Computer Vision and Image Processing

Homework 2 Scale-space blob detection

BHAVIK N GALA

UBIT: bhavikna

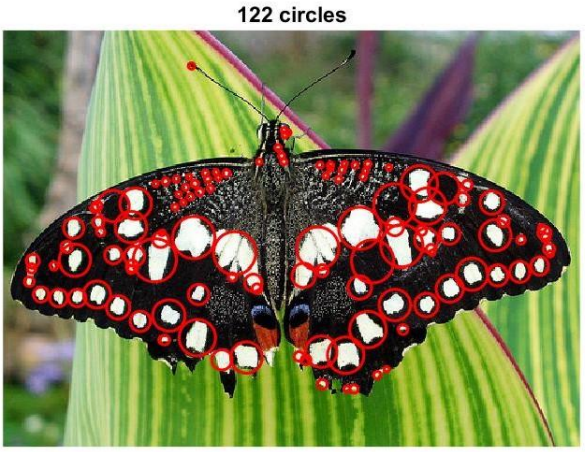
Person number: 50248608

Q 1. The output if your circle detector on all the images (four provided and four of your own choice), together with a comparison of running times for both the “efficient” and the “inefficient” implementation. (The comparison is graded, not the running times.)

Outputs:

Inefficient Method

Efficient Method



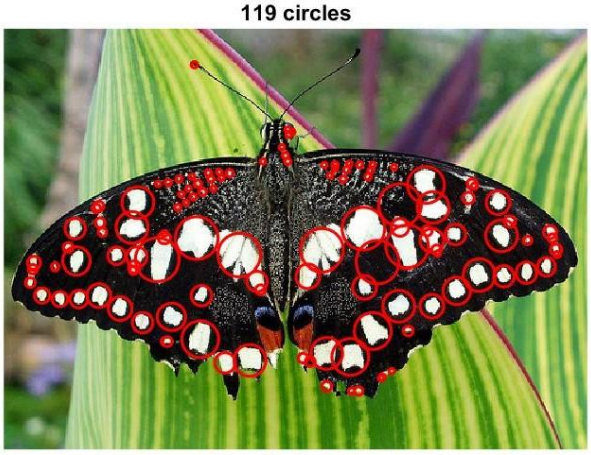
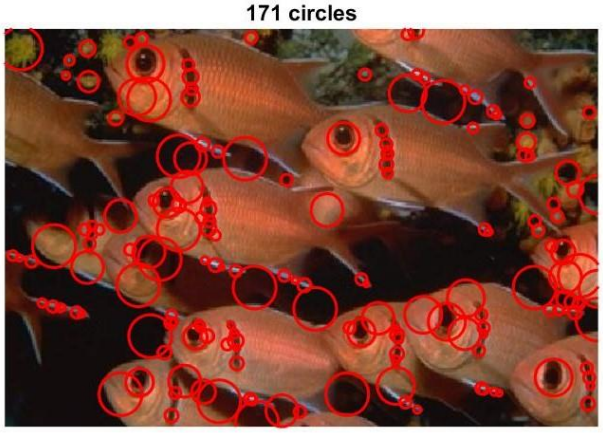


FIGURE 1 & 2: butterfly

FIGURE 3 & 4: sunflowers



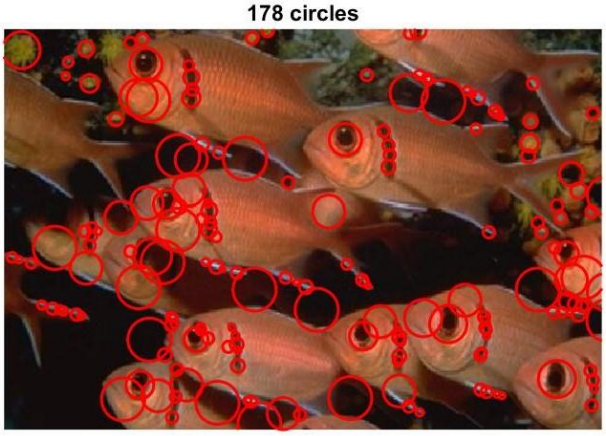
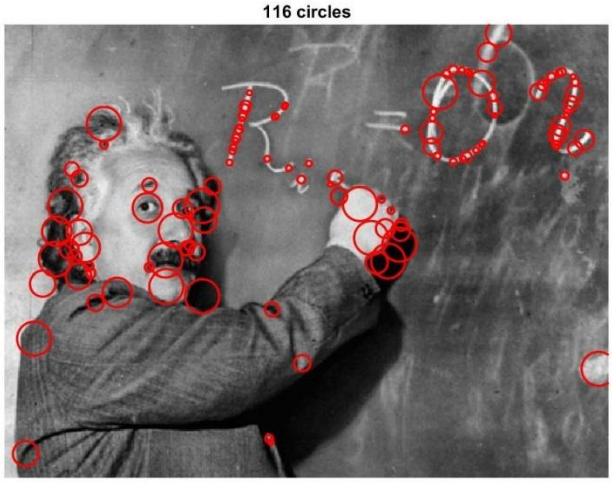


FIGURE 5 & 6: fishes



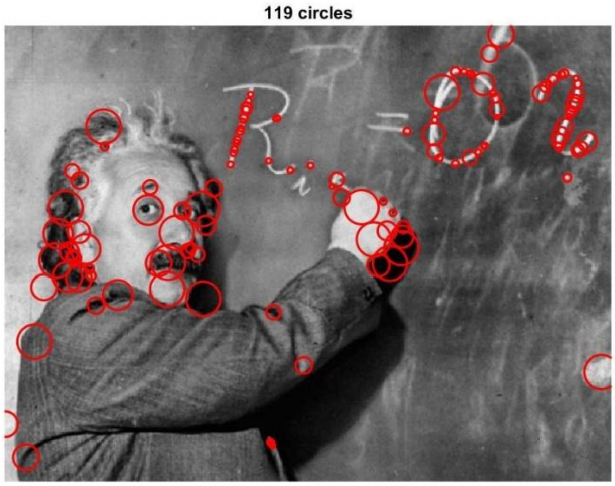
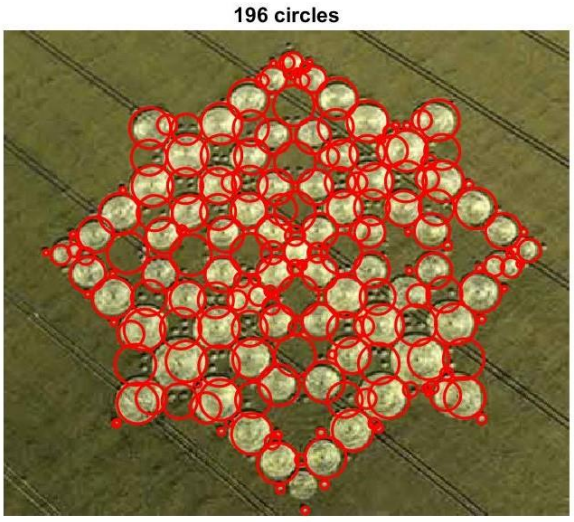


FIGURE 7 & 8: einstein

FIGURE 9 & 10: cheetah



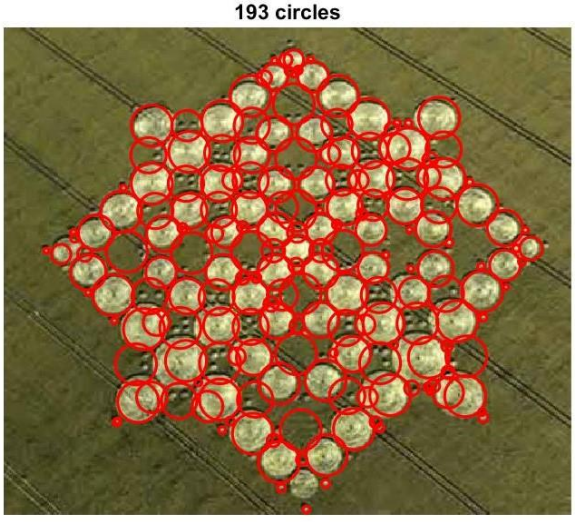
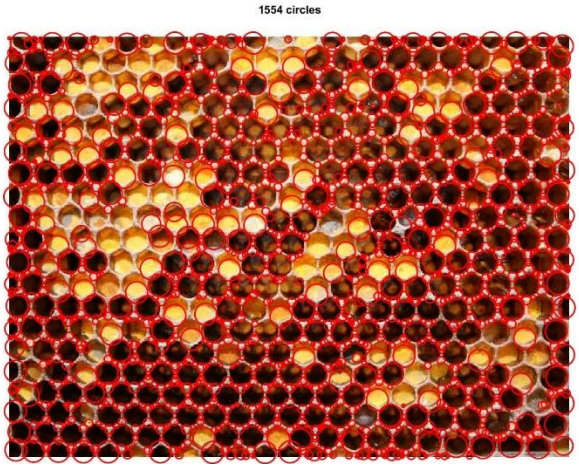


FIGURE 11 & 12: cropcircles

FIGURE 13 & 14: emojis



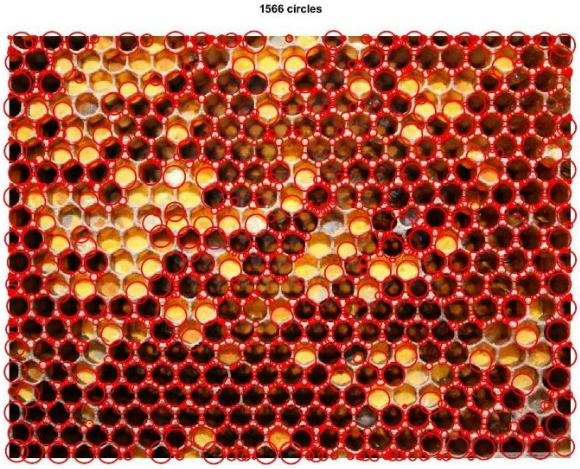


FIGURE 15 & 16: honeycomb

Running times for the 2 methods:

Sr No	Figure Name	Inefficient Method(sec)	Efficient Method(sec)	Difference(sec)
1	Butterfly	4.139704	3.260989	0.878715
2	Sunflowers	2.264561	2.214945	0.049616
3	Fishes	3.525738	2.965009	0.560729
4	Einstein	5.294507	4.954638	0.339869
5	Cheetah	4.717399	4.163711	0.553688
6	Cropcircles	4.472361	3.877832	0.594529
7	Emojis	2.463193	2.240015	0.223178
8	Honeycomb	15.964035	14.739292	1.224743

Comparison:

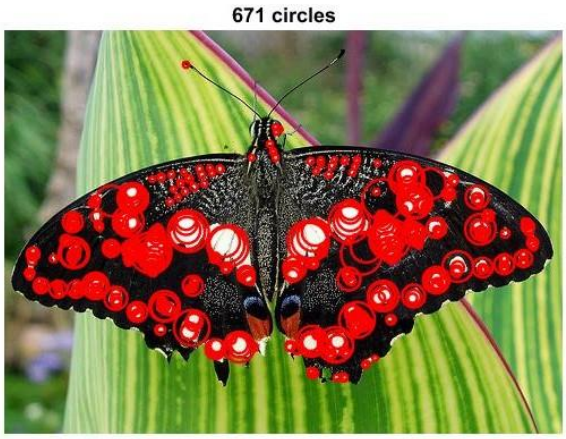
1. Downsampling and upsampling the image instead of applying filters of increasing kernel size does improve the speed of the program.
2. The reason for the speed up is as the filter kernel size remains constant and the image size decreases the number of computations decrease, increasing the speed, while in the case of increasing the scale of the filter coupled with the increase in the kernel size only increases the computations.
3. Some images showed a speed increase of around 1.2 second (honeycomb) while in case of some images the increase was only about 0.55 seconds (Einstein and honeycomb).
4. One major difference about the efficient method is that the interpolation shifted the centers of the blobs by few pixels to the left.
5. Another difference is that in some images more blobs were detected in the efficient method.
6. The interpolation method used was the default bicubic interpolation. Interpolation methods bicubic, cubic, and lanczos3 showed indistinguishable results. Hence I used the default bicubic interpolation.

Q 2. An explanation of any “interesting” implementation choices that you made.

1. I decided to use 1.618 as the initial sigma value, since everything in nature appears in this ratio/proportion, thus the blobs should also be of this ratio and get detected better.
2. Instead of putting a rigid value to threshold, I set it as a percentage of the maximum pixel value in the that layer, which is passed as a parameter. This gives a much better control for fine tuning.
3. In non-maximum suppression step I used `colfilt` function, which drastically reduced the execution time. I had previously used `nlfilter` function, and the executions time would be in double digits. After using `colfilt` function execution time dropped to single digits in seconds.
4. Smaller blobs nesting inside the bigger blobs is a common issue (figure 17). In order to deal with this, I traversed over the center array sorted in decreasing radius order. Then for any 2 centers, I checked for the condition:

   `Euclidean distance between the centers <= 1.3 * the radius of the bigger circle`

   This meant that the smaller blob was either inside the bigger blob or the majority of the smaller blob area was overlapping with the bigger blobs (figure 18). The above formula removed such blobs. This is a very naïve approach at removing overlapping blobs. Such non-maximum suppression technique is used in object detection algorithms.



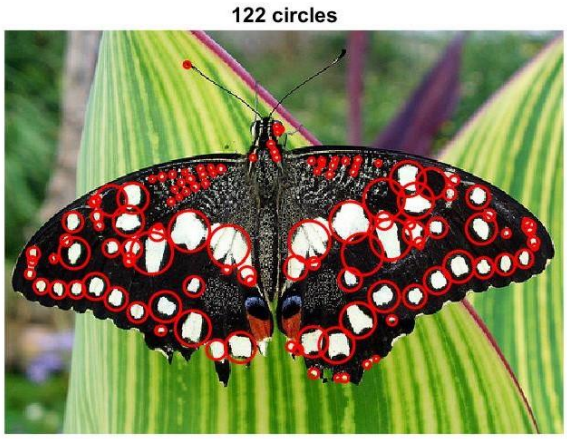
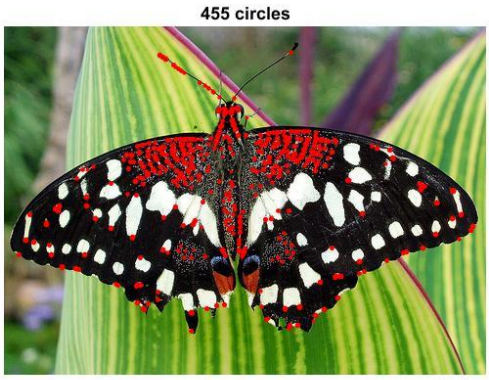


FIGURE 17 (left): Overlapping blobs. FIGURE 18 (right): Overlapping blobs removed with above formula.

Q 3. An explanation of parameter values you have tried and which ones you found to be optimal.

1. I first started with a very low sigma, tried 0.1, 0.3. 0.5 with the scale factor of 1.1. These values are just too small, the highest sigma at the 15th layer will be 1.14 with the initial sigma of 0.3. Therefore, values below 1 were just unsuitable. These values just detect lots of very small blobs (figure 19) and larger blobs are just left undetected.
2. Then I tried values above 1. I tried values such as 1.414, which is square root of 2, with a scaling factor of 1.1. Which did work good enough for small blobs (figure 20).
3. Since everything in nature has the golden ratio, I thought maybe initial sigma of 1.618 should be very well suited for the task. And thus I tried the initial sigma value of 1.618 with different scaling factors such as 1.1, 1.2, 1.3, etc. After some trial and error with the butterfly image, I found out that the scaling factor of 1.159 works really well (figure 21).



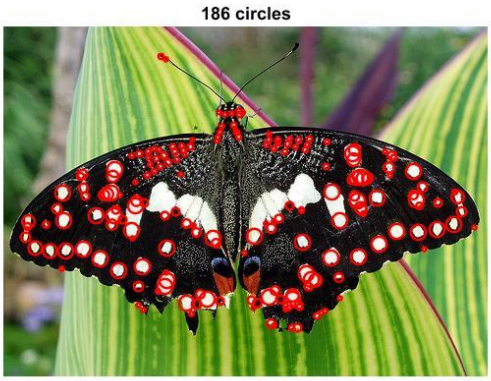
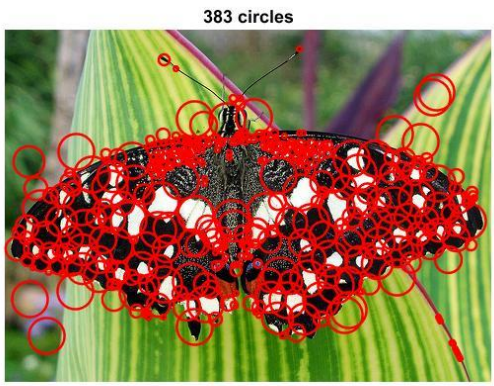


FIGURE 19 (left): Initial sigma is 0.3 and scaling factor is 1.1. Lots of small clusters of pixels are detected as blobs. Larger blobs, which are actually blobs, are not detected.
FIGURE 20 (right): Initial sigma is 1.414 and scaling factor is 1.1. Blobs are detected but many larger blobs are left undetected.

FIGURE 21: Initial sigma of 1.618 and scaling factor of 1.159.

4. The threshold value is set to 40% (figure 24) of the max value of the filter response in respective layer. Lower thresholds (figure 22) allow a lot of points to be detected as blobs and a high threshold allows comparatively lower number of blobs to be detected. A higher threshold (figure 23) gets rids of the blur, faded blobs in the background, blobs on the edges, and some blobs due to noise as well. However, too high threshold will lead to loss of some blobs in the foreground as well.



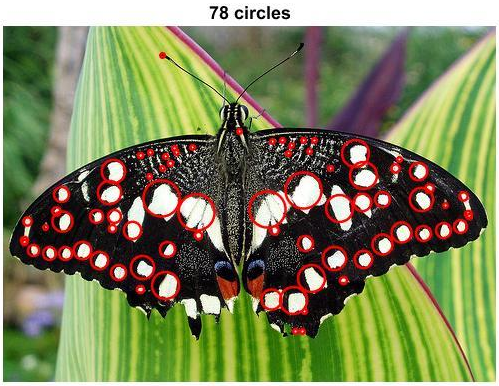


FIGURE 22 (left): Threshold value is 10% of the max value in the respective layer. Too many blobs due to a low threshold value; blobs in the background, and edges are detected.
FIGURE 23 (right): Threshold value is 60% of the max value in the respective layer. Number of blobs is less and ones formed by weaker pixels are removed in thresholding.

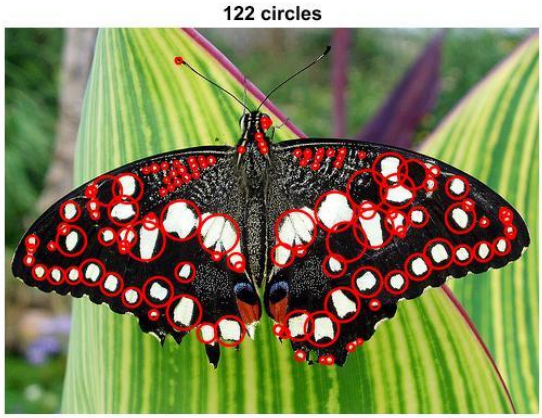


FIGURE 24: Threshold value is set to 40% of max in the respective layer.

5. Number of layers is also an important parameter. If the image is too big (figure 25), then the blob sizes will be too big and thus filter with smaller sigma values will not be able to detect big blobs. A solution to this problem is to increase the number the layers. Alternatively, the image can be resized to a smaller resolution so that all the blobs are detected (figure 26).

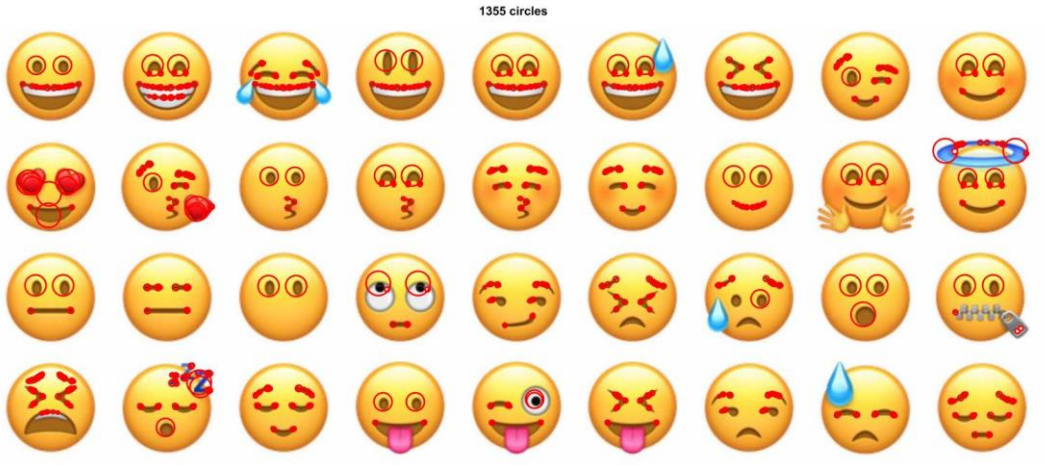


FIGURE 25: High resolution image. Only smaller blobs such as the eyes of the emojis are detected.

FIGURE 26: Low resolution image. Faces of emojis are detected as blobs.

Additional question: Filter width be odd or even?

In my opinion it should be odd. When the width is odd, the pixel on which the filter is applied will align directly with the center of the filter mask. In case of even width there will not be a center in the mask perse and thus any pixel on the in filter mask might have to be chosen as center pixel, which might lead to improper alignment of the filter function, Laplacian of Gaussian in our case, with the pixel on which the filter is applied. Meaning in case of the even width the peak of the LoG will not align with the pixel. Thus, the filter width should be odd.

Code documentation:

Flowchart for both the methods is as follows:

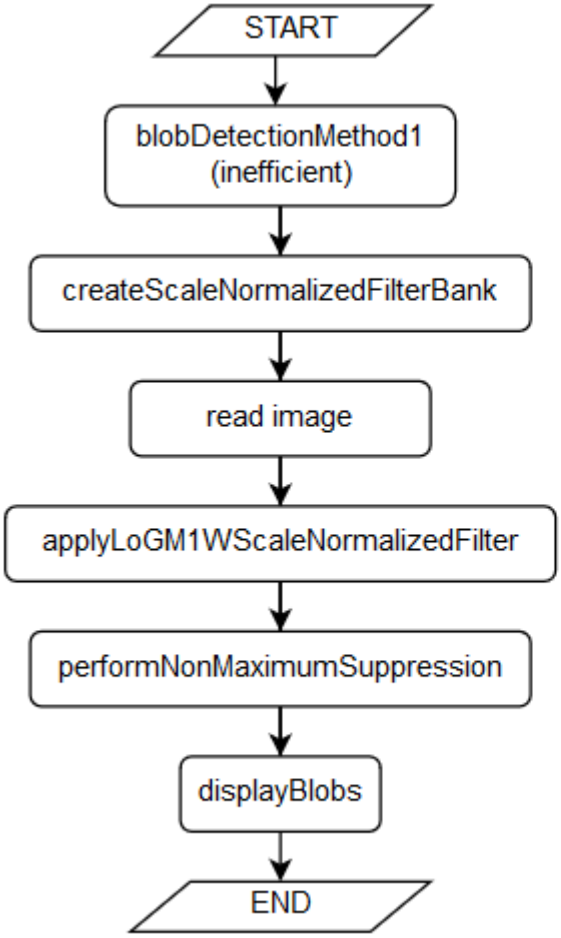


FIGURE 27: inefficient method

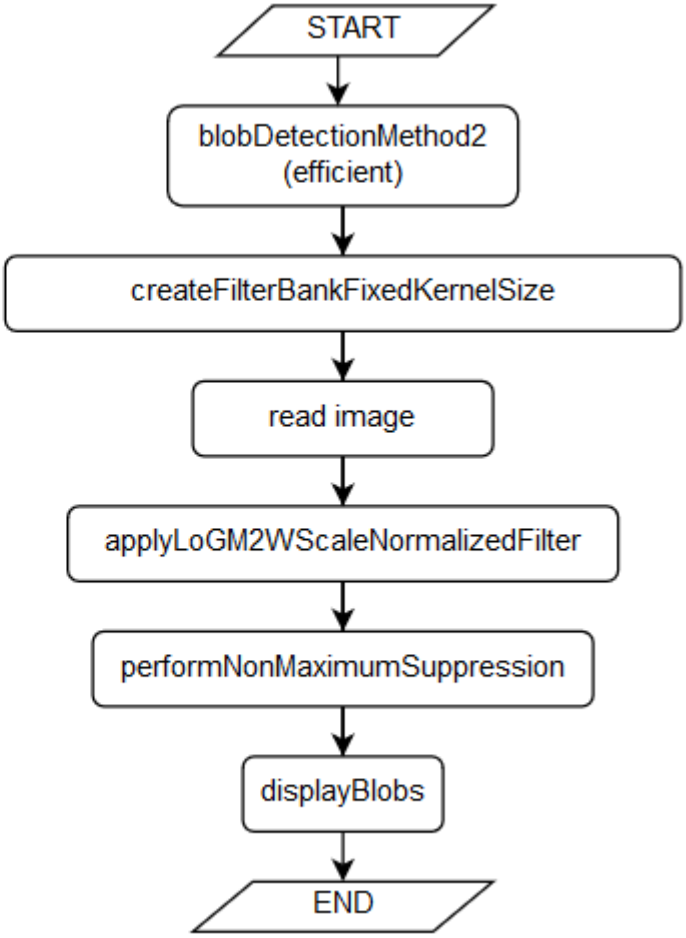


FIGURE 28: efficient method

1. General outline of both the methods is that first a filter bank is created – `createScaleNormalizedFilterBank` and `createScaleFilterBankFixedKernelSize`. In case of method 1 sigma is scaled generating different filter masks for each layer The filterbank is scale normalized. In case of method 2, array of scaling factor is generated along with a fixed filter mask. Here the filter is not scale normalized since filter mask is fixed, thus its magnitude will remain constant i.e. wont decrease with increase in scale.
2. After filter bank is generated, image is read and filter is applied of each layer – `applyLoGM1WScaleNormalizedFilter` and `applyLoGM2WScaleNormalizedFilter`.
3. Non-maximum suppression is performed on the filter responses of each layer – `performNonMaximumSuppression`.
4. After this step, centers and radius for the blobs are calculated and displayed on the original image – `displayBlobs`.

Folder structure is as follows: Parent folder is 50248608_hw2/

--./code/
--------blobDetections.m
--------blobDetectionMethod1.m
--------blobDetectionMethod2.m
--------createScaleNormalizedFilterBank.m
--------createFilterBankFixedKernelSize.m
--------applyLoGM1WScaleNormalizedFilter.m
--------applyLoGM2WScaleNormalizedFilter.m

--------performNonMaximumSuppression.m
--------removeBelowLocalMaxima.m
--------displayBlobs.m
--------describe.m
--------show_all_circles.m
--------harris.m
--./data/
--------myimages (images of my choice)
--------butterfly, einstein, fishes, sunflowers.jpg
--./results
--------method1/ (result for method 1)
--------method2/ (result for method 2)

Steps to run the code:

1. To run method 1 – inefficient method. Call function – **`blobDetectionMethod1.`**

   Its documentation is as follows:
   Function prototype: `blobDetectionMethod1(imPath, sigma, k, layers, thresholdPercent)`
   Function to perform blob detection on a given image. It generates filter masks for each layer and applies the filter for each layer on the image iteratively. The size of the filter mask is calculated by the formula: 2 * ceil(3 * sigma) + 1, which changes for every layer. After filtering is done, non-maximum suppression is applied on the image and the remainder blobs are displayed.

   INPUTS:

   a. imPath: Path of the image on which blob detection is to be performed.
   b. sigma: standard deviation of the first layer.
   c. k: scale factor used to scale the sigma for the following layer.
   d. layers: number of layers in the scale space.
   e. thresholdPercent: the percentage of the maximum response value to be used for thresholding. It should in the range 0-100.

   OUTPUTS:

   Displays the original image with detected blobs.

   RETURNS: None

2. To run method 2 – efficient method. Call function **`blobDetectionMethod2.`**

   Its documentation is as follows:
   Function prototype: `blobDetectionMethod2(imPath, sigma, k, layers, thresholdPercent)`
   Function to perform blob detection on a given image. It generates only one filter mask which is used for all the layers. The size of the filter mask is calculated as 2 * ceil(3 * sigma) + 1, which remains constant for all the layers. It generates an array of scales used to downsample and upsample the image in respective layer. After filtering is done, non-maximum suppression is performed and the remainder blobs are displayed on the original image.
   INPUTS:
   a. imPath: Path of the image on which blob detection is to be performed.
   b. sigma: standard deviation of the first layer.
   c. k: scale factor used to scale the sigma for the following layer.
   d. layers: number of layers in the scale space.
   e. thresholdPercent: the percentage of the maximum response value to be used for thresholding. It should in the range 0-100.

   OUTPUTS:

displays the original image with detected blobs

RETURNS: None

Important thing to note is that image path is provided in the argument instead of the image matrix. Secondly, the threshold provided is in terms of percentage with a scale of 0-100.

3. Alternatively, the code can be run from the script file **`blobDetections.m.`** The paths of the image needs to be specified in a cell array named **`imNames`**. It is declared and defined at the beginning of the script file. Edit the values to the location of the test images. The parameters can also be specified in the script file. Edit them and run the script file. The script will display images for the methods and output the time taken.

Code documentation for all other functions is as follows:

```
function [filter, scaleFactors] = createFilterBankFixedKernelSize(sigma,
scaleFactor, numScaleSpaceSize)

    % Create one filter masks and an array which contains scaling factors for
    % each layer in the scale space. The size of the mask is calculated by
    the
    % formula: 2 * ceil(3 * sigma) + 1.

    % INPUTS:
    % sigma: standard deviation of the first layer.
    % scaleFactor: scale factor for the scale space i.e k.
    % numScaleSpaceSize: number of layers in the scalespace.

    % RETURNS:
    % filter: filter mask which is used in all the layers
    % scaleFactors: array of scaling factors for every layer. Calculated by
    the
    % formula scaleFactor ^ (i - 1), i=1:numScaleSpaceSize



function [filterBank] = createScaleNormalizedFilterBank(sigma, ...
    scaleFactor, numScaleSpaceSize, displayFlag)

    % Create filter masks for each layer in the scale space. The size of the
    % mask is calculated by the formula: 2 * ceil(3 * sigma) + 1.

    % INPUTS:
    % sigma: standard deviation of the first layer.
    % scaleFactor: scale factor for the scale space i.e k.
    % numScaleSpaceSize: number of layers in the scalespace.
    % displayFlag: if true, surface plot for the filter is generated.

    % RETURNS:
    % filterBank: cell array of size [1 numScaleSpaceSize 2], contains sigma
    % and filter masks for each layer at {1, numScaleSpaceSize, 1} and
    % {1, numScaleSpaceSize, 2} respectively.


function [imFilterResponses] = applyLoGM1WScaleNormalizedFilter(im, ...
    filterBank, thresholdPercent, displayFlag)

    % This function iteratively applies LoG filter to the image using
    % corresponding filter mask for all the layers in the scale space. The
    % response of the filter is squared. Threshold is applied to the squared
    % response as the last stp of the function. Final response for each layer
    % along with their sigma value are stored in a cell array of
    % size(1, layers in scale space, 2) and returned.
```

```
    % INPUTS:
    % im: gray scale input image
    % filterBank: cell array of size(1, numScaleSpaceSize, 2), contains sigma
    % and filter masks for each layer
    % thresholdPercent: percentage of maximum in the response to be used as
    % threshold. It should be in range 0-100
    % displayFlag: if true, response at each step is displayed for each
    layer.

    % RETURNS:
    % imFilterResponses: cell array of size(1, numScaleSpaceSize, 2).
    Contains
    % sigma and final response for the image at each layer. sigma value is
    % stored at index{1, numScaleSpacceSize, 1} and response is stored at
    % index{1, numScaleSpaceSize, 2}


function [imFilterResponses] = applyLoGM2WScaleNormalizedFilter( im, ...
    filter, sigma, scaleFactors, thresholdPercent, displayFlag )

    % This function iteratively applies LoG filter to the image. The filter
    % mask is fixed. In each iteration, the image is first downsampled by the
    % factor k for the respective layer. Then the filter is applied to the
    % downsampled image. The filter response is then upsampled by the same
    % factor k. This response is squared and threshold is applied. This
    process
    % is repeated for all the layer. The final response and effective sigma
    % for each layer is stored in a cell array of size(1, numScaleSpaceSize,
    % 2) and returned.

    % INPUTS:
    % im: gray scale input image
    % filter: fixed filter mask which is used for all the layers.
    % sigma: standard deviation used to generate the filter mask.
    % scaleFactors: array of scaling factors for each layer.
    % thresholdPercent: percentage of maximum in the response to be used as
    % threshold. It should be in range 0-100
    % displayFlag: if true, response at each step is displayed for each
    layer.

    % RETURNS:
    % imFilterResponses: cell array of size(1, numScaleSpaceSize, 2).
    Contains
    % sigma and final response for the image at each layer. sigma value is
    % stored at index{1, numScaleSpacceSize, 1} and response is stored at
    % index{1, numScaleSpaceSize, 2}

function [imNonMaximum] = performNonMaximumSuppression( ...
    imFilterResponses, displayFlag)

    % This function performs non maximum suppression on the filter response.
    It
    % find the local maximum in the neighbourhood of 5 * 5 pixels and sets
    the
    % pixels less than the local maxima to zero. Matlab function colfilt with
    % sliding window and function handle removeBelowLocalMaxima are used for
    % the purpose.

    % INPUTS:
    % imFilterResponses: cell array of size(1, numScaleSpaceSize, 2).
    Contains
```

```
    % final response of the filter stage at index {1, numScaleSpaceSize, 2}
    and
    % sigma for respective layer at index {1, numScaleSpaceSize, 1}.
    % displayFlag: if true, display the non-maxima suppressed image for each
    % layer.

    % RETURNS:
    % imNonMaximum: cell arry of size(1, numScaleSpaceSize, 2). Contains
    % non-maxima suppressed image at index {1, numScaleSpaceSize, 2} and
    % sigma for respective layer at index {1, numScaleSpaceSize, 1}.

function Y = removeBelowLocalMaxima(X)

    % This function checks whether the centre pixel is greater than equal to
    % the maxima or not and then sets the output pixel to zero if it is less
    % than the local maxima or sets it to the value of the centre pixel if
    the
    % condition is satisfied.

    % INPUTS:
    % X: matrix of size(numPixelsInNeighbourHood, numNeighbourhoods). This
    % matrix is generated by the colfilt function.

    % RETURNS:
    % Y: row vector of size(1, number of columns in X). It contains either 0
    or
    % the value of the centre pixel depending on the condition.


function displayBlobs(im, imNonMaximum)

    % This function calculates radius for each blob detected in all the
    layers.
    % It then filters the centers, it selects the center with the largest
    % radius only. After this step it performs one more step to remove the
    % nested blobs. Once the centers are filtered, show_all_circles.m script
    is
    % called to display all the blobs on the original image.

    % INPUTS:
    % im: original image on which blob detection is performed.
    % imNonMaximum: non-maximum suppressed filter responses.

    % OUTPUTS:
    % displays the blobs on the original image.

    % RETURNS: None
```