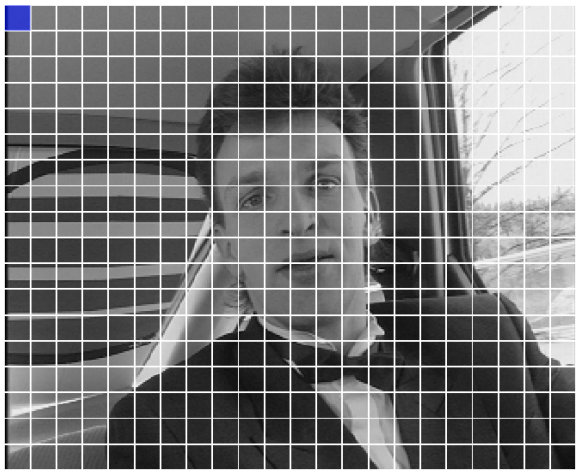
CTMC



Comparative of different block search methods in the block matching algorithm

Jordi Cenzano Ferret
2006-04-02

Index

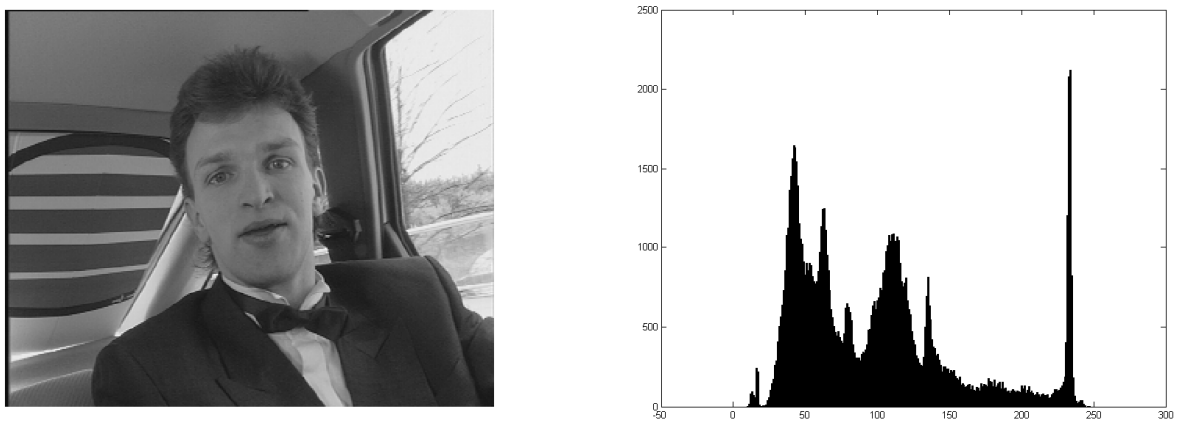
1 Introduction

1.1 Block matching (BM)

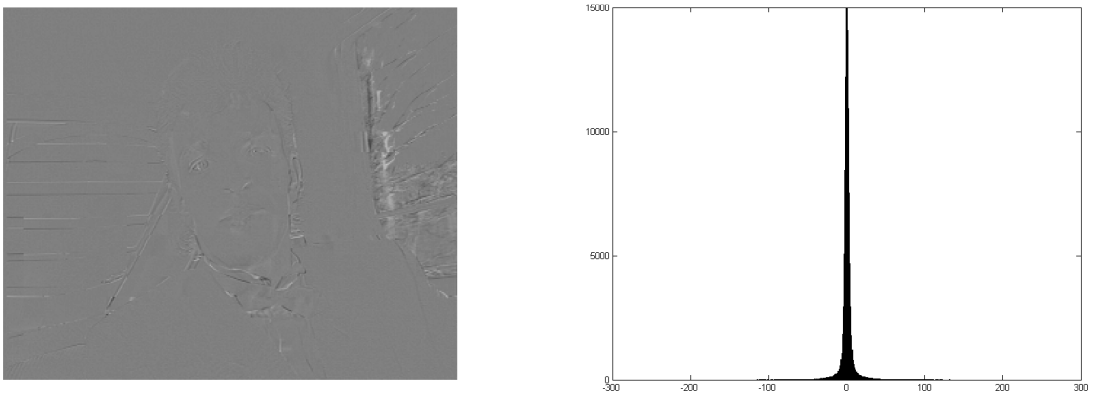
The block matching is a technique that tries to reduce the temporal redundancy in video sequences.

In natural videos, the information in frame N+1 is very similar to the information in the frame N (except some cases: shot changes, very fast pan, etc...). The BM tries to do the best representation of frame N+1 based on frame N.

(a)Mirror borders



(b) Original image and its histogram



(c) Error image and its histogram

Figure 1 *(Block matching algorithm)*

In the *Figure 1* we can see the basic steps of BM. First, divide the N+1 frame in X*Y blocks (usually 16x16 or 8x8), after that for each divided block it tries to find the best fit in the search area of the frame N, we used the minimum square error (MSE) *Figure 2* as best fit criteria.

In order to do that the BM puts each of N+1 frame in all positions in the search area of the frame N (exhaustive search) and it calculates the MSE in each position, and the displacement that has lower MSE will be the best fit for that block.

Finally the blocks of the N+1 image are represented with a two integers vector (motion vector) [Xsource – Xdest, Ysource –Ydest] and an error image block, that is the difference between the original block in the frame N+1 and the position of its best representation in frame N.

Usually we send the error image (a composition of error image blocks) with the motion vectors to describe the frame, this error image is the difference between the original frame N+1 and the reconstructed frame (see the histograms in *Figure 1*b, *Figure 1*c). The error image is easy to compress, because it has a very thin histogram.

1.2 Block matching search methods

We used as distortion or best fit criteria the minimum square error (MSE).

$$MSE=\frac{\sum_{R}(I(i,j)-I'(i,j))^2}{N^2}$$

Figure 2 *(MSE formula)*

1.2.1 Exhaustive

Is the obvious one, it calculates distortion of the source block in all positions of the search area in the frame N.

Is the best method to find the minimum error, it calculates all possible positions, but on the other hand is the method that requires more computational resources (time), the logical way is try to find a trade-off.

This method uses a fixed number of block comparisons: SearchAreaWidth * SearchAreaHeight per each block. If the SearchAreaWith = SearchAreaHeight = 16, the number of block comparisons per block search is 256.

1.2.2 3 Steps

This method uses 3 stages to find the best fit (*Figure 3*):

1. Test 8 points around the center (distance d = +-4) and the center (cyan), the point with less distortion becomes the center of the stage 2.
2. The distance is reduced (d = 2) and the 8 point around the new center are checked (magenta), the center point is already checked in the previous stage.
3. The distance is reduced another time (d = 1) and the 8 point around the new center are checked (green) the center point is already checked in the previous stage. Now the point with minimum distortion will be the best fit.

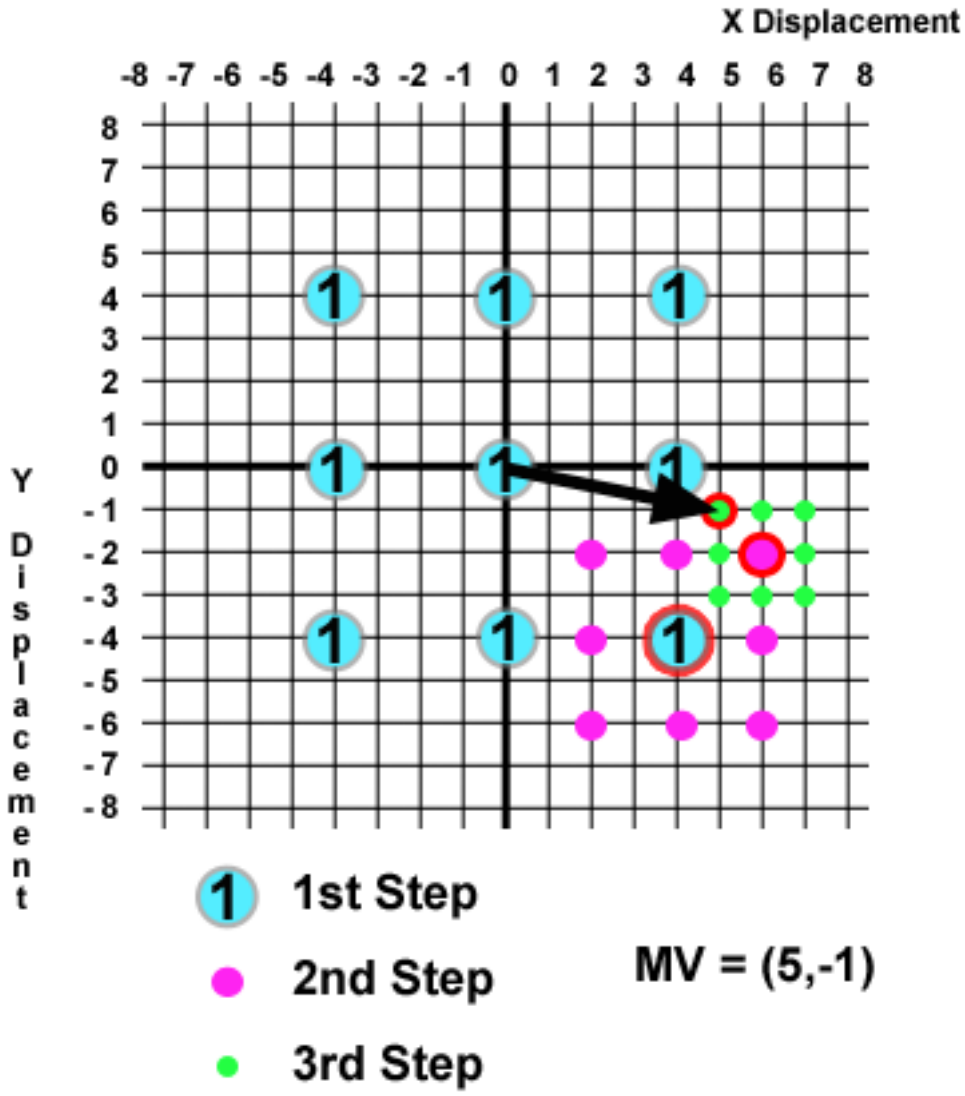


Figure 3 *(3 steps search method)*

This method uses a fixed number of block comparisons like the exhaustive method, but this method does 9 + 8 + 8 = 25 block comparisons in each block search.

Note: In this method, if we use a first distance d = 4 we fix the search area to: Xsrc – 4... Xsrc + 4 and Ysrc – 4... Ysrc + 4

1.2.3 Conjugate directions

1. To start this method calc the distortion of the central point and the adjacent points in the horizontal direction (cyan). If the minimum distortion point is the central point, the method proceeds to the next stage. If the minimum distortion is in the left or in the right point the method moves the center to the minimum and calc the three points again (note that only 1 point has to be calculated, the other 2 have been calculated in the previous step). If the right or left points are in the bounds of the search area the method goes to stage 2.
2. Do the same as in the stage 1 but in vertical direction, (magenta) and the start central point is the last minimum distortion point found in the stage 1
3. Do the same as in the stage 1 but in diagonal direction (green), and the start central point is the last minimum distortion point found in the stage 2

The number of block comparisons is variable depending on the distance of the best fit.

Best case: When the motion vector is (0,0), the number of block comparisons in that case is: 3 + 2 + 2 = 7

Worst case: When the motion vector is (SaerchAreaWidth/2, SaerchAreaHeight/2), the number of block comparisons in that case is: SaerchAreaWidth/2 + SaerchAreaHeight/2 + 1. If the SearchAreaHeight =SearchAreaHeight = 16. The number of block comparisons will be 17.

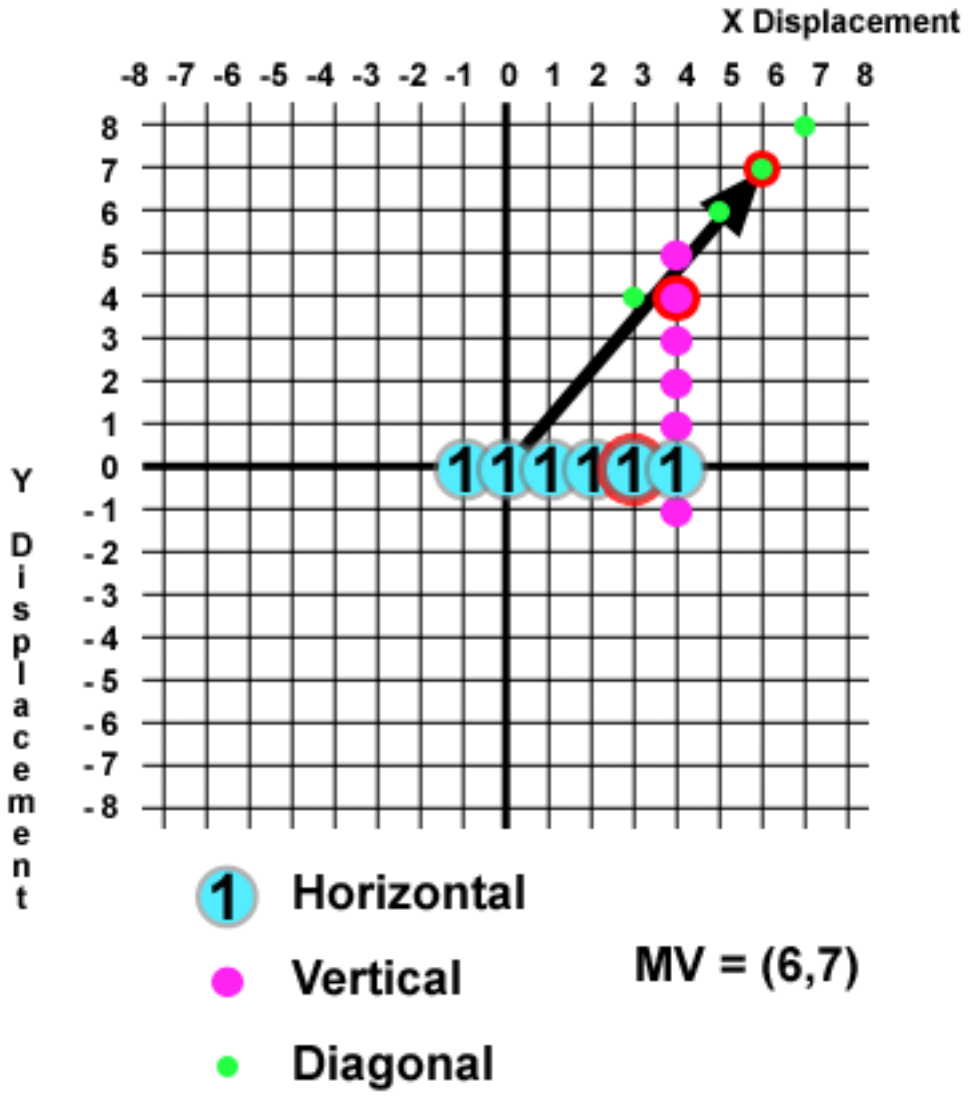


Figure 4 *(conjugate directions method)*

1.2.4 2D Log

1. This method uses a cross shape (+), first compute the minimum distortion in the central point and in the four points at distance d (d $= 2^{\log 2(searchRadius)-1}$), up, down, left, and right (cyan).
2. If the minimum distortion point is one of the four outer points of the cross, it moves the center of the cross to that point and iterates over this stage (magenta). If the minimum distortion point is the center or one of the outer points is in the bounds of the search area reduces the distance d (d $= 2^{\log 2(searchRadius)-2}$).Iterates this stage until d = 1 (green).
3. Calc the 8 points around the center, and the center (yellow) and the point with less distortion will be the best fit.

The number of block comparisons is variable depending on the distance of the best fit.

Best case: When the motion vector is (0,0), the number of block comparisons in that case is: 5 + 4 + 8 = 17.

Worst case: It depends on the path that the method follows, if the SearchAreaWidth = SearchAreaHeight = 16, the number of block comparisons are : 5 + 4 + 4 + 4 + 4 + 4 + 8 = 33.

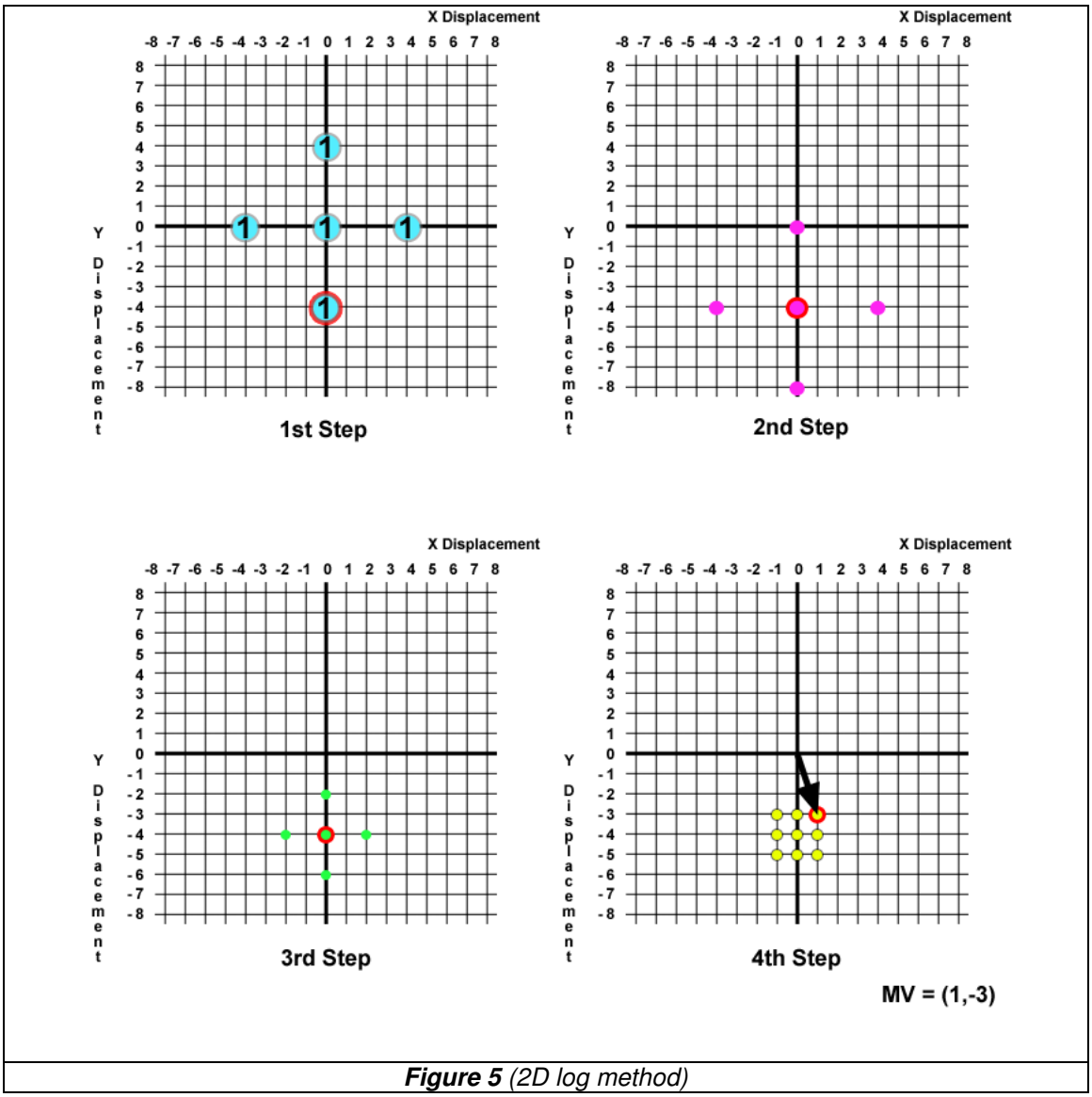


Figure 5 *(2D log method)*

2 Goals

The goal of this work is to compare the performance of the exhaustive search, 3 steps, conjugate directions, and 2Dlog methods in the BM algorithm, in order to do that we have to find some metrics. I have chosen:

1. Number of block comparisons
2. Time to compute block matching (*)
3. PSNR of the reconstructed image (obviously without compute the error image)

(*) We know that the number of comparisons is directly related with the time to compute. I only put here the time to compute to show the time that block matching spend in a normal PC computer (using matlab environment).

3 Metrics

3.1 Number of block comparisons

I think that this is the best measure to know the performance of different methods (in conjunction with PSNR). It is computer independent, and gives us the number of operations (product and add are considered only one operation, and I assumed that block is a squares) that the algorithm does (*Figure 6*).

$$OPperComparison = BlockSize^2$$

Figure 6 *(Number of operations per block comparison)*

The method that uses more block comparisons is the exhaustive method, because it compares all possible block positions in the search area (*Figure 7*). We have to see how the other methods improve the number of block comparisons.

$$OPperBlock = OPperComparision \cdot SearchArea^2$$

$$OPperFrame = OPperBlock \cdot \frac{\mathrm{Im}ageWidth}{BlockSize} \cdot \frac{\mathrm{Im}ageHeight}{BlockSize}$$

$$OPperFrame = SaerchArea^{2} \cdot \mathrm{Im}ageWidth \cdot \mathrm{Im}ageHeight$$

Figure 7 *(Block comparisons in all frame of the exhaustive method)*

3.2 Time to compute block matching

Is the time that matlab has used to in a particular PC computer (*) to compute block matching, we store the time used in each frame.

This measure is related to the computer that we have used to calc the block matching.

To know the time that matlab uses to compute the block matching we used the algorithm in the *Figure 8*.

(*) Laptop SONY Vaio VS4 (512MB, Intel centrino 1.6GHz)

```
tic;
[V numcomparesframe] = BlockMatching (y2, y1, blockSize, SearchArea,
Method);
timeUsed = toc;
```

Figure 8 *(Matlab get block matching time algorithm)*

3.3 PSNR

It is a measure that shows how the reconstructed frame looks like the original one, it's an objective measure, and I know that in image processing the best measure over quality is the human eye, but I have to get some metric.

$$MSE = \frac{\sum_{R} (I(i,j) - I'(i,j))^2}{N^2}$$

$$PSNR = 10 Log_{10} \left(\frac{MAXi^2}{MSE} \right)$$

Figure 9 *(PSNR formula)*

4 Tools

To implement the BM and its different methods I have used matlab. I have programmed the next functions:

4.1 Matlab functions

4.1.1 loadFileYuv

This function loads a video frame of a YUV raw video file, supports 4:2:0 and 4:4:4 chroma sample types.

```
% function [Y,U,V,status] = loadFileYuv(fileName, width, height ,
ChromaSample ,NumFrame)

function [Y,U,V,status] = loadFileYuv(fileName, width, height , ChromaSample ,NumFrame)
% loads YUV frames from YUV video file

status = 0;

fileId = fopen(fileName, 'r');

if (fileId >= 0)

    nChromaSample = 1; %4:4:4

    if (ChromaSample  == '4:2:0')
        nChromaSample = 0.5; %4:2:0
    end;

    whidthChroma  = width * nChromaSample;
    heightChroma  = height * nChromaSample;

    FrameSize = width * height +  2 * (whidthChroma * heightChroma);

    status = fseek(fileId, FrameSize * (NumFrame - 1) + 1, 'bof');
    if ((status >= 0)&(feof(fileId) == 0))
        status = fseek(fileId, FrameSize * (NumFrame - 1), 'bof');

        buf = fread(fileId, width * height, 'uchar'); %Y
        Y(:,:) = reshape(buf, width, height).'; % reshape

        buf = fread(fileId, whidthChroma * heightChroma, 'uchar'); %U
        U(:,:) = reshape(buf, whidthChroma, heightChroma).'; % reshape

        buf = fread(fileId, whidthChroma * heightChroma, 'uchar');%V
        V(:,:) = reshape(buf, whidthChroma, heightChroma).'; % reshape

        status = 1;
    else
        Y(:,:) = zeros (height,width,1);
        U(:,:) = zeros (heightChroma,whidthChroma,1);
        V(:,:) = zeros (heightChroma,whidthChroma,1);
    end;

    fclose(fileId);
end;
```

Figure 10 *(loadFileYuv matlab function)*

4.1.2 Psnr

This function calculates the PSNR between two frames.

```
% function [psnrDB] = psnr (frame1,frame2)

function [psnrDB] = psnr (frame1,frame2)
% calc the psnr between frames

error(nargchk(2, 2, nargin))

[FR1height,FR1widht]= size (frame1);
[FR2height,FR2widht]= size (frame2);

if ((FR1height ~= FR2height)|(FR1widht ~= FR2widht))
    error('The frame dimensions are diferent');
end;

mseTmp = 0;
for y = 1:FR1height
    for x = 1:FR1widht
        mseTmp = mseTmp + (frame1 (y,x) - frame2 (y,x))^2;
    end;
end;

mse = mseTmp / (FR1height * FR1widht);

psnrDB = 10 * log10 ((255^2)/mse);
```

Figure 11 *(psnr matlab function)*

4.1.3 BMMirrorFrame

This function creates a new frame with mirrored borders.

```
% function [FrameMirrowed] = BMMirrorFrame (Frame,extraWhidth,extraHeight)

function [FrameMirrowed] = BMMirrorFrame (Frame,extraWhidth,extraHeight)
% Create a frame with mirrowed borders

[FRheight,FRwidht]= size (Frame);

%Mirror borders
FrameMirrowed = zeros(FRheight + 2 * extraHeight,FRwidht + 2 * extraWhidth);
%Copy frame
FrameMirrowed((extraHeight+1):(FRheight+extraHeight),(extraWhidth+1):(FRwidht+extraWhidth))
= Frame(:,:);
%up
FrameMirrowed(1:extraHeight,(extraWhidth+1):(FRwidht+extraWhidth)) =
flipud(Frame(1:extraHeight,1:FRwidht));
%down
FrameMirrowed((FRheight+extraHeight+1):(FRheight + 2 *
extraHeight),(extraWhidth+1):(FRwidht+extraWhidth)) = flipud(Frame((FRheight-
extraHeight+1):FRheight,1:FRwidht));
%Left
FrameMirrowed((extraHeight+1):(extraHeight + FRheight),1:extraWhidth) =
fliplr(Frame(1:FRheight,1:extraWhidth));
%Right
FrameMirrowed((extraHeight+1):(extraHeight + FRheight),(FRwidht + extraWhidth + 1):(FRwidht
+ 2 * extraWhidth)) = fliplr(Frame(1:FRheight,(FRwidht - extraWhidth + 1):FRwidht));
%Up-Left
FrameMirrowed(1:extraHeight,1:extraWhidth) =
flipud(fliplr(Frame(1:extraHeight,1:extraWhidth)));
%Up-right
FrameMirrowed(1:extraHeight,(FRwidht + extraWhidth + 1):(FRwidht + 2 * extraWhidth)) =
flipud(fliplr(Frame(1:extraHeight,(FRwidht - extraWhidth + 1):FRwidht)));
%Down-left
FrameMirrowed((FRheight+extraHeight+1):(FRheight + 2 * extraHeight),1:extraWhidth) =
flipud(fliplr(Frame((FRheight-extraHeight+1):FRheight,1:extraWhidth)));
%Down-right
FrameMirrowed((FRheight+extraHeight+1):(FRheight + 2 * extraHeight),(FRwidht + extraWhidth
```

```
+ 1):(FRwidht + 2 * extraWhidth)) = flipud(fliplr(Frame((FRheight-
extraHeight+1):FRheight,(FRwidht - extraWhidth + 1):FRwidht)));
```

Figure 12 *(BMMirrorFrame function)*

4.1.4 BMReconstruct

This function reconstructs the motion estimated frame from its motion vectors matrix. I used to check the block matching algorithms.

```
% function [FrameDst] = BMReconstruct (FrameSrc,V,searcharea)

function [FrameDst] = BMReconstruct (FrameSrc,V,SearchArea)
% Recontruct a frame using frame source and BM vectors

error(nargchk(2, 3, nargin))

s = sprintf ('RECONSTRUCTING IMAGE!');
disp (s);

[FRheight,FRwidht]= size (FrameSrc);
[BlocksH,BlocksW,xy]= size (V);
BLheight = FRheight / BlocksH;
BLwidth = FRwidht / BlocksW;

FrameDst = zeros (FRheight,FRwidht);

if (nargin < 3)
    SearchArea = BlocksW * 4;
end;

FrameSrcMirror = BMMirrorFrame (FrameSrc,SearchArea,SearchArea);

bln  = 1;
for y = 1:BlocksH
    for x = 1:BlocksW
        xstart = ((x -1) * BLwidth) + 1;
        ystart = ((y -1) * BLheight) + 1;
        xend = ((x -1) * BLwidth) + BLwidth;
        yend = ((y -1) * BLheight) + BLheight;

        xsm = xstart + V(y,x,1) + SearchArea;
        xem = xsm + BLwidth - 1;
        ysm = ystart + V(y,x,2) + SearchArea;
        yem = ysm + BLheight - 1;

        FrameDst (ystart:yend,xstart:xend) = FrameSrcMirror(ysm:yem,xsm:xem);

        %s = sprintf ('Assing block number: %d',bln);
        %disp (s);

        bln = bln + 1;
    end;
end;
```

Figure 13 *(BMReconstruct function)*

4.1.5 BMSearchBlockPos

This is the largest function. This function searches for a given block, inside the given search area the minimum distortion position (using MSE), it can use 4 search methods: Exhaustive, 3step, conjugate directions and 2d log.

```
% function [pos numcompares] = BMSearchBlockPos
(FrameMirror,Block,xOrigin,yOrigin,SearchArea,Method)

function [pos numcompares] = BMSearchBlockPos
(FrameMirror,Block,xOrigin,yOrigin,SearchArea,Method)
% Search the better block position in frame

error(nargchk(5, 6, nargin))

if (nargin < 6)
    Method = 'exhaustive';
end;

[BLheight,BLwidth]= size (Block);
[FRMirrorHeight,FRMirrorWidth]= size (FrameMirror);

minCoef = BLheight * BLwidth * 255^2;

numcompares = 0;

searchRadius = SearchArea / 2;

if (strcmp(Method,'exhaustive') == 1)

    %Define the searcharea
    yMin = (yOrigin + searchRadius) - searchRadius;
    yMax = (yOrigin + searchRadius) + searchRadius -1;
    xMin = (xOrigin + searchRadius) - searchRadius;
    xMax = (xOrigin + searchRadius) + searchRadius -1;

    for y = yMin:yMax
        for x = xMin:xMax
            yh = y + BLheight - 1;
            xw = x + BLwidth - 1;

            srcBlock = FrameMirror(y:yh,x:xw);
            difBlock = (srcBlock - Block);
            coef = sum(sum (difBlock.^2));

            numcompares = numcompares + 1;
            if (coef < minCoef)
                pos = [x - searchRadius , y - searchRadius];
                minCoef = coef;
            end;
        end;
    end;
end;

if (strcmp(Method,'3step') == 1)

    addFirst = 4;

    %Check borders
    if (searchRadius < (addFirst + addFirst / 2 + addFirst/ 4))
        error ('In the 3step method the search raduis must be greater or equal than 7
(4 + 2 + 1)');
    end;

    posStep = zeros (3,3,2);

    for n= 1:3

        %Step N *************************

        nAdd = addFirst;
        if (n == 1)
```

```
            pos(1) = xOrigin ;
            pos(2) = yOrigin ;
        end;
        if (n == 2)
            nAdd = addFirst / 2;
        end;
        if (n == 3)
            nAdd = 1;
        end;

        posStep(1,1,1) = pos(1) + searchRadius - nAdd; posStep(1,1,2) = pos(2) +
searchRadius - nAdd;
        posStep(1,2,1) = pos(1) + searchRadius ; posStep(1,2,2) = pos(2) + searchRadius
- nAdd;
        posStep(1,3,1) = pos(1) + searchRadius + nAdd; posStep(1,3,2) = pos(2) +
searchRadius - nAdd;
        posStep(2,1,1) = pos(1)  + searchRadius - nAdd ; posStep(2,1,2) = pos(2)+
searchRadius;
        posStep(2,2,1) = pos(1) + searchRadius; posStep(2,2,2) = pos(2) + searchRadius;
        posStep(2,3,1) = pos(1) + searchRadius + nAdd; posStep(2,3,2) = pos(2) +
searchRadius;
        posStep(3,1,1) = pos(1) + searchRadius - nAdd; posStep(3,1,2) = pos(2) +
searchRadius + nAdd;
        posStep(3,2,1) = pos(1) + searchRadius ; posStep(3,2,2) = pos(2) + searchRadius
+ nAdd;
        posStep(3,3,1) = pos(1) + searchRadius + nAdd; posStep(3,3,2) = pos(2) +
searchRadius + nAdd;

        for iy = 1:3
            for ix = 1:3
                if ((iy == 2) & (ix == 2) & (n > 1))
                    coef = minCoef;
                else
                    x = posStep(iy,ix,1);
                    y = posStep(iy,ix,2);
                    yh = y + BLheight - 1;
                    xw = x + BLwidth - 1;

                    srcBlock = FrameMirror(y:yh,x:xw);
                    difBlock = (srcBlock - Block);
                    coef = sum(sum (difBlock.^2));

                     numcompares = numcompares + 1;
                end;

                if (coef < minCoef)
                    pos = [x - searchRadius,y - searchRadius];
                    minCoef = coef;
                end;
            end;
        end;
    end;
end;

if (strcmp(Method,'conjugatedir') == 1)
    %ini pos
    posStep = zeros (3,3); %Per row [x,y,coef]
    add = 1;

    %Define the searcharea
    yMin = (yOrigin + searchRadius) - searchRadius;
    yMax = (yOrigin + searchRadius) + searchRadius -1;
    xMin = (xOrigin + searchRadius) - searchRadius;
    xMax = (xOrigin + searchRadius) + searchRadius -1;

    posStep (1,1) = xOrigin + searchRadius - add; posStep (1,2) = yOrigin +
searchRadius;posStep (1,3) = -1;
    posStep (2,1) = xOrigin + searchRadius; posStep (2,2) = yOrigin +
searchRadius;posStep (2,3) = -1;
    posStep (3,1) = xOrigin + searchRadius + add; posStep (3,2) = yOrigin +
searchRadius;posStep (3,3) = -1;

    for n=1:3

        bmidexit = 0;

        if (n == 2)
```

```
            posStep (1,1) = posStep (2,1); posStep (1,2) = posStep (2,2) - add; posStep
(1,3) = -1;
            %posStep (2,...) Remains the same
            posStep (3,1) = posStep (2,1); posStep (3,2) = posStep (2,2) + add; posStep
(3,3) = -1;
        end;
        if (n == 3)
            posStep (1,1) = posStep (2,1) + add; posStep (1,2) = posStep (2,2) - add;
posStep (1,3) = -1;
            %posStep (2,...) Remains the same
            posStep (3,1) = posStep (2,1) - add; posStep (3,2) = posStep (2,2) + add;
posStep (3,3) = -1;
            if (posStep (1,1) > xMax)|(posStep (1,2) < yMin)|(posStep (3,1) <
xMin)|(posStep (3,2) > yMax)
                bmidexit = 1;
            end;
        end;

        while (bmidexit == 0)
            for s = 1:3
                if (posStep(s,3) < 0)
                    x = posStep(s,1);
                    y = posStep(s,2);
                    yh = y + BLheight - 1;
                    xw = x + BLwidth - 1;

                    srcBlock = FrameMirror(y:yh,x:xw);
                    difBlock = (srcBlock - Block);
                    posStep(s,3) = sum(sum (difBlock.^2));

                    numcompares = numcompares + 1;
                end;
            end;

            [m,i] = min(posStep(:,3));
            if (i == 2)
                bmidexit = 1;
            else
                if (n == 1) %HOR
                    if (i == 1) %left
                        %Check border left
                        if ((posStep (1,1) - add)< xMin)
                            bmidexit = 1;
                        else
                            posStep (3,1) = posStep (2,1);posStep (3,3) = posStep
(2,3); %y remains the same
                            posStep (2,1) = posStep (1,1);posStep (2,3) = posStep
(1,3) ; %y remains the same
                            posStep (1,1) = posStep (1,1) - add;posStep (1,3) = -1; %y
remains the same
                        end;
                    end;
                    if (i == 3) %right
                        %Check border right
                        if ((posStep (3,1) + add + BLwidth) > xMax)
                            bmidexit = 1;
                        else
                            posStep (1,1) = posStep (2,1);posStep (1,3) = posStep
(2,3); %y remains the same
                            posStep (2,1) = posStep (3,1);posStep (2,3) = posStep
(3,3); %y remains the same
                            posStep (3,1) = posStep (3,1) + add;posStep (3,3) = -1; %y
remains the same
                        end;
                    end;
                end;
                if (n == 2) %VERT
                    if (i == 1) %up
                        %Check border up
                        if ((posStep (1,2) - add) < yMin)
                            bmidexit = 1;
                        else
                            posStep (3,2) = posStep (2,2);posStep (3,3) = posStep
(2,3); %x remains the same
                            posStep (2,2) = posStep (1,2);posStep (2,3) = posStep
(1,3) ; %x remains the same
                            posStep (1,2) = posStep (1,2) - add;posStep (1,3) = -1; %x
```

```
remains the same
                        end;
                    end;
                    if (i == 3) %down
                        %Check border down
                        if ((posStep (3,2) + add + BLheight) > yMax)
                            bmidexit = 1;
                        else
                            posStep (1,2) = posStep (2,2);posStep (1,3) = posStep
(2,3); %x remains the same
                            posStep (2,2) = posStep (3,2);posStep (2,3) = posStep
(3,3); %x remains the same
                            posStep (3,2) = posStep (3,2) + add;posStep (3,3) = -1; %x
remains the same
                        end;
                    end;
                end;
                 if (n == 3) %DIAG
                    if (i == 1) %up-right
                        %Check border x up - right
                        if (((posStep (1,1) + add + BLwidth) > xMax)|((posStep (1,2) -
add)< yMin))
                            bmidexit = 1;
                        else
                            posStep (3,1) = posStep (2,1);posStep (3,2) = posStep
(2,2);posStep (3,3) = posStep (2,3);
                            posStep (2,1) = posStep (1,1);posStep (2,2) = posStep
(1,2);posStep (2,3) = posStep (1,3);
                            posStep (1,1) = posStep (1,1) + add;posStep (1,2) = posStep
(1,2) - add; posStep (1,3) = -1;
                        end;
                    end;
                    if (i == 3) %down-left
                        %Check border x down - left
                        if (((posStep (1,1) -add)< xMin)|((posStep (1,2) + add +
BLheight)> yMax))
                            bmidexit = 1;
                        else
                            posStep (1,1) = posStep (2,1);posStep (1,2) = posStep
(2,2);posStep (1,3) = posStep (2,3);
                            posStep (2,1) = posStep (3,1);posStep (2,2) = posStep
(3,2);posStep (2,3) = posStep (3,3);
                            posStep (1,1) = posStep (1,1) - add;posStep (1,2) = posStep
(1,2) + add; posStep (1,3) = -1;
                        end;
                    end;
                end;
            end;
        end;
    end;

    pos = [posStep(2,1) - searchRadius, posStep(2,2) - searchRadius];
    minCoef = posStep(2,3);

end;


if (strcmp(Method,'2dlog') == 1)
    posStep = zeros (5,3); %Per row [x,y,coef], 1- left, 2-up, 3- center, 4- right, 5-
down
    add = searchRadius/2;

    bmidexit = 0;

    %Define the searcharea
    yMin = (yOrigin + searchRadius) - searchRadius;
    yMax = (yOrigin + searchRadius) + searchRadius -1;
    xMin = (xOrigin + searchRadius) - searchRadius;
    xMax = (xOrigin + searchRadius) + searchRadius -1;

    if (xOrigin == 305)
        xOrigin = xOrigin ;
    end;
    %ini pos
    posStep (1,1) = xOrigin + searchRadius - add; posStep (1,2) = yOrigin +
searchRadius ;posStep (1,3) = -1;
    posStep (2,1) = xOrigin + searchRadius ; posStep (2,2) = yOrigin + searchRadius -
```

```
add;posStep (2,3) = -1;
    posStep (3,1) = xOrigin + searchRadius ; posStep (3,2) = yOrigin +
searchRadius ;posStep (3,3) = -1;
    posStep (4,1) = xOrigin + searchRadius + add; posStep (4,2) = yOrigin +
searchRadius ;posStep (4,3) = -1;
    posStep (5,1) = xOrigin + searchRadius ; posStep (5,2) = yOrigin + searchRadius +
add;posStep (5,3) = -1;

    while (bmidexit == 0)
        for s = 1:5
            if (posStep(s,3) < 0)
                x = posStep(s,1);
                y = posStep(s,2);
                yh = y + BLheight - 1;
                xw = x + BLwidth - 1;

                srcBlock = FrameMirror(y:yh,x:xw);
                difBlock = (srcBlock - Block);
                posStep(s,3) = sum(sum (difBlock.^2));

                numcompares = numcompares + 1;
            end;
        end;

        [m,i] = min(posStep(:,3));
        if (i == 3) %Center
            if (add <= 1)
                bmidexit = 1;
            else
                add = floor(add / 2);
            end;
        else %out point
            posStep(3,1) = posStep(i,1); posStep(3,2) = posStep(i,2); posStep(3,3) =
posStep(i,3);
        end;

        if (bmidexit == 0)
            %Check borders
            while ((((posStep(3,1) - add) < xMin)|((posStep(3,2) - add) <
yMin)|((posStep(3,1) + add) > xMax)|((posStep(3,2) + add) > yMax))&(add > 0))
                add = floor(add / 2);
            end;
            if (add > 0)
                posStep (1,1) = posStep(3,1) - add; posStep (1,2) =
posStep(3,2);posStep (1,3) = -1;
                posStep (2,1) = posStep(3,1); posStep (2,2) = posStep(3,2) -
add;posStep (2,3) = -1;
                %posStep (3,1)
                posStep (4,1) = posStep(3,1) + add; posStep (4,2) =
posStep(3,2);posStep (4,3) = -1;
                posStep (5,1) = posStep(3,1); posStep (5,2) = posStep(3,2) +
add;posStep (5,3) = -1;
            else
                bmidexit = 1;
            end;
        end;
    end;

    %Check 9 positions around center point
    posStepFinal = zeros (9,3); %Per row [x,y,coef]
    add = 1;

    xf = posStep(3,1);
    yf = posStep(3,2);
    minCoef = posStep(3,3);

    %ini pos
    posStepFinal (1,1) = xf - add; posStepFinal (1,2) = yf - add; posStepFinal (1,3) =
-1; %up-left
    posStepFinal (2,1) = xf; posStepFinal (2,2) = yf - add; posStepFinal (2,3) = -
1; %up-center
    posStepFinal (3,1) = xf + add; posStepFinal (3,2) = yf - add; posStepFinal (3,3) =
-1; %up-right
    posStepFinal (4,1) = xf - add; posStepFinal (4,2) = yf; posStepFinal (4,3) = -
1; %center-left
    posStepFinal (5,1) = xf; posStepFinal (5,2) = yf; posStepFinal (5,3) =
minCoef; %center
```

```
    posStepFinal (6,1) = xf + add; posStepFinal (6,2) = yf; posStepFinal (6,3) = -
1; %center-right
    posStepFinal (7,1) = xf - add; posStepFinal (7,2) = yf + add; posStepFinal (7,3) =
-1; %down-left
    posStepFinal (8,1) = xf; posStepFinal (8,2) = yf + add; posStepFinal (8,3) = -
1; %down - center
    posStepFinal (9,1) = xf + add; posStepFinal (9,2) = yf + add; posStepFinal (9,3) =
-1; %down - right

     for s = 1:9
        if (posStepFinal(s,3) < 0)
            x = posStepFinal(s,1);
            y = posStepFinal(s,2);
            yh = y + BLheight - 1;
            xw = x + BLwidth - 1;

            if ((x > xMin)&(y > yMin)&(xw < xMax)&(yh < yMax))
                srcBlock = FrameMirror(y:yh,x:xw);
                difBlock = (srcBlock - Block);
                posStepFinal(s,3) = sum(sum (difBlock.^2));
            else
                posStepFinal(s,3) = BLheight * BLwidth * 255^2;
            end;
            numcompares = numcompares + 1;
        end;
    end;

    [m,i] = min(posStepFinal(:,3));
    pos = [posStepFinal(i,1) - searchRadius ,posStepFinal(i,2) - searchRadius];
    minCoef = posStepFinal(i,3);

end;
```

Figure 14 *(BMSearchBlockPos function)*

4.1.6 BlockMatching

This function implements the block matching helped by BMSearchBlockPos function.

```
% function [V numcompares] = BlockMatching
(FrameBlocks,FrameToSearch,BlockSize,SearchArea,Method)

function [V numcompares] = BlockMatching
(FrameBlocks,FrameToSearch,BlockSize,SearchArea,Method)
% Search the better block position in frame

[FRheight,FRwidht]= size (FrameBlocks);

BlocksW = FRwidht / BlockSize;
BlocksH = FRheight / BlockSize;

bln = 1;

numcompares = 0;

FrameToSearchMirror = BMMirrorFrame (FrameToSearch,SearchArea/2,SearchArea/2);

V = zeros (BlocksH,BlocksW,2);

for y = 1:BlocksH
    for x = 1:BlocksW
        xstart = ((x -1) * BlockSize) + 1;
        ystart = ((y -1) * BlockSize) + 1;
        xend = ((x -1) * BlockSize) + BlockSize;
        yend = ((y -1) * BlockSize) + BlockSize;

        Vsrc = [xstart , ystart];
        Block = FrameBlocks(ystart:yend,xstart:xend);

        [Vdst numcomparestmp]= BMSearchBlockPos (FrameToSearchMirror,Block, xstart ,
ystart, SearchArea,Method);

        numcompares = numcompares + numcomparestmp;
```

```
        VFin = Vdst - Vsrc;

        V(y,x,1) = VFin(1);
        V(y,x,2) = VFin(2);

        s = sprintf ('Block number: %d .Block position x: %d y: %d .Vector: (%d,%d)
Num. compares: %d',bln,xstart,ystart,VFin(1),VFin(2),numcomparestmp);
        disp (s);

        bln = bln + 1;
    end;
end;
```

Figure 15*(Block matching matlab function)*

4.1.7 BMTestVideo

This function performs the BM algorithm in all video file, it reads frame by frame, calculates its motion vectors, reconstructs the frame and stores the PSNR, the number on comparisons, and the time used to compute BM.

```
% function [times,psnrframeDB,numcompares] = BMTestVideo
(Directory,FileNameYuv,Width,Height,numFrames,ChromaSample,blockSize,SearchArea,Method)

function [times,psnrframeDB,numcompares] = BMTestVideo
(Directory,FileNameYuv,Width,Height,numFrames,ChromaSample,blockSize,SearchArea,Method)
% Block matching

error(nargchk(6, 9, nargin))

if (nargin < 7)
    blockSize = 16;
end;
if (nargin < 8)
    SearchArea = 65535; %All image
end;

if (nargin < 9)
    Method = 'exhaustive';
end;

cd (Directory);

st1 = 1;
st2 = 1;
nFrame = 1;

times = zeros (1,numFrames);
numcompares = zeros (1,numFrames);
psnrframeDB = zeros (1,numFrames);

while ((st1 > 0) & (st2 > 0))
    %Load Frames N,N+1
    [y1,u1,v1,st1] = loadfileyuv (FileNameYuv,Width,Height,ChromaSample,nFrame);
    [y2,u2,v2,st2] = loadfileyuv (FileNameYuv,Width,Height,ChromaSample,nFrame + 1);

    if ((st1 > 0) & (st2 > 0))
        %Search blocks of y2 over y1
        tic;
        [V numcomparesframe] = BlockMatching (y2,y1,blockSize,SearchArea,Method);
        times (nFrame + 1) = toc;
        numcompares (nFrame + 1) = numcomparesframe;

        %Check the BM
        frRC = BMReconstruct (y1,V);
        frDif = y2 - frRC;

        %Calc PSNR
        psnrframeDB(nFrame + 1) = psnr (y2,frRC);

        %Plot results
```

```
        %subplot (2,2,1);imshow(y2,[0 255]);title ('Original frame N+1');
        %subplot (2,2,2);imshow (frRC,[0 255]);title ('Reconstructed frame N+1');
        %subplot (2,2,3);imshow(frDif,[-255 255]);title ('Dif (Original -
Reconstructed)');
        %[difH,difW] = size(frDif);
        %frDifv = reshape (frDif,1,difH*difW);
        %x = -255:1:255;
        %subplot (2,2,4);hist(frDifv,x);title ('Histogram of the Dif image');

        %Show Final results
        s = sprintf ('Frame %d. Time: %d  Num. compares: %d',nFrame+1,times(nFrame +
1),numcompares);
        disp(s);

        nFrame = nFrame+1;
    end;
end;
```

Figure 16*(BMTestVideo matlab function)*

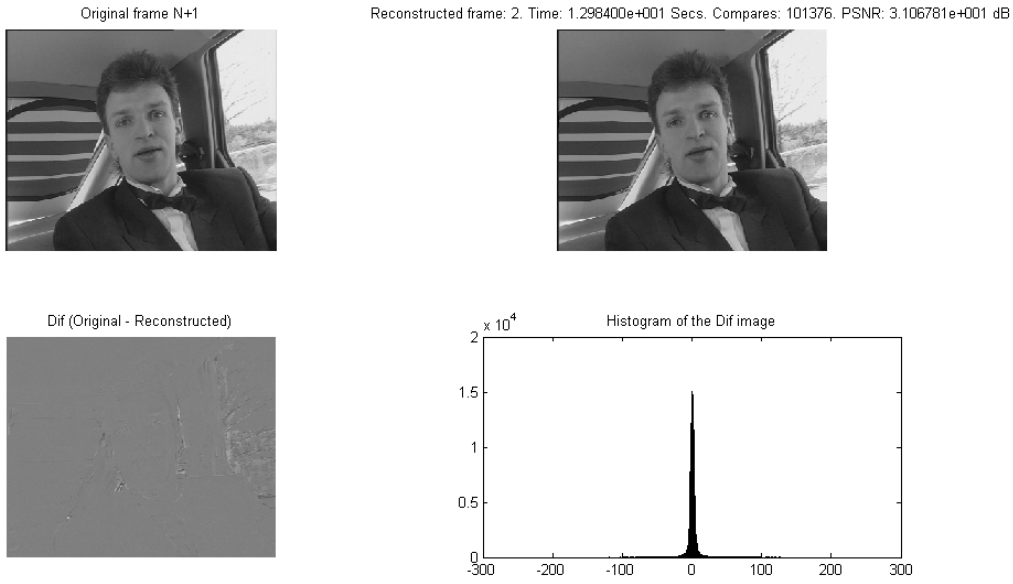
4.2 Checking tools

Before any performance test we have to check that our tools work well. In order to do that I have developed the BMTestFrame matlab function (*Figure 17*) that does the BM in only one frame. It can use any of all programmed methods, reconstructs the frame, and give us some data (psnr of reconstructed frame, error image, histogram of error image, time to compute block matching, etc...). We can see the results of the test in *Figure 18*.

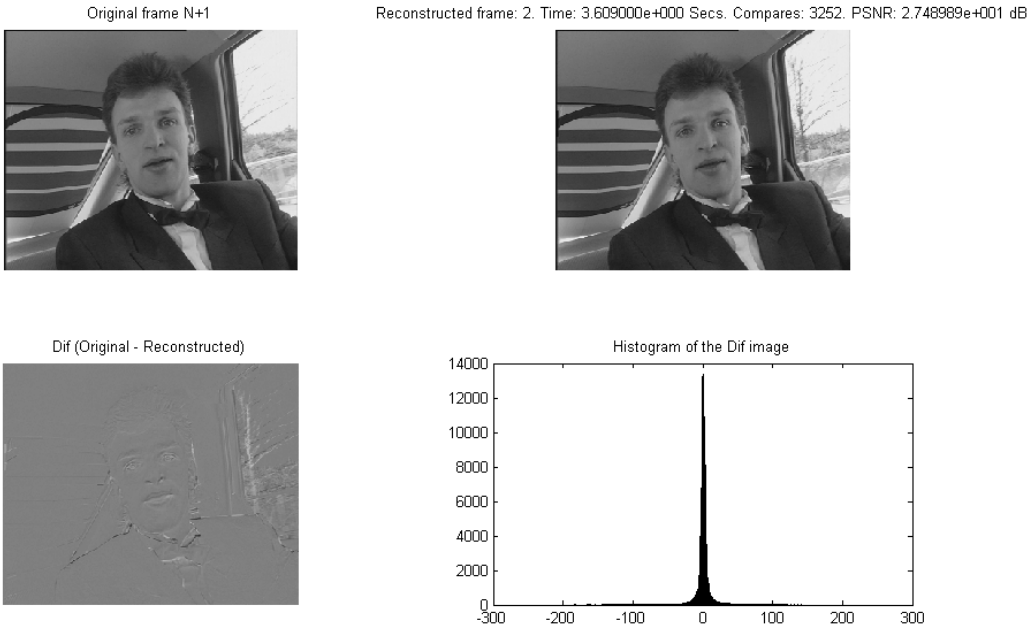
```
% function [] = BMTestFrame
(Directory,FileNameYuv,Width,Height,numFrame,ChromaSample,blockSize,SearchArea,Method)

function [] = BMTestFrame
(Directory,FileNameYuv,Width,Height,numFrame,ChromaSample,blockSize,SearchArea,Method)
% Block Matching test

error(nargchk(6, 9, nargin))

if (nargin < 9)
    Method = 'exhaustive';
end;

if (numFrame == 1)
    error ('The frame number must be greater than 1');
end;

if (nargin < 7)
    blockSize = 16;
end;
if (nargin < 8)
    SearchArea = 65535; %All image
end;

cd (Directory)
[y1,u1,v1,st1] = loadfileyuv (FileNameYuv,Width,Height,ChromaSample,numFrame - 1);
[y2,u2,v2,st2] = loadfileyuv (FileNameYuv,Width,Height,ChromaSample,numFrame);

if ((st1 <=0)|(st2 <= 0))
    error ('Loading frame');
end;

tic;
[V numcompares]= BlockMatching (y2,y1,blockSize,SearchArea,Method); %Search blocks of
y2 over y1
time = toc();

V

frRC = BMReconstruct (y1,V);

frDif = y2 - frRC;
```

```
psnrdb = psnr (y2,frRC);

subplot (2,2,1);imshow(y2,[0 255]);title ('Original frame N+1');
s = sprintf ('Reconstructed frame: %d. Time: %d Secs. Compares: %d. PSNR: %d
dB',numFrame,time,numcompares,psnrdb);
subplot (2,2,2);imshow (frRC,[0 255]);title (s);
subplot (2,2,3);imshow(frDif,[-255 255]);title ('Dif (Original - Reconstructed)');
[difH,difW] = size(frDif);
frDifv = reshape (frDif,1,difH*difW);
x = -255:1:255;
subplot (2,2,4);hist(frDifv,x);title ('Histogram of the Dif image');

s = sprintf ('Frame: %d. Time: %d Secs Compares: %d. PSNR: %d
dB',numFrame,time,numcompares,psnrdb);
disp(s);
```

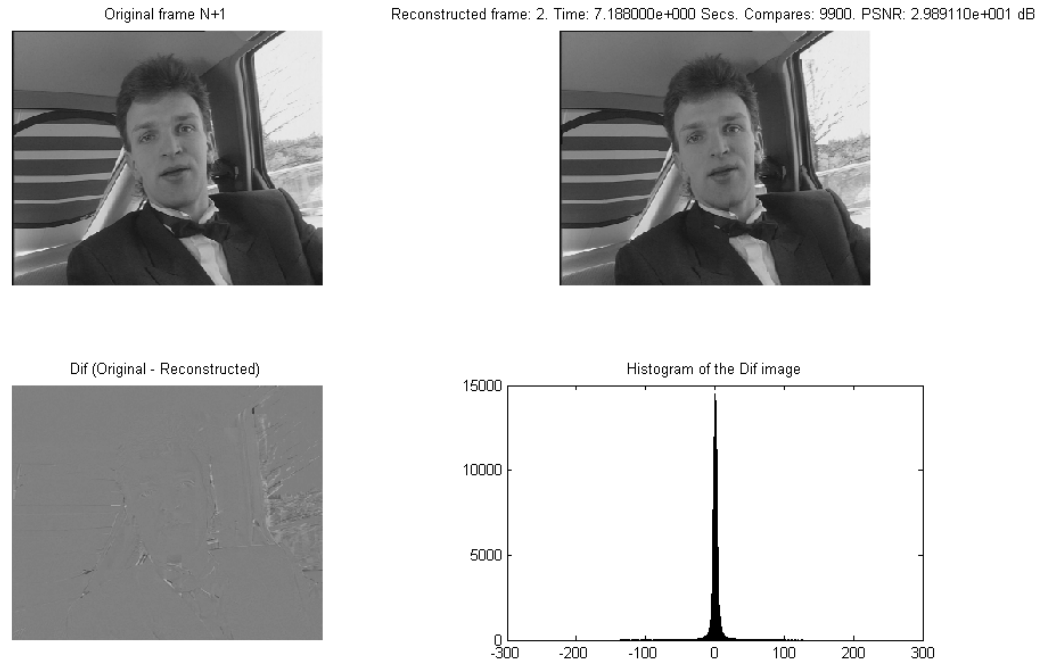
Figure 17 *(BMTestFrame matlab function)*



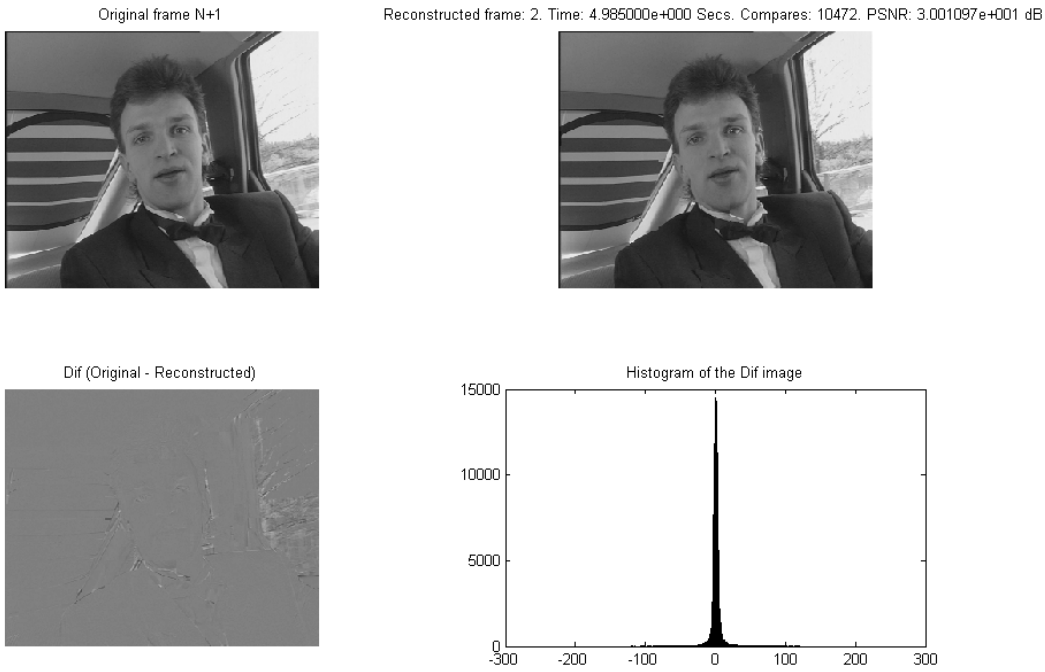
(a) Exhaustive method



(b) Conjugate directions



(c) 3 steps



(d) 2D Log

Figure 18 *(Test results)*

We obtained the results of *Figure 18* using a standard block size of 16x16 pixels and a search area of 16x16 pixels.

We can see that the four PSNR are very high, this is due to the images (frame N and frame N+1) are very similar.

Just using the test tool, we can already see an important improvement of speed of non exhaustive methods against the exhaustive one.

5 Results

To test the different BM methods I used 4 video sequences (*Table 1*):

Name	***Width***	***Height***	***Num. frames***	***Chroma sampling***
carphone	352	288	382	4:2:0
foreman	352	288	400	4:2:0
motherdaughter	352	288	300	4:2:0
table	176	144	300	4:2:0

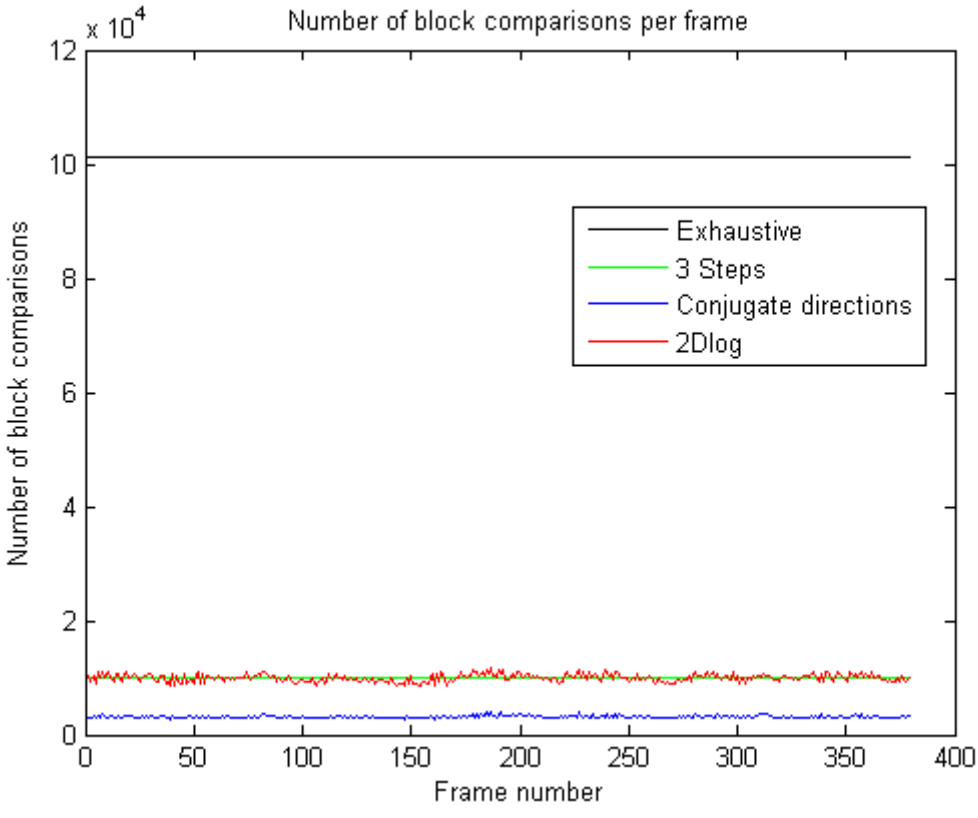
Table 1 *(Video sequences used)*

In all test I used a **block size of 16x16** and a **search area of 16x16**.

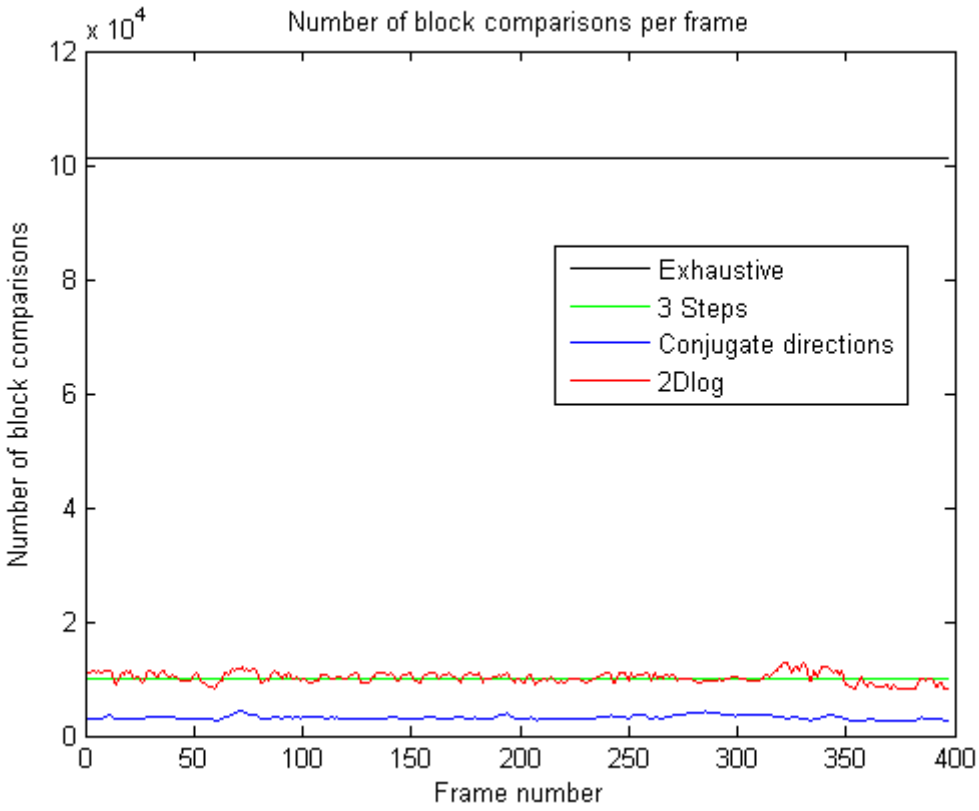
In the following tests I don't use the color information.

5.1 Number of block comparisons

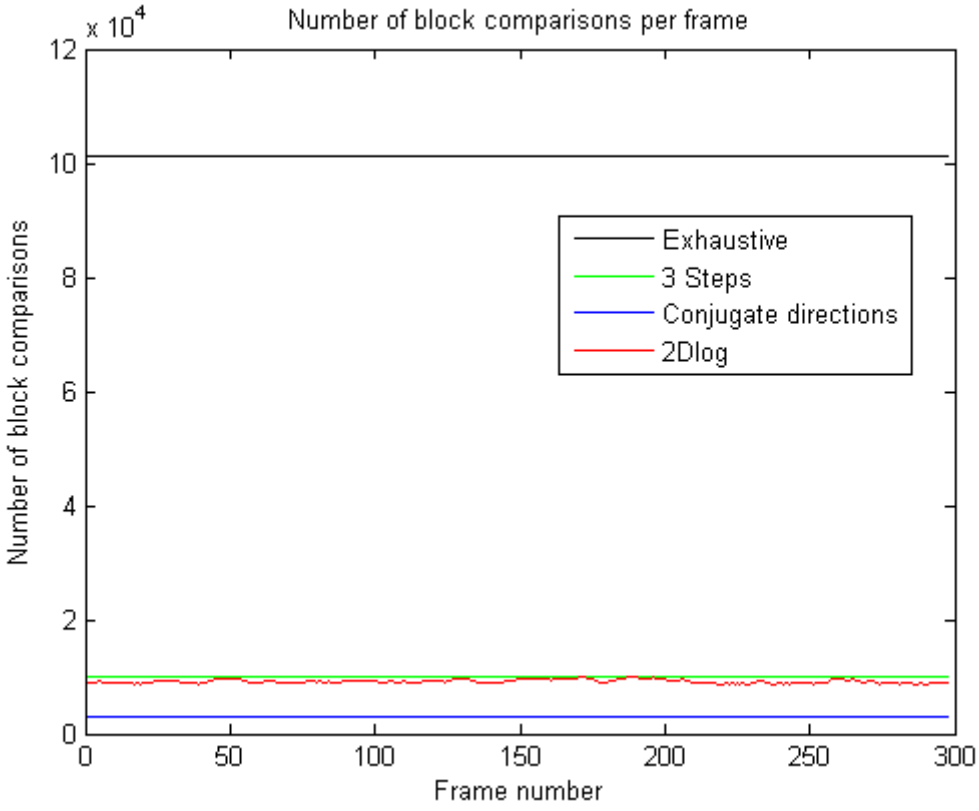
In then *Figure 19* we can see a plot with the number of block comparisons per frame of different video sequences. In the *Table 2* are the data of the plots.



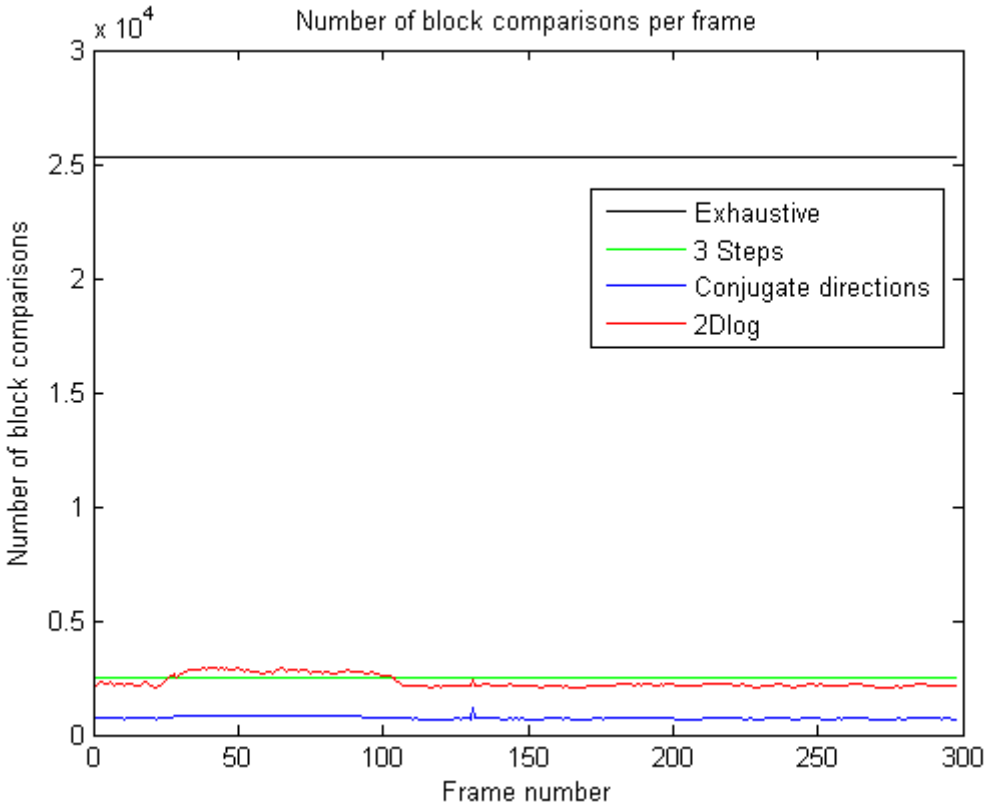
(a) carphone



(b) foreman



(c) motherdoughter



(d) table

Figure 19 *(Block comparisons per frame)*

carphone (CIF)

Method	***Mean***	***Std deviation***
Exhaustive	**101376**	**0**
3 Steps	**9900**	**0**
Conjugate directions	**3143**	**218**
2DLog	**9962**	**641**

foreman (CIF)

Method	***Mean***	***Std deviation***
Exhaustive	**101376**	**0**
3 Steps	**9900**	**0**
Conjugate directions	**3204**	**348**
2DLog	**10214**	**909**

motherdaughter (CIF)

Method	*Mean*	*Std deviation*
Exhaustive	**101376**	**0**
3 Steps	**9900**	**0**
Conjugate directions	**2902**	**60**
2DLog	**9160**	**330**

table (QCIF)

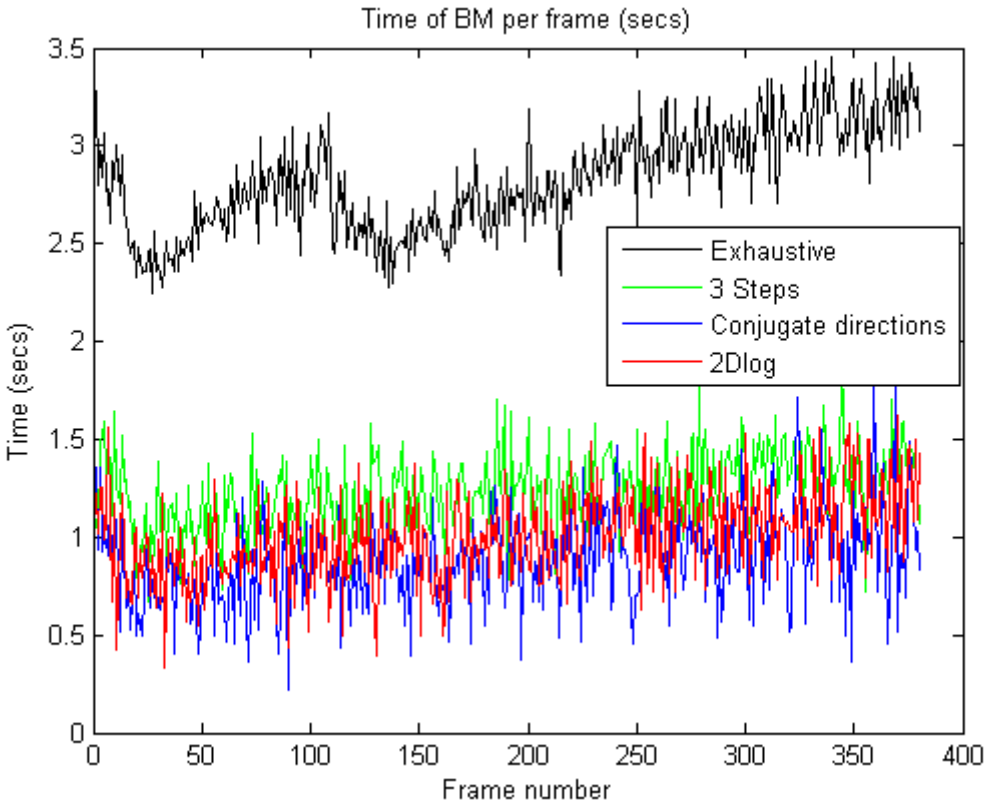
Method	*Mean*	*Std deviation*
Exhaustive	**25344**	**0**
3 Steps	**2475**	**0**
Conjugate directions	**740**	**60**
2DLog	**2322**	**274**

Table 2 *(Block comparisons data)*

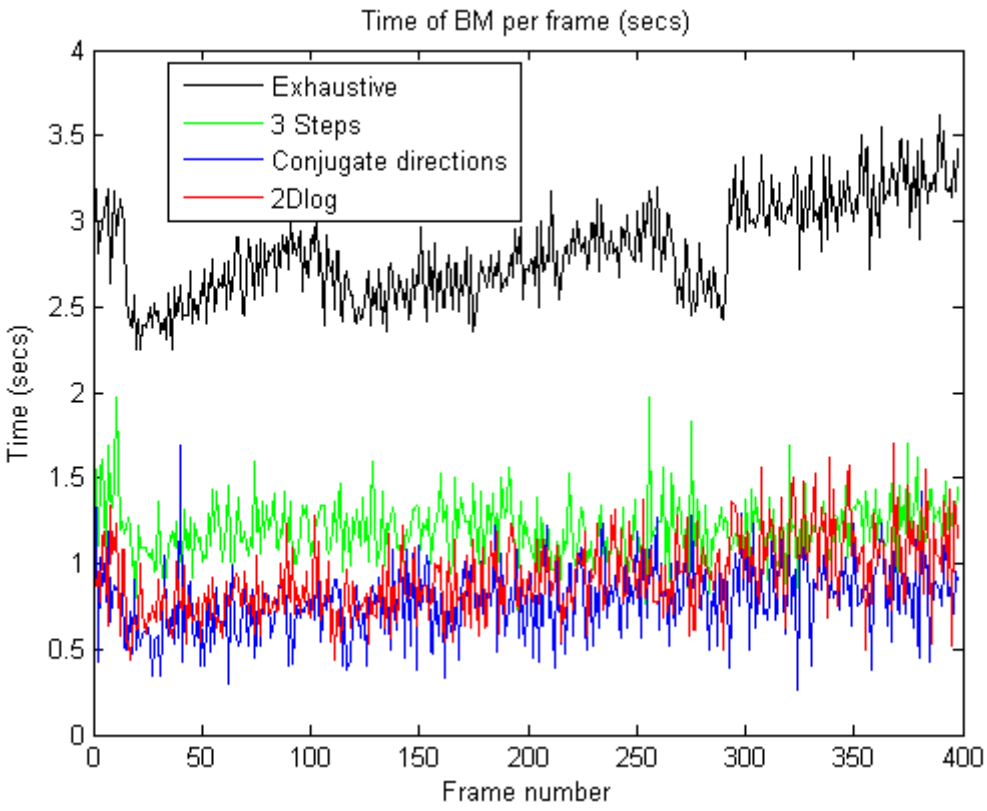
In the plots we see that the exhaustive method and the 3 steps method uses a fixed number of block comparisons per frame, and that 2DLog and conjugate direction uses a variable number of block comparisons. We can see as well that the method that does less block comparisons is clearly the conjugate directions followed by 2Dlog and 3 steps that are both very similar.

5.2 Time to compute Block matching

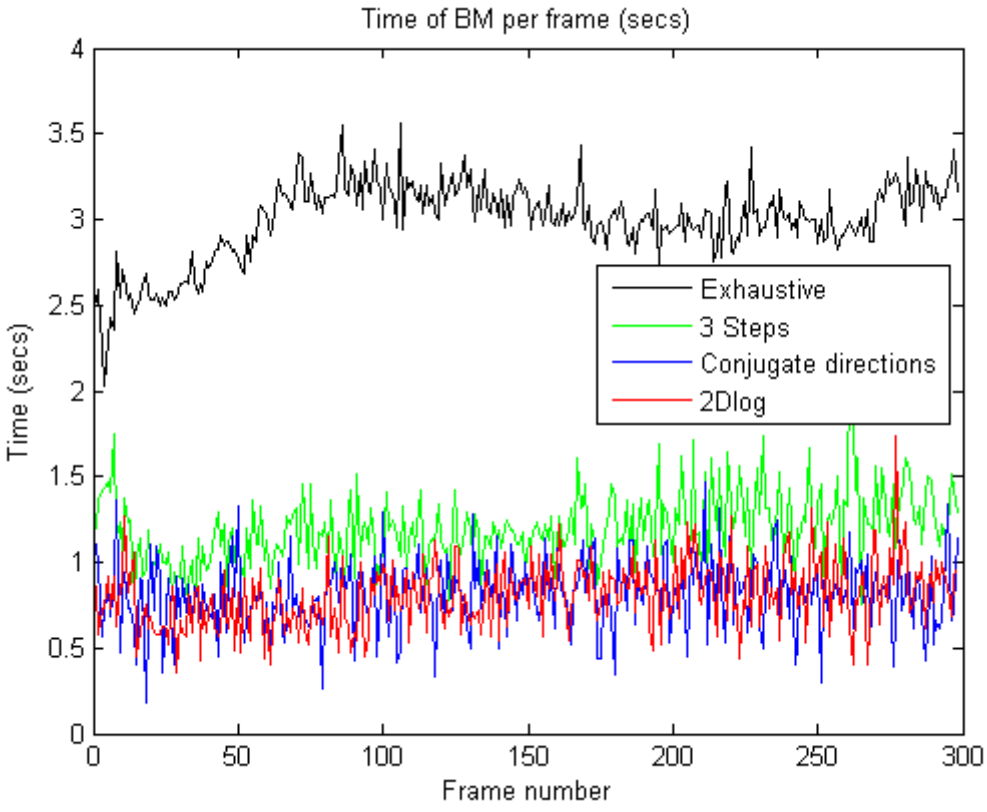
In the *Figure 20* we can see the time to compute BM per frame of different video sequences. In the *Table 3* are the data of plots.



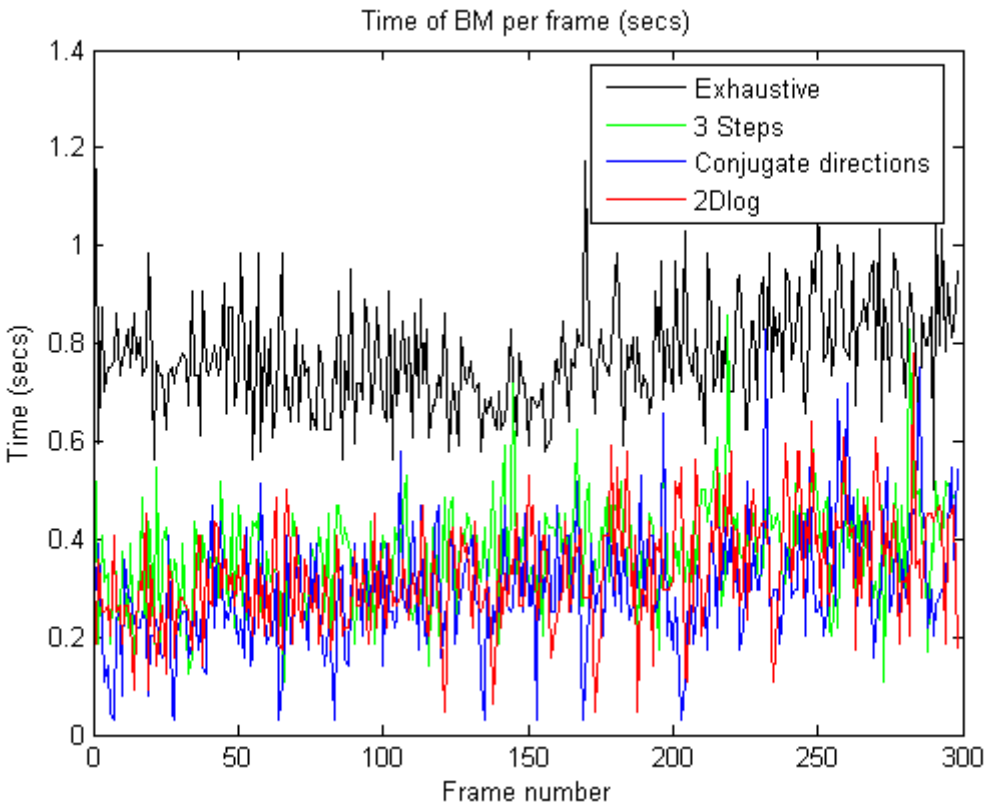
(a) carphone



(b) foreman



(c) motherdoughter



(d) table

Figure 20 *(Time to compute BM per frame)*

carphone (CIF)		
Method	***Mean (secs)***	***Std deviation (secs)***
Exhaustive	**2.82**	**0.25**
3 Steps	**1.23**	**0.20**
Conjugate directions	**0.89**	**0.25**
2DLog	**1.01**	**0.24**
foreman (CIF)		
Method	***Mean (secs)***	***Std deviation (secs)***
Exhaustive	**2.82**	**0.28**
3 Steps	**1.20**	**0.18**
Conjugate directions	**0.80**	**0.22**
2DLog	**0.92**	**0.23**
motherdaughter (CIF)		

Method	*Mean (secs)*	*Std deviation (secs)*
Exhaustive	**2.99**	**0.23**
3 Steps	**1.20**	**0.21**
Conjugate directions	**0.82**	**0.21**
2DLog	**0.81**	**0.20**

table (QCIF)

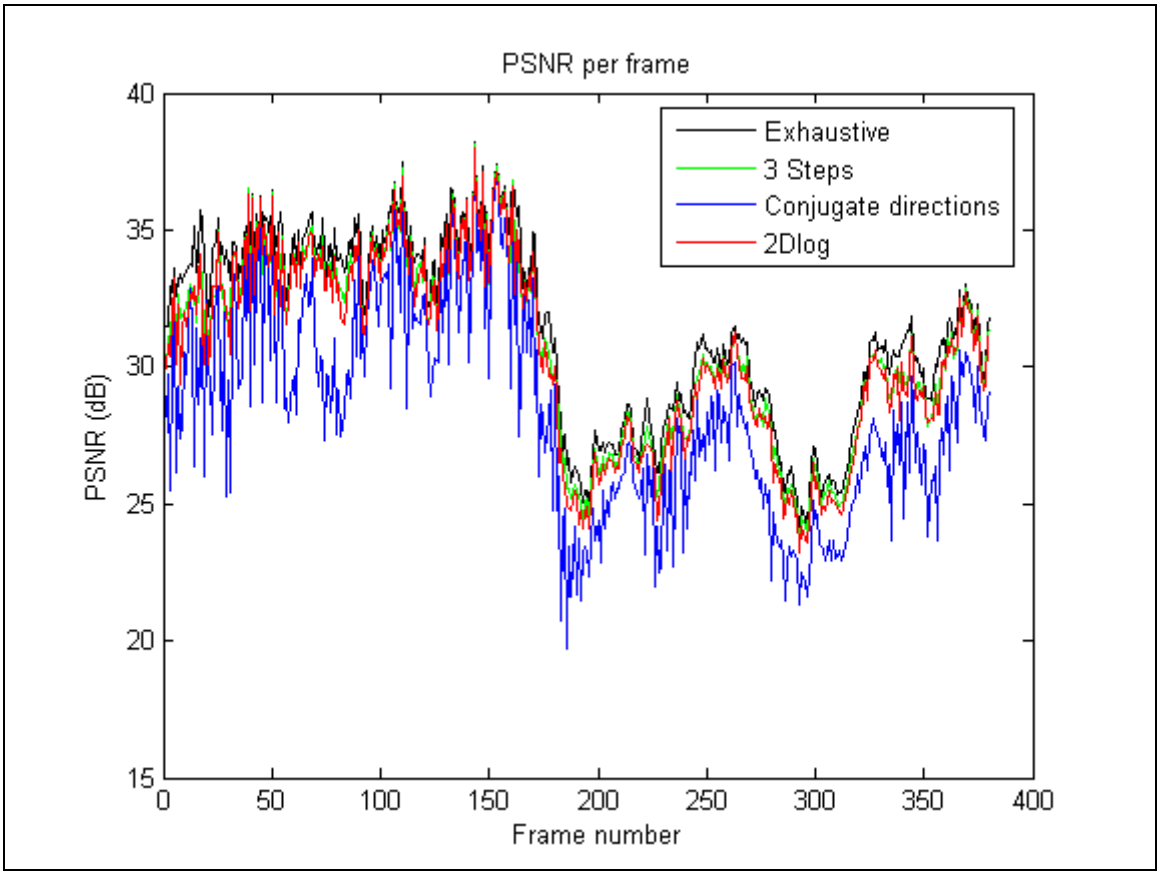
Method	*Mean (secs)*	*Std deviation (secs)*
Exhaustive	**0.77**	**0.11**
3 Steps	**0.37**	**0.11**
Conjugate directions	**0.30**	**0.12**
2DLog	**0.33**	**0.11**

Table 3 *(Time to compute BM data)*

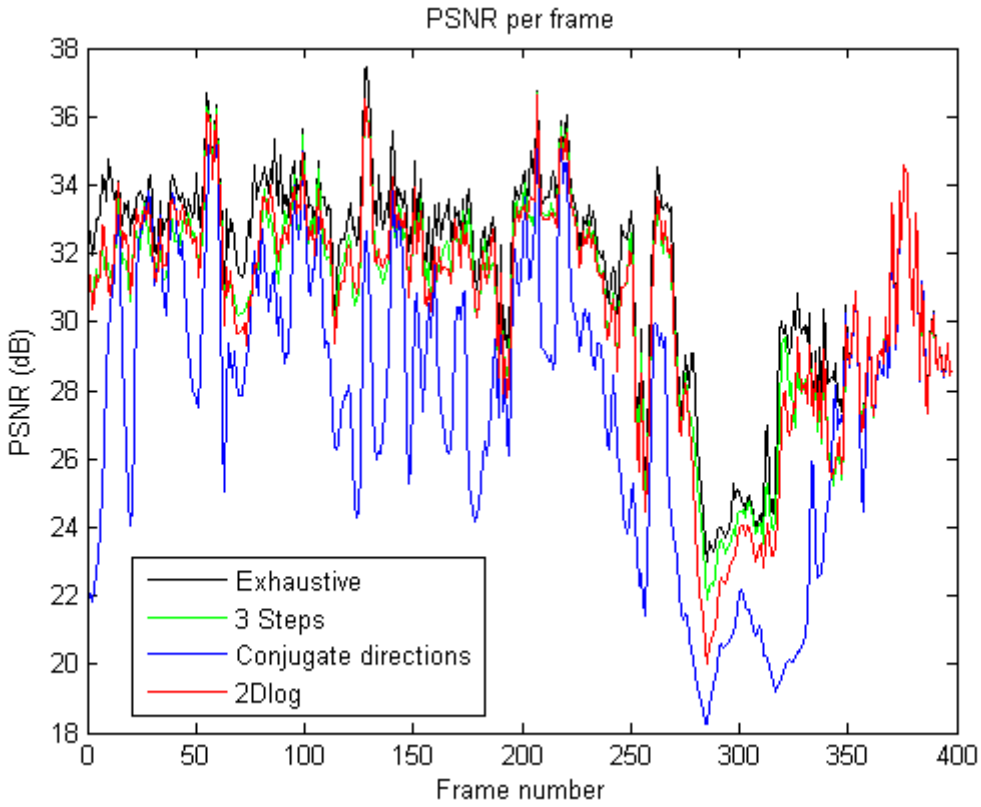
Obviously the method who has done less block comparisons, the conjugate directions, is the fastest one as well.

5.3 PSNR

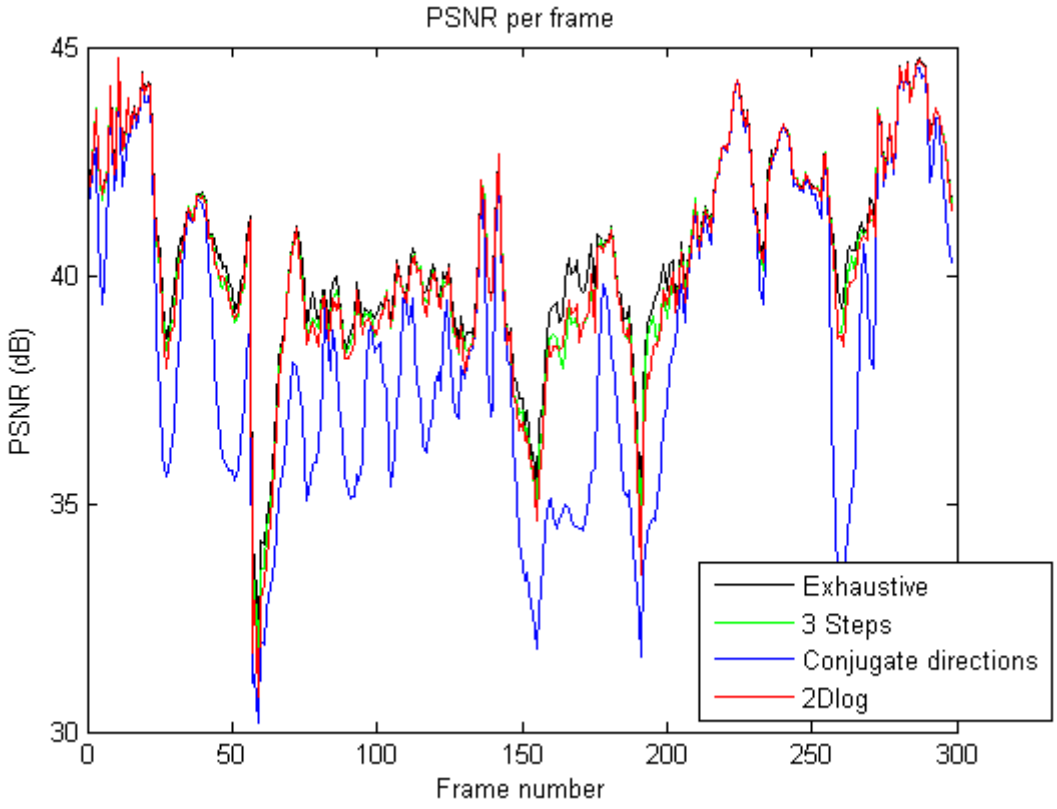
In the *Figure 21* we can see the PSNR of each frame. In the *Table 4* are the data of plots.



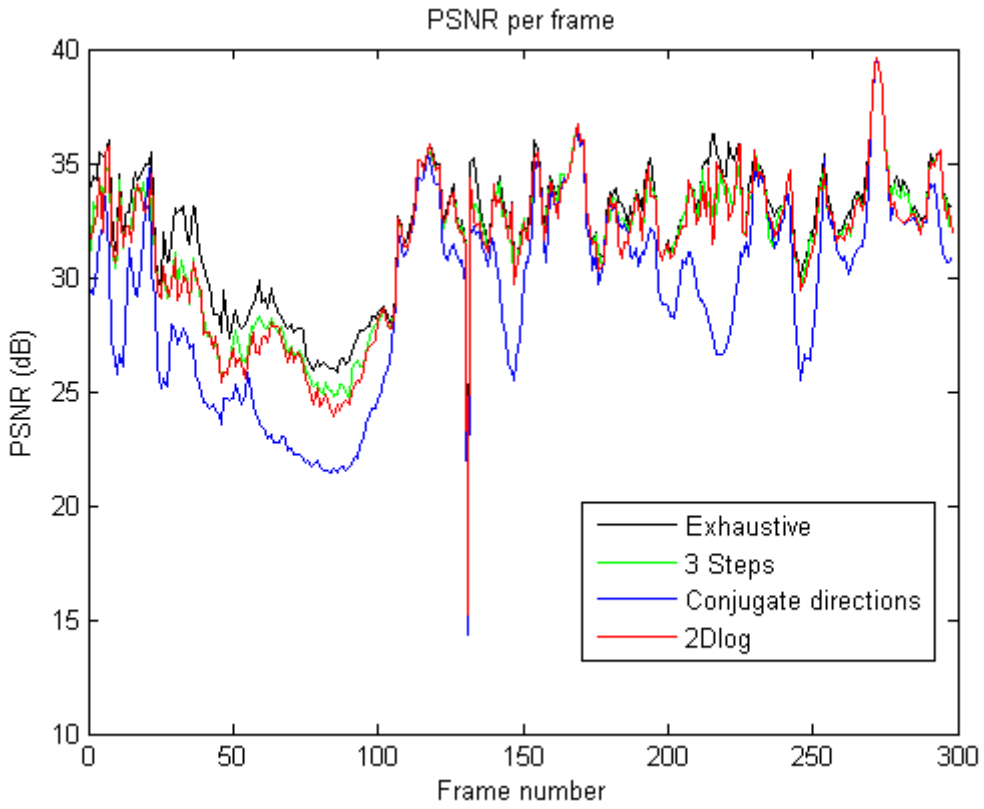
(a) carphone



(b) foreman



(c) motherdoughter



(d) table

Figure 21 *(PSNR per frame)*

carphone (CIF)		
Method	***Mean (dB)***	***Std deviation (dB)***
Exhaustive	**31.34**	**3.40**
3 Steps	**30.65**	**3.37**
Conjugate directions	**28.45**	**3.76**
2DLog	**30.46**	**3.48**
foreman (CIF)		
Method	***Mean (dB)***	***Std deviation (dB)***
Exhaustive	**31.43**	**3.11**
3 Steps	**30.53**	**3.05**
Conjugate directions	**27.70**	**4.19**
2DLog	**30.48**	**3.35**

motherdaughter (CIF)		
Method	***Mean (dB)***	***Std deviation (dB)***
Exhaustive	**40.53**	**2.19**
3 Steps	**40.32**	**2.32**
Conjugate directions	**38.63**	**3.41**
2DLog	**40.25**	**2.39**
table (QCIF)		
Method	***Mean (dB)***	***Std deviation (dB)***
Exhaustive	**32.17**	**3.00**
3 Steps	**31.50**	**3.22**
Conjugate directions	**29.26**	**4.20**
2DLog	**31.32**	**3.36**

Table 4*(PSNR data)*

In the *Figure 21*a we see that the PSNR of exhaustive method is always greater that the PSNR of the other method. This demonstrates an obvious thing: that exhaustive method is the method that gives the best PSNR for a given block size and search area.

The PSNR of the BM gives us an idea of variability of image content. For instance, in the *Figure 21*a (carphone) we see that the difference between the minimum and the maximum is 17dB, if we play the video we will see that there are low variability of image content, is a fixed shot of man talking to a fixed camera in a car, the only thing that changes in the video is a small part of the background: the car window.

In the *Figure 21*b (foreman) we see that around the frame 290 the PNSR decreases considerably, if we watch the video we will see that there is a fast pan between frames 275 and 330.

In the *Figure 21*c (motherdaughter) we see that the PSNR variability is similar to the variability of *Figure 21*a (carphone) this due to the video sequence is a fixed camera shot with a mother and her daughter talking to camera, and the only hard variation in the image content is around the frame 60, where appears the arm of the mother.

In the *Figure 21*d (table) the most important change in PSNR of this video is around the frame 130, there is hard low peak of PSNR, when we see the video we realize that this peak is due to a shot change in the sequence.

6 Conclusions

We can see in the results section that the **conjugate directions** method is the one that offers the lowest number of block comparisons. This means that it will be the fastest one. On the other hand is the one with the worst PSNR. We will choose this method in applications that have low computational power (real time) or that we need speed on coding (non real time).

If we need to reach the best PSNR and the time that the method spend is not very important in our application we can choose the **exhaustive** one.

In other applications where we need a trade off between speed and quality (PSNR) a good choice will be the **2Dlog** or the **3Steps** methods.

I have seen that in natural images (the sequence that we have used in this work) the best fit for a given block in the previous frame is usually very close of the position of the original block (very short motion vector). All studied methods (except the exhaustive one) try to use this property to minimize the block comparisons.

In future woks I would try what happens if I change the size of blocks or the size of the search areas. And I would try to increase the distance between the source and the predicted frame.

7 References

1. Wikipedia (PSNR Definition)
< http://en.wikipedia.org/wiki > [Visited: 1 April 2006]

2. Tarres Ruiz, Francesc. *Sistemas audiovisuales*. Barcelona: Edicions UPC,2000. ISBN 84-8301-393-2

3. New media republic (BM Algorithms)
< http://www.newmediarepublic.com/dvideo/compression/adv12.html > [Visited: 01 April 2006]