `image2pyramid.m`, implement a MATLAB function with header

```
function [img, L] = pyramid2image(B)
```

that takes a bandpass pyramid `B` produced by `image2pyramid.m` and returns the original image `img` at full resolution.

Your function should also return `L`, a lowpass pyramid stored as a cell vector of `double` images. The pyramid `L` contains as many average images $A^{(\ell)}$ (that is, scaled down versions of `img`) as the input bandpass pyramid `B` has levels. To keep the correspondence between levels in the bandpass and lowpass pyramid simple, the full image should *not* be part of `L`, and this is why `img` is returned separately. Remember that lower levels in a pyramid represent higher resolution images. As an example, if the original image is 512×512, then `L` contains 9 square images with $256, 128, 64, 32, 16, 8, 4, 2, 1$ pixels on a side.

All calculations are done in `double`, and the images in `L` should be in `double` as well. However, the image `img` should be `uint8` (Hint: do `img = uint8(round(img));` at the very end, for proper rounding). Your code should work on both color and gray-level pyramids.

Hand in your code (in your PDF and as a separate `.m` file) and show in your PDF file the image `fL` returned by the following code:

```
load('bandPass.mat')
% B = green(B);      % You will uncomment this line later
[I, L] = pyramid2image(B);
fB = flattenBandPass(B);
fL = flattenLowPass(I, L);
figure(1), clf, imshow(fB)
figure(2), clf, imshow(fL)
```

The first instruction loads a bandpass pyramid `B` into the MATLAB workspace. The data file `bandPass.mat` and functions other than `pyramid2image.m` are provided with this assignment. Do *not* show `fB`.

9. Show in your PDF file the image `fL` produced by the code snippet in the previous problem, after uncommenting the second line. This snippet will check your code on the green band of the image. By itself, this band is a gray-level image. The function `green.m` is provided with this assignment.