Mini Project -3

DetectBlobs.m:

- Blobs are regions on the image which have constant properties.
- We first convert the image to double and grayscale

```
im=rgb2gray(im);
im=im2double(im);
```

- Here, to detect blobs, after many trials, I found that detecting blobs at different 25 scales,i.e. with 25 different sigma gives us the best output
- ```
  n=25;
  scalespace = zeros(h,w,n);
  inc=1.1;
  ```

  Here n is the number of scales is 25. The factor by which the scale is incremented at every stage is k=1.1
- Now we at every scale we apply a Laplacian of a Gaussian filter on the image
- ```
  i=2;
  while(k<=n)
  filt_size=2*ceil(3*i)+1;
  filter=i*i*fspecial('log',filt_size,i);
  ```

- Initially I set scale=2
- Then I create a LoG filter with kernel size as filt_size, which varies at every iteration with the change in the scale.
- To obtain a scale normalized LoG filter, I multiply the resultant filter by the square of the scale=i, in this case.
- ```
  Lnorm=((imfilter(im,filter,'same','circular')).^2);
  scalespace(:,:,k)=Lnorm;
  scale(k)=i;
  i=i*inc;
  k=k+1;
  ```

- After that, I apply my filter on the image using imfilter. To handle the borders, I pass 2 parameters-'same' and 'circular' to the imfilter. And store the resultant h*w image obtained after applying the LoG at scale k, in the scalespace array- scalespace(:,:,k)
- Then I increment 1.1 times,i,e, **i=i*1.1** to obtain the new scale
- After this I used ordfilt to find the maximum pixel at a point in a 3*3 neighborhood.
- ```
  ord=zeros(h,w,n);
  for i = 1:n
      ord(:,:,i) = ordfilt2(scalespace(:,:,i), 9, ones(3,3));
  end
  ```

- This way, we get the maximum among the 3*3 neighborhood at each scale

- Now to find the maximum among all the 26 neighbors we simply find the maximum among between the current and the upper and the lower layer.
- ```
  for i = 1:n
  ```

  ```
          ord(:,:,i) = max(ord(:,:,max(i-1,1)),ord(:,:,i));
  ```

  ```
          ord(:,:,i) = max(ord(:,:,min(i+1,n)),ord(:,:,i));
  ```

  ```
  end
  ```
- Now we need maximum of only the 26 neighborhoods. So we check the values of the pixel in the scalespace array at every scale and compare it with the the pixel values in the ord array. If they are the same then we store that value in the ord array
- ```
  ord = ord .* (ord == scalespace);
  ```
- Hence now our **ord** array consists of max of all 26 neighborhoods
- Now we set the threshold to 0.0045(I arrived to this value through experimenting multiple values and found this to give the best results) and find record the x and y coordinate, the radius and the number of blobs of every and the score i.e, the response of the image (pixel value) after applying the LoG filter on the image
- ```
  threshold=0.0045;
  ```

  ```
  ord(ord<threshold)=0;
  ```

  ```
  for i=1:n
  ```

  ```
      [rows, cols,value] = find(ord(:,:,i));
  ```

  ```
      numBlobs = length(rows);
  ```

  ```
      radii =  scale(i) * sqrt(2);
  ```

  ```
      radii = repmat(radii, numBlobs, 1);
  ```

  ```
      r = [r; rows];
  ```

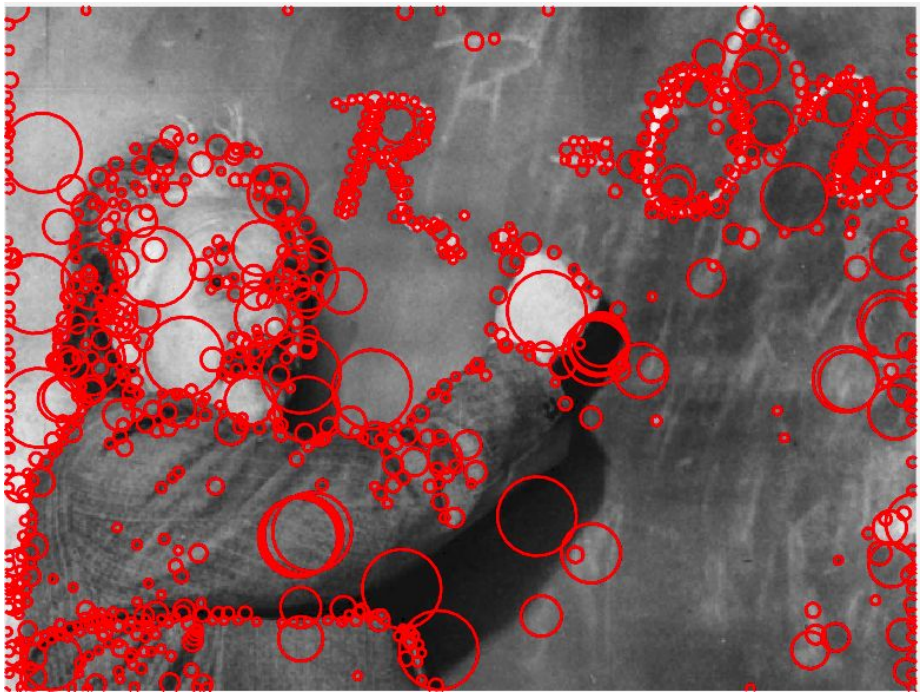
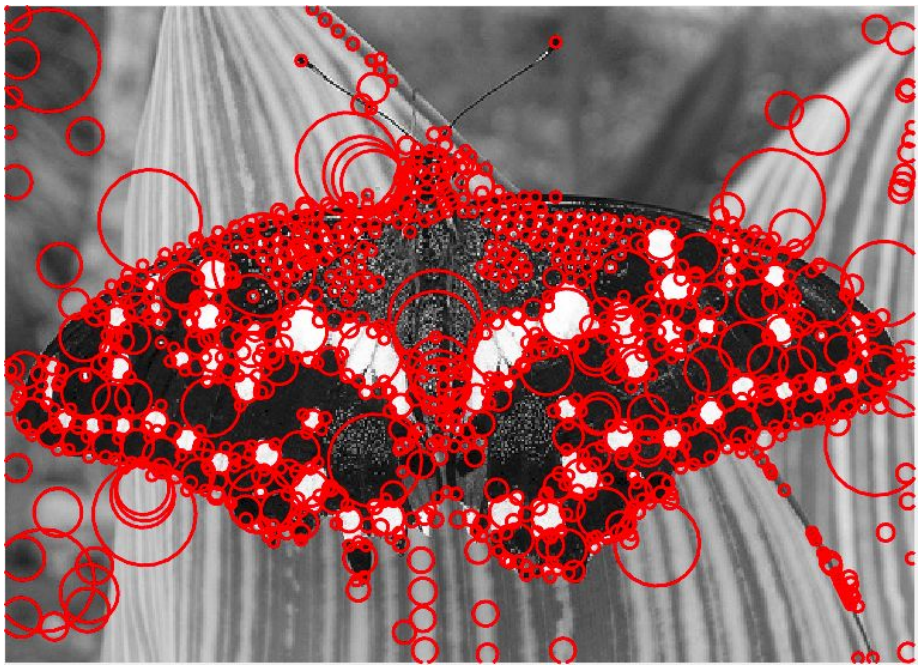
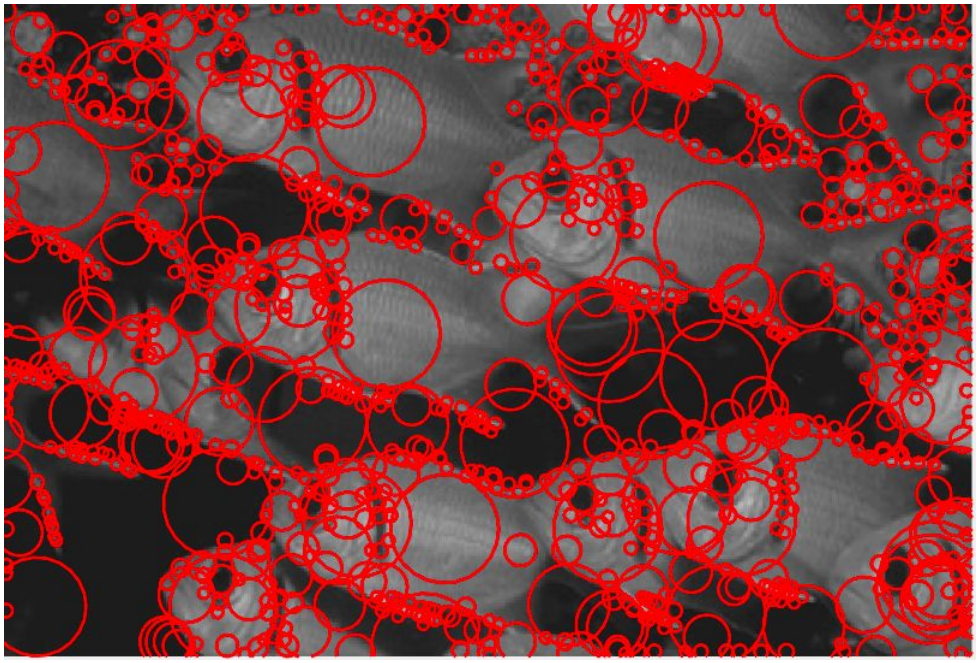
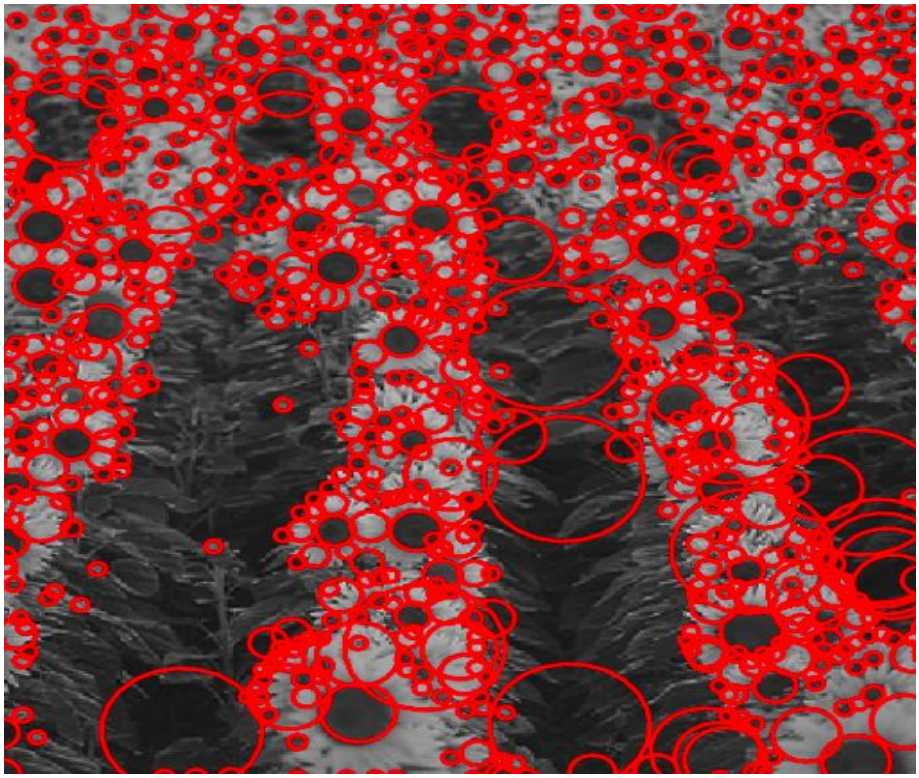
  ```
      c = [c; cols];
  ```

  ```
      rad = [rad; radii];
  ```

  ```
      v=[v;value];
  ```

  ```
  end
  ```

  ```
  blobs=[c,r,rad,v];
  ```

RESULTS:
Number of blobs=1000
  ```









2.computeMatches.m

- Here we are given sift descriptors for each blob of both the images,
- We find the sum of squared to distances of each each blob of the first image with each blob of the second image
- ```
  for i=1:n
    ssd=0;
    min1=intmax;
    min2=intmax;
    indice1=0;
      for j=1:m

          ssd=sum(sum((f1(i,:)-f2(j,:)).^2));
  ```

- Then we find the 2 minimum distances-min1 and min2

```
    if(ssd<min1)
            min1=ssd;
            indice1=j;
            if(min2==intmax)
                min2=min1;
            end


        elseif(ssd<min2)
            min2=ssd;

        end

    end
```

- If min1/min2 is lesser than the threshold of 0.8, we store that blob else we store 0
- ```
  if(min1/min2<0.8)
      arr1=[arr1;indice1];

    else
      arr1=[arr1;0];
    end
  ```

- Hence in the end we have an array matches of length Nx1 with mapping each blob to a blob in image 2 in the range of [1,M] if the threshold is lesser than 0.8

Estimating affine transformation using RANSAC

Ransac.m

- We use ransac here to find the inliers which will help us refine our transformation by getting rid of the noise in the image.
- Minimum number of iterations required to find the correct solution with a probability greater than 0.99 for fitting, k=3 points is 35.
- Now to approximate the affine transform, we take select 3 indices randomly.
- Save it in index
- If matches(index)==0, discard that index and find another index.
- If 2 or index match 2 or more blobs in matrix c1 to same blobs in c2, then discard that index as it will result in a system of linear equations which are not independent , thus not giving unique results for the affine transform matrix
- ```
  while(count<=3)%loop till we find 3 indeices

          if(matches(r,1)~=0)% check if a match exists

          rand_arr=[rand_arr;r];%store it
          count=count+1;
          end
          r=randi([1,N],1,1);
          if(length(rand_arr)~=0)
          for w=1:length(rand_arr)
              num=rand_arr(w);

          while(r==num)% check if 2 or more blobs map to same blob
              r=randi([1,N],1,1);%if yes, select another indice or
  else it will result in a system
                                % of linear equations which are not
  independent.
          end

      end
          end
  end
  ```
  ```

- Now we store the x and y coordinate of the 3 blobs with index belonging to the index_array, in the source image in matrix A
- ```
  for i=1:length(rand_arr)
  index=rand_arr(i);

  x1=c1(index,1);
  y1=c1(index,2);

  A=[A;([x1 y1])];%3X2
  ```

- Similarly, we store the corresponding matches(index) blobs of the second image in B

- ```
  x_ans=c2(matches(index),1);
  y_ans=c2(matches(index),2);

  B=[B;([x_ans,y_ans,1])];%3X3
  end
  ```

- **Matrix A** has dimensions **3x2** and dimensions of **B** are **3x3**
- Now we find the approximate affine transform matrix by taking the inverse
- We make the adjustments in the matrix dimensions to find the transform i.e our matrix
- **transform=B\A which is a 3x2 matrix**
- Now we create a temp matrix containing the **x and y** coordinates of the the blobs of the c2 matrix, which is the target image matrix in our case.
- ```
  M=size(c2,1);
  temp=[];
  for i=1:M
  temp=[temp;([c2(i,1) c2(i,2)])];
  end

  adjust=ones(M,1);
  temp=[temp adjust];
  ```

-
- We make temp a **Mx3** matrix
- Now we multiply, the approximate transform matrix obtained with the temp matrix.
-
- ```
  result=temp*transform;
  ```

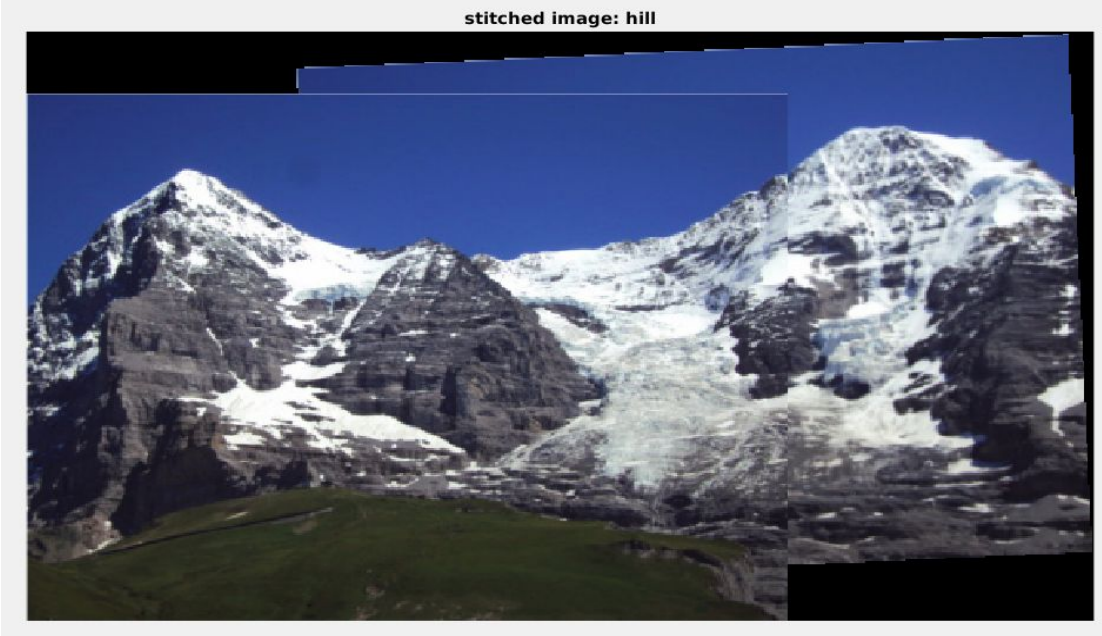
- **Result is a Mx2 (temp(Mx3)*transform(3x2)) matrix**
- Result contains the expected values of the x and y coordinates if the source image
- So now we compare the **result** matrix with the x and y coordinates of the source image.
-
- ```
  inlier=0;


  optimal=[];
  for i=1:N


  if(matches(i)~=0)
  x1=(c1(i,1));
  y1=(c1(i,2));
  if(((x1-result(matches(i),1))^2+((y1)-result(matches(i),2))^2)<=3)
  inlier=inlier+1;
  optimal=[optimal;i];
  end
  end
  end
  ```
- We find the squared distances and if it is less than threshold we append it to the inlier array .
- FInally if number of inliers found exceed the maximum number of inliers found till now, we store that inliers array and the transform matrix
- ```
  if(inlier>maxinlier)
      inliers=optimal;
      maxinlier=inlier;
      transf=transform';
  ```

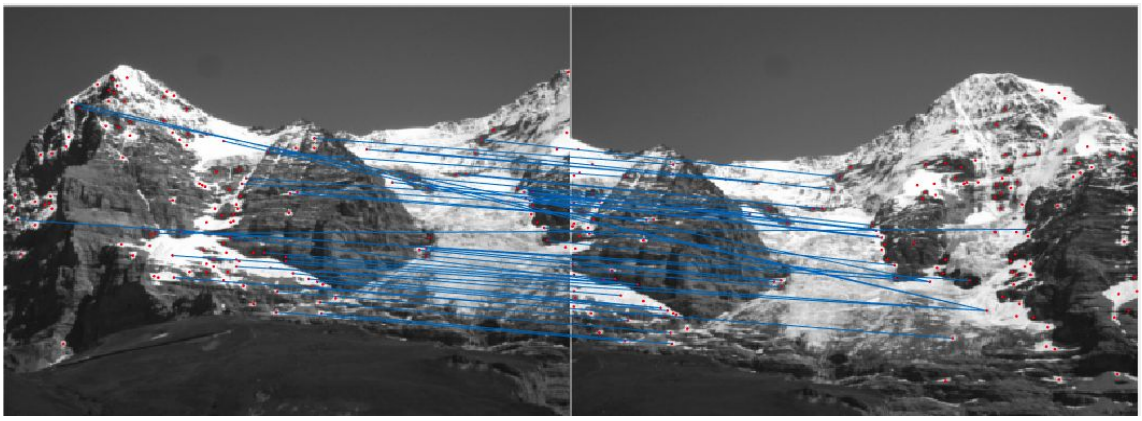
RESULTS:

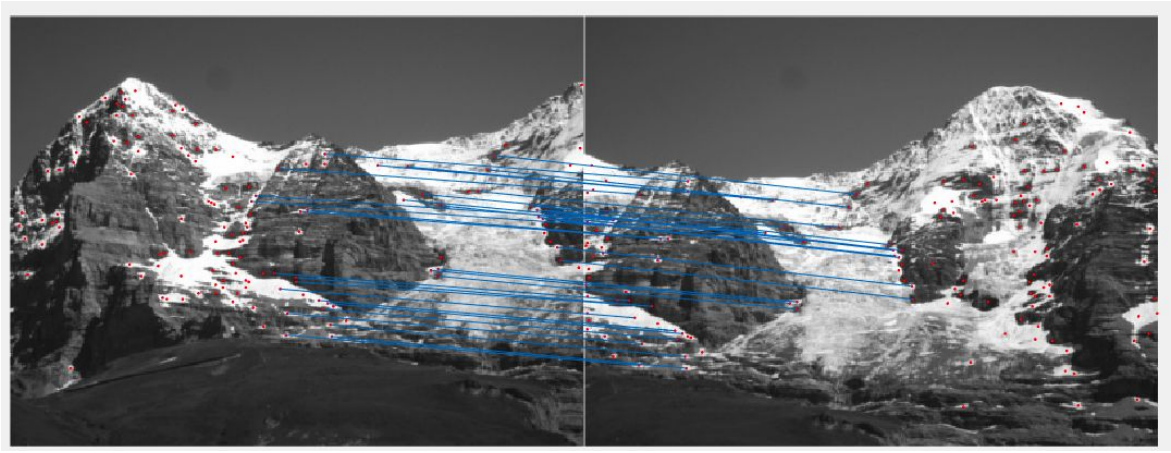
I obtained these results by experimenting with different threshold values to find inliers for ransac for different images.

1. **Hill**

affine: 1.0163 0.0421 141.4708
-0.0483 0.9848 -14.7708

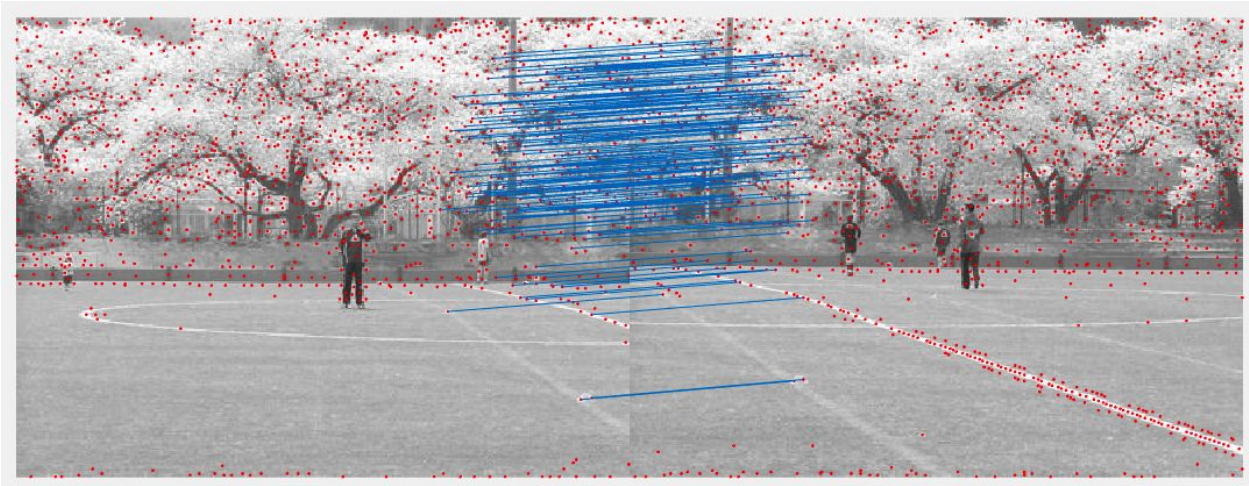






2.field:

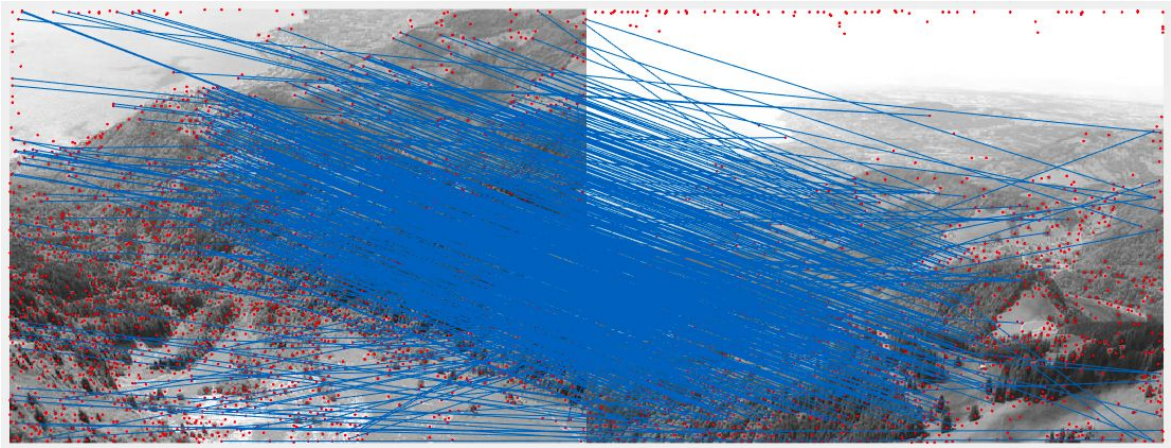
0.9945 0.0121 256.8629
0.0000 1.0000 10.0000

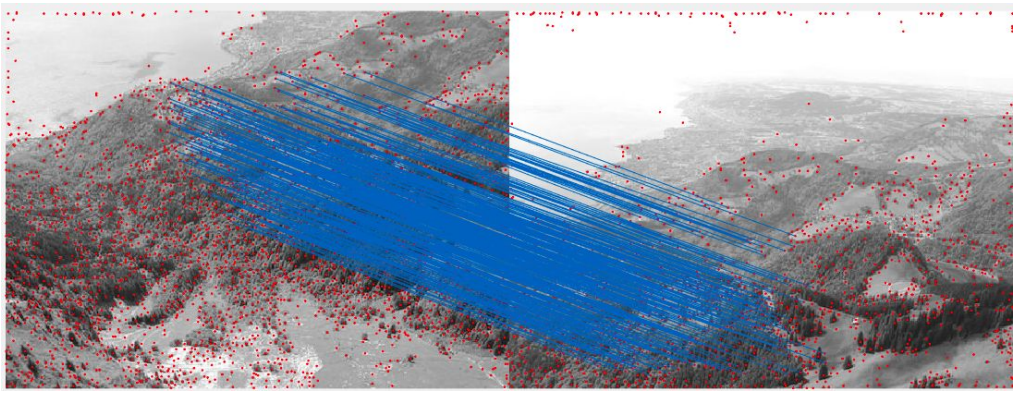


3. Ledge:

0.9752 -0.0537 148.7879

0.0478 0.9758 -131.3968

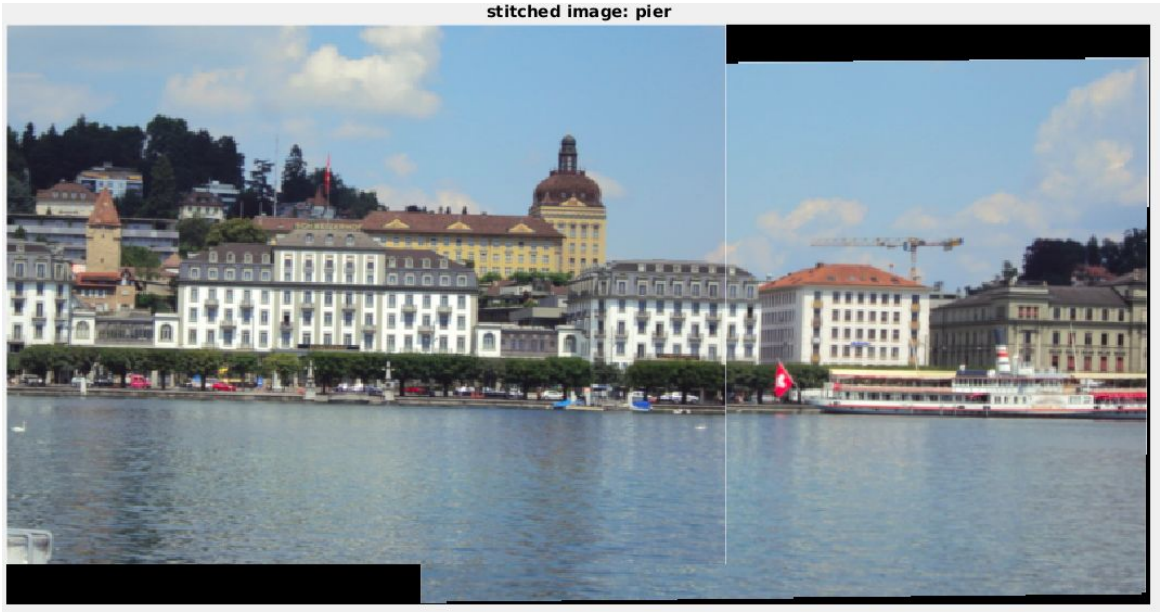


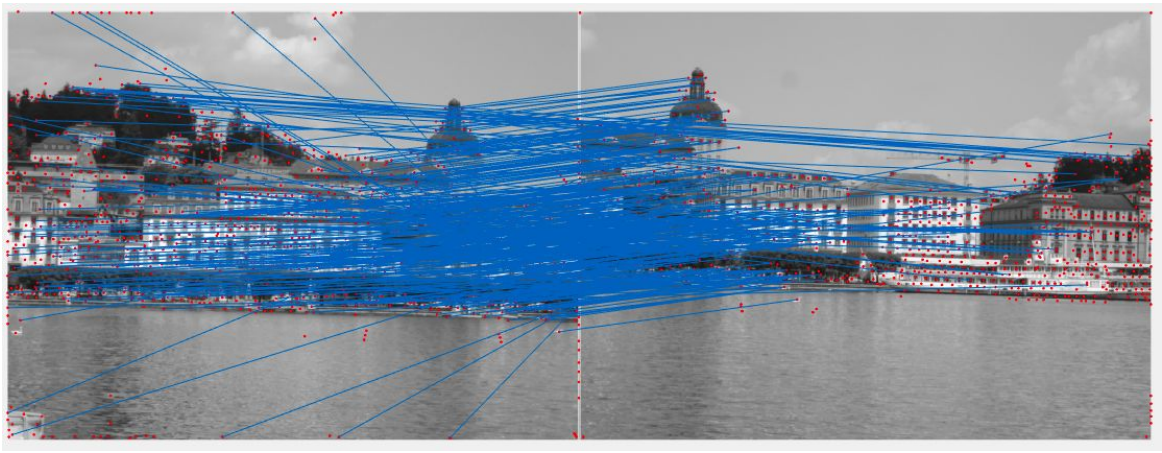


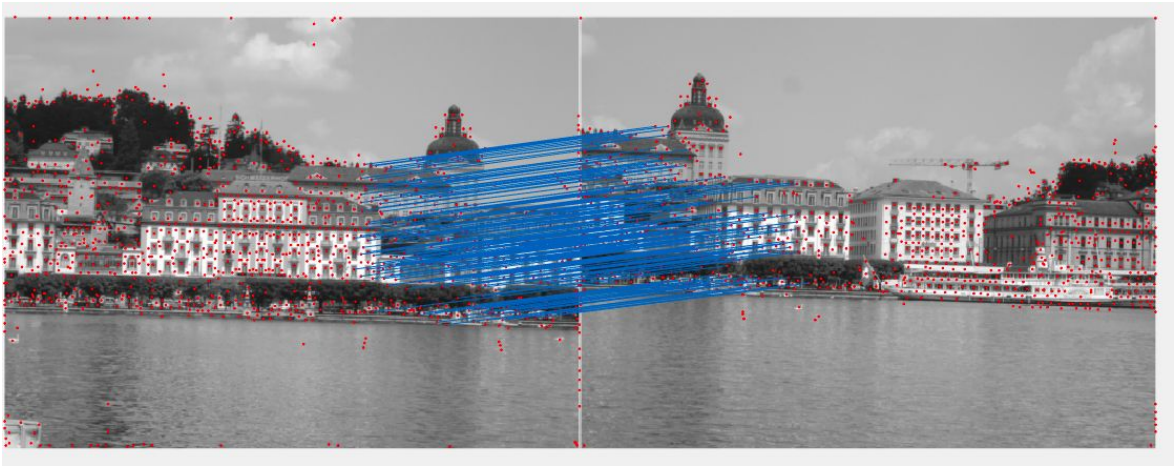
4. Pier

1.0019 0.0112 286.5251

0 1.0000 27.0000

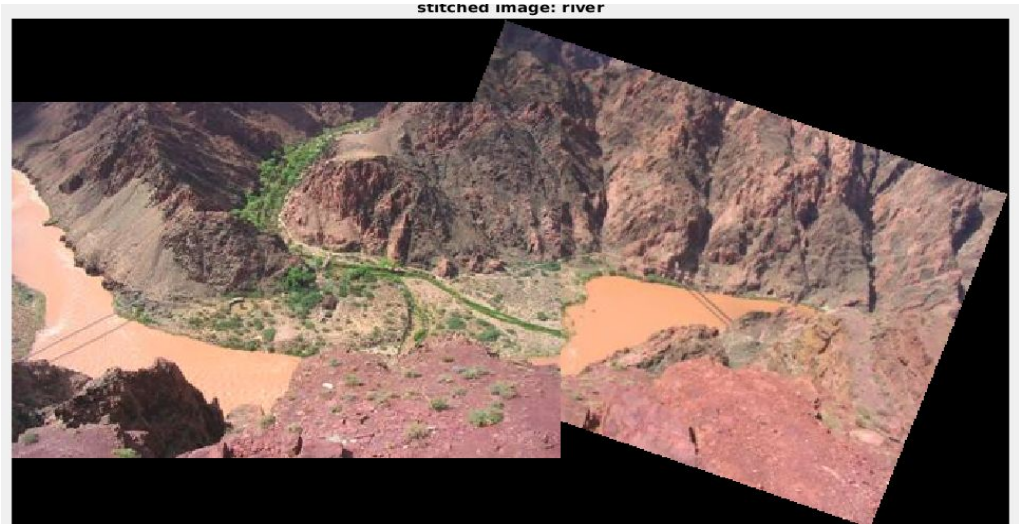


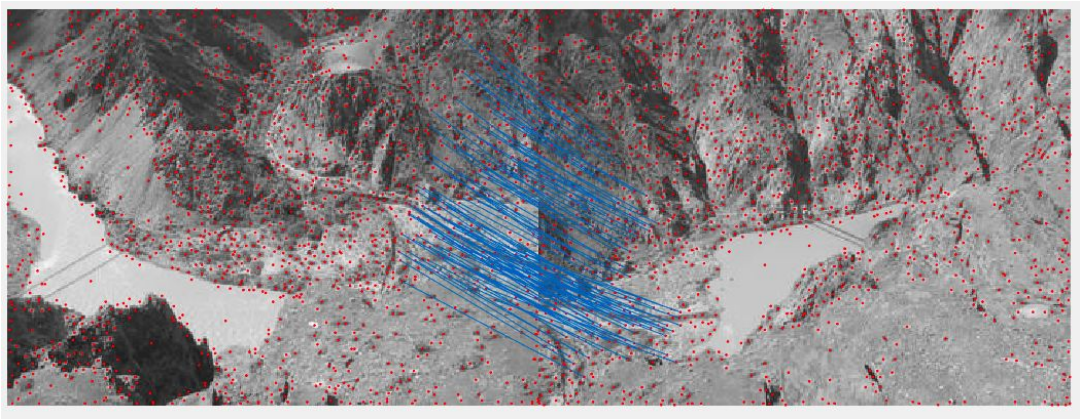


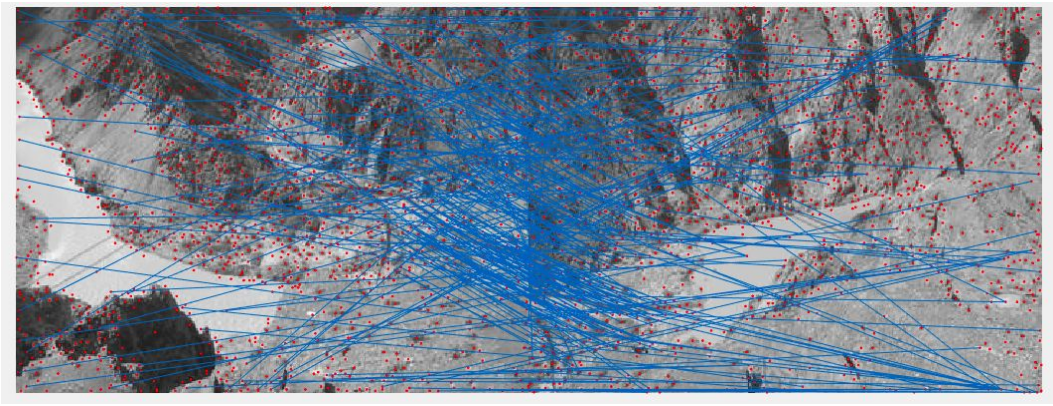


5. river

0.9175 -0.3359 323.3835
0.3654 0.9410 -62.3269

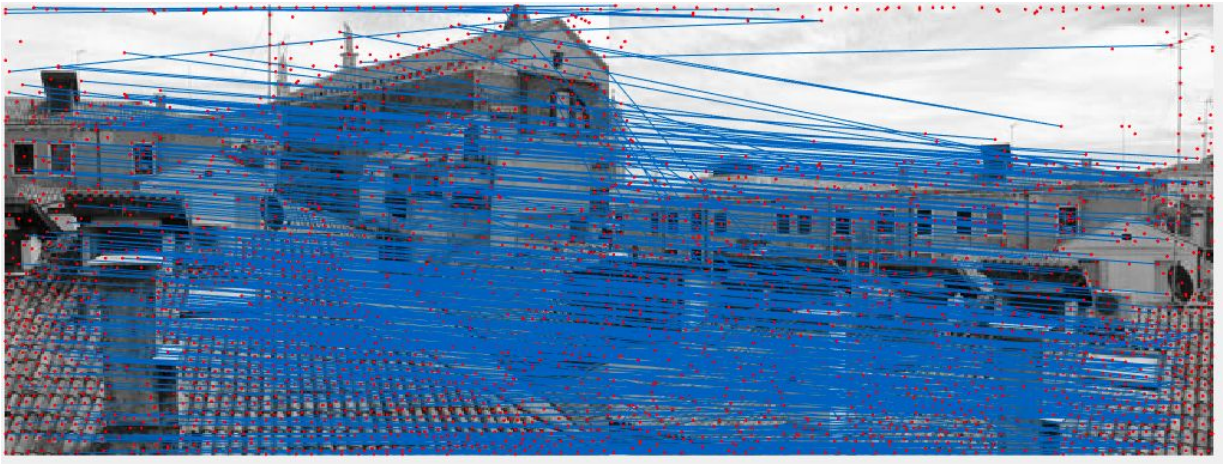


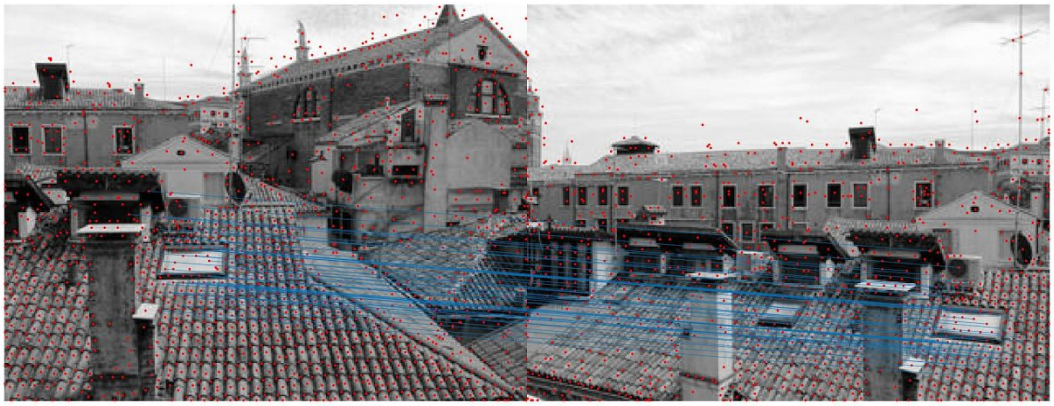




6. roof

1.1173 0.1038 -205.1173
-0.1475 1.1350 -23.8525

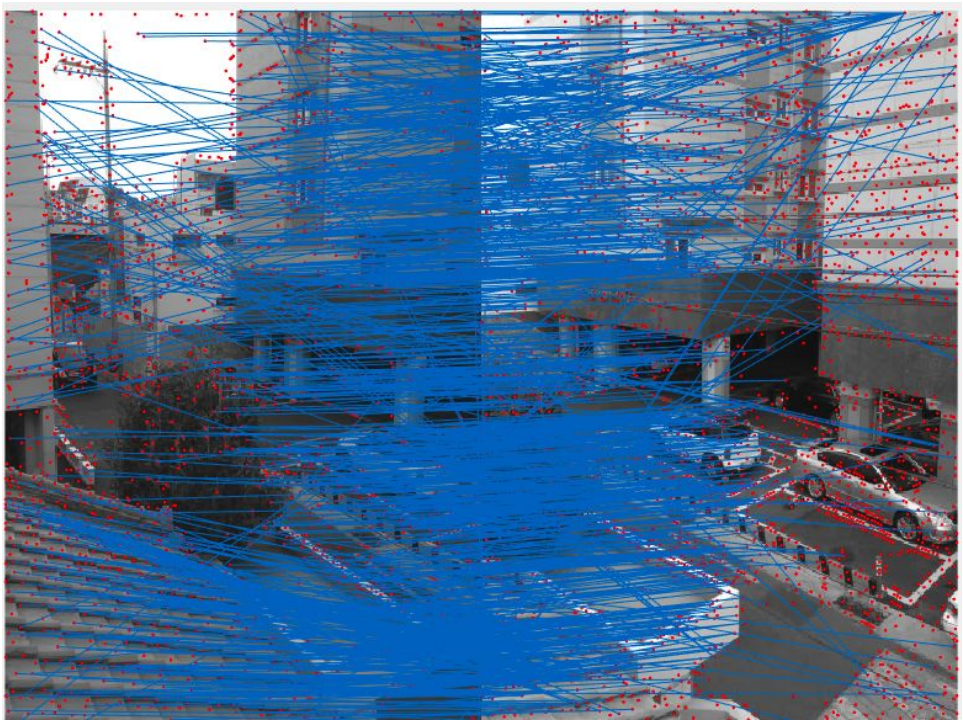


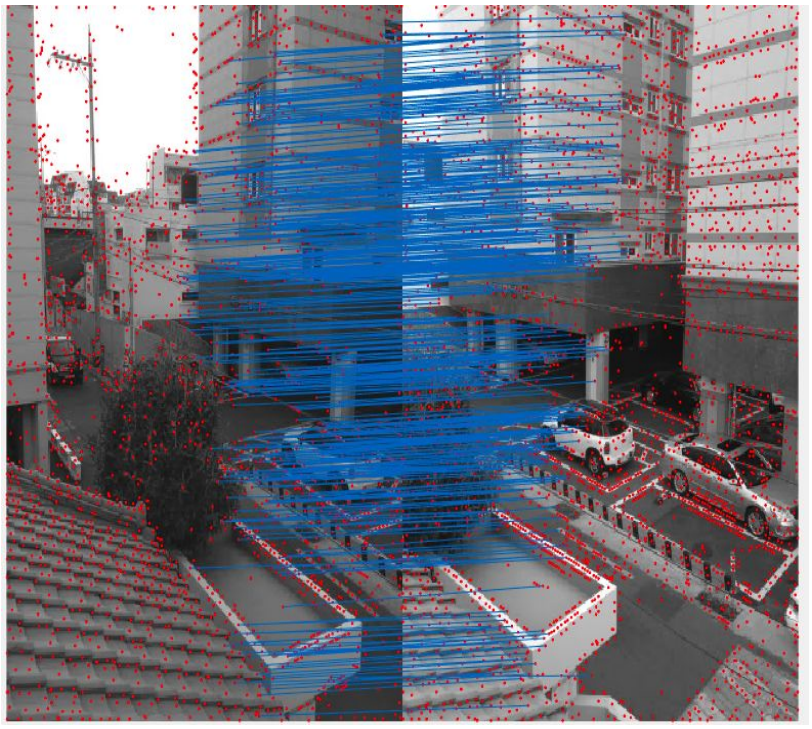


.

7. Building:

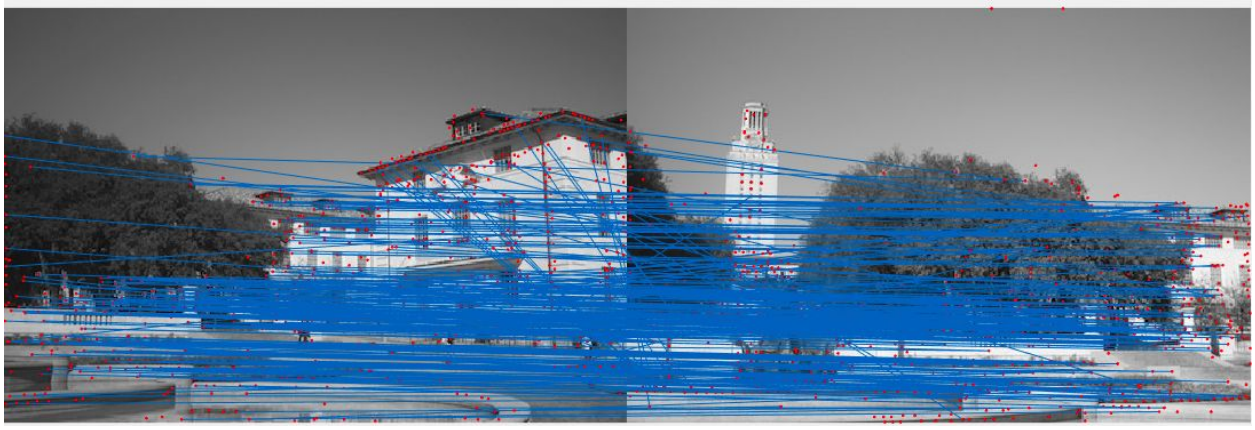
Affine:
0.9970 -0.0381 124.5071
-0.0431 0.9935 15.6395

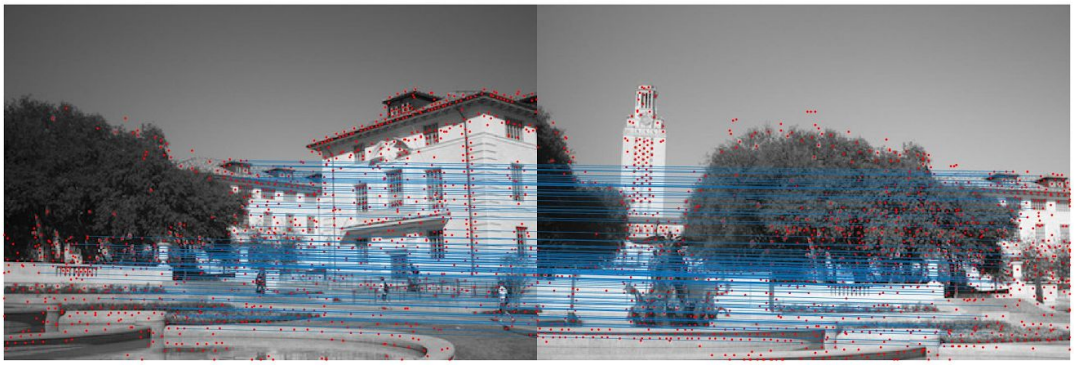




8. uttower

1.0066 -0.0758 -197.0714
0.0309 0.9794 -22.0000





EXTRA CREDIT 1:

1. **Blob detection using iteratively downsample the image and filter it with the kernel of fixed size.**

1. Here instead of changing the kernel size I iteratively down downsample the image
2. set sigma=2
3. Find the kernel size and the LoG filter.
4. Set the number of levels . here n=25
5. FIlter the image at each level
6. Upsample the filtered image using imresize and bicubic interpolation and save it
7. Downsample the original image by factor of (1/(k^i)). Here k is the increment factor which is set to 1.1

•

```
filt_size = 2*ceil(3*sigma)+1;
LoG =  sigma^2 * fspecial('log', filt_size, sigma);
inc=1.1
imRes = im;
for i = 1:n
```

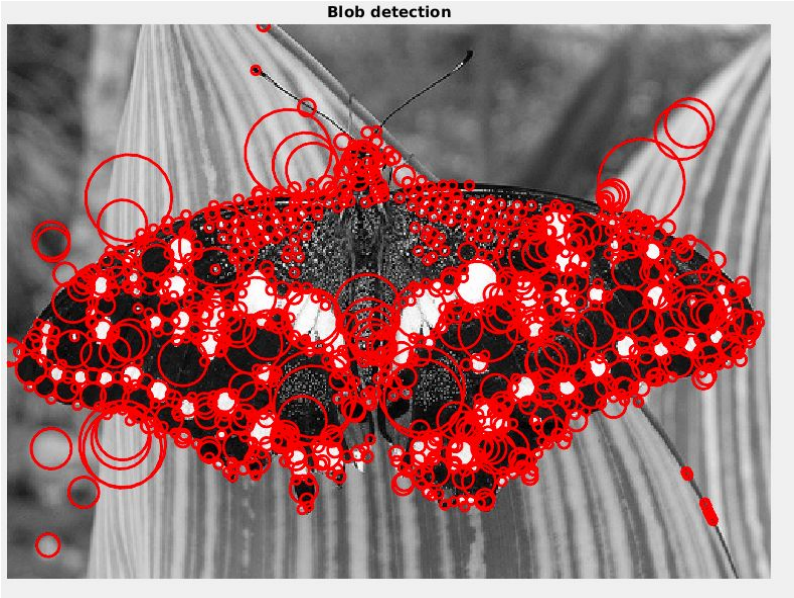
```
    %fprintf('Sigma %f\n', sigma * k^(i-1));
    imFiltered = imfilter(imRes, LoG, 'same', 'replicate'); % filter the
image with LoG
    % no scale normalization is needed:  the filter
    % remains the same size while the image is downsampled
    % so response of the filter is scale-invariant
    imFiltered = imFiltered .^ 2; % save square of the response for
current level

    % upsample the LoG response to the original image size
    scalespace(:,:,i) = imresize(imFiltered, size(im), 'bicubic'); %
bilinear supersampling will result in a loss of spatial resolution
    if i < n
        imRes = imresize(im, 1/(1.1^i), 'bicubic');
    end
end
```

COMPARING METHODS:

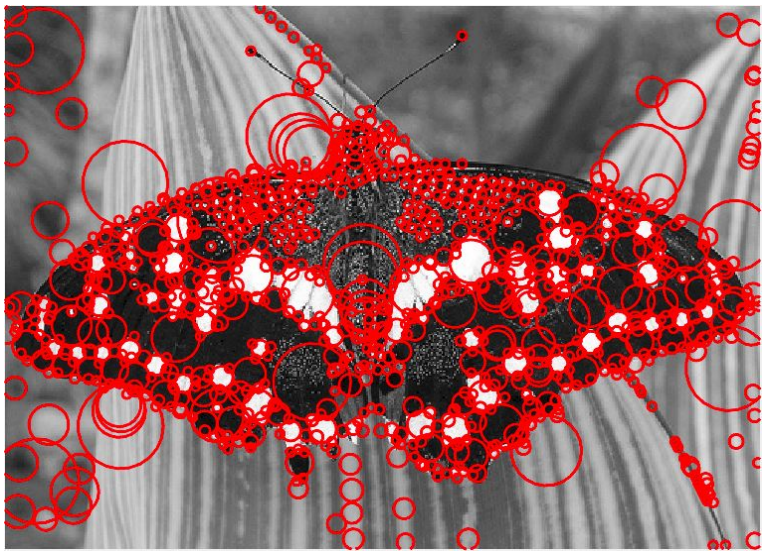
Elapsed time for iteratively filtering the image with kernel of increasing size is 0.446516 seconds.
Elapsed time for iteratively downsampling the image and filtering it with the kernel of fixed size is 0.296909 seconds.

RESULTS:

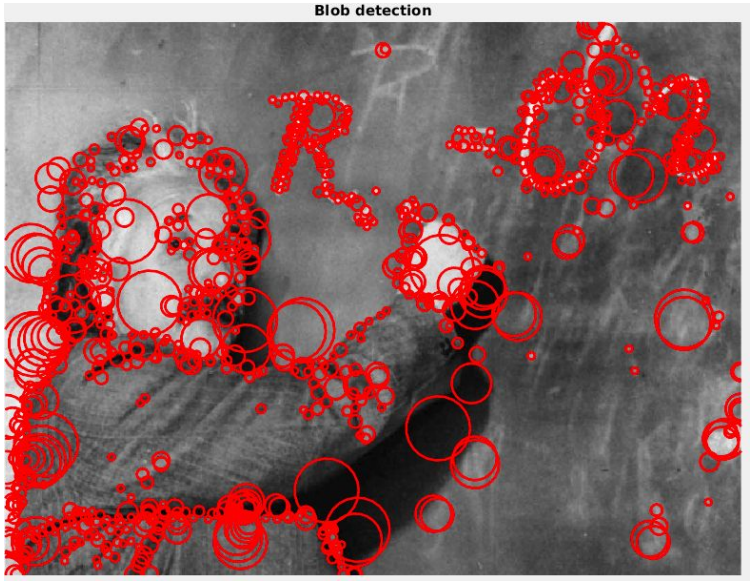
1. **Faster method-downsampling**



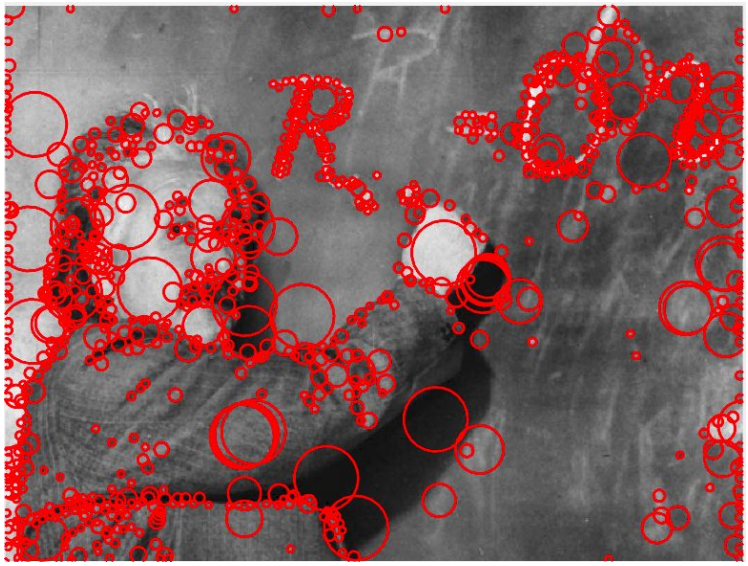
2. Slower method-changin kernel size



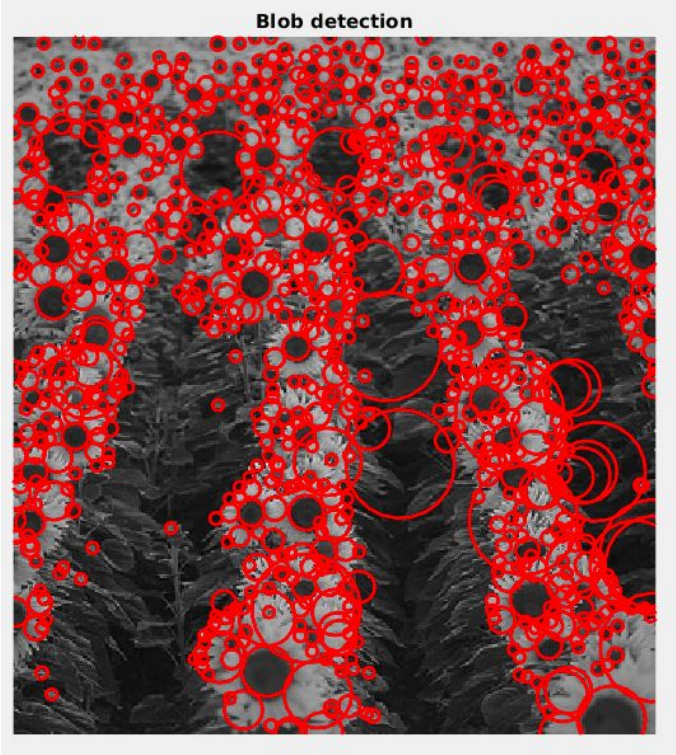
1. **Faster method**



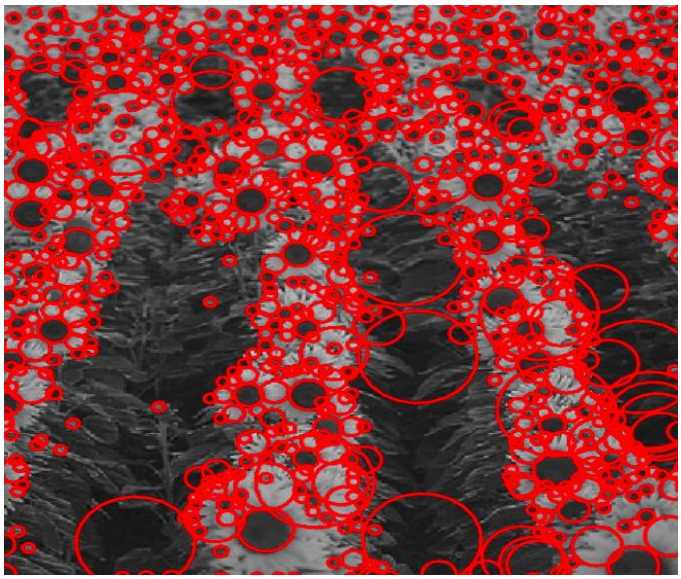
2.slower method



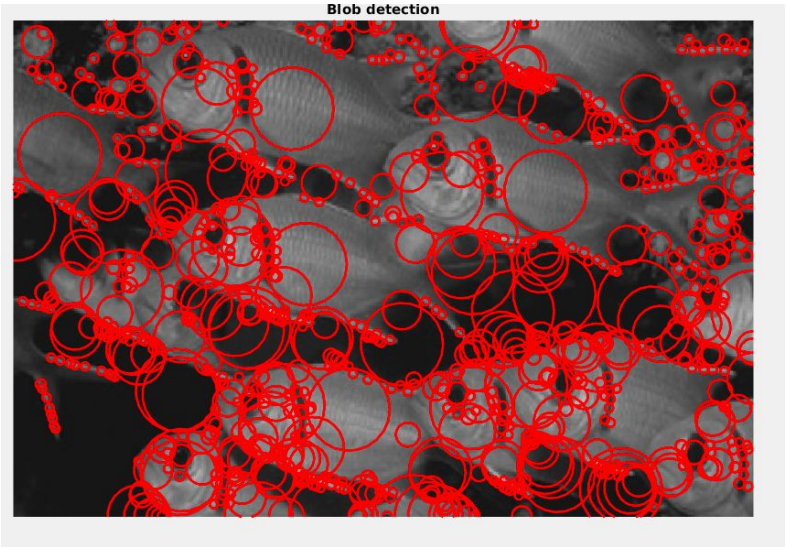
3. Faster method



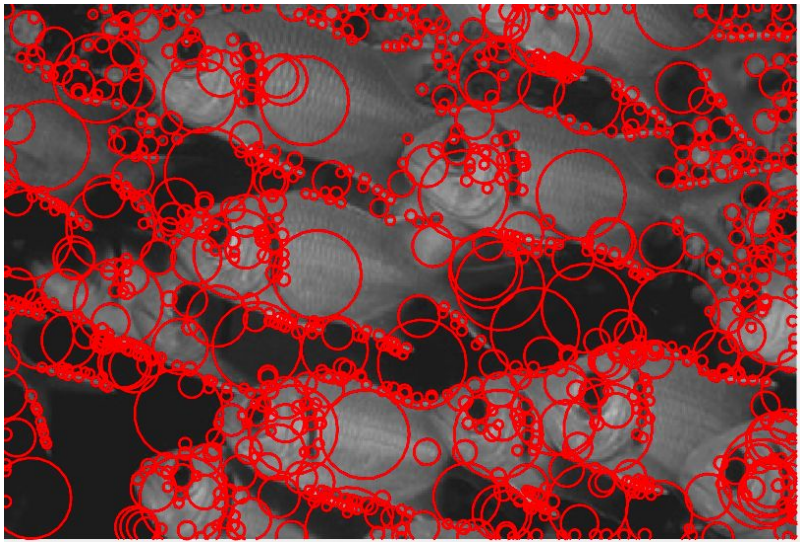
Slower method:



4.



Slower method

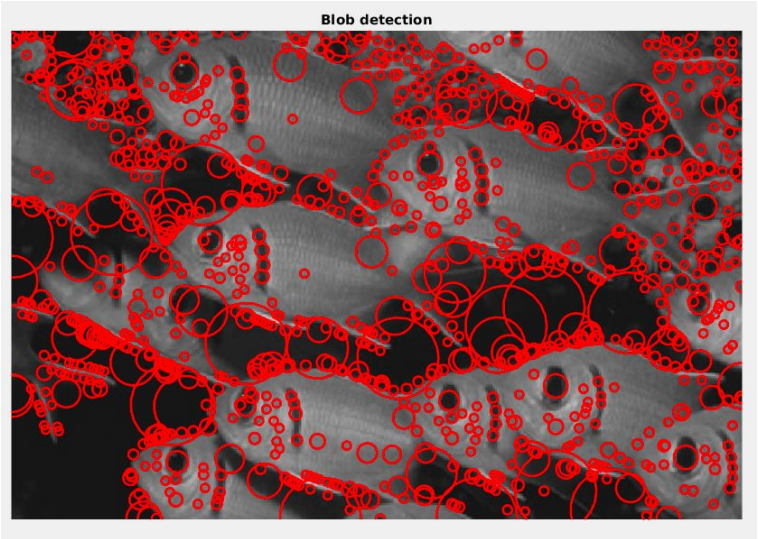


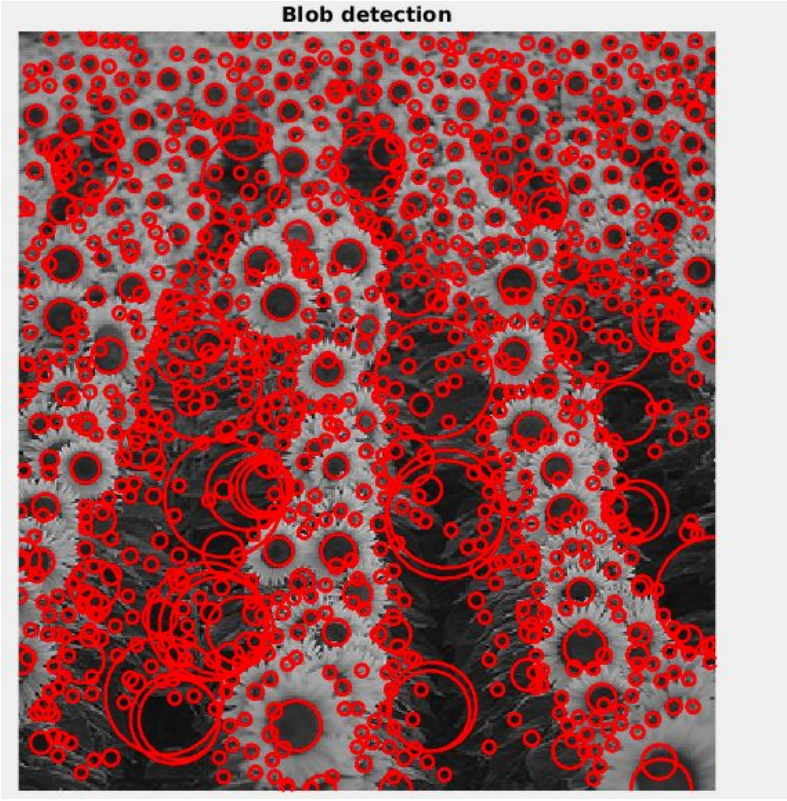
EXTRA CREDIT 2:

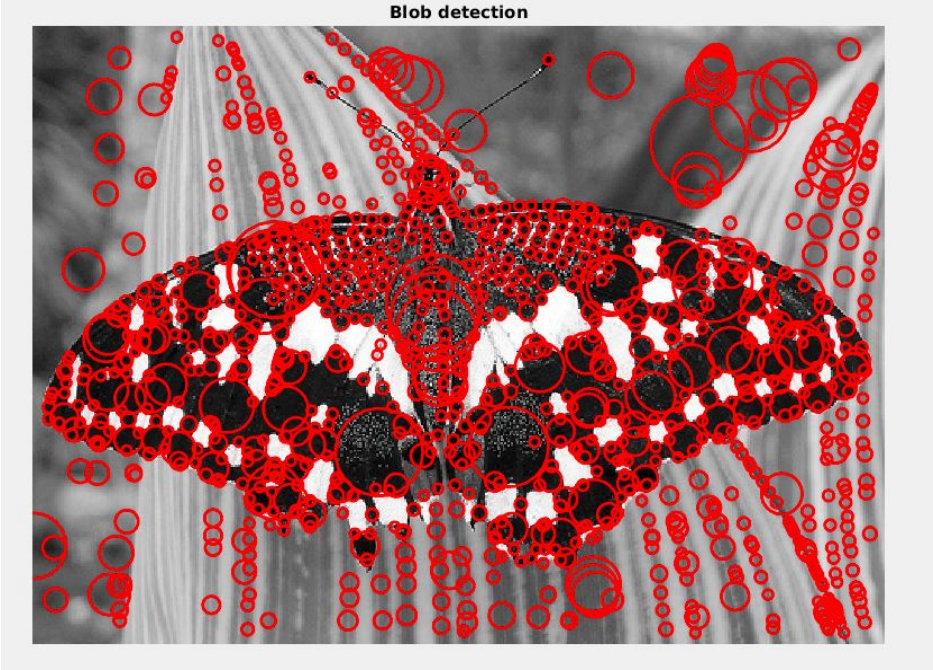
Using difference-of-Gaussian pyramid for blob detection:

```
g1 = fspecial('gaussian', filt_size, 1.1*i);
    g2 = fspecial('gaussian', filt_size, i);
    dog = g1 - g2;
    f = conv2(double(im),dog,'same');
```

```
    scalespace(:,:,k) = f;
```







COMPARING PERFORMANCES:

For Laplacian of Gaussian method:

Elapsed time is 0.4252344 seconds.

For Difference of Gaussian method:

Elapsed time is 0.384345 seconds.

CODE:
DETECT BLOBS:

```
function blobs = detectBlobs(im, param)
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji

% Input:
%   IM - input image
%
% Ouput:
%   BLOBS - n x 4 array with blob in each row in (x, y, radius, score)
%
% Dummy - returns a blob at the center of the image



im=rgb2gray(im);
im=im2double(im);

[h,w]=size(im);

n=25;
scalespace = zeros(h,w,n);
i=2;
k=1;
inc=1.1;
scale=zeros(n);
%score=zeros(n);

while(k<=n)
filt_size=2*ceil(3*i)+1;
filter=i*i*fspecial('log',filt_size,i);
Lnorm=((imfilter(im,filter,'same','circular')).^2);
scalespace(:,:,k)=Lnorm;
scale(k)=i;
i=i*inc;
k=k+1;
end
ord=zeros(h,w,n);
```

```
for i = 1:n
   ord(:,:,i) = ordfilt2(scalespace(:,:,i), 9, ones(3,3));
end

for i = 1:n
       ord(:,:,i) = max(ord(:,:,max(i-1,1)),ord(:,:,i));
       ord(:,:,i) = max(ord(:,:,min(i+1,n)),ord(:,:,i));
end
ord = ord .* (ord == scalespace);

r=[];
c=[];
rad=[];
v=[];
%score=[];
threshold=0.007;
ord(ord<threshold)=0;
%disp(sum(sum(ord>0)));
for i=1:n
   [rows, cols,value] = find(ord(:,:,i));
   numBlobs = length(rows);
   radii =  scale(i) * sqrt(2);
   radii = repmat(radii, numBlobs, 1);
   r = [r; rows];
   c = [c; cols];
   rad = [rad; radii];
   v=[v;value];

   %score=scalespace(:,:,);
end
blobs=[c,r,rad,v];

disp(size(blobs,1));
```

COMPUTE MATCHES:

```
function m = computeMatches(f1,f2)



[n ,d]=size(f1);
[m ,d]=size(f2);
%disp(n);
%disp(m);
%disp(n);
%disp(m);
%disp(d);
arr1=[];
%arr2=zeros(m);

for i=1:n
  ssd=0;
  min1=intmax;
  min2=intmax;
  indice1=0;
    for j=1:m

        ssd=sum(sum((f1(i,:)-f2(j,:)).^2));

        if(ssd<min1)
            min1=ssd;
            indice1=j;
            if(min2==intmax)
                min2=min1;
            end


        elseif(ssd<min2)
            min2=ssd;

        end

    end

    if(min1/min2<0.8)
     arr1=[arr1;indice1];
```

```
   else
    arr1=[arr1;0];
   end
end

m=arr1;
```

RANSAC:

```
function [inliers, transf] = ransac(matches, c1, c2, method)
% This code is part of:
%
%   CMPSCI 670: Computer Vision
%   University of Massachusetts, Amherst
%   Instructor: Subhransu Maji
length(matches)

N = length(matches);
M=size(c2,1);
temp=[];
for i=1:M
temp=[temp;([c2(i,1) c2(i,2)])];
end

adjust=ones(M,1);
temp=[temp adjust];
maxinlier=0;
inliers=[];
for pk=1:35
rand_arr=[];
r=randi([1,N],1,1);
count=1;

check=[];
while(count<=3)%loop till we find 3 indeices

      if(matches(r,1)~=0)% check if a match exists

      rand_arr=[rand_arr;r];%store it
      count=count+1;
      end
```

```
        r=randi([1,N],1,1);
        if(length(rand_arr)~=0)
        for w=1:length(rand_arr)
           num=rand_arr(w);

        while(r==num)% check if 2 or more blobs map to same blob
           r=randi([1,N],1,1);%if yes, select another indice or else it will result in a system
                        % of linear equations which are not independent.
        end

     end
        end
end
A=[];
B=[];

for i=1:length(rand_arr)
index=rand_arr(i);

x1=c1(index,1);
y1=c1(index,2);

A=[A;([x1 y1])];%3X2

x_ans=c2(matches(index),1);
y_ans=c2(matches(index),2);

B=[B;([x_ans,y_ans,1])];%3X3
end


%Ax=b---->(3X3  3X2  =  3X2)
transform=B\A;%3X2
result=temp*transform;

inlier=0;

optimal=[];
for i=1:N

if(matches(i)~=0)
x1=(c1(i,1));
y1=(c1(i,2));
```

```
if(((x1-result(matches(i),1))^2+((y1)-result(matches(i),2))^2)<=20)
inlier=inlier+1;
optimal=[optimal;i];
end
end
end

%while(matches)

%disp(inlier);
if(inlier>maxinlier)
   inliers=optimal;
   maxinlier=inlier;
   transf=transform';
   disp(transf);
   %disp(inlier)

end
%transf=reshape(transform_inv,[2 3]);%
%transf=pinv(transf);
%transf(1,3)=transf(1,3)*-1;
%transf(2,3)=transf(2,3)*-1;

%inliers=
%transf=([m1 m2 t1;m3 m4 t2]);

end
```