2008

An Investigation of Block Searching Algorithms for Video Frame Codecs

Jerome Casey
Dublin Institute of Technology, jerome.casey@dit.ie

An Investigation of Block Searching Algorithms for Video Frame Codecs

by

Jerome Casey

A dissertation submitted in partial fulfilment of the requirements for the DIT's Master of Science Degree in Applied Computing for Technologists, Faculty of Engineering, DIT, Bolton Street.

Supervisor: Anselm Griffin

I certify that this dissertation, which I now submit for examination for the award of MSc. in Applied Computing for Technologists, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation has not been submitted in whole or in part for an award in any other Institute or University.

The Institute has permission to keep, to lend or to copy this dissertation in whole or in part, on condition that any use of the material of the dissertation be duly acknowledged.

Signature: Jerome Casey Date: 5 December 2008

Candidate

Abbreviations

B-Picture – Bidirectionally predictive-coded Picture.

BDM – Block distortion measure.

BMA – Block motion algorithm.

BME – Block motion estimation.

CSP – Cross Shaped Pattern.

IEEE – Institute of Electrical and Electronics Engineers.

I-Picture – Intra-coded Picture.

ISO – International Organization for Standardization.

ITU-R – International Telecommunications Union.

LCSP – Large Cross Shaped Pattern.

LDSP – Large Diamond Shaped Pattern.

MB – Macroblock.

MAD – Mean Absolute Difference.

M-code – MATLAB code.

MPEG – Motion Picture Experts Group.

MSE – Mean Square Error.

PSNR – Peak-Signal-to-Noise-Ratio.

P-Picture – Predictive-coded Picture.

SAD – Sum of Absolute Differences.

SCSP – Small Cross Shaped Pattern.

SDSP – Small Diamond Shaped Pattern.

Table of Contents

List of Tables

List of Figures

Abstract

Block matching is the most computationally demanding aspect of the video encoding process. In many applications real-time video encoding is desired and therefore it is important that the encoding is fast. Also where handheld devices such as a PDA or mobile phone are concerned a less computationally intensive algorithm means a simpler processor can be used which saves on hardware costs and also extends battery life. An optimised algorithm also allows these devices to be used in low bandwidth wireless networks. The challenge is to decrease the computational load on the system without compromising the quality of the video stream too much, thus enabling easier and less expensive implementations of real-time encoding.

This thesis appraises some of the principal Block Search Algorithms used in Video compression today. This work follows on from the work of Aroh Barjatya who implemented 7 common Block Search Algorithms to predict P-frames in MATLAB. Three further hybrid DS algorithms are implemented in MATLAB. Additional code is added to produce plots of the main metrics and to calculate some statistics such as *Average Searching Points*, *Average PSNR* and the *Speed Improvement Ratio* with respect to the Diamond Search and the Exhaustive Search.

For a comparative analysis with previous studies 3 standard industry test sequences are used. The first sequence, *Miss America* is a typical videoconferencing scene with limited object motion and a stationary background. The second sequence, *Flower Garden* consists mainly of stationary objects, but with a fast camera panning motion. The third sequence, *Football* contains large local object motion. The performance of the 3 implemented algorithms were assessed by the aforementioned statistics.

Simulation results showed that the NCDS was the fastest algorithm amongst the 3 hybrid DS algorithms simulated. A speedup ranging from 10% for the complex motion sequence *Flower Garden* to nearly 54% for the low motion video conferencing sequence *Miss America* was recorded.

All 3 algorithms performed very competitively in terms of PSNR compared to the DS even though they use a lower number of search points on average. It was shown that the NCDS has marginally worse PSNR performance than the DS compared to the other 2 algorithms – the highest being a drop in PSNR of 0.680dB for the *Flower Garden* sequence. However, the speed improvements for NCDS are quite substantial and thus would justify its use over the DS. The results from the implementation concurred with the literature therefore validating the implementation.

The implementation was used as a guide in nominating a ‘robust’ Block Search Algorithm. When the DS, CDS, SCDS and the NCDS were compared with ARPS it was shown that ARPS generally gave both higher PSNR and higher search speed for all 3 sequences. The reason for the good performance of ARPS is that it quickly directs the search into the local region of the global minimum by calculating the Predicted Motion Vector. The minimum error from a rood pattern of nodes is found and then a final refined search calculates the motion vector.

Simulation results showed that ARPS was the best algorithm amongst the 10 algorithms simulated from the point of view of speed (lowest number of search points used per macroblock) and video quality (PSNR). For real-time encoding of video the best fast block motion algorithm to advise is ARPS.

1.0 Introduction

A video sequence consists of a series of frames viewed at a sufficiently fast frame rate to give the illusion of motion. Video generally contains a lot of data which places a large demand on video systems for both storage and transmission of digital video. For example, uncompressed CCIR601 active digital video requires a bandwidth in excess of 158Mbps – over 300 times the capacity of a 512kbps ADSL connection and only just over one hour recording on a 80GB hard disk (Chapman and Chapman, 2004). These demands have lead to a large body of research in the area of video compression, particularly motion estimation (Barjatya, 2004). Block-based motion estimation algorithms have seen widespread use in many codecs due to their effectiveness and simplicity of implementation (Tham *et al*, 1998). They have been used in MPEG-1, MPEG-2, H.261, H.262 and more recently in MPEG-4 and H.264 (Wiegand *et al*, 2003). A frame is selected as a reference frame and subsequent frames are predicted from the reference. An encoder will output a series of motion vectors and a difference image for each original uncompressed frame. At the decoder the original frame is reconstructed from the summation of the motion compensated image (produced from its motion vectors and corresponding reference frame) and the motion compensated difference image. Much less data is required to code the motion vectors and the motion compensated difference image than is needed to code the original image and so compression is achieved.

The main focus of this work is on the analysis of current fast block search algorithms for motion estimation with a conclusion of 'best-practice' when nominating a Block Search Algorithm. The optimal algorithm will have a low computational cost whilst not degrading the quality of the encoded video.

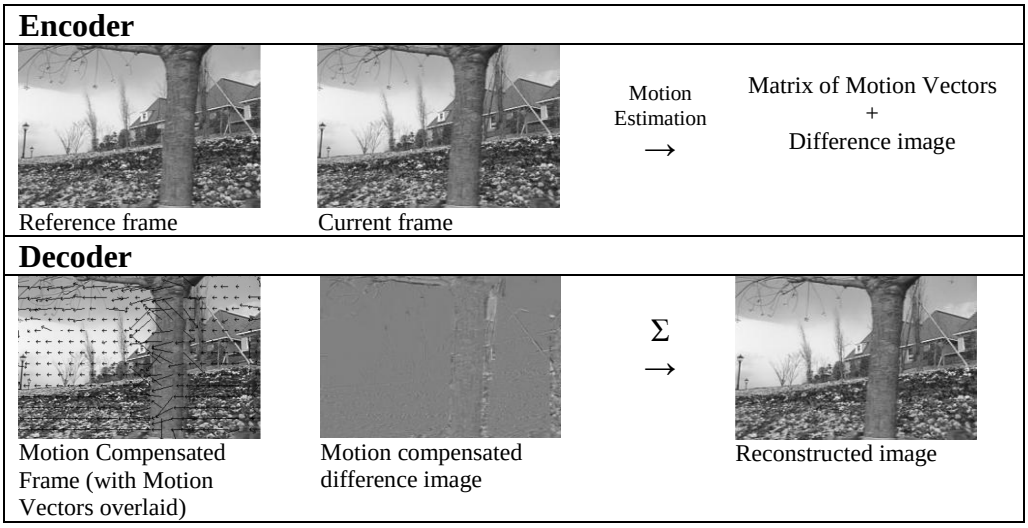


Fig. 1.1 Encoding and Decoding of a video sequence and the role of Motion Vectors.
The *Flower Garden* sequence involves a pan from left to right, a motion vector points to the best match macroblock in the reference frame, hence the predominance of motion vectors pointing to the left. *Source*: Girod, (2008).

1.1 Fast Motion Estimation Algorithms

Block matching is the most computationally demanding aspect of the video encoding process. In many applications real-time video encoding is desired and therefore it is important that the encoding is fast. Also where handheld devices are concerned a less computationally intensive block matching algorithm means a simpler processor can be used saving hardware costs and extending battery life (Dahlstrand, 2001). An optimised algorithm also allows these devices to be used in low bandwidth wireless networks.

As can be seen from table 1.1 which shows the number of operations per second required for a real-time implementation of a video encoder (Cheung, 1998) the motion estimation operation is by far the largest component of video encoding. It requires 608 million operations per second or 63% of the computational load. This is a considerable problem when trying to achieve real-time coding of video streams. The challenge is to decrease the motion estimation load on the system without compromising the quality of the video stream too much, thus enabling easier and less expensive implementations of real-time coding.

Additionally the more accurate the motion vector prediction the smaller the motion compensated difference image and hence the better the compression efficiency. This reduces the overall bandwidth requirements (such as in IP Video systems) but more importantly it can significantly reduce the amount of storage required for recording the video, often one of the most expensive items in a system (Keepence, 2008).

Table 1.1: MOPS requirement for real-time implementation for H.261 compression

Operation	**MOPS**
RGB to YC_bC_r	27
Motion estimation (25 searches in a region)	608
Inter/Intraframe coding	40
Loop filtering	55
Pixel prediction	18
2-D DCT	60
Quantization, zig-zag scanning	44
Entropy coding	17
Frame reconstruction	99
Total	**968**

Source: Cheung, (1998).

1.2 Achieving Motion Estimation

In a sequence of frames, the current frame is predicted from a previous frame known as a reference frame. The current frame is divided into non overlapping macroblocks, typically 16 pixels x 16 pixels in size. This choice of size is a good trade-off between accuracy and computational cost. However, motion estimation techniques may choose different block sizes, and may vary the size of the blocks within a given frame.

Each macroblock in the current frame is compared to a macroblock in the reference frame using some cost function, and the best matching macroblock is selected. The search is conducted within a predetermined search window defined by the search parameter *p*. Typically *p* is set to ±7 pixels. A vector denoting the displacement of the macroblock in the reference frame with respect to the macroblock in the current frame, is determined. This vector is known as the motion vector.

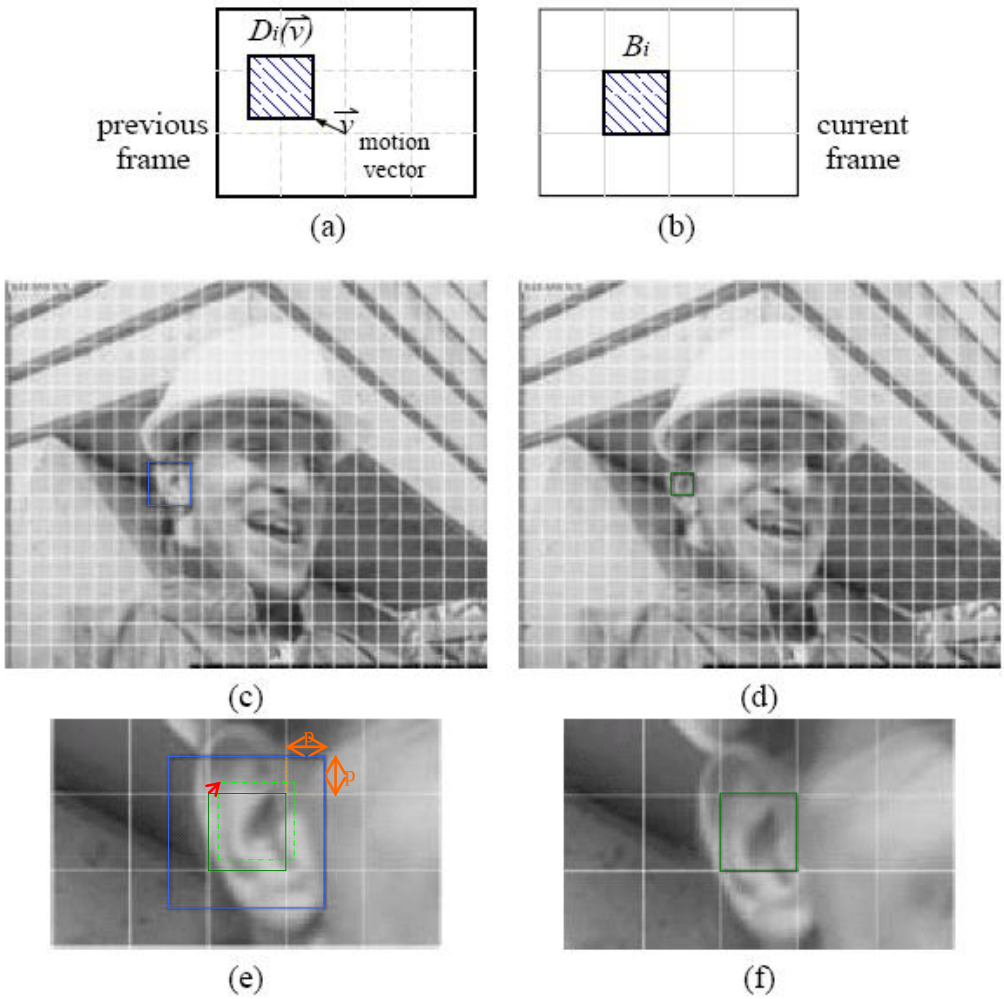


Fig. 1.2 How the Motion Vector is found in the Reference frame.
Block-matching motion estimation algorithms find the motion vector of the current block by finding the best-matching displaced block in the reference frame. For example, (c) and (d) show the 137th frame and the 138th frame of the *foreman* sequence. The current frame (138) is divided into non-overlapping blocks as shown in (d). The motion vector for the current block of the current frame - shown in (f) in green - is found by locating the best-matching displaced block within the corresponding search window in the range [-p, p] - shown in (e) in blue - in the previous frame (137). The displacement vector which produces the minimal matching-error via a cost function is the motion vector – shown above as a red arrow.
Source: Chen, (1998).

1.3 Cost Function

The matching of one macroblock with another is based on the output of a cost function also referred to as a block distortion measure (BDM). The macroblock that results in the least cost is the one that matches the current block the closest. There are various cost functions, of which the most popular and less computationally expensive is:

The Mean Absolute Difference (MAD) given by equation (i).

$$MAD = \frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left|C_{ij} - R_{ij}\right| \qquad \text{(i)}$$

Another cost function is the Mean Squared Error (MSE) given by equation (ii).

$$MSE = \frac{1}{N^2}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\left(C_{ij} - R_{ij}\right)^2 \qquad \text{(ii)}$$

where N is the size of the macrobock, C_{ij} and R_{ij} are the pixels being compared in the current macroblock and the reference macroblock, respectively.

Ghanbari (1990) states that the use of mean absolute error rather than the more complex mean square error as the distortion measure, results in slightly better entropy performance (almost 0.8% lower prediction error).

For each cost function, a comparison is made pixel by pixel using the luma value only (Richardson, 2002). These errors are summed over the macroblock and if this error is less than the previous error, the location of the macroblock in the reference picture is saved. Once all macroblocks in the search window have been examined, the motion vector is determined based on the macroblock with the lowest error measure.

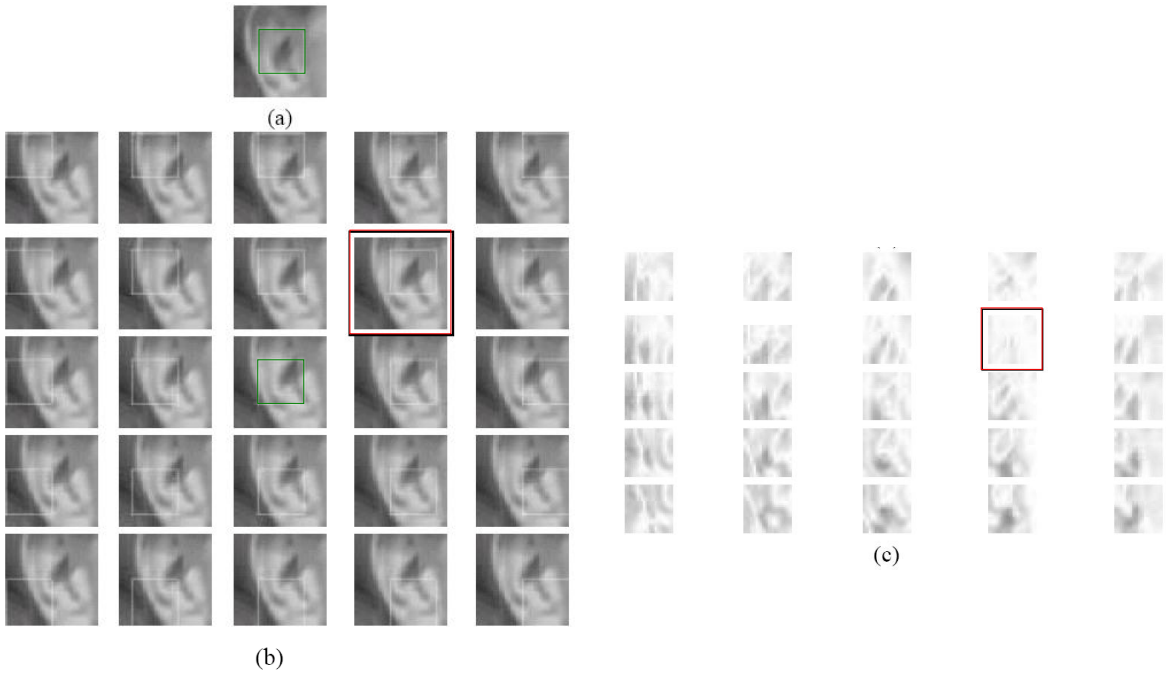


Fig. 1.3 A Cost Function being applied at a number of locations within the search window. (a) shows the current block of the current frame in green (b) shows different displaced blocks in the reference frame including the corresponding location of the current block in green (c) shows the corresponding residuals (matching errors). The displacement (upper-right) that finds the best-matching block (marked in red) is the motion vector.
Source: Chen, (1998).

1.4 Rate Distortion Performance-PSNR

Peak-Signal-to-Noise-Ratio (PSNR) given by equation (iii) characterizes the quality of the motion compensated image (created by using motion vectors and the macroblocks from the reference frame) compared to the original image:

$$PSNR = 10Log_{10}\left[\frac{(Peak\ to\ peak\ value\ of\ original\ data)^2}{MSE}\right] \qquad \text{(iii)}$$

The higher the value of PSNR, the smaller will be the error residual giving a more improved video quality (Nie and Ma, 2002). This metric is used as a quality indicator when comparing various block match algorithms.

1.5 Monotonic Error Surface

The idea behind many fast block search algorithms is that the error surface due to motion in every macroblock is unimodal. A unimodal surface is a bowl shaped surface such that the weights generated by the cost function increase monotonically from the global minimum. Some of the earlier algorithms rely on this assumption for their search strategies and therefore that the local minimum is actually the global minimum. Since this unimodal assumption is sometimes not valid, these algorithms are susceptible to being trapped at local minima, and as a result, do not achieve the same rate distortion performance as the full search.

However, these algorithms drastically reduce the number of search positions over the full search strategy. For software implementations, this results in a substantial reduction in the computational load and so the implemental benefit is worth the loss in compression efficiency for many applications (Booth, 2003).

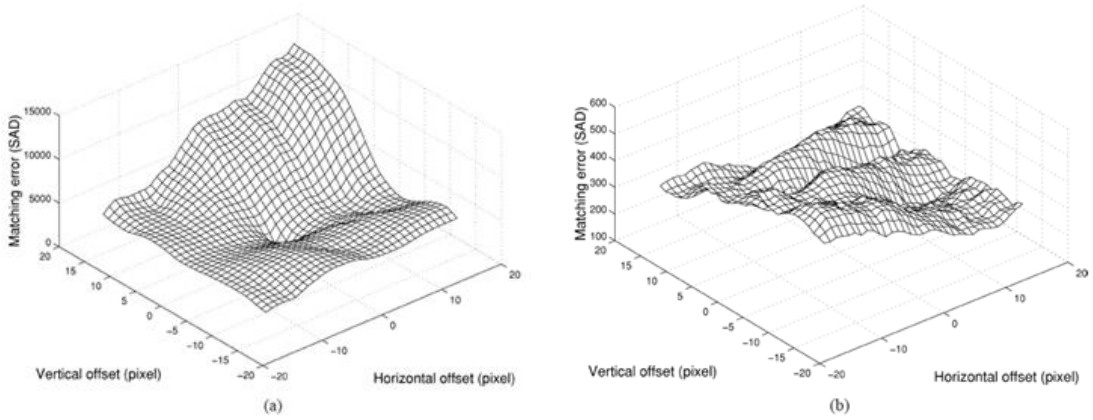


Fig. 1.4 Examples of Matching Error Surfaces.
(a) Unimodal error surface with a global minimum error point. (b) Non-unimodal error surface with multiple local minimum error points.
Source: Yao Nie and Kai-Kuang Ma (2002).

2.0 Literature Review

The Block-Matching technique for Motion Estimation was originally described by Jain and Jain (1981). It was easy to implement and thus widely adopted. Each image frame was divided into a fixed number of (usually) square blocks. For each block in the frame, a search is made in the reference frame over an area of the image that allows for the maximum translation that the coder can use. The search is for the best matching macroblock that gives the least prediction error - usually mean absolute difference (MAD) which is the easiest to compute.

Typical block sizes are of the order of 16 pixels x 16 pixels, and the maximum displacement might be ±64 pixels from a block's original position. Several search strategies are possible, usually using some kind of sampling mechanism, but the most straightforward approach is the Exhaustive Search. This is computationally demanding in terms of data throughput, but algorithmically simple, and relatively easily implemented.

A good match during the search means that a good prediction can be made, but the improvement in prediction must outweigh the cost of transmitting the motion vector. A good match requires that the whole macroblock has undergone the same translation, and the macroblock should not overlap objects in the image that have different degrees of motion, including the background.

The choice of macroblock size to use for motion compensation is always a compromise, smaller and more numerous blocks can better represent complex motion than fewer large ones. This reduces the work and transmission costs of subsequent correction stages but with greater cost for the motion information itself. An appraisal of some of the principal Block Search Algorithms used in Video compression today is now presented.

2.1 Exhaustive Search (ES)

The exhaustive search algorithm checks every possible motion vector candidate in a search window using a distortion measure and finds the motion vector within that window that minimizes the distortion. Although ES finds the best motion vector in a global sense, the large number of distortion calculations that it requires adds to the computational cost of a video coder and limits the algorithm's practical implementations.

- The most computationally expensive block matching algorithm of all is the Exhaustive Search since it calculates the cost function at each possible location in the search window - 225 locations (15x15).
- As a result it finds the best possible match and gives the highest PSNR amongst any block matching algorithm.
- The obvious disadvantage of ES is that the larger the search window gets the more computations it requires.

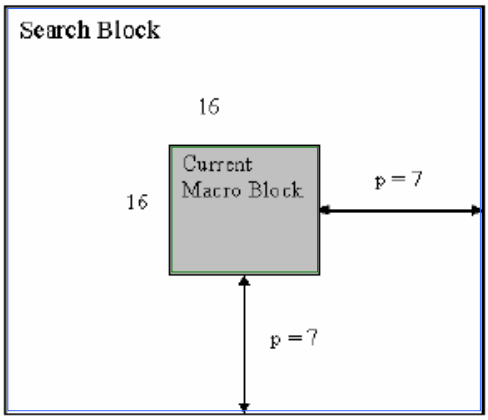


Fig. 2.1 Exhaustive Search Algorithm.
Source: Barjatya, 2004.

2.2 Three Step Search (3SS)

Fast block matching algorithms began to emerge in the early eighties trying to achieve the same PSNR as Exhaustive Search but using a lower number of search points to reduce the computational complexity. One early fast algorithm was the Three Step Search introduced by Koga *et al* (1981).

- The 1st step of the 3SS begins with the search location at the centre and sets the 'step size' S = 4, instead of the usual search parameter value of 7 for ES. It then searches at eight locations +/- S pixels in the x and y direction around location (0,0) as well as the centre location.
- From these **nine** locations searched it picks the one giving the least cost and makes it the new search origin - (4, 0) in the diagram. The 2nd step begins with this origin and with the new step size of S = S/2 = 2, and repeats a similar search finding the best match again - at (6, -2) in the diagram. This is made the new search origin for the 3rd and final step which will have the new step size S = S/2 = 1.
- The best-match macroblock in the 3rd step is found and this is the best match overall. Its location is the motion vector value. The calculated motion vector is then saved for transmission. The 3SS gives a flat reduction in computation by a factor of 9 compared to ES - since 25 locations in total are checked (9+8+8) as compared to 225 for ES.

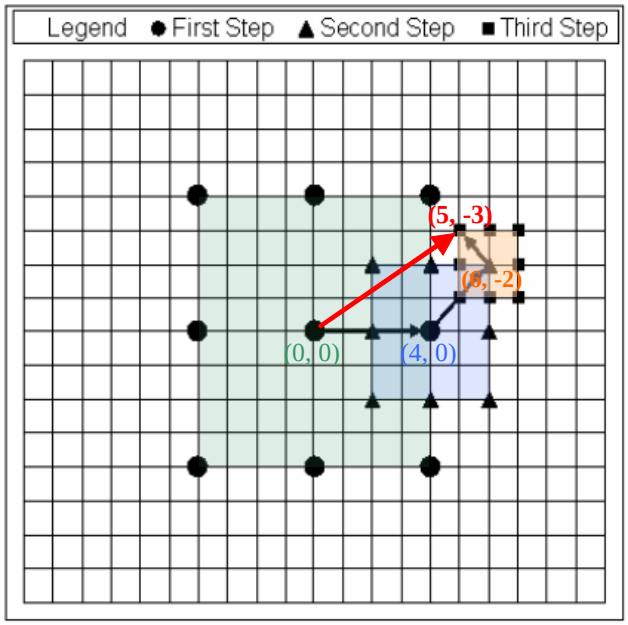


Fig. 2.2 Three Step Search Algorithm procedure.
Motion Vector is (+5, -3). *Source*: Barjatya, 2004.

2.3 New Three Step Search (NTSS)

Li *et al*, (1994) developed the New Three Step Search (NTSS) to improve on the TSS which was good for large motions but was prone to missing small motions (Jing and Chau, 2004). It used a centre biased searching scheme like TSS but it had a provision for a half way stop after the first or second step - thus reducing computational cost. It was one of the first widely accepted fast algorithms and frequently used for implementing earlier standards like MPEG 1 and H.261 (Barjatya, 2004).

- In the 1st step 16 points are checked in addition to the search origin for lowest weight using a cost function. 8 are a distance of S = 4 away (similar to TSS) and the other 8 are at a distance S = 1 away from the search origin. If the lowest cost is at the origin then the searching is stopped and the motion vector is set as (0, 0).
- If the lowest weight is at any one of the 8 locations at S = 1, then the origin of the search is changed to that point and the weights adjacent to it are checked. Depending on whether the origin is located at the middle of a horizontal or vertical axis or a corner a further 3 or 5 adjacent points are checked. The location that gives the lowest weight is the closest match and the motion vector is set to that location. This scenario results in a total of either 20 or 22 search points being checked for these quasistationary (within a central 2x2 area) macroblocks (Tham *et al*, 1998).
- Alternatively if the lowest weight after the first step was one of the 8 locations at S = 4, then the normal TSS procedure is followed.
- Hence although this process might need a minimum of 17 (8+9) points to check every macroblock (stationary), it also has the worst-case scenario of 33 (8+9+8+8) locations to check.

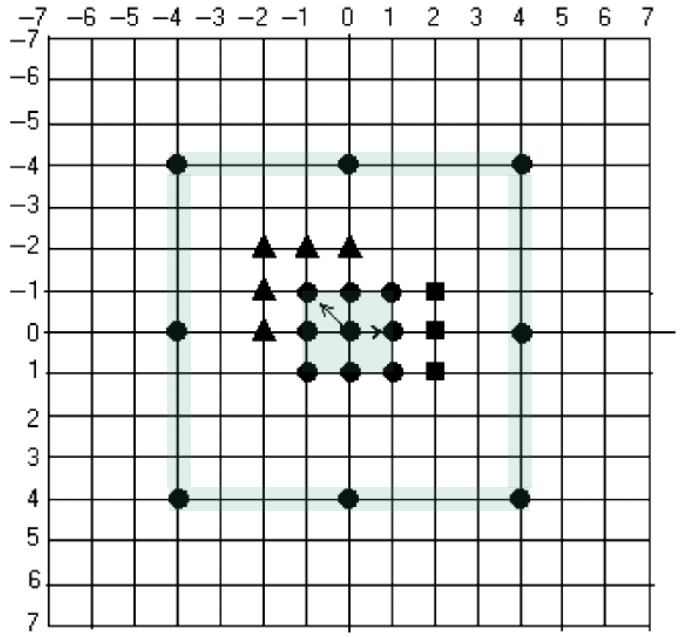


Fig. 2.3 New Three Step Search Algorithm procedure.
Source: Jing and Chau, 2004. Big circles are checking points in the first step of TSS with the extra 8 points added in the first step of NTSS. Triangles and squares are second step of NTSS showing 3 points and 5 points being checked when least weight in first step is at one of the 8 neighbours of the window centre.

2.4 Simple and Efficient Search (SES)

Lu and Liou (1997) developed a Simple And Efficient Search Algorithm which improved on TSS by halving the number of computations while keeping the same regularity and good performance. The algorithm halved the number of computations of the TSS on the basis that for a unimodal surface there cannot be two minimums in opposite directions. Thus the 8 point fixed pattern search of TSS can be changed to incorporate this and save on computations. The algorithm still has three steps like TSS, but with each step having two phases.

- The search area is divided into four quadrants and the algorithm checks three locations A, B and C as shown. A is at the origin and B and C are S = 4 locations away from A in orthogonal directions.
- Depending on the MAD calculated at each of the three locations the first phase chooses which one of the possible 4 quadrants to search for the second phase. The rules for determining which quadrant is searched in the second phase are as follows:

If MAD(A) ≥ MAD(B) and MAD(A) ≥ MAD(C), select (a);
If MAD(A) ≥ MAD(B) and MAD(A) < MAD(C), select (b);
If MAD(A) < MAD(B) and MAD(A) < MAD(C), select (c);
If MAD(A) < MAD(B) and MAD(A) ≥ MAD(C), select (d);

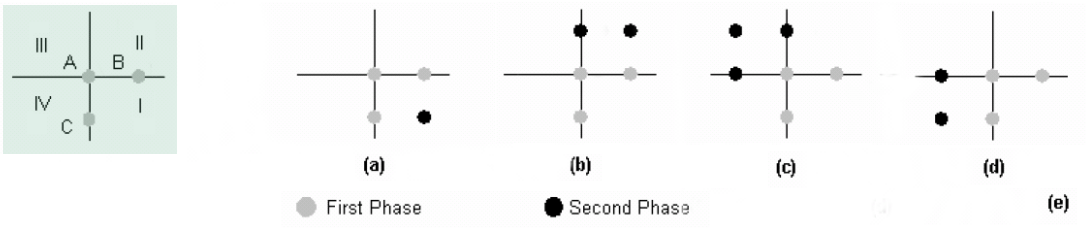


Fig. 2.4 Search patterns corresponding to each selected quadrant of the SES.
(a) to (d) show when an individual quadrant from I to IV is selected. *Source*: Lu and Liou (1997).

- Some additional points are then selected depending on the quadrant chosen and the second phase finds the location with the lowest weight and sets it as the new origin. The step size is changed to S = 2 - similar to the TSS - and the process is repeated until S = 1 is reached. The location with the lowest weight is the motion vector.

Although SES saves a lot on computation as compared to TSS, it was not widely accepted for two reasons (Barjatya, 2004).
Firstly, in reality the error surfaces are not strictly unimodal and hence the PSNR achieved by SES is poor compared to TSS.
Secondly, the Four Step Search, published the year before offered low computational cost compared to TSS and gave significantly better PSNR.

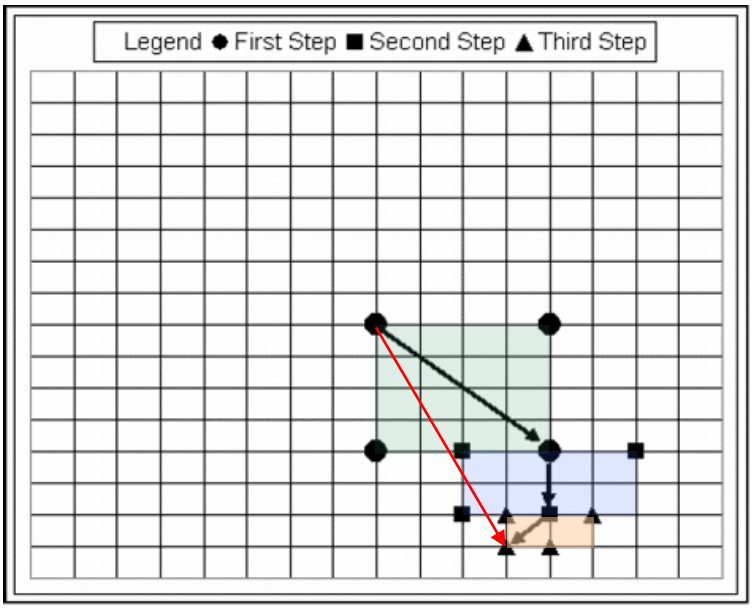


Fig. 2.5 Simple and Efficient Search Algorithm procedure.
Motion Vector is (+3, +7). *Source*: Barjatya, 2004.

2.5 Four Step Search (4SS)

Po and Ma (1996) introduced the four-step search (4SS) algorithm using a centre-biased checking point pattern with a halfway-stop technique. This meant that the algorithm could have 2, 3 or 4 search steps and thus the total number of checking points could vary from 17 (9+8) for best-case to 27 (9+5+5+8) for worst case such as when estimating large movement (Po and Ma, 1996). Po and Ma showed that 4SS performed better than the popular TSS in terms of motion compensation errors (albeit with two block matches more in the worst-case) and had similar performance to the NTSS. In addition, the 4SS also reduced the worst-case computational requirement by 6 block matches from 33 for N3SS to 27 search points and the average computational requirement from 21 to 19 search points giving it an edge over N3SS.

The 4SS algorithm (Po and Ma, 1996) is summarized as follows:

- Step 1: A minimum block distortion measure (BDM) point is found from a pattern of 9 checking points in a 5×5 window (i.e. S = 2) as shown in Fig. 2.6a. If the minimum BDM point is found to be at the centre of the search window, the search jumps to Step 4; otherwise if it is at one of the other 8 points this is made the new origin and the search moves on to Step 2.

- Step 2: The search window size is maintained at 5×5. However, the search pattern will depend on the position of the previous minimum BDM point.

  a) If the previous minimum BDM point is located at the corner of the previous search window, 5 additional checking points as shown in Fig. 2.6b are used.
  b) If the previous minimum BDM point is located at the middle of a horizontal or vertical axis of the previous search window, 3 additional checking points as shown in Fig. 2.6c are used.

If the minimum BDM point is found to be at the centre of the search window at this step, the search jumps to Step 4; otherwise the search moves on to Step 3.

- Step 3: The searching pattern strategy is the same as in Step 2, but finally it will go to Step 4.

- Step 4: The search window is reduced to 3×3 (i.e. S = 1) as shown in Fig. 2.6d and the direction of the overall motion vector is considered as the minimum BDM point among these final 9 searching points.

An advantage of 4SS is that the intermediate steps may be skipped and then jumped to the final step with a 3×3 window if at any time the minimum BDM point is located at the centre of the search window. Based on this 4SS pattern, the whole 15×15 displacement window can be covered even though only the small 5×5 and 3×3 search windows are used.

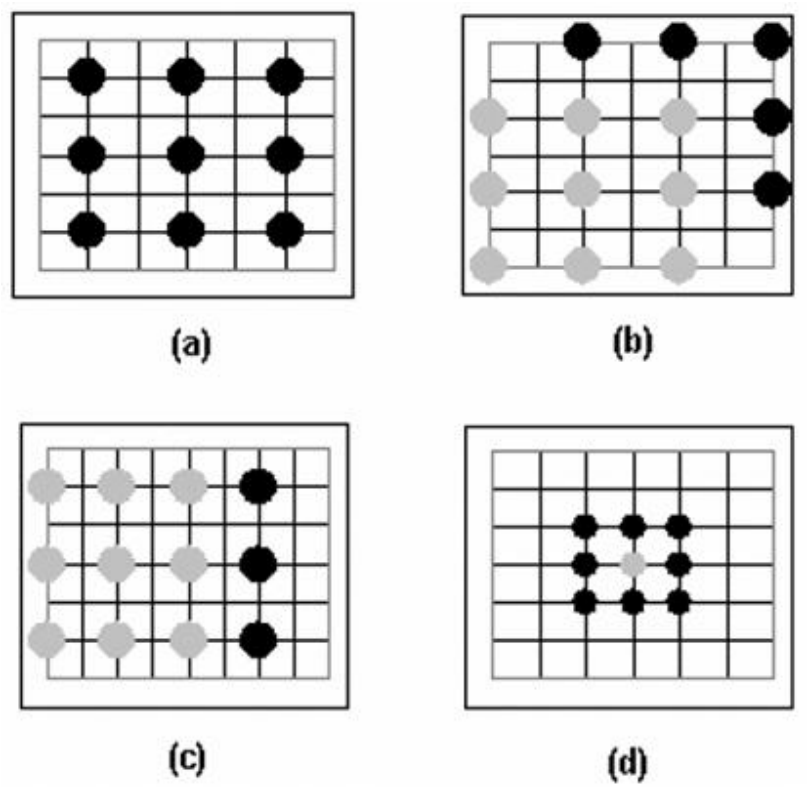


Fig. 2.6 Search Patterns used in the Four Step Search Algorithm.
Source: Barjatya, 2004.

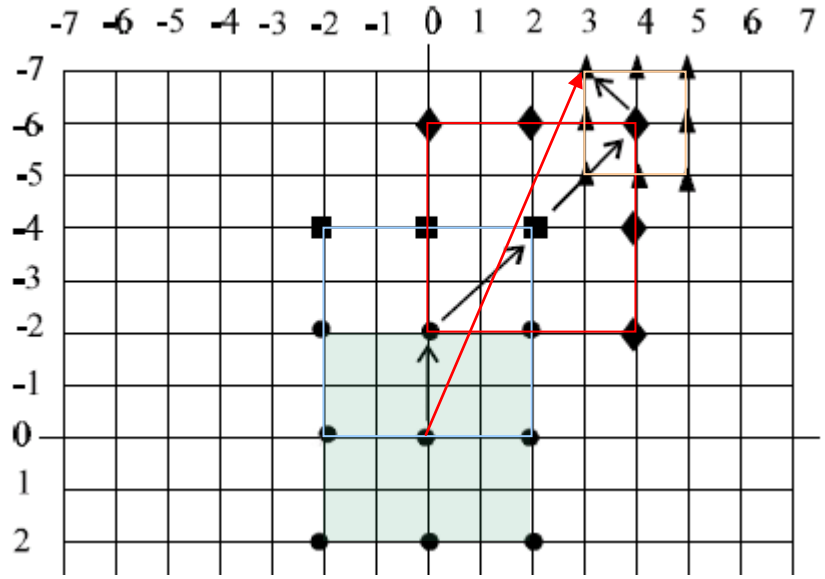


Fig. 2.7 Four Step Search Algorithm procedure.
Motion Vector is (+3, -7), 25 checking points used. *Source*: Po and Ma (1996).

2.6 Diamond Search (DS)

Tham *et al* (1998) introduced a Novel Unrestricted Centre-Biased Diamond Search Algorithm (UCBDS) - more commonly referred to as the Diamond Search Algorithm (DS). They reported it had a best case search of only 13 (9+4) search points and an average of 15.5 block matches - making it consistently faster than any of the previous suboptimal (non exhaustive) block-matching techniques.

The UCBDS algorithm (Tham *et al*, 1998) is summarized as follows:

- The algorithm uses two different types of search pattern: a Large Diamond Search Pattern (LDSP) as shown in Fig. 2.8a and a Small Diamond Search Pattern (SDSP) as shown in Fig. 2.8d.
- The first step uses a LDSP. A minimum block distortion measure (BDM) point is found at one of 9 checking points in a 5×5 window (i.e. S = 2) as shown in Fig. 2.8a.

1. If the minimum BDM point is at one of the four vertices then this point is made the centre of a new LDSP and 5 new candidate points are evaluated as shown in Fig. 2.8b.

2. If the minimum BDM point is at one of the other four points along a face then this point is made the centre of a new LDSP and 3 new candidate points are evaluated as shown in Fig. 2.8c.

These 2 scenarios are repeated without limit - all the time using LDSP - until the minimum BDM point is found to be at the centre of the search window.

3. If the minimum BDM point is found to be at the centre of the search window, the search changes to a SDSP with 4 more internal candidate points being evaluated as shown in Fig. 2.8d. The candidate point with the minimum BDM is chosen as the motion vector.

Tham *et al* (1998) concluded from their results that UCBDS was more efficient, effective, and robust when compared to the existing FS, TSS, NTSS, and FSS due to the following reasons:

• Efficiency—UCBDS is highly centre biased, and it has a very compact diamond search point configuration. This allowed a minimum of only 13 candidate search points per macroblock - resulting in a speed improvement of up to 31% over the FSS.

• Effectiveness—UCBDS has the freedom to search for the true motion vector due to its unrestricted search strategy. This indirectly reduces the chances of being trapped at a local minimum and leads to lower motion compensation errors.

• Robustness—As UCBDS is unrestricted and does not have a predetermined number of search steps, it is flexible enough to work well for any search range/window size.

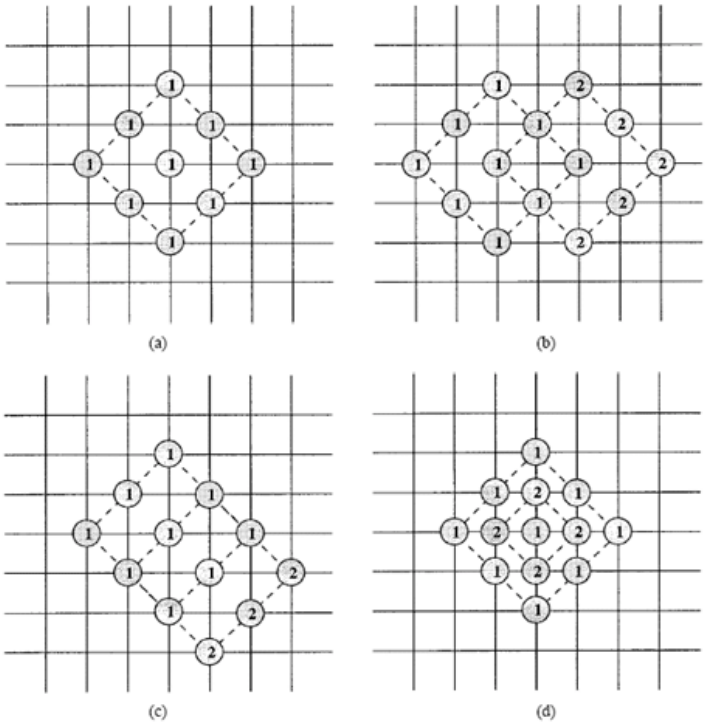


Fig. 2.8 Search Patterns used in the Diamond Search Algorithm.
(a) Original diamond search-point configuration. (b) Next step along a diamond's vertex. (c) Next step along a diamond's face. (d) Final step with a shrunk (small) diamond. *Source*: Tham *et al* (1998).

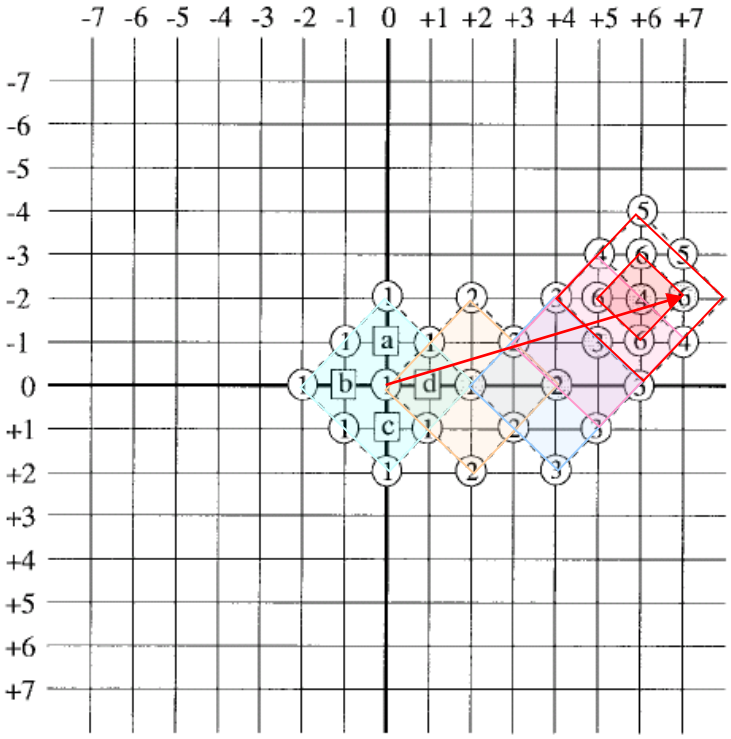


Fig. 2.9 Diamond Block Matching Algorithm procedure.
Motion Vector is (+7, -2), five LDSPs are needed in this example followed by a final SDSP. There are 28 (9+5+5+3+2+4) block evaluations in total. Note: any candidate points that extend beyond the search window of w = ±7 are ignored.
Source: Tham *et al* (1998).

2.7 Adaptive Rood Pattern Search (ARPS)

Nie and Ma (2002) outlined a number of flaws with the diamond search (DS) which was the best BMA at that time. They argued that when the size of the fixed search DS pattern does not match the magnitude of the actual motion, over search or under search will occur leading to certain search deficiency and inaccuracy:

1. For example, in DS, LDSP will be too large for searching a small motion vector with a length less than 2 pixels away from the search centre, thus causing unnecessary searches (i.e. over search).
2. On the other hand, in the case of large and complex motion (e.g. the *Foreman* sequence), the characteristic of centre-biased motion vector distribution is very weak, and the unimodal error surface assumption is no longer valid [see Fig. 1.2 (b)]. A LDSP could be too small for searching a large motion vector (i.e. under search) and leads to either a long search path (causing unnecessary intermediate searches) or being trapped into a local minimum matching error point (yielding large residuals and degrading video quality).

These observations led them to develop an adaptive rood search pattern (Nie and Ma, 2002). The algorithm is summarized as follows:

- The motion vectors of the macroblocks in the neighbourhood of the current block are well correlated with the motion vector of the current block and are thus reliable for prediction. Nie and Ma, (2002) decided to use the motion vector of the macroblock directly to its left as a starting point, calling it the Predicted motion vector. This will be available as scanning is done in raster order. This step will direct the search into the local region of the global minimum.
- The predicted motion vector in Fig. 2.10 points to (+2, -1). This point is checked using a cost function as well as a rood pattern of locations. The rood pattern has a step size of S = Max (|X|, |Y|) where X and Y are the x-coordinate and y-coordinate of the predicted motion vector. For all macroblocks in the first column of the frame (where there is no macroblock directly to the left) the rood pattern step size is fixed at 2 pixels.
- The point that has the least weight becomes the origin for subsequent search steps to essentially perform a refined local search. The assumption of unimodal error surface formed in this area is valid, hence a fixed, compact, unrestricted and small search pattern such as SDSP is used.
- The SDSP is repeated until the least weighted point is found to be at the centre of the SDSP.
- A further small improvement in the algorithm can be to check for Zero Motion Prejudgment. If the least weighted point is already at the centre of the rood pattern the search is stopped half way.

The main advantage of this algorithm over DS is that if the predicted motion vector is (0, 0), it does not waste computational time in doing LDSP, it rather directly starts using SDSP. Additionally, if the predicted motion vector is far away from the centre, ARPS again saves on computations compared to DS by directly jumping to that vicinity and using SDSP whereas DS takes its time doing LDSP.

Care has to be taken during the unrestricted SDSP step not to repeat computations at points that were checked earlier. A checkmatrix is utilised: 0 representing locations not yet checked and 1 representing those that have been checked. In addition when the predicted motion vector turns out to match one of the rood pattern locations double computations have also to be avoided e.g. if the Predicted Motion Vector below was (+2, 0).

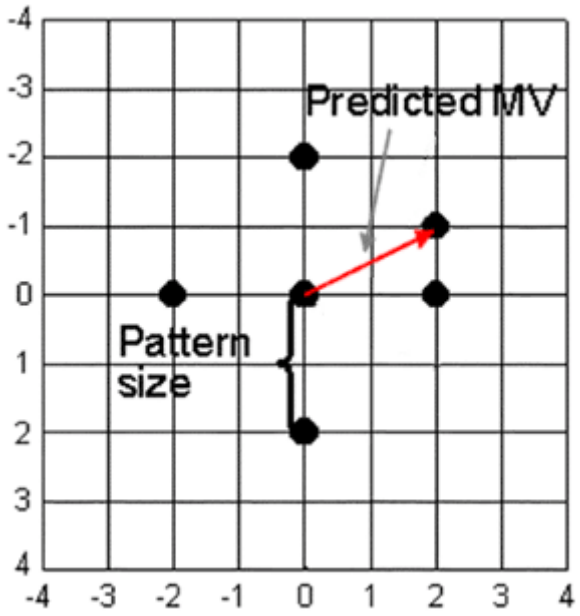


Fig. 2.10 Adaptive Rood Search Pattern.

The initial stage calculates the Predicted Motion Vector as (+2, -1) which directs the search into the local region of the global minimum, the minimum error from a rood pattern of nodes (step size S = Max(|2|, |-1|) = 2) is found and then the final stage performs a fixed search to calculate the motion vector.

Source: Yao Nie and Kai-Kuang Ma (2002).

2.8 Hexagon Based Search Pattern (HEXBS)

Zhu *et al* (2002) proposed the hexagon based search pattern (HEXBS) as an alternative to the diamond search pattern. They noted that the DS was sensitive to motion vectors in different directions – since its eight checking points have different distances from the centre point the advancing speed for the DS per step is 2 pels horizontally and vertically but only $\sqrt{2}$ pels diagonally. They stated that ideally a circle-shaped search pattern with a uniform distribution of a minimum number of search points was more desirable to achieve the fastest search speed uniformly. This search pattern should have a minimum number of search points distributed uniformly where each search point is used equally with maximum efficiency and where the redundancy among search points should be removed maximally. As a result, they devised the hexagon based search pattern (HEXBS) which has a more circle-approximated pattern. The pattern consists of six endpoints with the two horizontal points being 2 pels from the centre and the remaining four points $\sqrt{5}$ pels from the centre - thus the six endpoints are approximately uniformly distributed. Their analysis showed a speed improvement rate of as high as over 80% for locating some motion vectors in certain scenarios. Generally, the larger the motion vector, the more search points the HEXBS algorithm saved compared to DS. This was explained by the HEXBS algorithm only needing to evaluate 3 new checking points for each new search step compared with 3 or 5 in the Diamond Search.

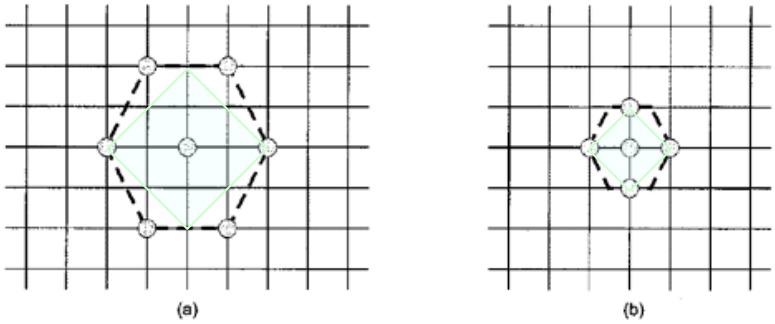


Fig. 2.11 Search Patterns used in the HEXBS algorithm.
(a) large HEXBS pattern with the LDSP overlaid for comparison (b) small HEXBS pattern with the SDSP overlaid for comparison. *Source*: Zhu *et al* (2002).

The HEXBS algorithm (Zhu *et al*, 2002) is summarized as follows:

Step 1: (**Starting**) The large hexagon with seven checking points is centred at (0, 0), the centre of a predefined search window in the motion field. If the minimum BDM point is found to be at the centre of the hexagon, proceed to Step 3; otherwise, proceed to Step 2.
Step 2: (**Searching**) With the minimum BDM point in the previous search step as the centre, a new large hexagon is formed. Three new candidate points are checked, and the minimum BDM point is again identified. If the minimum BDM point is still the centre point of the newly formed hexagon, then go to Step 3; otherwise, repeat this step continuously.
Step 3: (**Ending**) Switch the search pattern from the large to the small size of the hexagon. The four points covered by the small hexagon are evaluated to compare with the current minimum BDM point. The new minimum BDM point is the final solution of the motion vector.

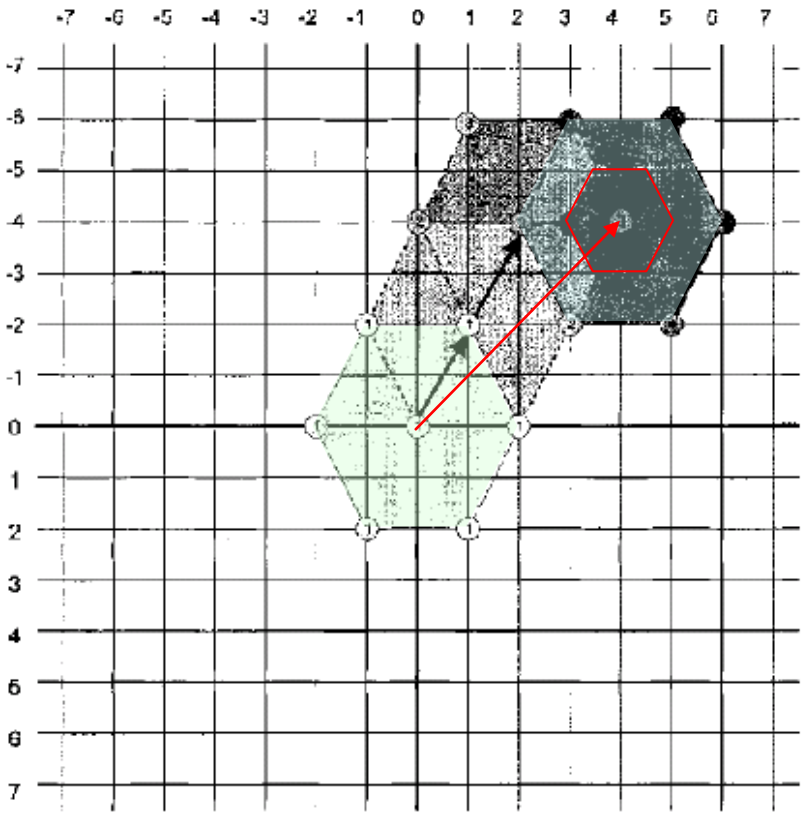


Fig. 2.12 Hexagon Based Search Pattern Procedure.
HEXBS pattern search path example locating the motion vector (+4,-4). Note: a small HEXBS pattern is applied in the final step after the best candidate search point at step 3 remains the best at step 4. In total, 20 (7+3+3+3+4) search points are evaluated in five steps.
Source: Zhu *et al* (2002).

2.9 Cross Diamond Search (CDS)

Cheung and Po (2002b) proposed the CDS algorithm which used a cross-search pattern as the initial step and large/small diamond search (DS) patterns in the subsequent steps. In their analysis of 6 real-world sequences they found that over 80% of the motion vectors were located within a 5x5 search grid. In addition they described the cross-centre-biased (CCB) motion vector distribution - locations where there was a high probability of the motion vector being found. This formed the basis of for their algorithm and their selection of the 9 highly probable candidate points located horizontally and vertically at the centre of a 5x5 search grid. The algorithm employed 2 halfway-stop techniques which meant small motion vectors were found with fewer search points than the DS algorithm while maintaining similar or even better search quality. The first step stop involved a search of only 9 search points compared to 13 for DS (9+4) while the second step stop required only 11 search points compared to a best case search of 16 for DS (9+3+4). Cheung and Po (2002b) reported a speedup of up to 40% over DS in some cases. They also reported that CDS was more robust and provided faster searching speed and smaller distortions than other popular fast block-matching algorithms of the time.

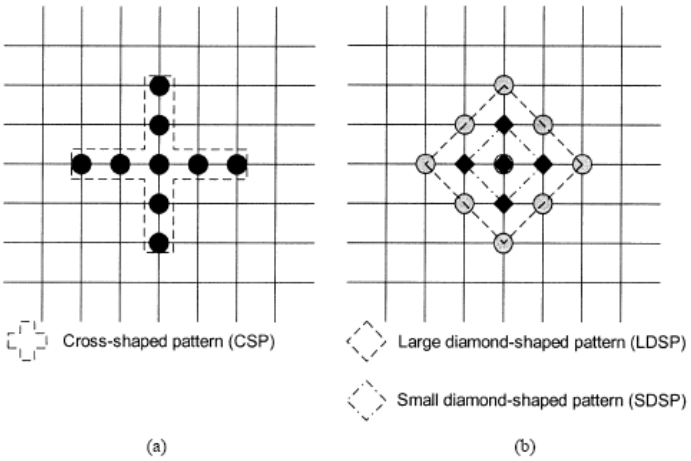


Fig. 2.13 Search patterns used in the Cross Diamond Search algorithm.
(a) CSP (b) LDSP and SDSP. *Source*: Cheung and Po (2002b).

The CDS algorithm Cheung and Po (2002b) is summarized as follows:

- Step 1: (**Starting**) A minimum BDM is found from the nine search points of the CSP located at the centre of the search window. If the minimum BDM point occurs at the centre of the CSP, the search stops. This is called the first-step-stop as shown in Fig. 2.14(a). Otherwise, go to Step 2.
- Step 2: (**Half-diamond Searching**) Two additional search points of the central LDSP closest to the current minimum of the central CSP are checked, i.e. two of the four candidate points located at (±1, ±1). If the minimum BDM found in step 1 is located at the middle wing of the CSP, i.e. (±1, 0) or (0, ±1), and the new minimum BDM found in this step still coincides with this point, then the search stops. This is called the second-step stop, e.g. Fig. 2.14(b). Otherwise, go to Step 3.
- Step 3: (**Searching**) A new LDSP is formed by repositioning the minimum BDM found in the previous step as the centre of the LDSP. If the new minimum BDM point is still at the centre of this newly formed LDSP, then go to Step 4; otherwise, this step is repeated again.
- Step 4: (**Ending**) With the minimum BDM point in the previous step as the centre, a new SDSP is formed. The location of the minimum BDM point found for this step is the motion vector.

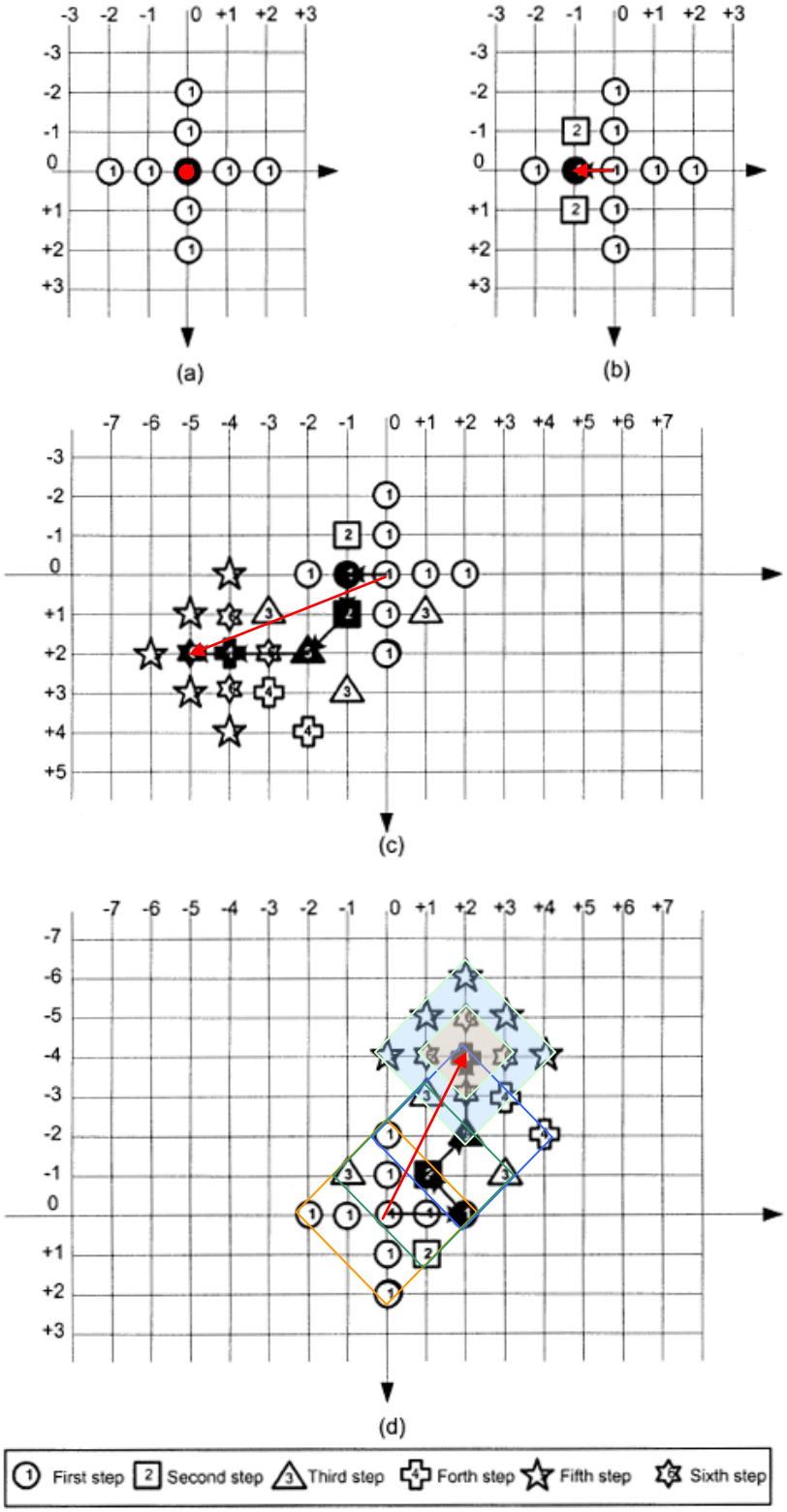


Fig. 2.14 Cross Diamond Block Matching Algorithm procedure.
Examples show each candidate point marked with its corresponding step number. The minimum BDM point at the end of each step is shown filled. (a) First-step-stop with MV(0,0). (b) Second-step-stop with MV(-1, 0). (c) An Unrestricted search path with MV(-5, +2) and (d) MV(+2, -4), respectively. In (d), the best-matched point at step 6 coincides with that at steps 5 and 4. In total, 27 (9+2+4+3+5+4) search points are evaluated. For comparison a DS overlay is shown – this would have found the motion vector by evaluating only 24 (9+3+3+5+4) search points. *Source*: Cheung and Po (2002b).

2.10 Small Cross Diamond Search (SCDS)

Cheung and Po (2002a) introduced the SCDS in the same year that they introduced CDS (Cheung and Po, 2002b). It differed by having a smaller cross pattern in the initial step - using 5 points instead of 9. The algorithm also employed 2 halfway-stop techniques which meant small motion vectors were found using fewer search points than with the DS algorithm. The first step stop involved a search of only 5 search points compared to 13 for DS (9+4) while the third step stop required only 11 (5+4+2) search points compared to a best case search of 16 for DS (9+3+4). An unrestricted large diamond search (DS) pattern was employed in the subsequent steps followed by a final small diamond search. Cheung and Po (2002a) reported a speedup of up to 146% over DS for the *Akiyo* QCIF video conference sequence.

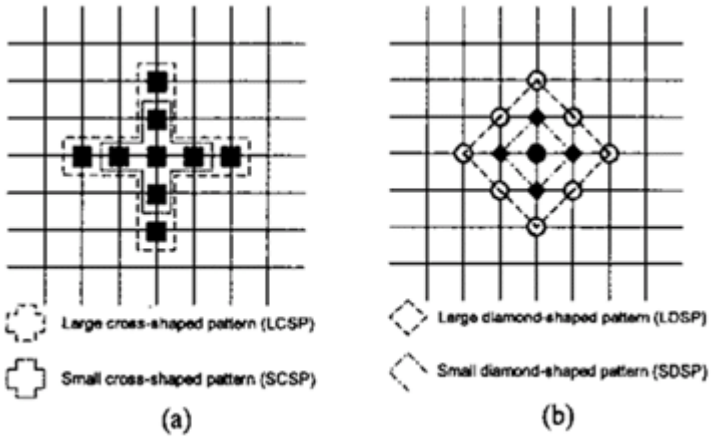


Fig. 2.15 Search patterns used in the Small Cross Diamond Search algorithm.
(a) LCSP and SCSP (b) LDSP and SDSP. *Source*: Cheung and Po (2002a).

The SCDS algorithm Cheung and Po (2002a) is summarized as follows:

- Step 1: (**Starting**) A minimum BDM is found from the five search points of the SCSP located at the centre of the search window. If the minimum BDM point occurs at the centre of the SCSP, the search stops. This is called the first-step-stop as shown in Fig. 2.16(a). Otherwise, go to Step 2.
- Step 2: (**Large Cross Searching**) The four outermost search points of the central LCSP are checked, i.e. the four candidate points located at (0, ±2) and (±2, 0). This step guides the possible correct direction for the subsequent steps. Then go to Step 3.
- Step 3: (**Half-Diamond Searching**) Two additional search points of the central LDSP closest to the current minimum of the central LCSP are checked, i.e. two of the four candidate points located at (±1, ±1). If the minimum BDM found in step 1 is located at the middle wing of the CSP, i.e. (±1, 0) or (0, ±1), and the new minimum BDM found in this step still coincides with this point, then the search stops. This is called the third-step stop, e.g. Fig. 2.16(b). Otherwise, go to Step 4.
- Step 4: (**Searching**) A new LDSP is formed by repositioning the minimum BDM found in the previous step as the centre of the LDSP. If the new minimum BDM point is still at the centre of this newly formed LDSP, then go to Step 5; otherwise, this step is repeated again.
- Step 5: (**Ending**) With the minimum BDM point in the previous step as the centre, a new SDSP is formed. The location of the minimum BDM point found for this step is the motion vector.

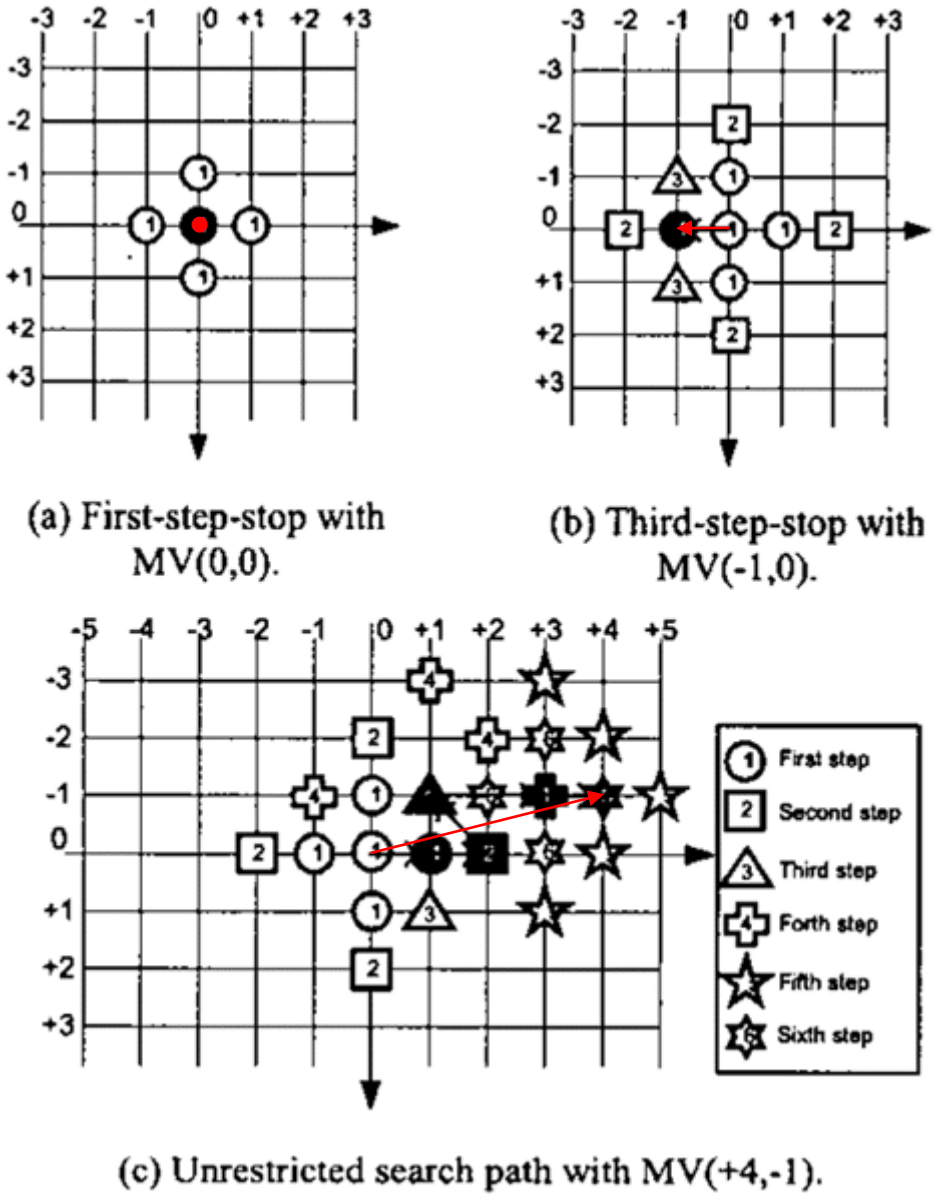


Fig. 2.16 Small Cross Diamond Block Matching Algorithm procedure.
Examples show each candidate point marked with its corresponding step number. The minimum BDM point at the end of each step is shown filled.
(a) First-step-stop with MV(0, 0). (b) Third-step-stop with MV(-1, 0). (c) An Unrestricted search path with MV(+4, -1). In (c), the best-matched point at step 6 coincides with that at steps 5 and 4. *Source*: Cheung and Po (2002a).

2.11 New Cross Diamond Search (NCDS)

Lam *et al* (2003) introduced the NCDS which like SCDS the previous year (Cheung and Po, 2002a) used a 5 point small cross shape pattern (SCSP) as the initial step. The 5 point pattern is repeated in the second step if needed which makes the algorithm more efficient than both the previous SCDS or CDS and thus making a saving on the number of search points for stationary or quasi-stationary blocks. The algorithm also employed 2 halfway-stop techniques which meant small motion vectors were found with fewer search points than the DS algorithm. The first step stop involved a search of only 5 search points compared to 13 for DS (9+4) while the second step stop required only 8 (5+3) search points compared to a best case search of 16 for DS (9+3+4). An unrestricted large diamond search (DS) pattern was employed in the subsequent steps followed by a final small diamond search. Lam *et al* (2003) reported a speedup of up to 58% over DS for the *Claire* CIF video conference sequence.

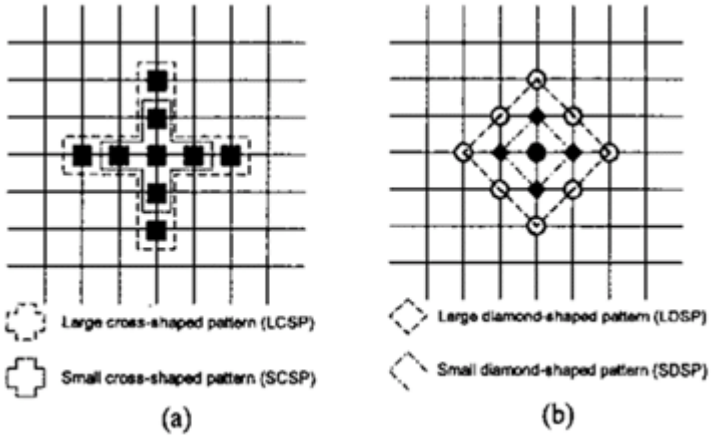


Fig. 2.17 Search patterns used in the New Cross Diamond Search algorithm. (a) LCSP and SCSP (b) LDSP and SDSP. *Source*: Cheung and Po (2002a).

The NCDS algorithm Lam *et al* (2003) is summarized as follows:

- Step 1: (**Starting**) – **Small Cross Shape Pattern (SCSP)** A minimum BDM is found from the five search points of the SCSP located at the centre of the search window. If the minimum BDM point occurs at the centre of the SCSP, the search stops. This is called the first-step-stop as shown in Fig. 2.18(a). Otherwise, go to Step 2.
- Step 2: (**SCSP**) With the vertex (minimum BDM point) from the first SCSP as the centre, a new SCSP is formed. If the minimum BDM point occurs at the centre of this SCSP, the search stops. This is called the second-step-stop as shown in Fig. 2.18(b). Otherwise, go to Step 3.
- Step 3: *Guiding* **Large Cross Shape Pattern** (**LCSP**) The three unchecked outermost search points of the central LCSP are checked. This step is trying to guide the possible correct direction for the subsequent steps. Go to Step 4.
- Step 4: (**Searching**) A new LDSP is formed by repositioning the minimum BDM found in the previous step as the centre of the LDSP. If the new minimum BDM point is still at the centre of this newly formed LDSP, then go to Step 5; otherwise, this step is repeated again.
- Step 5: (**Ending**) With the minimum BDM point in the previous step as the centre, a new SDSP is formed. The location of the minimum BDM point found for this step is the motion vector.

Note: Around this time another New Cross Diamond Search algorithm was also developed by Jia and Zhang (2004).

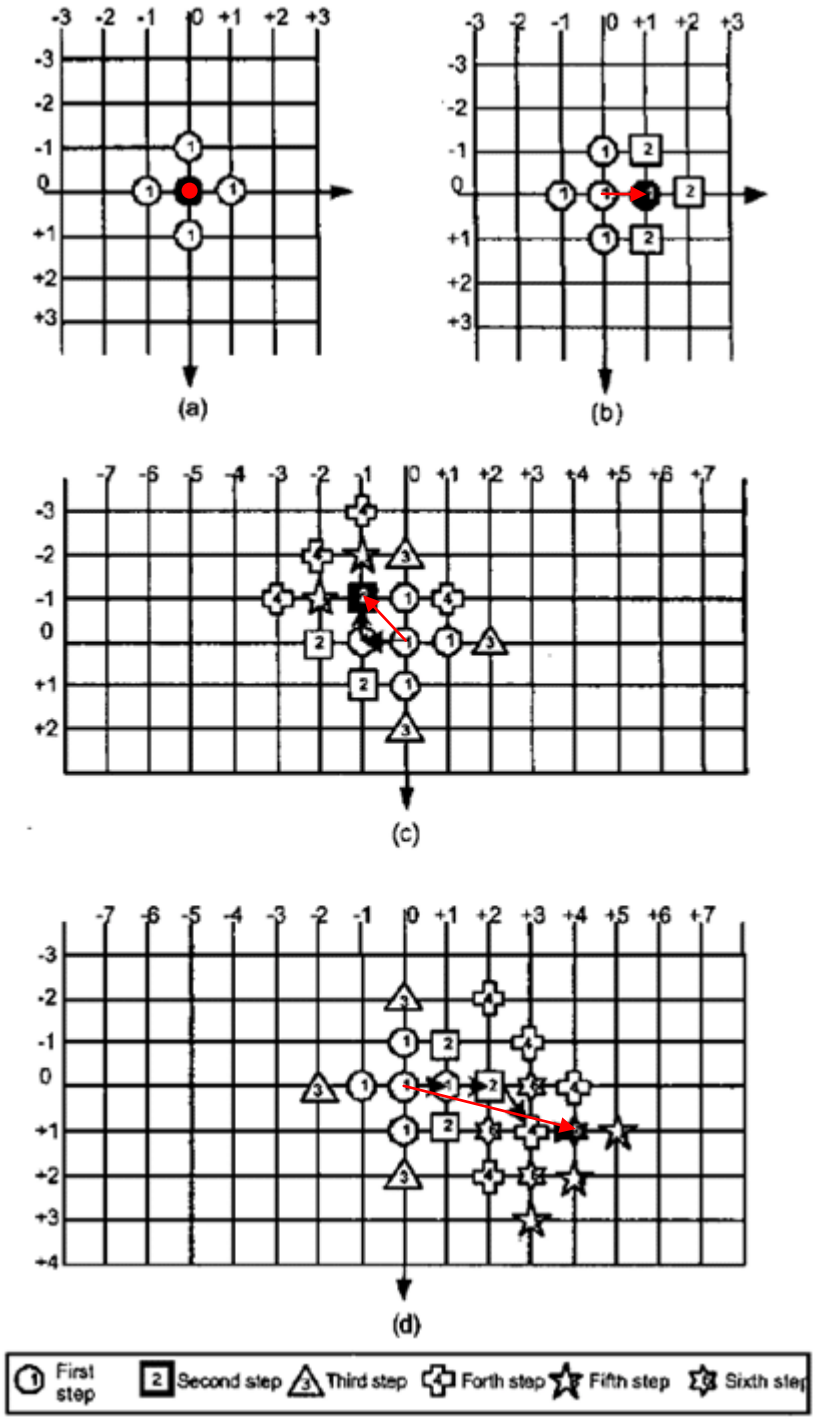


Fig. 2.18 New Cross Diamond Block Matching Algorithm procedure.
Examples show each candidate point marked with its corresponding step number. The minimum BDM point at the final step is shown filled.
(a) First-step-stop with MV(0,0). (b) Second-step-stop with MV(+1, 0). (c) and (d) An Unrestricted search path with MV(-1, -1) and MV(+4, +1) respectively. In (c), the best-matched point at step 5 coincides with that at steps 4 and 2. *Source*: Lam *et al* (2003).

3.0 Implementation of Algorithms

3.1 Video Sequences used for Analysis

Motion vectors are typically estimated from the luma component only (Richardson, 2002). The frames to be input into the various algorithms are stored in Sun Rasterfile format (*.ras*) which is an uncompressed greyscale format. The standard video sequences used for algorithm analysis are saved as CIF (Common Intermediate Format) or QCIF (Quarter CIF format) which in some sources are stored in the *.yuv* file format. These were then converted to usable images which are used as input by the various block search algorithms. In the conversion the C_b and C_r components are suppressed while the Y (luma) component is retained. The MATLAB code used is given in Appendix D.

A variety of sequences were chosen as described by Tham *et al* (1998):

1. The first sequence "*Miss America*" is a typical videoconferencing scene with limited object motion and a stationary background.
2. The second sequence "*Flower Garden*" consists mainly of stationary objects, but with a fast camera panning motion.
3. The third sequence "*Football*" contains large local object motion.

Using the study of Barjatya (2004) as a basis, motion vectors will be predicted for 30 frames using a distance of 2 between the current frame and the reference frame. Thus only the first 32 frames of each sequence need to be examined.

Table 3.1: Video Sequences used for Analysis

Frame Format (Frame Size, Number of Frames)	Sequences
CIF (352 x 288, 32 frames)	Flower Garden
SIF (352 x 240, 32 frames)	Football
QCIF (176 x 144, 32 frames)	Miss America ǂ

Sources: CIPR, (2008) and VTRG ǂ, (2008).

Miss America ǂ

Flower Garden

Football

Fig. 3.1 Stills of the Video Sequences Analysed.
Sources: CIPR, (2008) and VTRG ǂ, (2008).

3.2 Algorithms to be Implemented

The Diamond Search algorithm (DS) proved to be the best block matching algorithm for many years after it was introduced in 1998 (Barjatya, 2004). Towards the end of 2002 some hybrid DS algorithms began to appear. Cheung and Po (2002b) introduced Cross Diamond Search (CDS) and Small Cross Diamond Search (SCDS) (2002a) and Lam *et al* (2003) introduced New Cross Diamond Search (NCDS). All improved on the performance of Diamond Search (DS) by modifying the starting search pattern from Large Diamond Search Pattern (LDSP) to the Cross Search Pattern (CSP) originated by Ghanbari (1990). The three algorithms differed with respect to the number of points being used out of the CSP as shown in **Fig. 3.2**. CDS uses all 9 points whereas SCDS and NCDS use only the inner 5 points to start and then expand their search. In addition they improved on DS by providing half-way stops for stationary or quasi-stationary sequences, thus helping to reduce the number of points searched. After applying the initial cross search pattern these CSP based variants follow the normal DS procedure - that of an unrestricted LDSP followed by a final Small Diamond Search Pattern (SDSP).

In his 2004 study, Barjatya referenced these three CSP algorithms but did not provide an implementation. He stated that they improved on DS and that of the three, NCDS came closest to the performance of ARPS - which was the best performing of the 7 algorithms studied.

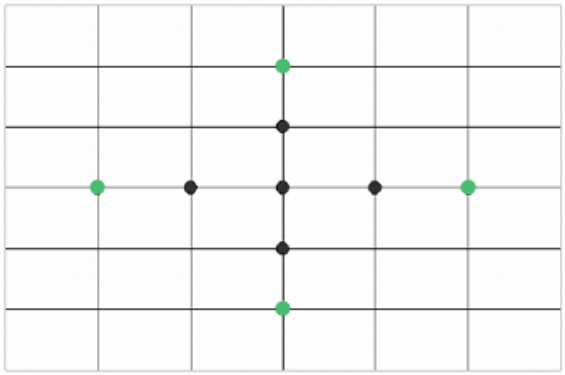


Fig. 3.2 The Cross Search Pattern used by CDS, SCDS, and NCDS.
CDS uses all 9 points, SCDS and NCDS use only the inner 5 points.
Source: Barjatya, (2004).

3.3 Thesis Aims

The aims of this thesis are to:

1. provide a detailed description of the many block search algorithms available today.
2. code an implementation in MATLAB for the 3 hybrid DS algorithms.
3. validate the results obtained from the implementation against results from the literature.
4. quantify the performance of the 3 algorithms against the Diamond Search algorithm.
5. quantify the performance of the 3 algorithms against the ARPS algorithm and
6. make a recommendation of 'best-practice' when nominating a Block Search Algorithm.

3.4 Coding the Algorithms

As an initial stage in coding all 3 algorithms their flowcharts are drawn and presented below. Their MATLAB implementation is presented in Appendix C.

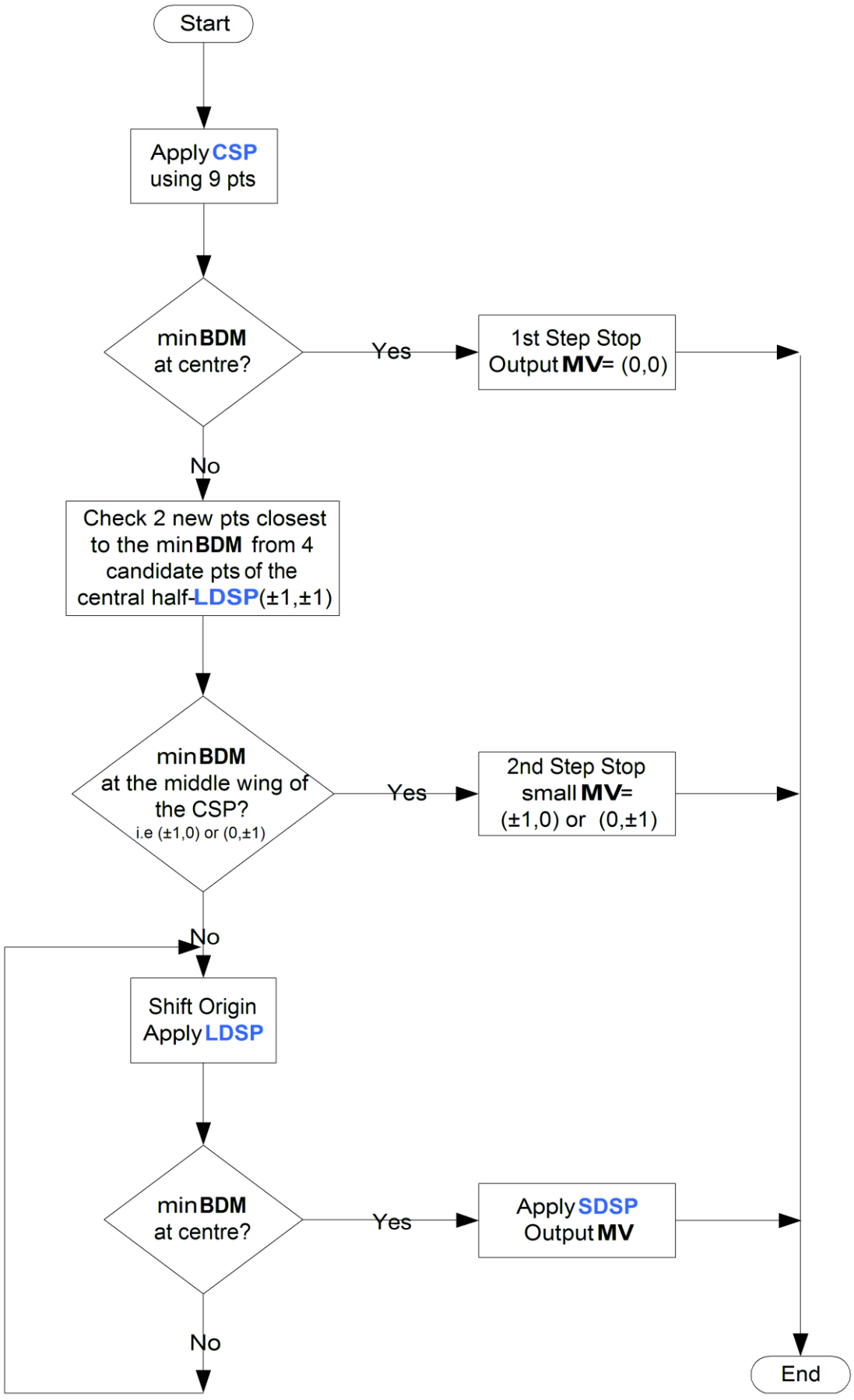


Fig. 3.3 Flowchart for the Cross Diamond Search Algorithm.

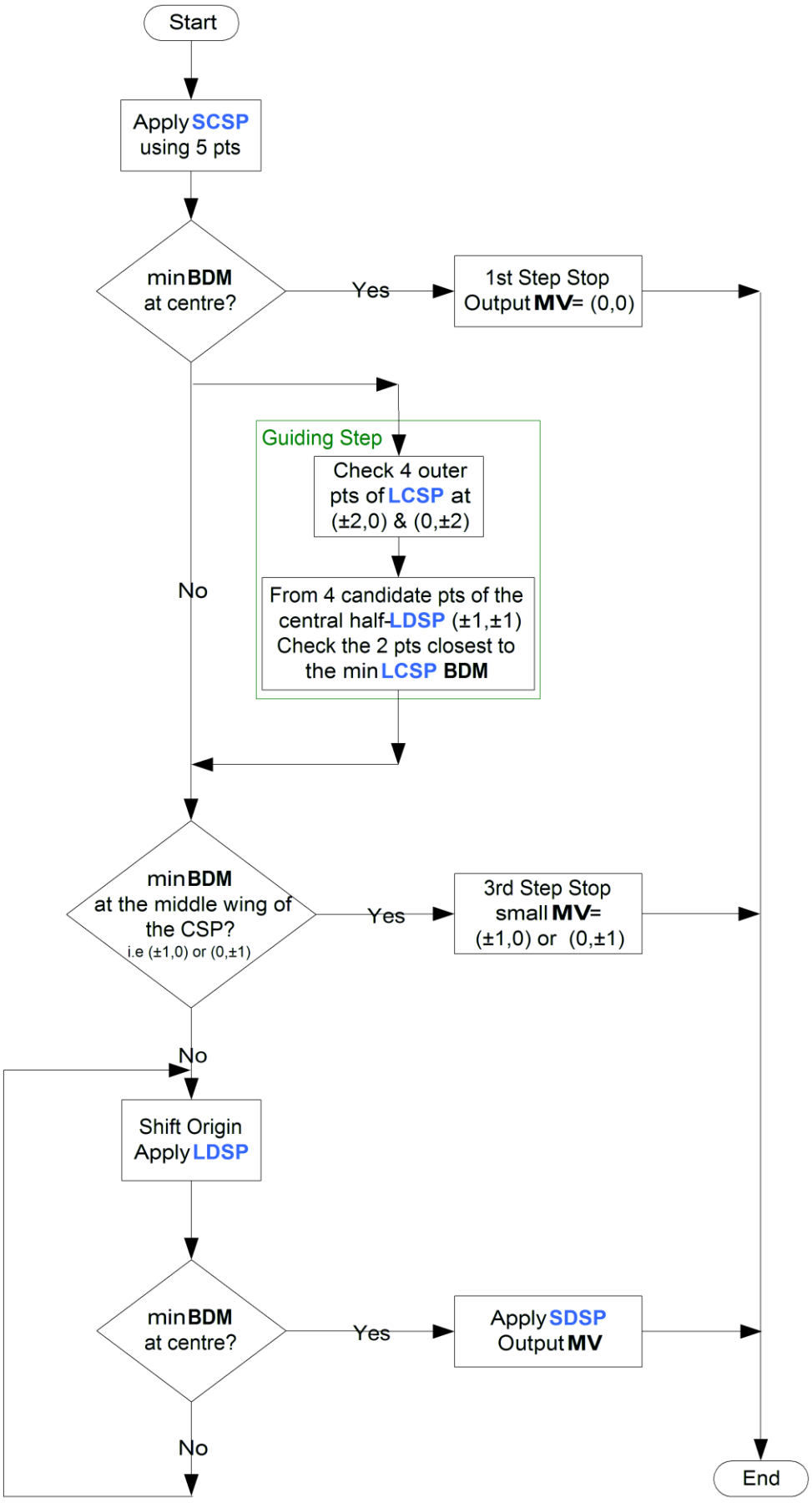


Fig. 3.4 Flowchart for the Small Cross Diamond Search Algorithm.

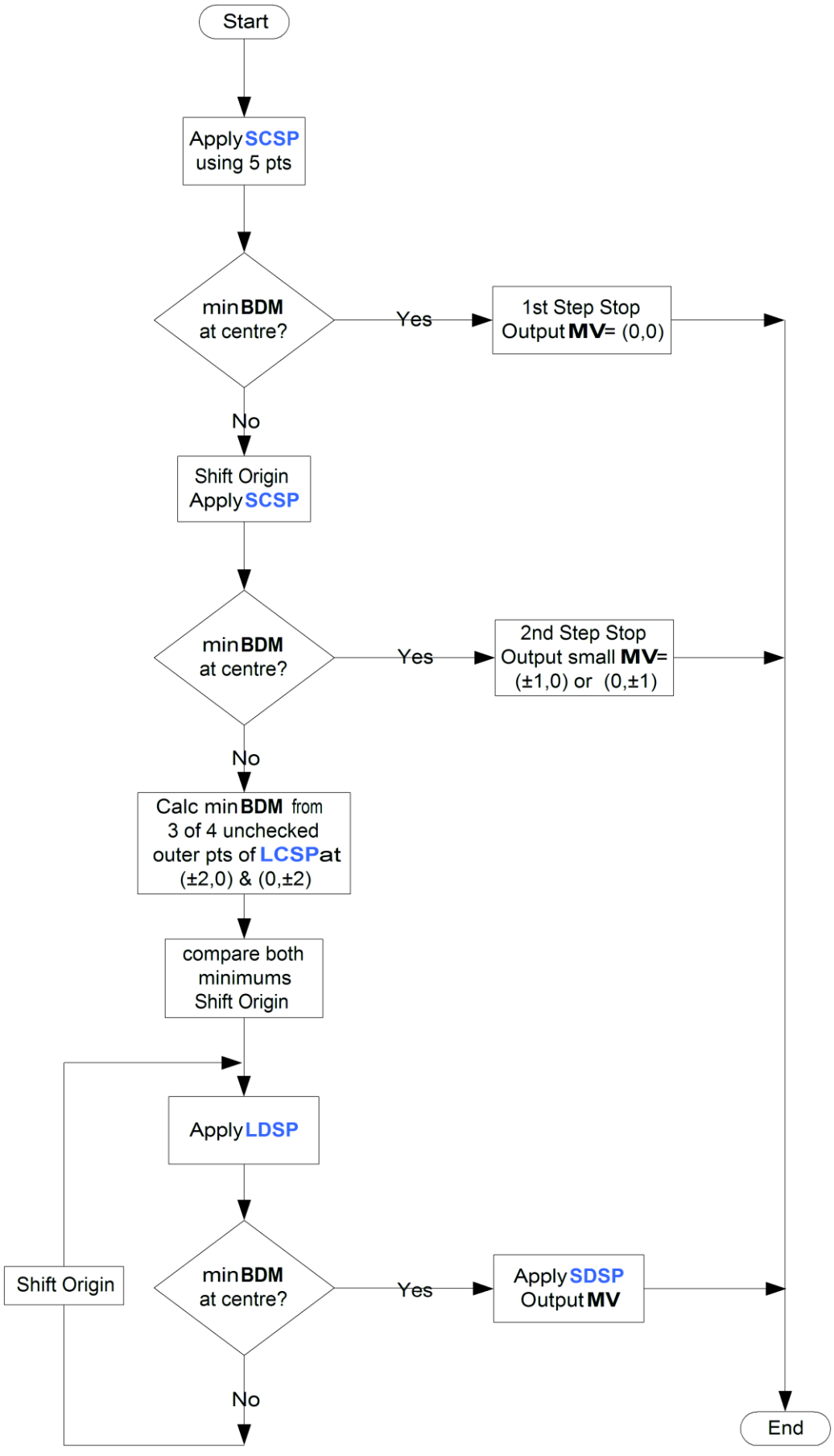


Fig. 3.5 Flowchart for the New Cross Diamond Search Algorithm.

3.5 Program Execution

motionsEstAnalysis.m – the main script to execute all algorithms is run in the MATLAB command window. Initially 2 frames of a particular video sequence are loaded into the workspace – the first is the reference frame and the second is the frame to be predicted (encoded). The first block match algorithm is called. The block distortion measure (BDM) used is the mean absolute difference (MAD). The macroblock size is set at 16 pixels x 16 pixels and the maximum displacement in the search area is ±7 pixels in both the horizontal and the vertical directions. A frame difference of 2 was used in calculating predicted frames.

The algorithm function called returns the motion vector matrix for the predicted frame – one motion vector for every macroblock in the frame.
The average number of points searched to calculate each motion vector within the predicted frame is also returned.

The motion vector matrix is then input into the **motionComp.m** function which creates the motion compensated image from each motion vector and its corresponding macroblock in the reference frame.

The PSNR of the motion compensated image with respect to the original frame is then calculated and recorded by calling the **imgPSNR.m** function – one value for each predicted frame.

The next algorithm is then called and the process repeats for a complete analysis of 10 algorithms. The process then loads the next frame to be encoded along with its reference frame, runs all 10 algorithms again, and then loops until 30 frames in total are predicted.

The main script also contains code to produce 2 comparative plots of the main metrics for all 10 algorithms – *Search points per macroblock Vs Frame Number* and *PSNR Vs Frame Number*. The code saves these to disk in jpeg format. In addition the main script outputs some statistics for each algorithm such as *Average Searching Points*, *Average PSNR* and *Speed Improvement Ratio*. The latter was used for the 3 implemented algorithms to quantify their performance compared to the Diamond Search and the Exhaustive Search.

For the analysis of another video sequence the main script is updated to point to its location and is then run again.

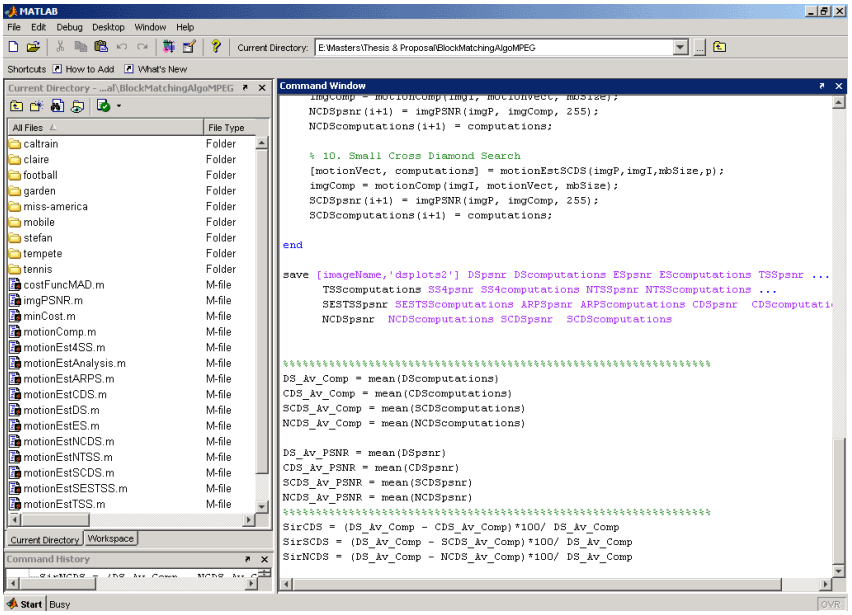


Fig. 3.6 MATLAB environment showing the main script running for the *Football* sequence.

4.0 Analysis of the Implemented Block Search Algorithms

4.1 Experimental Results

The simulation is performed on 3 sequences with different degrees and types of motion-content. The CDS, SCDS and NCDS were compared against 7 other algorithms using the following test criteria:

1) Average points searched – the average number of search points used to find the motion vector as shown in Table 4.1
2) the average Peak-Signal-to-Noise-Ratio (PSNR) as shown in Table 4.2
3) the average Speed Improvement Ratio (SIR) with respect to the DS and the ES as shown in Tables 4.3 and 4.4
4) the difference in average PSNR compared to the DS as shown in Table 4.5

Table 4.1: Average Points Searched for selected Fast BMAs over 3 sequences

Sequence	ES	TSS	NTSS	4SS	SES	DS	CDS	SCDS	NCDS	ARPS
flower CIF (352x288)	210.317	23.813	24.570	19.989	15.930	18.303	18.413	17.176	16.530	9.562
football SIF (352x240)	202.049	23.095	22.799	19.355	15.842	18.527	17.385	16.071	15.401	11.014
Ms America QCIF (176x144)	195.963	22.503	16.241	15.594	16.575	12.427	9.260	6.380	5.778	5.645

Table 4.2: Average PSNR (dB) for selected Fast BMAs over 3 sequences

Sequence	ES	TSS	NTSS	4SS	SES	DS	CDS	SCDS	NCDS	ARPS
flower	24.373	23.819	24.145	23.379	23.561	23.329	23.130	22.885	22.649	24.295
football	20.307	20.128	20.101	19.949	19.630	19.886	19.830	19.778	19.757	19.898
Ms America	39.378	39.354	39.377	39.360	39.153	39.375	39.324	39.317	39.323	39.332

Table 4.3: Average Speed Improvement Ratio (%) over ES for 3 sequences

Sequence	CDS	SCDS	NCDS	ARPS
flower	91.245	91.834	92.140	95.454
football	91.396	92.046	92.378	94.549
Ms America	95.275	96.744	97.052	97.119

Table 4.4: Average Speed Improvement Ratio (%) over DS for 3 sequences

Sequence	CDS	SCDS	NCDS	ARPS
flower	-0.601	6.158	9.682	47.757
football	6.165	13.256	16.874	40.549
Ms America	25.487	48.663	53.508	54.574

Table 4.5: Difference in Average PSNR (dB) over DS for 3 sequences

Sequence	CDS	SCDS	NCDS	ARPS
flower	0.199	0.444	0.680	-0.966‡
football	0.0558	0.1086	0.1294	-0.0119 ‡
Ms America	0.0514	0.0581	0.0516	0.0433

‡ a negative value indicates a PSNR greater than the DS value was achieved.

Initial observations of Table 4.1 show that in many cases the actual number of search points is lower than the theoretical estimation e.g. 225 for Exhaustive Search theoretically versus ~ 200 experimentally. This is due to truncation of the search window at picture boundaries and truncation of searching patterns at window boundaries which end up saving many search points practically (Cheung and Po, 2002b).

Fig. 4.1 and Fig. 4.2 show a frame-by-frame comparison of search point number per block and PSNR respectively for the different algorithms applied to the *Football* sequence. Fig. 4.1 shows a curve that fluctuates quite intensely for all 3 hybrid DS algorithms representing the high motion content of the sequence. There appear to be spikes of more intense motion around frames 7, 17 and 24 representing a transition from small to large motion and then back to small motion. It is noted that the number of search points fluctuate much more sharply for the 3 hybrid DS algorithms and the DS than for the ARPS algorithm.

Fig. 4.3 and Fig. 4.4 plot a frame-by-frame comparison of search point number per block and PSNR respectively for the different algorithms applied to the *Flower Garden* sequence. Fig. 4.3 shows that the average number of search points per macroblock with NCDS < SCDS < CDS. There is a deviation from the expected improvement of CDS over DS – the CDS in fact takes more search points than the DS for most of the frames predicted – this may be due to a number of factors which are discussed below. Fig. 4.4 also demonstrates that this sequence displays the largest degradation of video quality for any of the algorithms compared to the Exhaustive search.

Fig. 4.5 and Fig. 4.6 plot a frame-by-frame comparison of search point number per block and PSNR respectively for the different algorithms applied to the *Miss America* sequence. Fig. 4.5 shows that the average number of search points per macroblock with NCDS < SCDS < CDS < DS. Also this is the only sequence of the three examined where NCDS comes close to matching ARPS for performance. Fig. 4.6 also demonstrates that there is almost no degradation of video quality for any of the algorithms compared to the Exhaustive search.

The Exhaustive Search is not graphed since it has the largest number of search points requiring ~200 searches per macroblock for each sequence. Although PSNR performance of 4SS, DS, and ARPS is relatively the same, ARPS takes a factor of 2 less computations in some sequences and hence is the best of the fast block matching algorithms studied.

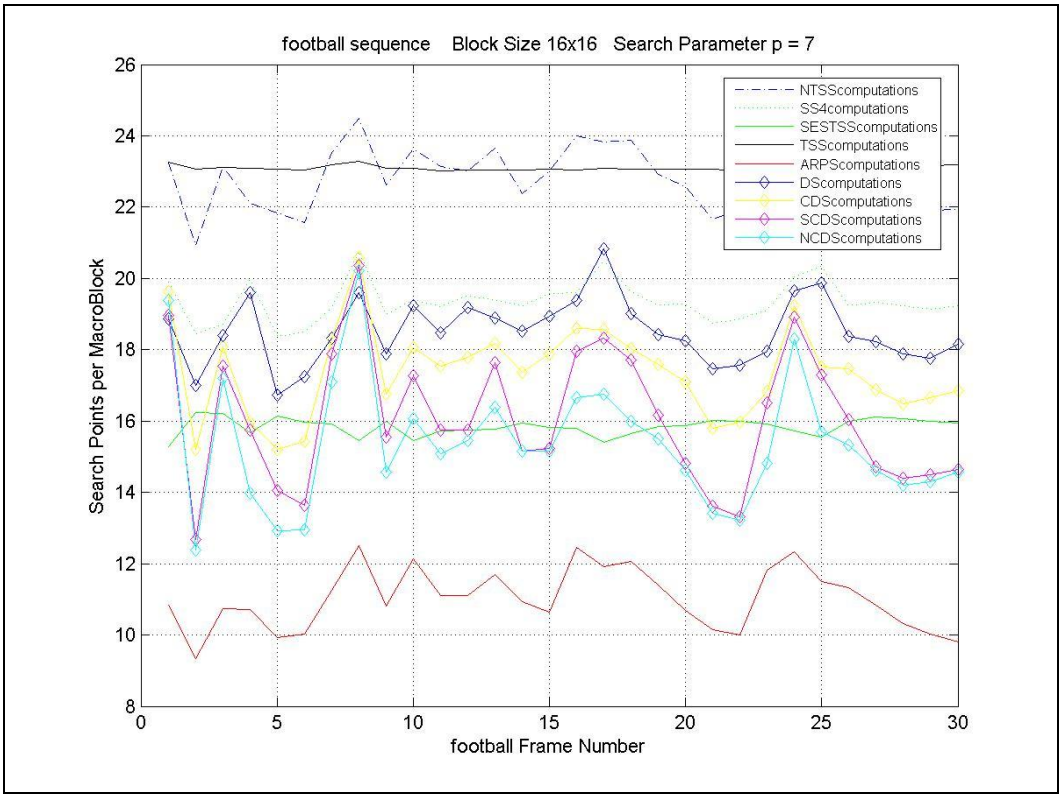


Fig. 4.1 Search points per macroblock for selected Fast BMAs applied to *Football*.

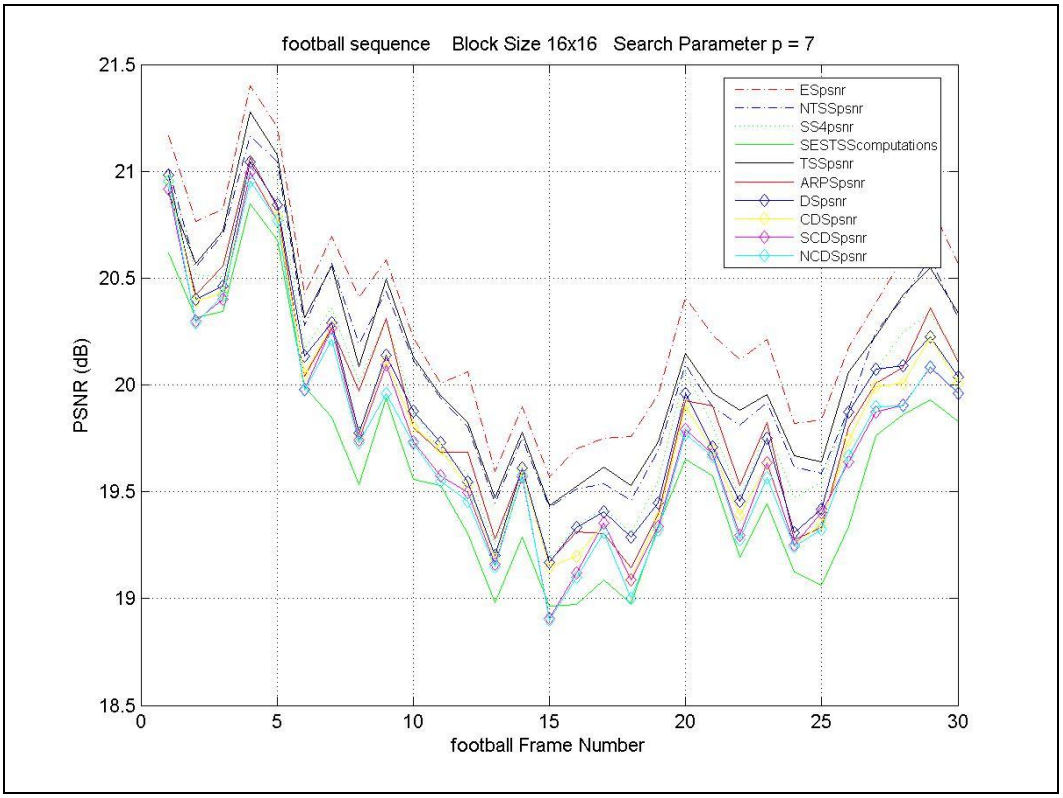


Fig. 4.2 PSNR performance for selected Fast BMAs applied to *Football.*

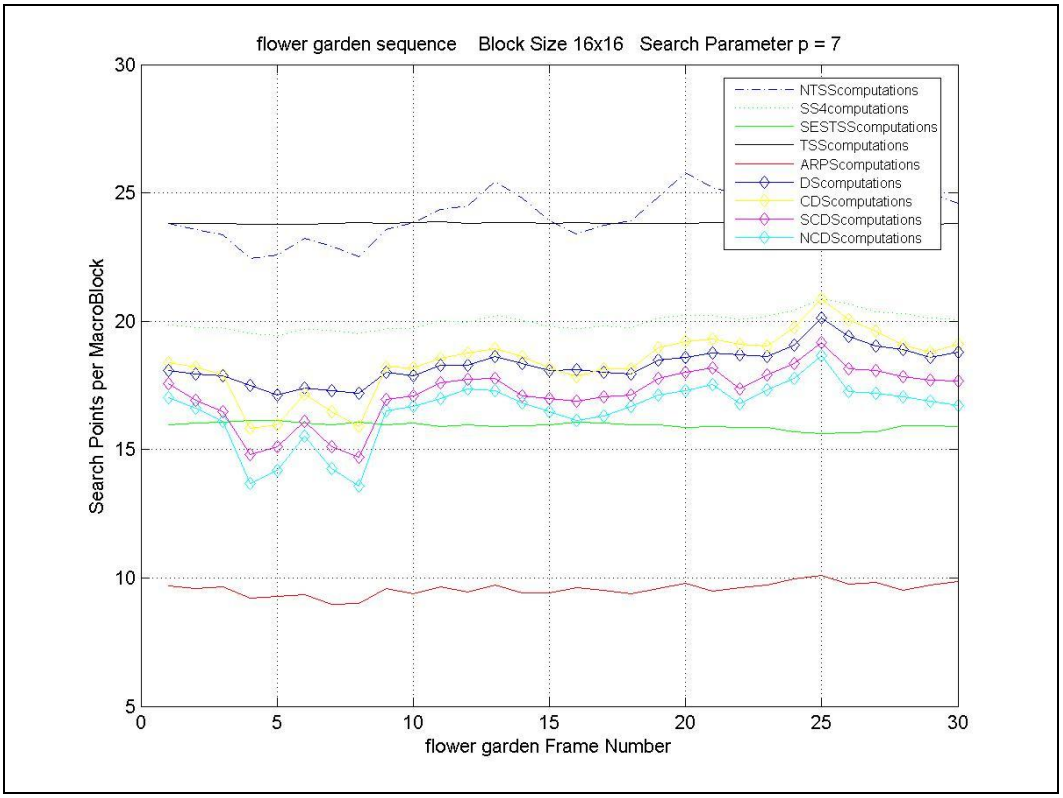


Fig. 4.3 Search points per macroblock for selected Fast BMAs applied to *Flower garden*.

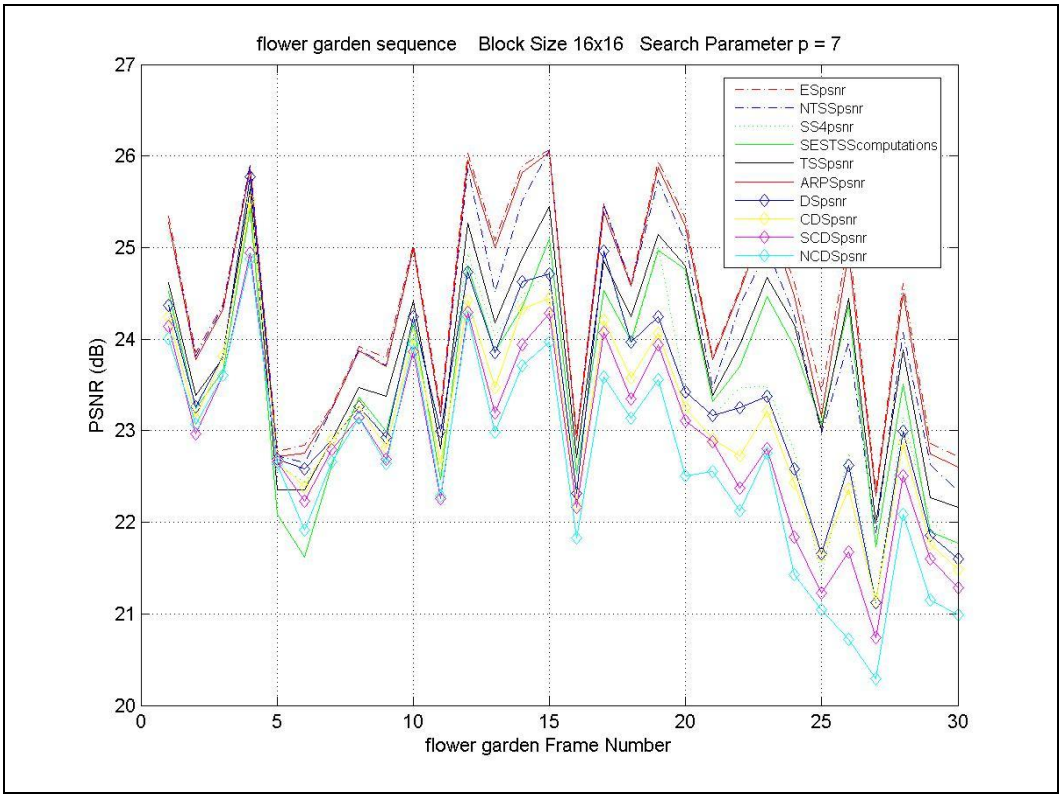


Fig. 4.4 PSNR performance for selected Fast BMAs applied to *Flower garden.*

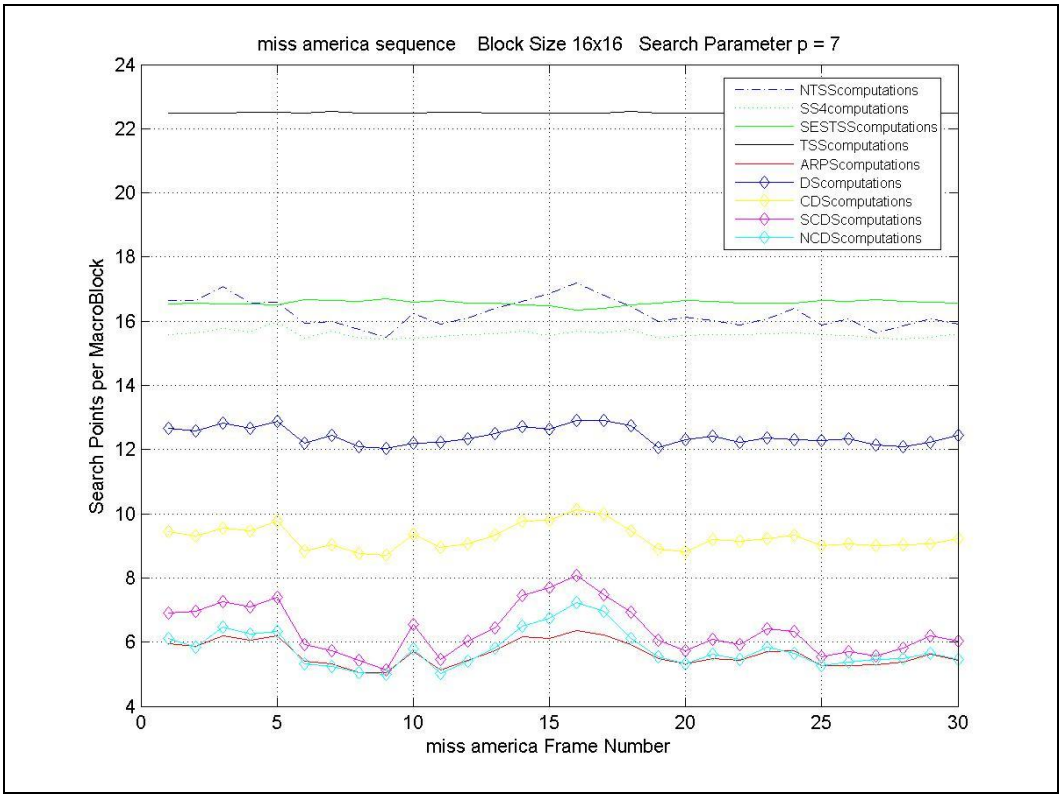


Fig. 4.5 Search points per macroblock for selected Fast BMAs applied to *Miss America*.

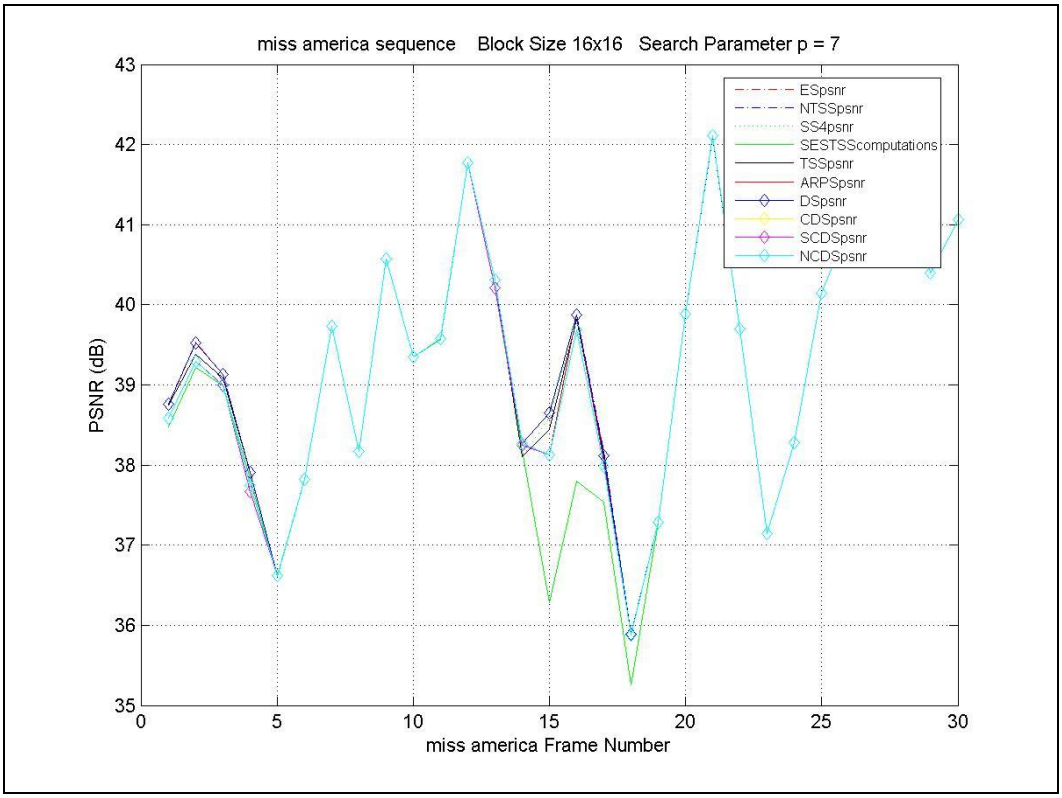


Fig. 4.6 PSNR performance for selected Fast BMAs applied to *Miss America*.

4.2 Validation of the Implementation

Jia and Zhang (2004) tested the *Miss America* sequence (CIF, 150 frames) and the *Football* sequence (SIF, 125 frames) for various algorithms including the ES, NTSS, DS and the CDS. They did not test the *Flower Garden* sequence.

For the *Miss America* sequence they reported 17.314 search points for the DS algorithm and 12.419 for the CDS while for the *Football* sequence the corresponding figures were 17.376 search points for the DS algorithm and 15.634 for the CDS. These results agree closely with our implementation for the *Football* sequence of 18.527 search points for the DS algorithm and 17.385 for the CDS – however they did use more frames at 125.
Their results vary slightly from our implementation for the *Miss America* sequence of 12.427 search points for the DS algorithm and 9.260 for the CDS – however they did use the larger CIF resolution so a larger number of search points would be expected.

Their PSNR for the DS and the CDS were 37.097dB and 37.305dB respectively for the *Miss America* sequence, while the *Football* sequence obtained a PSNR of 21.892dB and 21.803dB respectively. These values compare very favourably with our results.

Lam *et al* (2003) – who introduced the NCDS – tested the DS, CDS, SCDS and the NCDS for each of the 3 sequences we employed: the *Miss America* sequence (CIF, 80 frames), the *Flower Garden* sequence (SIF, 80 frames) and the *Football* sequence (SIF, 80 frames).

For the *Miss America* sequence Lam *et al* (2003) reported 16.36 search points for the DS algorithm, 11.75 for the CDS, 10.75 for the SCDS and 8.7745 for the NCDS.

As stated previously a smaller resolution for the *Miss America* sequence was used resulting in fewer search points for each algorithm – the large drop in search points from the DS to the CDS is also reproduced in our results.

For the *Flower Garden* sequence the corresponding figures were 16.84 search points for the DS algorithm, 15.09 for the CDS, 14.87 for the SCDS and 13.4562 for the NCDS.

We also used the larger CIF resolution for this sequence so our number of search points would be expected to be higher. The rate of decrease in the number of search points is lower than for the slower *Miss America* sequence and this is reproduced in our results as in Lam *et al* (2003). The 3 hybrid algorithms work best for low motion video conferencing sequences – indeed the values for the CDS that we obtained are actually higher than DS. This may be due to the complexity of the frames selected.

The sequence of frames we used contains both rotational motion (a windmill) as well as the translational motion to do with the panning from left to right. The difficulty in making a comparison here with the findings of Lam *et al* (2003) is that only a portion of the available 115 frames (CIPR, 2008) are being used. Lam *et al* (2003) used 80 frames but the trend in their results may point to the fact that they used frames with translational motion only. Our 30 frame sequence has both, making it more complex and requiring more search points than translational motion alone.

For the *Football* sequence Lam *et al* (2003) reported 13.67 search points for the DS algorithm, 10.96 for the CDS, 8.24 for the SCDS and 7.9022 for the NCDS.

These values are a great deal lower than in our implementation – however Lam *et al* (2003) did use 80 frames. Again the difficulty in making a comparison here with the findings of Lam *et al* (2003) is that only a portion of the available 125 frames (CIPR, 2008) are being used.

Lam *et al* (2003) used the MSE to ascertain the effect on video quality instead of the PSNR measure and so no comparison could be made for distortion.

In conclusion, considering the various resolutions used, the number of frames used and variation in the complexity of the motion depending on which frames of a sequence were selected to be tested, this implementation effectively reproduces the findings of Jia and Zhang (2004) and Lam *et al* (2003).

4.3 Performance of the 3 Hybrid DS Algorithms versus DS

Table 4.4 shows that the CDS is nearly 26% faster than the DS for the *Miss America* sequence, slightly slower than the DS for the *Flower Garden* sequence (though this deviation is explained below) and 6% faster than the DS for the *Football* sequence.

Both the SCDS and the NCDS improve further on the DS. Table 4.4 shows that the SCDS is nearly 49% faster than the DS for the *Miss America* sequence, the NCDS is almost 54% faster; the SCDS is 6% faster than the DS for the *Flower Garden* sequence, the NCDS is nearly 10% faster; the SCDS is 13% faster than the DS for the *Football* sequence, the NCDS is nearly 17% faster.

For the *Miss America* sequence with motion vectors limited within a small region around (0, 0), the 3 hybrid DS algorithms achieve a considerable speed improvement over the DS. They reduce computations significantly over the DS particularly for low bit-rate video applications with 1) gentle or no motion, such as background information and 2) small motion. Both types of motion estimation are accomplished by the first and second-step stop respectively.

For the *Flower Garden* sequence with medium motion, there is in fact a lower average SIR for the CDS over the DS as shown in Table 4.4. Figure 4.7 shows an overlay of motion vectors for the predicted picture. Some of the true motion vectors as calculated by the Exhaustive Search (Zhu *et al*, 2002) can be as large as (-5, 0) to (-7, 0) which are at the limits of the search window. As mentioned above the sequence of frames we used contains both rotational motion (a windmill) as well as the translational motion to do with the panning from left to right. This makes the sequence more complex and would require more search points than for translational motion alone. This could explain why there are a larger number of points searched for the CDS than for the DS and hence why there is a decrease in the SIR for the CDS compared to the DS. Cheung and Po (2002b) state that although both first-step-stop and second-step-stop halfway techniques employed in the CDS algorithm can optimize the highly probable CCB characteristics, the DS algorithm does in fact seem to be more efficient beyond the central 3x3 cross-shaped region. Figure 4.8 shows that when a motion vector occurs outside the central 3x3 cross-shaped region the CDS algorithm actually uses more search points. This suggests strongly that the CDS is only advantageous for slow moving videoconferencing sequences.

Fig. 4.7 The *Flower Garden* sequence with its Motion Vectors overlaid.
The *Flower Garden* sequence involves a pan from left to right, a motion vector points to the best match macroblock in the reference frame, hence the predominance of motion vectors pointing to the left. *Source*: Girod, (2008).

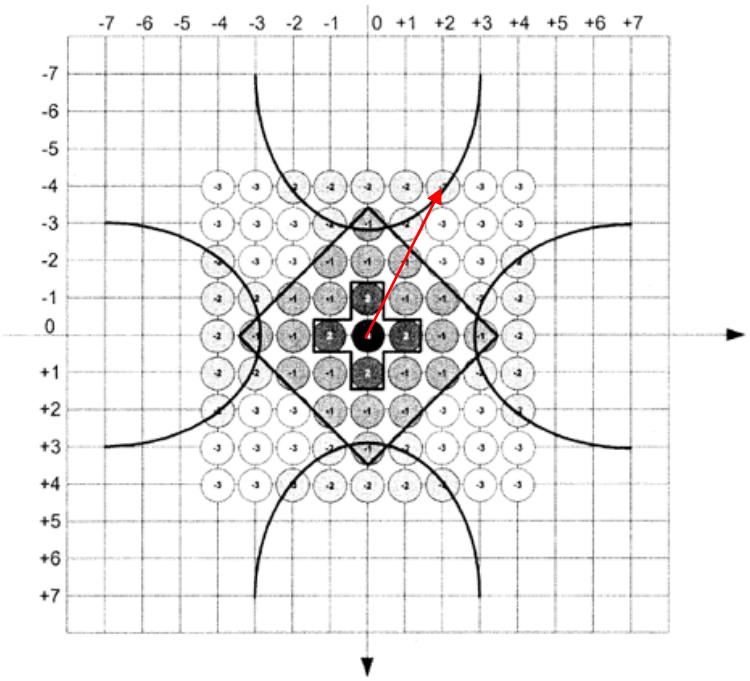


Fig. 4.8 Maximum number of search points saved / used by the CDS compared to the DS.
The search points saved are denoted as +ve while those used as –ve for the corresponding motion vector location. Outside of the central 3x3 region the DS begins to outperform the CDS. For example to find the MV(+2, -4) the CDS will require 3 more search points than the DS. This was previously demonstrated for the CDS procedure in chapter 2.
Source: Cheung and Po (2002b).

For the *Football* sequence, as shown by Table 4.4, each of the 3 hybrid DS algorithms are shown to be faster than the DS. The camera is stationary in this sequence and the object movement is due to the players arriving within the shot. Thus there are areas of the pitch with zero or no motion which would benefit from the halfway stops of the 3 hybrid DS algorithms. The larger the motion in a video sequence, the smaller the speed improvement rate of the 3 hybrid algorithms over the DS or the other fast algorithms will be – this can be seen when contrasted with the slower moving *Miss America* sequence. This again demonstrates that the CDS, the SCDS and the NCDS are better choices than the DS for slow moving videoconferencing sequences.

Comparing the performance of the 3 hybrid algorithms with each other in Table 4.1 we see that in terms of average number of search points used NCDS < SCDS < CDS < DS. For larger motions all 3 hybrid algorithms take 11 points to reach the unrestricted LDSP – CDS uses 9 initial points, plus 2 from HDSP; SCDS uses 5 initial points, 4 for LCSP and 2 from HDSP and NCDS uses 5 initial points, 3 for its second SCSP and 3 for LCSP. Thus the saving must occur within its halfway stops. The CDS performs poorer than the SCDS or the NCDS for low motion sequences or for sequences with local areas of low motion since it uses more points in reaching its halfway stops.

In general in using fewer search points a fast block match algorithm trades off block distortion for higher search speed (Tham *et al*, 1998). From Table 4.2 it can be observed that all 3 algorithms perform very competitively in terms of PSNR compared to the DS even though they lower the average number of search points. For a better comparison of the trade off between PSNR and search speed, Tables 4.4 and 4.5 give the percentage SIR over the DS and the Difference in Average PSNR for the 3 sequences. It can be seen that the NCDS has marginally worse PSNR performance than the DS compared to the other 2 techniques – the highest being a drop in PSNR of 0.680dB for the *Flower Garden* sequence. However, the speed improvements with the NCDS are quite substantial – up to 53% for the low motion sequence *Miss America* and thus justifies its use over the DS.

Table 4.5 shows that the CDS, the SCDS and the NCDS have a marginally lower PSNR than the DS – the highest being 0.680dB for the *Flower Garden* sequence. They consistently perform better than the DS algorithm with respect to speed, in particular for the low motion video conferencing sequence.

In conclusion, the CDS, the SCDS and the NCDS had a higher search speed than the DS for all 3 sequences with only a minimal loss in PSNR. From experimental results shown in Table 4.1, the NCDS takes the smallest average number of search points per block among the 3 hybrid cross diamond algorithms or the DS for each of the three test sequences. The NCDS is thus the fastest of the 3 hybrid cross diamond algorithms.

4.4 Performance of the 3 Hybrid DS Algorithms and the DS versus ARPS

Table 4.4 shows that ARPS is nearly 55% faster than the DS for the *Miss America* sequence, 48% faster than the DS for the *Flower Garden* sequence and nearly 41% faster than the DS for the *Football* sequence. ARPS is also significantly faster than the CDS, the SCDS and the NCDS for the *Flower Garden* sequence and the *Football* sequence and only the NCDS comes close to matching its speed for the *Miss America* sequence.

Table 4.5 shows that ARPS has a marginally lower PSNR than the DS by 0.0433dB for the *Miss America* sequence, but a higher PSNR than the DS by 0.966dB for the *Flower Garden* sequence and 0.0119dB higher than the DS for the *Football* sequence. It consistently performs better than the 3 hybrid DS algorithms with respect to PSNR.

Generally ARPS gave both higher PSNR and higher search speed than the DS, the CDS, the SCDS and the NCDS for all 3 sequences. The reason for the good performance of ARPS is that it quickly directs the search into the local region of the global minimum by calculating the Predicted Motion Vector, the minimum error from the rood pattern of nodes is found and then a final refined search calculates the motion vector.

4.5 Choosing a Block Motion Algorithm

Since with real-time encoding of video one may not always know the type of motion that will enter the encoder, the best fast block motion algorithm of the 10 algorithms studied is ARPS from the point of view of speed (lowest number of search points used per macroblock) and video quality (PSNR).

5.0 Conclusions & Future Work

Conclusions

A detailed description of the many block search algorithms available today was provided. An implementation for the 3 hybrid DS algorithms was coded in MATLAB. Three test sequences were examined.

Based on the cross-centre biased motion vector distribution of real world video sequences, the 3 hybrid DS algorithms were shown to improve on the DS algorithm by altering the starting pattern and providing a number of halfway stops. Simulation results showed that the NCDS was the fastest algorithm amongst the 3 hybrid DS algorithms simulated. A speedup ranging from 10% for the complex motion sequence *Flower Garden* to nearly 54% for the low motion video conference sequence *Miss America* was recorded.

All 3 algorithms performed very competitively in terms of PSNR compared to the DS even though they lower the average number of search points. It was shown that the NCDS has marginally worse PSNR performance than the DS compared to the other 2 algorithms – the highest being a drop in PSNR of 0.680dB for the *Flower Garden* sequence. However, the speed improvements for the NCDS are quite substantial and would thus justify its use over the DS. The results from the implementation concurred with the literature, therefore validating the implementation.

The implementation was used as a guide in nominating a 'robust' Block Search Algorithm. When the DS, CDS, SCDS or NCDS were compared with ARPS it was shown that ARPS generally gave both higher PSNR and higher search speed for all 3 sequences. The reason for the good performance of ARPS is that it quickly directs the search into the local region of the global minimum by calculating the Predicted Motion Vector, the minimum error from the rood pattern of nodes is found and then a final refined search calculates the motion vector.

Simulation results showed that ARPS was the best algorithm amongst the 10 algorithms simulated from the point of view of speed (lowest number of search points used per macroblock) and video quality (PSNR). For real-time encoding of video the best fast block motion algorithm to advise is ARPS.

Future Work

Future work could look at some other recent block search algorithms such as Kite Cross Diamond Search (Lam *et al*, 2004), Enhanced Hexagonal Search (Zhu *et al*, 2004) and Cross Diamond Hexagonal Search (Cheung and Po, 2005) – and provide implementations.

Another interesting area for analysis would be an investigation of the useful potential applications of Motion vectors – such as motion detection, object tracking, and even potential alternative encoding methods. A computational benefit is that an MPEG file does not need to be decoded to analyze its motion vectors.

Another area of investigation could be the analysis of flexible block sizes in motion estimation (Yu, 2004 and Servias *et al*, 2005). Traditional codecs commonly process frames at the macroblock level (16 pixels by 16 pixels). H.264, however can process on segments within a macroblock, ranging in size from the commonly used 16x16 to as small as 4x4, which helps to code complex motion in areas of high detail. The existing MATLAB code could be redeveloped to perform its processing on a variety of block sizes within a frame – benefiting scenes with complicated motion and thus providing higher quality in lower data rates. The existing code could also perhaps be developed to use both past and future frames in the motion estimation process as is the case with standard codecs.

References

1. J.R. Jain and A.K. Jain, (**Dec 1981**) *Displacement Measurement and its Application in Interframe Image Coding*, IEEE Trans. Commun., Vol. COM-29, No. 12, pp. 1799-1808.

2. T. Koga, K. Iinuma, A. Hirano, Y. Iijima and T. Ishiguro (**Nov-Dec 1981**) *Motion-Compensated Interframe Coding for Video Conferencing*, Proceedings National Telecommunications Conference, New Orleans, '81 (IEEE), p G.5.3.1 - G.5.3.5.

3. M. Ghanbari. (**July 1990**) *The Cross-Search Algorithm for Motion Estimation*, IEEE Transactions on Communications, Volume 38, No. 7, pp. 950–953.

4. Le Gail, D. (**April 1991**) *MPEG: A Video Compression Standard for Multimedia Applications*, Communications of the ACM, Vol. 34, No. 4, pp. 47-58.

5. Renxiang Li, Bing Zeng and Ming L. Liou, (**Aug. 1994**) *A New Three-Step Search Algorithm For Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 4, Issue 4, pp. 438–442.

6. Lai Man Po and Wing-Chung Ma, (**June 1996**) *A Novel Four-Step Search Algorithm For Fast Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 6, Issue 3, pp. 313–317.

7. Lurng-Kuo Liu and Ephraim Feig, (**Aug 1996**) *A Block-Based Gradient Descent Search Algorithm For Block Motion Estimation In Video Coding*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 6, Issue 4, pp.419–422.

8. Jianhua Lu and Ming L. Liou, (**Apr. 1997**) *A Simple And Efficient Search Algorithm For Block-Matching Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 7, Issue 2, pp. 429–433.

9. Cheung Chok Kwan (**July 1998**) *Fast Motion Estimation Techniques For Video Compression*, Thesis (M.Phil.), Dept. of Electronic Engineering, City University Of Hong Kong.

10. J. Y. Tham, S. Ranganath, M. Ranganath and A. A. Kassim, (**Aug. 1998**) *A Novel Unrestricted Center-Biased Diamond Search Algorithm For Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 8, pp. 369–377.

11. Yen-Kuang Chen **(Nov 1998)** *True Motion Estimation — Theory, Application, And Implementation*, Thesis (PhD), Department Of Electrical Engineering, Princeton University.

12. Shan Zhu and Kai-Kuang Ma, (**Feb. 2000**) *A New Diamond Search Algorithm For Fast Block-Matching Motion Estimation*, IEEE Transactions on Image Processing, Volume 9, Issue 2, pp. 287–290.

13. Dahlstrand, M. (**Dec 2001**) *Prediction Based Block Matching for Video Encoding*. A thesis presented to the Royal Institute Of Technology, Department Of Signals, Sensors & Systems Signal Processing, Stockholm [online]. Available from: http://www.ee.kth.se/php/modules/publications/reports/2001/IR-SB-EX-0125.pdf [Accessed Aug 03 2008].

14. Mei-Juan Chen, Ming-Chung Chu and Chih-Wei Pan. (**Apr. 2002**) *Efficient Motion-Estimation Algorithm For Reduced Frame-Rate Video Transcoder*. IEEE Transactions on Circuits and Systems for Video Technology, Volume 12, Issue 4, pp. 269–275.

15. Ce Zhu, Xiao Lin and Lap-Pui Chau, (**May 2002**) *Hexagon-Based Search Pattern For Fast Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 12, Issue 5, pp. 349–355.

16. Chun-Ho Cheung and Lai-Man Po, (**Sept. 2002a**) *A Novel Small-Cross-Diamond Search Algorithm For Fast Video Coding And Videoconferencing Applications*, Proceedings. IEEE 2002 International Conference on Image Processing, Volume 1, pp. 681–684.

17. Chun-Ho Cheung and Lai-Man Po, (**Dec. 2002b**) *A Novel Cross-Diamond Search Algorithm For Fast Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 12, Issue 12, pp. 1168–1177.

18. Yao Nie and Kai-Kuang Ma. (**Dec. 2002**) *Adaptive Rood Pattern Search For Fast Block-Matching Motion Estimation*, IEEE Transactions on Image Processing, Volume 11, Issue 12, pp. 1442–1449.

19. Richardson, I. (**2002**) *Video CODEC Design: Developing Image and Video Compression Systems*, 1st Edition, West Sussex: John Wiley & Sons.

20. Chau, L.P. and Jing, X. **(Apr. 2003)** *Efficient Three-Step Search Algorithm For Block Motion Estimation In Video Coding*. Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Volume 3, pp. 421-424.

21. Wiegand, T., Sullivan, G., Bjøntegaard, G. and Luthra, A. (**July 2003**) *Overview of the H.264/AVC Video Coding Standard*. IEEE Transactions on Circuits and Systems for Video Technology, Vol. 13, No. 7.

22. Lam, C.W., Po, L.M. and Cheung, C.H. (**Dec. 2003**) *A New Cross-Diamond Search Algorithm For Fast Block-Matching Motion Estimation*, IEEE International Conference on Neural Networks & Signal Processing, Volume 2, pp. 1262–1265.

23. Booth, S. P.W. (**2003**) *A New Fast Motion Search Algorithm For Block Based Video Encoders*. A thesis presented to the University of Waterloo, Ontario, Canada. [online]. Available from: www.eng.uwaterloo.ca/~dclausi/Theses/SimonBoothMASc2003.pdf [Accessed Aug 03 2008].

24. Ce Zhu, Xiao Lin, Lap-Pui Chau, Hock-Ann Ang and Choo-Yin Ong. (**Mar. 2004**). *Efficient Inner Search For Faster Diamond Search*, Signal Processing, Volume 84, Issue 3, pp. 527–533.

25. Barjatya, Aroh **(April 2004)** *Block Matching Algorithms* For *Motion Estimation*, DIP 6620 Final Project Paper in Digital Image Processing, Utah State University, pp. 1–6. [online]. Available from his homepage at http://cc.usu.edu/~arohb and also on the Mathworks file exchange site at http://www.mathworks.com/mathlabcentral. [Accessed Mar 06 2008].

26. Lam, C.W., Po, L.M. and Cheung, C.H. (**May 2004**) *A Novel Kite-Cross-Diamond Search Algorithm For Fast Block-Matching Motion Estimation*. Proceedings of the 2004 IEEE International Symposium on Circuits and Systems (ISCAS), Volume 3, pp. 729–732.

27. Jia, H. and Zhang, L. (**May 2004**) *A New Cross Diamond Search Algorithm for Block Motion Estimation*. Proceedings of the 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Volume 3, pp. 357–360.

28. Yu, A.C. (**May 2004**) *Efficient Block-Size Selection Algorithm For Inter-Frame Coding In H.264/MPEG-4 AVC*, Proceedings IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Volume 3, pp. 169–172.

29. Jing, X. and Chau, L.P. (**June 2004)** *An Efficient Three-Step Search Algorithm For Block Motion Estimation*, IEEE Transactions on Multimedia, Volume 6, Issue 3, pp. 435–438.

30. Zhu, C., Lin, X., Chau, L. and Po, L.M. (**Oct 2004**) *Enhanced Hexagonal Search for Fast Block Motion Estimation*, IEEE Transactions on Circuits and Systems for Video Technology, Volume 14, Issue 10, pp. 1210–1214.

31. Chapman, N. and Chapman, J. (**2004**) *Digital Multimedia*, 2nd Edition, West Sussex: John Wiley & Sons.

32. Cheung, C.H. and Po, L.M. (**Feb. 2005**) *Novel Cross-Diamond-Hexagonal Search Algorithms For Fast Block Motion Estimation*, IEEE Transactions on Circuits on Multimedia, Volume 7, No. 1, pp. 16–22.

33. Servias, M., Vlachos, T. and Davies, T. (**Sept. 2005**) *Motion-Compensation Using Variable-Size Block-Matching With Binary Partition Trees*, ICIP 2005. IEEE International Conference on Image Processing, Volume 1, pp. 157–160.

34. Amer, I. (**2008**) *DMET 1004 - Advanced Video Technology* Lecture Notes, German University in Cairo [online]. Available from: http://cs.guc.edu.eg/courses/_Spring2008/DMET1004/Tutorials/readyuv.m [Accessed Sept 06 2008].

35. CIPR (**2008**) *Center for Image Processing Research*, Electrical, Computer, and Systems Engineering Department, Rensselaer Polytechnic Institute, Troy, New York. [online]. Available from: http://www.cipr.rpi.edu/resource/sequences [Accessed Aug 03 2008].

36. Girod, B. (**2008**) *EE398 Image and Video Compression* Lecture Notes [online]. Available from: http://www.stanford.edu/class/ee398/handouts/lectures [Accessed Aug 06 2008].

37. Keepence, B. (**2008**) *IP Video CCTV and H.264* [online]. Available from: http://www.indigovision.com/learnabout-cctvh264.php [Accessed Sept 06 2008].

38. VTRG (**2008**) *Video Trace Research Group*, Arizona State University [online]. Available from: http://trace.eas.asu.edu/yuv/index.html [Accessed Aug 03 2008].

Appendix A: Glossary of Terms

Chrominance – This is the colour information for the pixel. In many applications, the luminance and chrominance are combined and displayed as RGB (red, green, blue) format rather than YUV (luminance and two chrominance components). RGB, YUV and others are known as colour spaces.

CIF (Common Intermediate Format) – a set of standard video formats used in videoconferencing, defined by their resolution 352x288. The original CIF is also known as Full CIF (FCIF).

Current Frame – The frame that is being predicted using blocks from a reference frame. A set of motion vectors results from the prediction.

Error Measure – The measure of how different one macroblock is to another. Some examples are Mean Absolute Error and Mean Square Error.

Luminance – This is the black and white content of the image or how light or dark a pixel is.

Macroblock – A group of 16x16 contiguous pixels within an image.

Motion Vector – A pair of numbers (a vector) representing the displacement between a macroblock in the current frame and a macroblock in the reference frame.

Motion estimation – the process done by the coder to find the motion vector pointing to the best prediction macroblock in a reference frame or field. Compression redundancy between adjacent frames can be exploited where a frame is selected as a reference and subsequent frames are predicted from the reference using motion estimation. The motion estimation process analyzes previous or future frames to identify blocks that have not changed, and motion vectors are stored in place of blocks. The process of video compression using motion estimation is also known as interframe coding.

QCIF (Quarter CIF) – a video format defined by a resolution 176x144.

Reference Frame – The frame that is used to make a prediction of another frame. The other frame may be a future or a previous frame.

Search Window – The area of the reference frame that is searched when motion estimation is performed. This is defined by the search parameter w which is typically set = ±7 pixels from the current macroblock position.

Appendix B: M-Code by Aroh Barjatya

Barjatya's project consisted of a project report and MATLAB source code as part of coursework for a Digital Image Processing Class at Utah State University. The *caltrain* test images, scripts and paper can be found at his homepage http://cc.usu.edu/~arohb and also on the Mathworks file exchange site at http://www.mathworks.com/mathlabcentral.

Approximately 1,700 lines of code were originally written by Aroh Barjatya. In all 7 block search algorithms were coded. Each algorithm calculates a matrix of motion vectors for each frame. The average number of locations searched per motion vector is recorded for each frame of the test sequence. The motion vectors are input into a motion compensated image creator script which reconstructs the image. The quality of the reconstructed image can be ascertained by comparing it with the original image and outputting the PSNR metric.

The following m files are his original work:

Table B.1: M files used by Barjatya and their role

	M-file Name	Description
1.	motionsEstAnalysis.m	Main Script to execute all Algorithms
2.	motionEstES.m	Exhaustive Search
3.	motionEstTSS.m	Three Step Search Algorithm
4.	motionEstNTSS.m	New Three Step Search Algorithm
5.	motionEstSESTSS.m	Simple And Efficient Search Algorithm
6.	motionEst4SS.m	Four Step Search Algorithm
7.	motionEstDS.m	Diamond Search Algorithm
8.	motionEstARPS.m	Adaptive Rood Pattern Search Algorithm
9.	costFuncMAD.m	Mean Absolute Difference Function
10.	minCost.m	Identifies minimum cost macroblock
11.	motionComp.m	Computes the motion compensated image using the given motion vectors
12.	imgPSNR.m	finds the PSNR of the motion compensated image w.r.t. original image

The MATLAB code used is shown below:

1. motionEstAnalysis.m Script to execute all Algorithms

```
% This script uses all the Motion Estimation algorithms written for the
% final project and save their results.
% The algorithms being used are Exhaustive Search, Three Step Search, New
% Three Step Search, Simple and Efficient Search, Four Step Search, Diamond
% Search, and Adaptive Rood Pattern Search.
%

close all
clear all

% the directory and files will be saved based on the image name
% Thus we just change the sequence / image name and the whole analysis is
% done for that particular sequence

imageName = 'caltrain';
mbSize = 16;
p = 7;

for i = 0:30

    imgINumber = i;
    imgPNumber = i+2;

    if imgINumber < 10
        imgIFile = sprintf('./%s/gray/%s00%d.ras',imageName, imageName, imgINumber);
    elseif imgINumber < 100
        imgIFile = sprintf('./%s/gray/%s0%d.ras',imageName, imageName, imgINumber);
    end

    if imgPNumber < 10
        imgPFile = sprintf('./%s/gray/%s00%d.ras',imageName, imageName, imgPNumber);
    elseif imgPNumber < 100
        imgPFile = sprintf('./%s/gray/%s0%d.ras',imageName, imageName, imgPNumber);
    end

    imgI = double(imread(imgIFile));
    imgP = double(imread(imgPFile));
    imgI = imgI(:,1:352);
    imgP = imgP(:,1:352);

        % Exhaustive Search
    [motionVect, computations] = motionEstES(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    ESpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    EScomputations(i+1) = computations;

    % Three Step Search
    [motionVect,computations ] = motionEstTSS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    TSSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    TSScomputations(i+1) = computations;

    % Simple and Efficient Three Step Search
    [motionVect, computations] = motionEstSESTSS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    SESTSSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    SESTSScomputations(i+1) = computations;
```

```
    % New Three Step Search
    [motionVect,computations ] = motionEstNTSS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    NTSSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    NTSScomputations(i+1) = computations;

    % Four Step Search
    [motionVect, computations] = motionEst4SS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    SS4psnr(i+1) = imgPSNR(imgP, imgComp, 255);
    SS4computations(i+1) = computations;

    % Diamond Search
    [motionVect, computations] = motionEstDS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    DSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    DScomputations(i+1) = computations;

    % Adaptive Rood Pattern Search
    [motionVect, computations] = motionEstARPS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    ARPSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    ARPScomputations(i+1) = computations;


end

save dsplots2 DSpsnr DScomputations ESpsnr EScomputations TSSpsnr ...
     TSScomputations SS4psnr SS4computations NTSSpsnr NTSScomputations ...
      SESTSSpsnr SESTSScomputations ARPSpsnr ARPScomputations
```

2. motionEstES.m Exhaustive Search Algorithm

```
% Computes motion vectors using exhaustive search method
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   EScomputations: The average number of points searched for a macroblock

function [motionVect, EScomputations] = motionEstES(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(2*p + 1, 2*p +1) * 65537;

computations = 0;

% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the exhaustive search starts here
      % we will evaluate cost for  (2p + 1) blocks vertically
      % and (2p + 1) blocks horizontaly
      % m is row(vertical) index
      % n is col(horizontal) index
      % this means we are scanning in raster order

      for m = -p : p
         for n = -p : p
            refBlkVer = i + m;   % row/Vert co-ordinate for ref block
            refBlkHor = j + n;   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                  || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
               continue;
            end
            costs(m+p+1,n+p+1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;

         end
      end

      % Now we find the vector where the cost is minimum
      % and store it ... this is what will be passed back.

      [dx, dy, min] = minCost(costs); % finds which macroblock in imgI gave us min Cost
      vectors(1,mbCount) = dy-p-1;    % row co-ordinate for the vector
      vectors(2,mbCount) = dx-p-1;    % col co-ordinate for the vector
```

```
        mbCount = mbCount + 1;
        costs = ones(2*p + 1, 2*p +1) * 65537;
    end
end

motionVect = vectors;
EScomputations = computations/(mbCount - 1);
```

3. motionEstTSS.m Three Step Search Algorithm

```
% Computes motion vectors using Three Step Search method
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   TSScomputations: The average number of points searched for a macroblock

function [motionVect, TSScomputations] = motionEstTSS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(3, 3) * 65537;

computations = 0;

% we now take effectively log to the base 2 of p
% this will give us the number of steps required

L = floor(log10(p+1)/log10(2));
stepMax = 2^(L-1);

% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the three step search starts
      % we will evaluate 9 elements at every step
      % read the literature to find out what the pattern is
      % my variables have been named aptly to reflect their significance

      x = j;
      y = i;

      % In order to avoid calculating the center point of the search
      % again and again we always store the value for it from teh
      % previous run. For the first iteration we store this value outside
      % the for loop, but for subsequent iterations we store the cost at
      % the point where we are going to shift our root.

      costs(2,2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                               imgI(i:i+mbSize-1,j:j+mbSize-1),mbSize);

      computations = computations + 1;
      stepSize = stepMax;
```

```
      while(stepSize >= 1)

         % m is row(vertical) index
         % n is col(horizontal) index
         % this means we are scanning in raster order
         for m = -stepSize : stepSize : stepSize
            for n = -stepSize : stepSize : stepSize
               refBlkVer = y + m;   % row/Vert co-ordinate for ref block
               refBlkHor = x + n;   % col/Horizontal co-ordinate
               if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                  || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                  continue;
               end

               costRow = m/stepSize + 2;
               costCol = n/stepSize + 2;
               if (costRow == 2 && costCol == 2)
                  continue
               end
               costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                  imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);

               computations = computations + 1;
            end
         end

         % Now we find the vector where the cost is minimum
         % and store it ... this is what will be passed back.

         [dx, dy, min] = minCost(costs);      % finds which macroblock in imgI gave us min Cost


         % shift the root for search window to new minima point

         x = x + (dx-2)*stepSize;
         y = y + (dy-2)*stepSize;

         % Arohs thought: At this point we can check and see if the
         % shifted co-ordinates are exactly the same as the root
         % co-ordinates of the last step, then we check them against a
         % preset threshold, and ifthe cost is less then that, than we
         % can exit from teh loop right here. This way we can save more
         % computations. However, as this is not implemented in the
         % paper I am modeling, I am not incorporating this test.
         % May be later...as my own addition to the algorithm

         stepSize = stepSize / 2;
         costs(2,2) = costs(dy,dx);

      end
      vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
      vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
      mbCount = mbCount + 1;
      costs = ones(3,3) * 65537;
   end
end

motionVect = vectors;
TSScomputations = computations/(mbCount - 1);
```

4. motionEstNTSS.m New Three Step Search Algorithm

```
% Computes motion vectors using *NEW* Three Step Search method
%
% Based on the paper by R. Li, b. Zeng, and M. L. Liou
% IEEE Trans. on Circuits and Systems for Video Technology
% Volume 4, Number 4, August 1994 :  Pages 438:442
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   NTSScomputations: The average number of points searched for a macroblock

function [motionVect, NTSScomputations] = motionEstNTSS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(3, 3) * 65537;

% we now take effectively log to the base 2 of p
% this will give us the number of steps required

L = floor(log10(p+1)/log10(2));
stepMax = 2^(L-1);

computations = 0;

% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the NEW three step search starts



      x = j;
      y = i;

      % In order to avoid calculating the center point of the search
      % again and again we always store the value for it from the
      % previous run. For the first iteration we store this value outside
      % the for loop, but for subsequent iterations we store the cost at
      % the point where we are going to shift our root.
      %
      % For the NTSS, we find the minimum first in the far away points
      % we then find the minimum for the close up points
      % we then compare the minimums and which ever is the lowest is where
      % we shift our root of search. If the minimum is the center of the
```

```
% current window then we stop the search. If its one of the
% immediate close to the center then we will do the second step
% stop. And if its in the far away points, then we go doing about
% the normal TSS approach
%
% more details in the code below or read the paper/literature

costs(2,2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(i:i+mbSize-1,j:j+mbSize-1),mbSize);
stepSize = stepMax;
computations = computations + 1;

% This is the calculation of the outer 8 points
% m is row(vertical) index
% n is col(horizontal) index
% this means we are scanning in raster order
for m = -stepSize : stepSize : stepSize
   for n = -stepSize : stepSize : stepSize
      refBlkVer = y + m;   % row/Vert co-ordinate for ref block
      refBlkHor = x + n;   % col/Horizontal co-ordinate
      if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
          || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
          continue;
      end

      costRow = m/stepSize + 2;
      costCol = n/stepSize + 2;
      if (costRow == 2 && costCol == 2)
         continue
      end
      costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
         imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
      computations = computations + 1;
   end
end

% Now we find the vector where the cost is minimum
% and store it ...

[dx, dy, min1] = minCost(costs);      % finds which macroblock in imgI gave us min Cost


% Find the exact co-ordinates of this point

x1 = x + (dx-2)*stepSize;
y1 = y + (dy-2)*stepSize;

% Now find the costs at 8 points right next to the center point
% (x,y) still points to the center

stepSize = 1;
for m = -stepSize : stepSize : stepSize
   for n = -stepSize : stepSize : stepSize
      refBlkVer = y + m;   % row/Vert co-ordinate for ref block
      refBlkHor = x + n;   % col/Horizontal co-ordinate
      if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
          || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
          continue;
```

```
        end

        costRow = m/stepSize + 2;
        costCol = n/stepSize + 2;
        if (costRow == 2 && costCol == 2)
           continue
        end
        costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
           imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
        computations = computations + 1;
     end
  end

  % now find the minimum amongst this

  [dx, dy, min2] = minCost(costs);      % finds which macroblock in imgI gave us min Cost


  % Find the exact co-ordinates of this point

  x2 = x + (dx-2)*stepSize;
  y2 = y + (dy-2)*stepSize;

  % the only place x1 == x2 and y1 == y2 will take place will be the
  % center of the search region

  if (x1 == x2 && y1 == y2)
     % then x and y still remain pointing to j and i;
     NTSSFlag = -1; % this flag will take us out of any more computations
  elseif (min2 <= min1)
     x = x2;
     y = y2;
     NTSSFlag = 1; % this flag signifies we are going to go into NTSS mode
  else
     x = x1;
     y = y1;
     NTSSFlag = 0; % This value of flag says, we go into normal TSS
  end


  if (NTSSFlag == 1)
     % Now in order to make sure that we dont calcylate the same
     % points again which were in the initial center window we take
     % care as follows

     costs = ones(3,3) * 65537;
     costs(2,2) = min2;
     stepSize = 1;
     for m = -stepSize : stepSize : stepSize
        for n = -stepSize : stepSize : stepSize
           refBlkVer = y + m;   % row/Vert co-ordinate for ref block
           refBlkHor = x + n;   % col/Horizontal co-ordinate
           if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                 || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              continue;
           end

           if ( (refBlkVer >= i - 1  && refBlkVer <= i + 1) ...
```

```
                && (refBlkHor >= j - 1  && refBlkHor <= j + 1) )
            continue;
        end

        costRow = m/stepSize + 2;
        costCol = n/stepSize + 2;
        if (costRow == 2 && costCol == 2)
            continue
        end
        costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
            imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
        computations = computations + 1;
      end
    end

    % now find the minimum amongst this

    [dx, dy, min2] = minCost(costs);      % finds which macroblock in imgI gave us min Cost

    % Find the exact co-ordinates of this point and stop

    x = x + (dx-2)*stepSize;
    y = y + (dy-2)*stepSize;

elseif (NTSSFlag == 0)
    % this is when we are going about doing normal TSS business
    costs = ones(3,3) * 65537;
    costs(2,2) = min1;
    stepSize = stepMax / 2;
    while(stepSize >= 1)
      for m = -stepSize : stepSize : stepSize
        for n = -stepSize : stepSize : stepSize
          refBlkVer = y + m;   % row/Vert co-ordinate for ref block
          refBlkHor = x + n;   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            continue;
          end

          costRow = m/stepSize + 2;
          costCol = n/stepSize + 2;
          if (costRow == 2 && costCol == 2)
            continue
          end
          costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
              imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations = computations + 1;

        end
      end

      % Now we find the vector where the cost is minimum
      % and store it ... this is what will be passed back.

      [dx, dy, min] = minCost(costs);      % finds which macroblock in imgI gave us min Cost


      % shift the root for search window to new minima point
```

```
            x = x + (dx-2)*stepSize;
            y = y + (dy-2)*stepSize;

            stepSize = stepSize / 2;
            costs(2,2) = costs(dy,dx);

         end
      end

      vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
      vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
      mbCount = mbCount + 1;
      costs = ones(3,3) * 65537;
   end
end

motionVect = vectors;
NTSScomputations = computations/(mbCount - 1);
```

5. motionEstSESTSS.m Simple And Efficient Search Algorithm

```
% Computes motion vectors using Simple and Efficient TSS method
%
% Based on the paper by Jianhua Lu and Ming L. Liou
% IEEE Trans. on Circuits and Systems for Video Technology
% Volume 7, Number 2, April 1997 :  Pages 429:433
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   SESTSScomputations: The average number of points searched for a macroblock
%
% Written by Aroh Barjatya


function [motionVect, SESTSScomputations] = motionEstSESTSS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);


% we now take effectively log to the base 2 of p
% this will give us the number of steps required

L = floor(log10(p+1)/log10(2));
stepMax =  2^(L-1);
costs = ones(1,6)*65537;

computations = 0;

% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the Simple and Efficient three step search starts here
      %
      % each step is divided into two phases
      % in the first phase we evaluate the cost in two ortogonal
      % directions at a distance of stepSize away
      % based on a certain relationship between the three points costs
      % we select the remaining TSS points in the second phase
      % At the end of the second phase, which ever point has the least
      % cost becomes the root for the next step.
      % Please read the paper to find out more detailed information

      stepSize = stepMax;
```

```
x = j;
y = i;
while (stepSize >= 1)
   refBlkVerPointA = y;
   refBlkHorPointA = x;

   refBlkVerPointB = y;
   refBlkHorPointB = x + stepSize;

   refBlkVerPointC = y + stepSize;
   refBlkHorPointC = x;

   if ( refBlkVerPointA < 1 || refBlkVerPointA+mbSize-1 > row ...
         || refBlkHorPointA < 1 || refBlkHorPointA+mbSize-1 > col)
      % do nothing %

   else
      costs(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                  imgI(refBlkVerPointA:refBlkVerPointA+mbSize-1, ...
                    refBlkHorPointA:refBlkHorPointA+mbSize-1), mbSize);
      computations = computations + 1;
   end

   if ( refBlkVerPointB < 1 || refBlkVerPointB+mbSize-1 > row ...
         || refBlkHorPointB < 1 || refBlkHorPointB+mbSize-1 > col)
      % do nothing %

   else
      costs(2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
            imgI(refBlkVerPointB:refBlkVerPointB+mbSize-1, ...
               refBlkHorPointB:refBlkHorPointB+mbSize-1), mbSize);
      computations = computations + 1;
   end


   if ( refBlkVerPointC < 1 || refBlkVerPointC+mbSize-1 > row ...
         || refBlkHorPointC < 1 || refBlkHorPointC+mbSize-1 > col)
      % do nothing %

   else
      costs(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
            imgI(refBlkVerPointC:refBlkVerPointC+mbSize-1, ...
               refBlkHorPointC:refBlkHorPointC+mbSize-1), mbSize);
      computations = computations + 1;
   end



   if (costs(1) >= costs(2) && costs(1) >= costs(3))
      quadrant = 4;
   elseif (costs(1) >= costs(2) && costs(1) < costs(3))
      quadrant = 1;
   elseif (costs(1) < costs(2) && costs(1) < costs(3))
      quadrant = 2;
   elseif (costs(1) < costs(2) && costs(1) >= costs(3))
      quadrant = 3;
   end
```

```
switch quadrant
   case 1
      refBlkVerPointD = y - stepSize;
      refBlkHorPointD = x;

      refBlkVerPointE = y - stepSize;
      refBlkHorPointE = x + stepSize;

      if ( refBlkVerPointD < 1 || refBlkVerPointD+mbSize-1 > row ...
            || refBlkHorPointD < 1 || refBlkHorPointD+mbSize-1 > col)
         % do nothing %

      else
         costs(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                  imgI(refBlkVerPointD:refBlkVerPointD+mbSize-1, ...
                     refBlkHorPointD:refBlkHorPointD+mbSize-1), mbSize);
         computations = computations + 1;
      end

      if ( refBlkVerPointE < 1 || refBlkVerPointE+mbSize-1 > row ...
            || refBlkHorPointE < 1 || refBlkHorPointE+mbSize-1 > col)
         % do nothing %

      else
         costs(5) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                  imgI(refBlkVerPointD:refBlkVerPointD+mbSize-1, ...
                     refBlkHorPointD:refBlkHorPointD+mbSize-1), mbSize);
         computations = computations + 1;
      end


   case 2
      refBlkVerPointD = y - stepSize;
      refBlkHorPointD = x;

      refBlkVerPointE = y - stepSize;
      refBlkHorPointE = x - stepSize;

      refBlkVerPointF = y;
      refBlkHorPointF = x - stepSize;


      if ( refBlkVerPointD < 1 || refBlkVerPointD+mbSize-1 > row ...
            || refBlkHorPointD < 1 || refBlkHorPointD+mbSize-1 > col)
         % do nothing %

      else
         costs(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                  imgI(refBlkVerPointD:refBlkVerPointD+mbSize-1, ...
                     refBlkHorPointD:refBlkHorPointD+mbSize-1), mbSize);
         computations = computations + 1;
      end

      if ( refBlkVerPointE < 1 || refBlkVerPointE+mbSize-1 > row ...
            || refBlkHorPointE < 1 || refBlkHorPointE+mbSize-1 > col)
         % do nothing %
```

```
        else
            costs(5) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVerPointE:refBlkVerPointE+mbSize-1, ...
                        refBlkHorPointE:refBlkHorPointE+mbSize-1), mbSize);
            computations = computations + 1;
        end

        if ( refBlkVerPointF < 1 || refBlkVerPointF+mbSize-1 > row ...
                || refBlkHorPointF < 1 || refBlkHorPointF+mbSize-1 > col)
            % do nothing %

        else
            costs(6) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVerPointF:refBlkVerPointF+mbSize-1, ...
                        refBlkHorPointF:refBlkHorPointF+mbSize-1), mbSize);
            computations = computations + 1;
        end


    case 3
        refBlkVerPointD = y;
        refBlkHorPointD = x - stepSize;

        refBlkVerPointE = y - stepSize;
        refBlkHorPointE = x - stepSize;

        if ( refBlkVerPointD < 1 || refBlkVerPointD+mbSize-1 > row ...
                || refBlkHorPointD < 1 || refBlkHorPointD+mbSize-1 > col)
            % do nothing %

        else
            costs(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVerPointD:refBlkVerPointD+mbSize-1, ...
                        refBlkHorPointD:refBlkHorPointD+mbSize-1), mbSize);
            computations = computations + 1;
        end

        if ( refBlkVerPointE < 1 || refBlkVerPointE+mbSize-1 > row ...
                || refBlkHorPointE < 1 || refBlkHorPointE+mbSize-1 > col)
            % do nothing %

        else
            costs(5) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVerPointD:refBlkVerPointD+mbSize-1, ...
                        refBlkHorPointD:refBlkHorPointD+mbSize-1), mbSize);
            computations = computations + 1;
        end


     case 4
        refBlkVerPointD = y + stepSize;
        refBlkHorPointD = x + stepSize;

        if ( refBlkVerPointD < 1 || refBlkVerPointD+mbSize-1 > row ...
                || refBlkHorPointD < 1 || refBlkHorPointD+mbSize-1 > col)
            % do nothing %
```

```
            else
               costs(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVerPointD:refBlkVerPointD+mbSize-1, ...
                           refBlkHorPointD:refBlkHorPointD+mbSize-1), mbSize);
               computations = computations + 1;
            end
          otherwise


        end


        % Now we find the vector where the cost is minimum
        % and store it ... this is what will be passed back.
        % we use the matlab function min() in this case and not the one
        % that is written by author: minCosts()


        [cost, dxy] = min(costs);      % finds which macroblock in imgI gave us min Cost

        switch dxy
            case 1
               % x = x; y = y;
            case 2
               x = refBlkHorPointB;
               y = refBlkVerPointB;
            case 3
               x = refBlkHorPointC;
               y = refBlkVerPointC;
            case 4
               x = refBlkHorPointD;
               y = refBlkVerPointD;
            case 5
               x = refBlkHorPointE;
               y = refBlkVerPointE;
            case 6
               x = refBlkHorPointF;
               y = refBlkVerPointF;

         end

        costs = ones(1,6) * 65537  ;
        stepSize = stepSize / 2;

      end

      vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
      vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
      mbCount = mbCount + 1;
   end
end

motionVect = vectors;
SESTSScomputations = computations/(mbCount - 1);
```

6. motionEst4SS.m Four Step Search Algorithm

```
% Computes motion vectors using Four Step Search method
%
% Based on the paper by Lai-Man Po, and Wing-Chung Ma
% IEEE Trans. on Circuits and Systems for Video Technology
% Volume 6, Number 3, June 1996 :  Pages 313:317
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   SS4computations: The average number of points searched for a macroblock

function [motionVect, SS4Computations] = motionEst4SS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(3, 3) * 65537;



% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it
computations = 0;



mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the 4 step search starts
      % we are scanning in raster order

      x = j;
      y = i;

      costs(2,2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(i:i+mbSize-1,j:j+mbSize-1),mbSize);
      computations = computations + 1;

      % This is the calculation of the 9 points
      % As this is the first stage, we evaluate all 9 points
      for m = -2 : 2 : 2
         for n = -2 : 2 : 2
            refBlkVer = y + m;   % row/Vert co-ordinate for ref block
            refBlkHor = x + n;   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
               continue;
            end
```

```
        costRow = m/2 + 2;
        costCol = n/2 + 2;
        if (costRow == 2 && costCol == 2)
           continue
        end
        costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
           imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
        computations = computations + 1;

     end
  end

  % Now we find the vector where the cost is minimum
  % and store it ...

  [dx, dy, cost] = minCost(costs);      % finds which macroblock in imgI gave us min Cost



  % The flag_4ss is set to 1 when the minimum
  % is at the center of the search area

  if (dx == 2 && dy == 2)
     flag_4ss = 1;
  else
     flag_4ss = 0;
     xLast = x;
     yLast = y;
     x = x + (dx-2)*2;
     y = y + (dy-2)*2;
  end

  costs = ones(3,3) * 65537;
  costs(2,2) = cost;

  stage = 1;
  while (flag_4ss == 0 && stage <=2)
     for m = -2 : 2 : 2
        for n = -2 : 2 : 2
           refBlkVer = y + m;   % row/Vert co-ordinate for ref block
           refBlkHor = x + n;   % col/Horizontal co-ordinate
           if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
              || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              continue;
           end

           if (refBlkHor >= xLast - 2 && refBlkHor <= xLast + 2 ...
                 && refBlkVer >= yLast - 2 && refBlkVer <= yLast + 2 )
              continue;
           end

           costRow = m/2 + 2;
           costCol = n/2 + 2;
           if (costRow == 2 && costCol == 2)
              continue
           end

           costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
```

```
                                imgI(refBlkVer:refBlkVer+mbSize-1, ...
                                  refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;


        end
    end

    [dx, dy, cost] = minCost(costs);


    if (dx == 2 && dy == 2)
        flag_4ss = 1;
    else
        flag_4ss = 0;
        xLast = x;
        yLast = y;
        x = x + (dx-2)*2;
        y = y + (dy-2)*2;
    end

    costs = ones(3,3) * 65537;
    costs(2,2) = cost;
    stage = stage + 1;


end  % while loop ends here


% we now enter the final stage

for m = -1 : 1 : 1
    for n = -1 : 1 : 1
        refBlkVer = y + m;   % row/Vert co-ordinate for ref block
        refBlkHor = x + n;   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
             || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            continue;
        end

        costRow = m + 2;
        costCol = n + 2;
        if (costRow == 2 && costCol == 2)
            continue
        end
        costs(costRow, costCol ) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
        imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
        computations = computations + 1;
    end
end

% Now we find the vector where the cost is minimum
% and store it ...

[dx, dy, cost] = minCost(costs);

x = x + dx - 2;
y = y + dy - 2;
```

```
        vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
        vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
        mbCount = mbCount + 1;
        costs = ones(3,3) * 65537;

    end
end

motionVect = vectors;
SS4Computations = computations/(mbCount - 1);
```

7. motionEstDS.m Diamond Search Algorithm

```
% Computes motion vectors using Diamond Search method
%
% Based on the paper by Shan Zhu, and Kai-Kuang Ma
% IEEE Trans. on Image Processing
% Volume 9, Number 2, February 2000 :  Pages 287:290
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   DScomputations: The average number of points searched for a macroblock
%
% Written by Aroh Barjatya


function [motionVect, DScomputations] = motionEstDS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(1, 9) * 65537;

% we now take effectively log to the base 2 of p
% this will give us the number of steps required

L = floor(log10(p+1)/log10(2));


% The index points for Large Diamond search pattern
LDSP(1,:) = [ 0 -2];
LDSP(2,:) = [-1 -1];
LDSP(3,:) = [ 1 -1];
LDSP(4,:) = [-2  0];
LDSP(5,:) = [ 0  0];
LDSP(6,:) = [ 2  0];
LDSP(7,:) = [-1  1];
LDSP(8,:) = [ 1  1];
LDSP(9,:) = [ 0  2];

% The index points for Small Diamond search pattern
SDSP(1,:) = [ 0 -1];
SDSP(2,:) = [-1  0];
SDSP(3,:) = [ 0  0];
SDSP(4,:) = [ 1  0];
SDSP(5,:) = [ 0  1];


% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it
```

```
computations = 0;

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the Diamond search starts
      % we are scanning in raster order

      x = j;
      y = i;

      costs(5) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                             imgI(i:i+mbSize-1,j:j+mbSize-1),mbSize);
      computations = computations + 1;

      % This is the first search so we evaluate all the 9 points in LDSP
      for k = 1:9
         refBlkVer = y + LDSP(k,2);   % row/Vert co-ordinate for ref block
         refBlkHor = x + LDSP(k,1);   % col/Horizontal co-ordinate
         if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
              || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            continue;
         end

         if (k == 5)
            continue
         end
         costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
              imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
         computations = computations + 1;
      end

      [cost, point] = min(costs);


      % The SDSPFlag is set to 1 when the minimum
      % is at the center of the diamond

      if (point == 5)
         SDSPFlag = 1;
      else
         SDSPFlag = 0;
         if ( abs(LDSP(point,1)) == abs(LDSP(point,2)) )
            cornerFlag = 0;
         else
            cornerFlag = 1; % the x and y co-ordinates not equal on corners
         end
         xLast = x;
         yLast = y;
         x = x + LDSP(point, 1);
         y = y + LDSP(point, 2);
         costs = ones(1,9) * 65537;
         costs(5) = cost;
      end
```

```
while (SDSPFlag == 0)
   if (cornerFlag == 1)
      for k = 1:9
         refBlkVer = y + LDSP(k,2);   % row/Vert co-ordinate for ref block
         refBlkHor = x + LDSP(k,1);   % col/Horizontal co-ordinate
         if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            continue;
         end

         if (k == 5)
            continue
         end

         if ( refBlkHor >= xLast - 1  && refBlkHor <= xLast + 1 ...
               && refBlkVer >= yLast - 1  && refBlkVer <= yLast + 1 )
            continue;
         elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
               || refBlkVer > i+p)
            continue;
         else
            costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                           refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
         end
      end

   else
      switch point
         case 2
            refBlkVer = y + LDSP(1,2);   % row/Vert co-ordinate for ref block
            refBlkHor = x + LDSP(1,1);   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              % do nothing, outside image boundary
            elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
               || refBlkVer > i+p)
              % do nothing, outside search window
            else
              costs(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                           refBlkHor:refBlkHor+mbSize-1), mbSize);
              computations = computations + 1;
            end

            refBlkVer = y + LDSP(2,2);   % row/Vert co-ordinate for ref block
            refBlkHor = x + LDSP(2,1);   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              % do nothing, outside image boundary
            elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
               || refBlkVer > i+p)
              % do nothing, outside search window
            else

              costs(2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                           refBlkHor:refBlkHor+mbSize-1), mbSize);
```

```
            computations = computations + 1;
        end

        refBlkVer = y + LDSP(4,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(4,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
            % do nothing, outside search window
        else

            costs(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end

    case 3
        refBlkVer = y + LDSP(1,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(1,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
            % do nothing, outside search window
        else

            costs(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end

        refBlkVer = y + LDSP(3,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(3,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
            % do nothing, outside search window
        else

            costs(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end

        refBlkVer = y + LDSP(6,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(6,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
```

```
            % do nothing, outside search window
        else

            costs(6) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end


    case 7
        refBlkVer = y + LDSP(4,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(4,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
            % do nothing, outside search window

        else
            costs(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end

        refBlkVer = y + LDSP(7,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(7,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
            % do nothing, outside search window

        else
            costs(7) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end

        refBlkVer = y + LDSP(9,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + LDSP(9,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
            || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
            || refBlkVer > i+p)
            % do nothing, outside search window

        else
            costs(9) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations = computations + 1;
        end
```

```
        case 8
            refBlkVer = y + LDSP(6,2);   % row/Vert co-ordinate for ref block
            refBlkHor = x + LDSP(6,1);   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              % do nothing, outside image boundary
            elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                || refBlkVer > i+p)
              % do nothing, outside search window

            else
              costs(6) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                          refBlkHor:refBlkHor+mbSize-1), mbSize);
              computations = computations + 1;
            end

            refBlkVer = y + LDSP(8,2);   % row/Vert co-ordinate for ref block
            refBlkHor = x + LDSP(8,1);   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              % do nothing, outside image boundary
            elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                || refBlkVer > i+p)
              % do nothing, outside search window

            else
              costs(8) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                          refBlkHor:refBlkHor+mbSize-1), mbSize);
              computations = computations + 1;
            end

            refBlkVer = y + LDSP(9,2);   % row/Vert co-ordinate for ref block
            refBlkHor = x + LDSP(9,1);   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              % do nothing, outside image boundary
            elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                || refBlkVer > i+p)
              % do nothing, outside search window

            else
              costs(9) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                          refBlkHor:refBlkHor+mbSize-1), mbSize);
              computations = computations + 1;
            end
        otherwise
    end
end

[cost, point] = min(costs);

if (point == 5)
    SDSPFlag = 1;
```

```
        else
            SDSPFlag = 0;
            if ( abs(LDSP(point,1)) == abs(LDSP(point,2)) )
                cornerFlag = 0;
            else
                cornerFlag = 1;
            end
            xLast = x;
            yLast = y;
            x = x + LDSP(point, 1);
            y = y + LDSP(point, 2);
            costs = ones(1,9) * 65537;
            costs(5) = cost;
        end

    end  % while loop ends here

    % we now enter the SDSP calculation domain
    costs = ones(1,5) * 65537;
    costs(3) = cost;

    for k = 1:5
        refBlkVer = y + SDSP(k,2);   % row/Vert co-ordinate for ref block
        refBlkHor = x + SDSP(k,1);   % col/Horizontal co-ordinate
        if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
             || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
            continue; % do nothing, outside image boundary
        elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                    || refBlkVer > i+p)
            continue;   % do nothing, outside search window
        end

        if (k == 3)
            continue
        end

        costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                    imgI(refBlkVer:refBlkVer+mbSize-1, ...
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
        computations = computations + 1;

    end

    [cost, point] = min(costs);

    x = x + SDSP(point, 1);
    y = y + SDSP(point, 2);

    vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
    vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
    mbCount = mbCount + 1;
    costs = ones(1,9) * 65537;

  end
end

motionVect = vectors;
DScomputations = computations/(mbCount - 1);
```

8. motionEstARPS.m Adaptive Rood Pattern Search Algorithm

```
% Computes motion vectors using Adaptive Rood Pattern Search method
%
% Based on the paper by Yao Nie, and Kai-Kuang Ma
% IEEE Trans. on Image Processing
% Volume 11 Number 12, December 2002 :  Pages 1442:1448
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter  (read literature to find what this means)
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   ARPScomputations: The average number of points searched for a macroblock
%
% Written by Aroh Barjatya


function [motionVect, ARPScomputations] = motionEstARPS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(1, 6) * 65537;



% The index points for Small Diamond search pattern
SDSP(1,:) = [ 0 -1];
SDSP(2,:) = [-1  0];
SDSP(3,:) = [ 0  0];
SDSP(4,:) = [ 1  0];
SDSP(5,:) = [ 0  1];

% We will be storing the positions of points where the checking has been
% already done in a matrix that is initialised to zero. As one point is
% checked, we set the corresponding element in the matrix to one.

checkMatrix = zeros(2*p+1,2*p+1);

computations = 0;

% we start off from the top left of the image
% we will walk in steps of mbSize
% mbCount will keep track of how many blocks we have evaluated

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      % the Adapive Rood Pattern search starts
      % we are scanning in raster order

      x = j;
      y = i;
```

```
costs(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                       imgI(i:i+mbSize-1,j:j+mbSize-1),mbSize);

checkMatrix(p+1,p+1) = 1;
computations =  computations + 1;
% if we are in the left most column then we have to make sure that
% we just do the LDSP with stepSize = 2
if (j-1 < 1)
   stepSize = 2;
   maxIndex = 5;
else
   stepSize = max(abs(vectors(1,mbCount-1)), abs(vectors(2,mbCount-1)));

   % now we have to make sure that if the point due to motion
   % vector is one of the LDSP points then we dont calculate it
   % again
   if ( (abs(vectors(1,mbCount-1)) == stepSize && vectors(2,mbCount-1) == 0) ...
        || (abs(vectors(2,mbCount-1)) == stepSize && vectors(1,mbCount-1) == 0)) ...

      maxIndex = 5; % we just have to check at the rood pattern 5 points

   else
      maxIndex = 6; % we have to check 6 points
      LDSP(6,:) = [ vectors(2, mbCount-1)  vectors(1, mbCount-1)];
   end
end

% The index points for first and only Large Diamond search pattern

LDSP(1,:) = [ 0 -stepSize];
LDSP(2,:) = [-stepSize 0];
LDSP(3,:) = [ 0  0];
LDSP(4,:) = [stepSize  0];
LDSP(5,:) = [ 0  stepSize];


% do the LDSP


for k = 1:maxIndex
   refBlkVer = y + LDSP(k,2);   % row/Vert co-ordinate for ref block
   refBlkHor = x + LDSP(k,1);   % col/Horizontal co-ordinate
   if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
        || refBlkHor < 1 || refBlkHor+mbSize-1 > col)

      continue; % outside image boundary
   end

   if (k == 3 || stepSize == 0)
      continue; % center point already calculated
   end
   costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
        imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
   computations =  computations + 1;
   checkMatrix(LDSP(k,2) + p+1, LDSP(k,1) + p+1) = 1;

end
```

```
        [cost, point] = min(costs);

        % The doneFlag is set to 1 when the minimum
        % is at the center of the diamond

        x = x + LDSP(point, 1);
        y = y + LDSP(point, 2);
        costs = ones(1,5) * 65537;
        costs(3) = cost;

        doneFlag = 0;
        while (doneFlag == 0)
           for k = 1:5
              refBlkVer = y + SDSP(k,2);   % row/Vert co-ordinate for ref block
              refBlkHor = x + SDSP(k,1);   % col/Horizontal co-ordinate
              if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                   || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                   continue;
              end

              if (k == 3)
                 continue
              elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                        || refBlkVer > i+p)
                    continue;
              elseif (checkMatrix(y-i+SDSP(k,2)+p+1 , x-j+SDSP(k,1)+p+1) == 1)
                 continue
              end

              costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                         imgI(refBlkVer:refBlkVer+mbSize-1, ...
                            refBlkHor:refBlkHor+mbSize-1), mbSize);
              checkMatrix(y-i+SDSP(k,2)+p+1, x-j+SDSP(k,1)+p+1) = 1;
              computations =  computations + 1;

           end

           [cost, point] = min(costs);

           if (point == 3)
              doneFlag = 1;
           else
              x = x + SDSP(point, 1);
              y = y + SDSP(point, 2);
              costs = ones(1,5) * 65537;
              costs(3) = cost;
           end

        end  % while loop ends here

        vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
        vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
        mbCount = mbCount + 1;
        costs = ones(1,6) * 65537;

        checkMatrix = zeros(2*p+1,2*p+1);
     end
end

motionVect = vectors;
ARPScomputations = computations/(mbCount-1) ;
```

9. costFuncMAD.m Mean Absolute Difference Function

```
% Computes the Mean Absolute Difference (MAD) for the given two blocks
% Input
%       currentBlk : The block for which we are finding the MAD
%       refBlk : the block w.r.t. which the MAD is being computed
%       n : the side of the two square blocks
%
% Output
%       cost : The MAD for the two blocks
%
% Written by Aroh Barjatya


function cost = costFuncMAD(currentBlk,refBlk, n)



err = 0;
for i = 1:n
   for j = 1:n
      err = err + abs((currentBlk(i,j) - refBlk(i,j)));
   end
end
cost = err / (n*n);
```

10. minCost.m Locates Minimum Cost Macroblock

```
% Finds the indices of the cell that holds the minimum cost
          % Input
%   costs : The matrix that contains the estimation costs for a macroblock
%
          % Output
%   dx : the motion vector component in columns
%   dy : the motion vector component in rows
%
% Written by Aroh Barjatya

function [dx, dy, min] = minCost(costs)

[row, col] = size(costs);

% we check whether the current
% value of costs is less then the already present value in min. If its
% indeed smaller then we swap the min value with the current one and note
% the indices.

min = 65537;

for i = 1:row
   for j = 1:col
      if (costs(i,j) < min)
         min = costs(i,j);
         dx = j; dy = i;
      end
   end
end
```

11. motionComp.m Motion Compensated Image Creator

```
% Computes motion compensated image using the given motion vectors
        % Input
%   imgI : The reference image
%   motionVect : The motion vectors
%   mbSize : Size of the macroblock
        % Ouput
%   imgComp : The motion compensated image
%
% Written by Aroh Barjatya

function imgComp = motionComp(imgI, motionVect, mbSize)

[row col] = size(imgI);

% we start off from the top left of the image
% we will walk in steps of mbSize
% for every marcoblock that we look at we will read the motion vector
% and put that macroblock from reference image in the compensated image

mbCount = 1;
for i = 1:mbSize:row-mbSize+1
   for j = 1:mbSize:col-mbSize+1

      % dy is row(vertical) index
      % dx is col(horizontal) index
      % this means we are scanning in order

      dy = motionVect(1,mbCount);
      dx = motionVect(2,mbCount);
      refBlkVer = i + dy;
      refBlkHor = j + dx;
      imageComp(i:i+mbSize-1,j:j+mbSize-1) = imgI(refBlkVer:refBlkVer+mbSize-1,
refBlkHor:refBlkHor+mbSize-1);

      mbCount = mbCount + 1;
   end
end


imgComp = imageComp;
```

12. imgPSNR.m Finds M.C. Image PSNR w.r.t. Reference Image

```
% Computes motion compensated image's PSNR
        % Input
%   imgP : The original image
%   imgComp : The compensated image
%   n : the peak value possible of any pixel in the images
        % Ouput
%   psnr : The motion compensated image's PSNR

function psnr = imgPSNR(imgP, imgComp, n)

[row col] = size(imgP);

err = 0;

for i = 1:row
   for j = 1:col
      err = err + (imgP(i,j) - imgComp(i,j))^2;
   end
end
mse = err / (row*col);

psnr = 10*log10(n*n/mse);
```

Appendix C: Corrections to Barjatya code by Jerome Casey

Some corrections to the original implementation by Barjatya have been added by me.
The following m-files have been updated:

motionEstSESTSS.m recalculates the cost at the central point A at the beginning of the while loop when the cost is already available at the end of the previous iteration. This will lead to an overestimation in the average number of points searched per macroblock for this algorithm. Thus initially calculate the cost of the centre point before entering the while loop as shown.

```
stepSize = stepMax;
      x = j;
      y = i;

     refBlkVerPointA = y;
     refBlkHorPointA = x;

        if ( refBlkVerPointA < 1 || refBlkVerPointA+mbSize-1 > row ...
             || refBlkHorPointA < 1 || refBlkHorPointA+mbSize-1 > col)
           % do nothing %

        else
           costs(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                      imgI(refBlkVerPointA:refBlkVerPointA+mbSize-1, ...
                        refBlkHorPointA:refBlkHorPointA+mbSize-1), mbSize);
           computations = computations + 1;
        end

      while (stepSize >= 1)

        refBlkVerPointB = y;
        refBlkHorPointB = x + stepSize;

        refBlkVerPointC = y + stepSize;
        refBlkHorPointC = x;


        if ( refBlkVerPointB < 1 || refBlkVerPointB+mbSize-1 > row ...
             || refBlkHorPointB < 1 || refBlkHorPointB+mbSize-1 > col)
           % do nothing %
```

The function also has an error in the location of point E in quadrant 3.

```
           case 3
              refBlkVerPointD = y;
              refBlkHorPointD = x - stepSize;

              refBlkVerPointE = y + stepSize;
              refBlkHorPointE = x - stepSize;
```

The code also orders the 4 quadrants differently to the original paper of Lu and Liou (1997) but is consistent in this. This is purely semantics and will not affect the PSNR or Average number of search points so the code does not need to be changed.

The minimum cost calculated at the end of an iteration should be stored for the next iteration. In the existing code it is recalculated again. In addition when a motion vector is found the cost matrix should be reset for the next motion vector iteration.

```
        costs = ones(1,6) * 65537  ;
        stepSize = stepSize / 2;
        costs (1) = cost ;
    end

    vectors(1,mbCount) = y - i;    % row co-ordinate for the vector
    vectors(2,mbCount) = x - j;    % col co-ordinate for the vector
    mbCount = mbCount + 1;
    costs = ones(1,6) * 65537  ;
  end
end
```

costFuncMAD.m is speeded up by vectorizing the for-loops within the M-file code. This optimises the function as well as the overall program execution since the function is called frequently.

```
err = sum(sum(abs(currentBlk - refBlk)));
```

imgPSNR.m is speeded up by vectorizing the for-loops within the M-file code.

```
err = (sum(sum(imgP - imgComp)))^2;
```

Appendix D: M-Code by Jerome Casey

The implementation added by me contains approximately 1,100 additional lines of code.

The following m files have been added:

Table D.1: M-files used by Casey and their role

	M-file Name	Description
1.	motionEstCDS.m	Cross Diamond Search Algorithm
2.	motionEstSCDS.m	Small Cross Diamond Search Algorithm
3.	motionEstNCDS.m	New Cross Diamond Search Algorithm
4.	plots.m	Produces frame-wise plots of Average Searching Points and Average PSNR
5.	stats.m	Calculates Average Searching Points, Average PSNR, Speed Improvement Ratio and the *PSNR difference* (w.r.t. Diamond Search) for the sequence overall.

motionsEstAnalysis.m - the main script - has also been updated to call the 3 additional algorithms and save the results to a *.mat* file.

```
  % 8 Cross Diamond Search
    [motionVect, computations] = motionEstCDS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    CDSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    CDScomputations(i+1) = computations;

  % 9 Small Cross Diamond Search
    [motionVect, computations] = motionEstSCDS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    SCDSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    SCDScomputations(i+1) = computations;

  % 10 New Cross Diamond Search
    [motionVect, computations] = motionEstNCDS(imgP,imgI,mbSize,p);
    imgComp = motionComp(imgI, motionVect, mbSize);
    NCDSpsnr(i+1) = imgPSNR(imgP, imgComp, 255);
    NCDScomputations(i+1) = computations;
 end
save dsplots2 DSpsnr DScomputations ESpsnr EScomputations TSSpsnr ...
     TSScomputations SS4psnr SS4computations NTSSpsnr NTSScomputations ...
      SESTSSpsnr SESTSScomputations ARPSpsnr ARPScomputations ...
CDSpsnr CDScomputations SDSpsnr SDScomputations NCDSpsnr NCDScomputations
```

The additional MATLAB code used in this work is shown below:

1. motionEstCDS.m Cross Diamond Search Algorithm

```
% Computes motion vectors using the Cross Diamond Search method
%
% Based on the paper by Chun-Ho Cheung and Lai-Man Po.
% IEEE Transactions on Circuits and Systems for Video Technology
% (Dec. 2002b)   Volume 12, Issue 12, pp. 1168–1177.
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   CDScomputations: The average number of points searched for a macroblock
%
% Written by Jerome Casey



function [motionVect, CDScomputations] = motionEstCDS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(1, 9) * 65537;  % 9 point cost matrix for the Cross Shaped pattern

% The index points for the Cross Shape pattern
CSP(1,:) = [ 0 -2];
CSP(2,:) = [ 0 -1];
CSP(3,:) = [-2  0];
CSP(4,:) = [-1  0];
CSP(5,:) = [ 0  0];
CSP(6,:) = [ 1  0];
CSP(7,:) = [ 2  0];
CSP(8,:) = [ 0  1];
CSP(9,:) = [ 0  2];

% we start off from the top left of the image and walk in steps of mbSize
% for every macroblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

% We will be storing the positions of points where the checking has been
% already done in a matrix that is initialised to 0. As a point is
% checked the corresponding element in the matrix to set to 1.

checkMatrix = zeros(2*p+1,2*p+1);
computations = 0;

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      x = j;
      y = i;
      xStart = j; % needed in step 2 if 2 of 4 points of a central-half LDSP
      yStart = i; % need to be calculated
```

```
        % apply the CSP and find the minimum BDM of the 9 points.
        % If the minimum BDM is at the centre then the search stops (first step stop).
        % If it is at another point then store the location and value of the min BDM.
        % and continue to the next step

        MVfoundFlag = 0;    % MVfoundFlag is set to 1 when Motion Vector is found
        SDSPFlag = 0;       % SDSPFlag is set to 1 when a SDSP needs to be executed

        % ********** Step 1 Uses a CSP to find a min BDM from 9 points************************************
        for k = 1:9
           refBlkVer = y + CSP(k,2);   % row/Vert co-ordinate for ref block
           refBlkHor = x + CSP(k,1);   % col/Horizontal co-ordinate
           if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              continue;   % since outside image boundary
           else
           costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
           computations =  computations + 1;
           checkMatrix(CSP(k,2) + p+1,CSP(k,1) + p+1) = 1; % row/Vert co-ord first, then col/Horiz
           end
        end

        [cost, point] = min(costs);
        if (point == 5)
           MVfoundFlag = 1;            % first step stop
        else
           xCSP = x + CSP(point, 1);    % shift centre to min BDM location for next step
           yCSP = y + CSP(point, 2);
           minCostCSP = cost;          % retain the cost for comparison with the min BDM from step 2
        end

        % points 2,4,6,8 are located at the middle wing of the CSP i.e.(±1,0) or (0,±1) if the min BDM
%occurs at any
        % of these 4 points and is still the overall min BDM by step 2 then we have a Second step stop

        if (mod(point,2)==0)   % remainder after division by 2 is zero for even numbers
           MiddleWingFlag = 1;
        else
           MiddleWingFlag = 0;
        end

        % ********** Step 2  A Half Diamond Search Pattern is applied**************************************

        if (MVfoundFlag == 0)

        % The index points for the Half Diamond Search pattern (±1,±1)
        HDSP(1,:) = [-1 -1];
        HDSP(2,:) = [1  -1];
        HDSP(3,:) = [-1  1];
        HDSP(4,:) = [ 1  1];

        HalfDiamondCosts = ones(1,4) * 65537; % initialise a new cost matrix to store costs for the 4 half
%diamond locations

        % Of the 4 candidate points in HDSP, just check the 2 points closest to the current min CSP
%BDM i.e. point
              switch point
```

```
            case {1 2}
               refBlkVer = yStart + HDSP(1,2);   % row/Vert co-ordinate for ref block
               refBlkHor = xStart + HDSP(1,1);   % col/Horizontal co-ordinate
               if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                  || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                  % do nothing, outside image boundary
               else
                  HalfDiamondCosts(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                           imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(1,2) + p+1,HDSP(1,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
               end

               refBlkVer = yStart + HDSP(2,2);   % row/Vert co-ordinate for ref block
               refBlkHor = xStart + HDSP(2,1);   % col/Horizontal co-ordinate
               if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                  || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                  % do nothing, outside image boundary
               else
                  HalfDiamondCosts(2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                           imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(2,2) + p+1,HDSP(2,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
               end

            case {3 4}
               refBlkVer = yStart + HDSP(1,2);   % row/Vert co-ordinate for ref block
               refBlkHor = xStart + HDSP(1,1);   % col/Horizontal co-ordinate
               if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                  || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                  % do nothing, outside image boundary
               else
                  HalfDiamondCosts(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                           imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(1,2) + p+1,HDSP(1,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
               end

               refBlkVer = yStart + HDSP(3,2);   % row/Vert co-ordinate for ref block
               refBlkHor = xStart + HDSP(3,1);   % col/Horizontal co-ordinate
               if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                  || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                  % do nothing, outside image boundary
               else
                  HalfDiamondCosts(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                           imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(3,2) + p+1,HDSP(3,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
               end

            case {6 7}
```

```
                refBlkVer = yStart + HDSP(2,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(2,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                   || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                    % do nothing, outside image boundary
                else
                  HalfDiamondCosts(2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                            imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(2,2) + p+1,HDSP(2,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

                refBlkVer = yStart + HDSP(4,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(4,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                   || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                    % do nothing, outside image boundary
                else
                  HalfDiamondCosts(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                            imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(4,2) + p+1,HDSP(4,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

            case {8 9}
                refBlkVer = yStart + HDSP(3,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(3,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                   || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                    % do nothing, outside image boundary
                else
                  HalfDiamondCosts(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                            imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(3,2) + p+1,HDSP(3,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

                refBlkVer = yStart + HDSP(4,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(4,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                   || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                    % do nothing, outside image boundary
                else
                  HalfDiamondCosts(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                            imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                  computations = computations + 1;
                  checkMatrix(HDSP(4,2) + p+1,HDSP(4,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

            otherwise
        end % end switch
```

```
      [minHalfDiamondcost, HalfDiamondpoint] = min(HalfDiamondCosts);

      xHDSP = xStart + HDSP(HalfDiamondpoint, 1);    % min BDM location is at one of the 4 points of
%the HDSP
      yHDSP = yStart + HDSP(HalfDiamondpoint, 2);    % record co-ords of min BDM HDSP point

   % decide overall min BDM from 2 costs
   % 1.minCostCSP at (xCSP,yCSP)   or    2.HalfDiamondcost at (xHDSP,yHDSP)
      OverallMinCost = ones(1, 2) * 65537;  % 2 point cost matrix

      OverallMinCost(1)= minCostCSP;
      OverallMinCost(2)= minHalfDiamondcost;

      [mincost, location] = min(OverallMinCost);

   % if the overall min BDM is at the middle wing of the CSP i.e.(±1,0) or (0,±1) then we have a Second
%step stop
       if (location == 1)
          if (MiddleWingFlag == 1)
              MVfoundFlag = 1;      % Second step stop
          end
         x = xCSP;
         y = yCSP;
       else
         x = xHDSP;
         y = yHDSP;
       end

       if (MVfoundFlag == 0)
         costs = ones(1,9) * 65537; % initialise a new cost matrix for the upcoming LDSP search if MV
%not found
         costs(5) = mincost;        % retain the cost so as not to calculate it again
       end

      end % end if from start of step 2

      % ******Step 3 An unrestricted Large Diamond Search Pattern is applied until the Min BDM
%occurs at the centre**
      while (MVfoundFlag == 0 &&  SDSPFlag == 0)

      % The index points for Large Diamond search pattern
      LDSP(1,:) = [ 0 -2];
      LDSP(2,:) = [-1 -1];
      LDSP(3,:) = [ 1 -1];
      LDSP(4,:) = [-2  0];
      LDSP(5,:) = [ 0  0];
      LDSP(6,:) = [ 2  0];
      LDSP(7,:) = [-1  1];
      LDSP(8,:) = [ 1  1];
      LDSP(9,:) = [ 0  2];

       for k = 1:9
          refBlkVer = y + LDSP(k,2);   % row/Vert co-ordinate for ref block
          refBlkHor = x + LDSP(k,1);   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
              || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
             continue;   % since outside image boundary
```

```
          end

          if (k == 5)
             continue;   % since centre point has already been calculated
          elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                       || refBlkVer > i+p)
             continue;       % since outside of search window
          elseif (checkMatrix(y-i+LDSP(k,2)+p+1,x-j+LDSP(k,1)+p+1) == 1)
             continue;
          else
          costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
               imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(y-i+LDSP(k,2)+ p+1,x-j+LDSP(k,1)+ p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
          end
        end % end for

        [cost, point] = min(costs);
        if (point == 5)           % The SDSPFlag is set to 1 when the minimum
          SDSPFlag = 1;           % cost occurs at the centre of the diamond
        else
          x = x + LDSP(point, 1); % shift centre to min BDM location for next step
          y = y + LDSP(point, 2);
          costs = ones(1,9) * 65537; % reset cost matrix for another LDSP loop
          costs(5) = cost;        % retain the cost so as not to calculate it again
        end
       end %end while

  %****** Step 4  A final Small Diamond Search Pattern is applied*********
       if (SDSPFlag == 1)

        % The index points for the Small Diamond search pattern
        SDSP(1,:) = [ 0 -1];
        SDSP(2,:) = [-1  0];
        SDSP(3,:) = [ 0  0];
        SDSP(4,:) = [ 1  0];
        SDSP(5,:) = [ 0  1];

        costs = ones(1,5) * 65537;
        costs(3) = cost;           % value of cost comes from final LDSP loop

         for k = 1:5
          refBlkVer = y + SDSP(k,2);   % row/Vert co-ordinate for ref block
          refBlkHor = x + SDSP(k,1);   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
             continue;   % do nothing, outside image boundary
          elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                       || refBlkVer > i+p)
             continue;   % do nothing, outside search window
          end

          if (k == 3)
             continue;   % since centre point has already been calculated
          elseif (checkMatrix(y-i+SDSP(k,2)+p+1,x-j+SDSP(k,1)+p+1) == 1)
             continue;
          else
          costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
```

```
            imgI(refBlkVer:refBlkVer+mbSize-1,refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(y-i+SDSP(k,2)+ p+1,x-j+SDSP(k,1)+ p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
           end
         end %end for

         [cost, point] = min(costs);
         x = x + SDSP(point, 1);
         y = y + SDSP(point, 2);
       end %end if from start of step 4

       vectors(1,mbCount) = y - i;   % row co-ordinate for the motion vector
       vectors(2,mbCount) = x - j;   % col co-ordinate for the motion vector
       mbCount = mbCount + 1;
       costs = ones(1, 9) * 65537;   % reset cost matrix for next MV search i.e. 9 point Cross Shaped
%pattern
       checkMatrix = zeros(2*p+1,2*p+1); % reset checkMatrix for next MV search
    end  %end for j
end %end for i

motionVect = vectors;
CDScomputations = computations/(mbCount - 1);
```

2. motionEstSCDS.m Small Cross Diamond Search Algorithm

```
% Computes motion vectors using the Small Cross Diamond Search method
%
% Based on the paper by Chun-Ho Cheung and Lai-Man Po.
% IEEE 2002 International Conference on Image Processing Proceedings
% (Sept. 2002a)   Volume 1, pp. 681–684.
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   SCDScomputations: The average number of points searched for a macroblock
%
% Written by Jerome Casey



function [motionVect, SCDScomputations] = motionEstSCDS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(1, 5) * 65537;  % 5 point cost matrix for Small Cross Shaped pattern

% The index points for the Small Cross Shaped pattern
SCSP(1,:) = [ 0 -1];
SCSP(2,:) = [-1  0];
SCSP(3,:) = [ 0  0];
SCSP(4,:) = [ 1  0];
SCSP(5,:) = [ 0  1];

% we start off from the top left of the image and walk in steps of mbSize
% for every macroblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

% We will be storing the positions of points where the checking has been
% already done in a matrix that is initialised to 0. As a point is
% checked the corresponding element in the matrix to set to 1.

checkMatrix = zeros(2*p+1,2*p+1);
computations = 0;

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      x = j;
      y = i;
      xStart = j; % needed if BDM of outer 4 points of a LCSP and 2 of 4 points of
      yStart = i; % a central-half LDSP need to be calculated in step 3

      % In order to avoid re-calculating the centre point of the search
      % we always store the value for it from the previous run.
      % For the first iteration of the While loop we store this value outside
      % the loop, but for subsequent iterations we store the cost at
```

```
        % the point where we are going to shift our root.
        %
        % For the SCDS we apply the Small CSP and find the minimum BDM of the 5 points.
        % If the minimum BDM is at the centre the search stops (first step stop).
        % If it is at one of the other 4 points, store the location and value of min BDM.
        % and continue to next step

        MVfoundFlag = 0;    % MVfoundFlag is set to 1 when Motion Vector is found
        SDSPFlag = 0;       % SDSPFlag is set to 1 when a SDSP needs to be executed

        % **********Step 1 Uses a SCSP to find a min BDM from 5 points************************************
        for k = 1:5
           refBlkVer = y + SCSP(k,2);   % row/Vert co-ordinate for ref block
           refBlkHor = x + SCSP(k,1);   % col/Horizontal co-ordinate
           if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              continue;   % since outside image boundary
           else
           costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
           computations =  computations + 1;
           checkMatrix(SCSP(k,2) + p+1,SCSP(k,1) + p+1) = 1; % row/Vert co-ord first, then col/Horiz
           end
        end

         [cost, point] = min(costs);
         if (point == 3)
           MVfoundFlag = 1;           % first step stop
         else
           xSCSP = x + SCSP(point, 1); % shift centre to min BDM location for next step
           ySCSP = y + SCSP(point, 2);
           minCostSCSP = cost;         % retain the cost so as not to calculate it again
         end

        % **********Step 2 (Guiding Step) Uses a LCSP to find a min BDM from the 4 outer
%points***********
        % this will guide the Half Diamond search pattern of step 3

        if (MVfoundFlag == 0)

         % The index points for the Large Cross Shape pattern
         LCSP(1,:) = [ 0 -2];
         LCSP(2,:) = [-2  0];
         LCSP(3,:) = [ 0  0];
         LCSP(4,:) = [ 2  0];
         LCSP(5,:) = [ 0  2];

         OuterCosts = ones(1,5) * 65537; % intialise a new cost matrix to store costs for the 4 outer points

         % original centre located at (xStart,yStart) - could also have used (j,i)here as well

         for k = 1:5
           refBlkVer = yStart + LCSP(k,2);   % row/Vert co-ordinate for ref block
           refBlkHor = xStart + LCSP(k,1);   % col/Horizontal co-ordinate
           if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
              continue;   % since outside image boundary
           end
```

```
        % no need to add code referring to p parameter just yet as search is still within the search
%window
        % but it will be needed for the unrestricted LDSP search

          if (k == 3)     % since this is the original centre point (xStart,yStart) and has already been
%calculated
             continue;   % in step 1 above
          else
          OuterCosts(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
               imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(LCSP(k,2) + p+1,LCSP(k,1) + p+1) = 1; % row/Vert co-ord first, then col/Horiz
          end
        end % end for

        [Outercost, Outerpoint] = min(OuterCosts);

        xLCSP = xStart + LCSP(Outerpoint, 1);    % min BDM location is at one of the 4 outer points of
%the LCSP
        yLCSP = yStart + LCSP(Outerpoint, 2);    % record co-ords of min BDM LCSP point

       %**********Step 3  A Half Diamond Search Pattern is applied**************************************

        % The index points for the Half Diamond Search pattern
        HDSP(1,:) = [-1 -1];
        HDSP(2,:) = [1  -1];
        HDSP(3,:) = [-1  1];
        HDSP(4,:) = [ 1  1];

        HalfDiamondCosts = ones(1,4) * 65537; % initialise a new cost matrix to store costs for the 4 half
%diamond locations

        % Of the 4 candidate points in HDSP, just check the 2 points closest to the min LCSP BDM i.e.
%Outerpoint
             switch Outerpoint
                case 1
                   refBlkVer = yStart + HDSP(1,2);   % row/Vert co-ordinate for ref block
                   refBlkHor = xStart + HDSP(1,1);   % col/Horizontal co-ordinate
                   if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                      || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                      % do nothing, outside image boundary
                   else
                     HalfDiamondCosts(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                               imgI(refBlkVer:refBlkVer+mbSize-1, ...
                                  refBlkHor:refBlkHor+mbSize-1), mbSize);
                     computations = computations + 1;
                     checkMatrix(HDSP(1,2) + p+1,HDSP(1,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                   end

                   refBlkVer = yStart + HDSP(2,2);   % row/Vert co-ordinate for ref block
                   refBlkHor = xStart + HDSP(2,1);   % col/Horizontal co-ordinate
                   if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                      || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                       % do nothing, outside image boundary
                   else
                     HalfDiamondCosts(2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
```

```
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                            refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(2,2) + p+1,HDSP(2,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

            case 2
                refBlkVer = yStart + HDSP(1,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(1,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                    || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                     % do nothing, outside image boundary
                else
                    HalfDiamondCosts(1) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                            refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(1,2) + p+1,HDSP(1,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

                refBlkVer = yStart + HDSP(3,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(3,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                    || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                     % do nothing, outside image boundary
                else
                    HalfDiamondCosts(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                            refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(3,2) + p+1,HDSP(3,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

            case 4
                refBlkVer = yStart + HDSP(2,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(2,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                    || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                     % do nothing, outside image boundary
                else
                    HalfDiamondCosts(2) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
                            refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(2,2) + p+1,HDSP(2,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

                refBlkVer = yStart + HDSP(4,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(4,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                    || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                     % do nothing, outside image boundary
                else
                    HalfDiamondCosts(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                        imgI(refBlkVer:refBlkVer+mbSize-1, ...
```

```
                        refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(4,2) + p+1,HDSP(4,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

            case 5
                refBlkVer = yStart + HDSP(3,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(3,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                    || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                     % do nothing, outside image boundary
                else
                    HalfDiamondCosts(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                            imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(3,2) + p+1,HDSP(3,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

                refBlkVer = yStart + HDSP(4,2);   % row/Vert co-ordinate for ref block
                refBlkHor = xStart + HDSP(4,1);   % col/Horizontal co-ordinate
                if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                    || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
                     % do nothing, outside image boundary
                else
                    HalfDiamondCosts(4) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                            imgI(refBlkVer:refBlkVer+mbSize-1, ...
                              refBlkHor:refBlkHor+mbSize-1), mbSize);
                    computations = computations + 1;
                    checkMatrix(HDSP(4,2) + p+1,HDSP(4,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
                end

            otherwise
        end % end switch

    [minHalfDiamondcost, HalfDiamondpoint] = min(HalfDiamondCosts);

    xHDSP = xStart + HDSP(HalfDiamondpoint, 1);    % min BDM location is at one of the 4 points of
%the HDSP
    yHDSP = yStart + HDSP(HalfDiamondpoint, 2);    % record co-ords of min BDM HDSP point

  %  decide overall min BDM from 3 costs
  %  1.minCostSCSP at (xSCSP,ySCSP) 2.Outercost at (xLCSP,yLCSP) and 3.HalfDiamondcost at
% (xHDSP,yHDSP)
    OverallMinCost = ones(1, 3) * 65537;  % 3 point cost matrix

    OverallMinCost(1)= minCostSCSP;
    OverallMinCost(2)= Outercost;
    OverallMinCost(3)= minHalfDiamondcost;

    [mincost, location] = min(OverallMinCost);
     if (location == 1)
        MVfoundFlag = 1;       % Third step stop
        x = xSCSP;
        y = ySCSP;
```

```
        elseif(location == 2)
          x = xLCSP;
          y = yLCSP;
        else
          x = xHDSP;
          y = yHDSP;
        end

        if (MVfoundFlag == 0)
          costs = ones(1,9) * 65537; % initialise a new cost matrix for the upcoming LDSP search if MV
%not found
          costs(5) = mincost;        % retain the cost so as not to calculate it again
        end

       end % end if from start of step 2

       %****** Step 4 An unrestricted Large Diamond Search Pattern is applied until the Min BDM
%occurs at the centre**
       while (MVfoundFlag == 0 &&  SDSPFlag == 0)

       % The index points for the Large Diamond search pattern
       LDSP(1,:) = [ 0 -2];
       LDSP(2,:) = [-1 -1];
       LDSP(3,:) = [ 1 -1];
       LDSP(4,:) = [-2  0];
       LDSP(5,:) = [ 0  0];
       LDSP(6,:) = [ 2  0];
       LDSP(7,:) = [-1  1];
       LDSP(8,:) = [ 1  1];
       LDSP(9,:) = [ 0  2];

        for k = 1:9
          refBlkVer = y + LDSP(k,2);   % row/Vert co-ordinate for ref block
          refBlkHor = x + LDSP(k,1);   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
              || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
             continue;   % since outside image boundary
          end

          if (k == 5)
             continue;   % since centre point has already been calculated
          elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                      || refBlkVer > i+p)
             continue;       % since outside of search window
          elseif (checkMatrix(y-i+LDSP(k,2)+p+1 , x-j+LDSP(k,1)+p+1) == 1)
             continue;
          else
          costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
               imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(y-i+LDSP(k,2) + p+1,x-j+LDSP(k,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
          end
        end % end for

        [cost, point] = min(costs);
        if (point == 5)           % The SDSPFlag is set to 1 when the minimum
          SDSPFlag = 1;           % cost occurs at the centre of the diamond
        else
```

```
          x = x + LDSP(point, 1); % shift centre to min BDM location for next step
          y = y + LDSP(point, 2);
          costs = ones(1,9) * 65537; % reset cost matrix for another LDSP loop
          costs(5) = cost;        % retain the cost so as not to calculate it again
        end

      end %end while from start of step 4

  % ****** Step 5  A final Small Diamond Search Pattern is applied*********
      if (SDSPFlag == 1)

      % The index points for the Small Diamond search pattern
      SDSP(1,:) = [ 0 -1];
      SDSP(2,:) = [-1  0];
      SDSP(3,:) = [ 0  0];
      SDSP(4,:) = [ 1  0];
      SDSP(5,:) = [ 0  1];

      costs = ones(1,5) * 65537;
      costs(3) = cost;           % value of cost comes from final LDSP loop

        for k = 1:5
          refBlkVer = y + SDSP(k,2);   % row/Vert co-ordinate for ref block
          refBlkHor = x + SDSP(k,1);   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
             continue;   % do nothing, outside image boundary
          elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                      || refBlkVer > i+p)
             continue;   % do nothing, outside search window
          end

          if (k == 3)
             continue;   % since centre point has already been calculated
          elseif (checkMatrix(y-i+SDSP(k,2)+p+1 , x-j+SDSP(k,1)+p+1) == 1)
             continue;
          else
          costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
               imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(y-i+SDSP(k,2) + p+1,x-j+SDSP(k,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
          end
        end %end for

        [cost, point] = min(costs);
        x = x + SDSP(point, 1);
        y = y + SDSP(point, 2);
      end %end if from start of step 5

      vectors(1,mbCount) = y - i;   % row co-ordinate for the motion vector
      vectors(2,mbCount) = x - j;   % col co-ordinate for the motion vector
      mbCount = mbCount + 1;
      costs = ones(1, 5) * 65537;   % reset cost matrix for next MV search i.e. 5 point Small Cross
%Shaped pattern
      checkMatrix = zeros(2*p+1,2*p+1); % reset checkMatrix for next MV search
   end  %end for j
end %end for i

motionVect = vectors;
SCDScomputations = computations/(mbCount - 1);
```

3. motionEstNCDS.m New Cross Diamond Search Algorithm

```
% Computes motion vectors using the New Cross Diamond Search method
%
% Based on the paper by Chi-Wai Lam, Lai-Man Po and Chun Ho Cheung
% IEEE International Conference on Neural Networks & Signal Processing
% (Dec. 2003)   Volume 2, pp. 1262–1265.
%
% Input
%   imgP : The image for which we want to find motion vectors
%   imgI : The reference image
%   mbSize : Size of the macroblock
%   p : Search parameter
%
% Ouput
%   motionVect : the motion vectors for each integral macroblock in imgP
%   NCDScomputations: The average number of points searched for a macroblock
%
% Written by Jerome Casey



function [motionVect, NCDScomputations] = motionEstNCDS(imgP, imgI, mbSize, p)

[row col] = size(imgI);

vectors = zeros(2,row*col/mbSize^2);
costs = ones(1, 5) * 65537;  % 5 point cost matrix for Small Cross Shaped pattern



% The index points for the Small Cross Shaped pattern
SCSP(1,:) = [ 0 -1];
SCSP(2,:) = [-1  0];
SCSP(3,:) = [ 0  0];
SCSP(4,:) = [ 1  0];
SCSP(5,:) = [ 0  1];

% we start off from the top left of the image and walk in steps of mbSize
% for every macroblock that we look at we will look for
% a close match p pixels on the left, right, top and bottom of it

% We will be storing the positions of points where the checking has been
% already done in a matrix that is initialised to 0. As a point is
% checked the corresponding element in the matrix to set to 1.

checkMatrix = zeros(2*p+1,2*p+1);
computations = 0;

mbCount = 1;
for i = 1 : mbSize : row-mbSize+1
   for j = 1 : mbSize : col-mbSize+1

      x = j;
      y = i;
      xStart = j; % needed if BDM of outer 3 points
      yStart = i; % of a LCSP needs to be calculated in step 3

      % In order to avoid re-calculating the centre point of the search
      % we always store the value for it from the previous run.
```

```
        % For the first iteration of the While loop we store this value outside
        % the loop, but for subsequent iterations we store the cost at
        % the point where we are going to shift our root.
        %
        % For the NCDS we apply the Small CSP and find the minimum BDM of the 5 points.
        % If the minimum BDM is at the centre the search stops (first step stop).
        % If it is at one of the other 4 points, make that the centre for a 2nd Small CSP
        % If the minimum BDM is at the centre at this step the search stops (second step stop).
        % If it is at one of the other 4 points, record the location and
        % continue to next While loop

        costs(3) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                               imgI(i:i+mbSize-1,j:j+mbSize-1),mbSize);
        checkMatrix(p+1,p+1) = 1;
        computations = computations + 1;

        MVfoundFlag = 0;    % MVfoundFlag is set to 1 when Motion Vector is found
        SDSPFlag = 0;       % SDSPFlag is set to 1 when a SDSP needs to be executed
        step = 1;           % about to start step 1

        % **********Step(s) 1/2 Uses a SCSP to find a min BDM from 5 points**********

        while (MVfoundFlag == 0 && step <= 2)
         for k = 1:5
            refBlkVer = y + SCSP(k,2);   % row/Vert co-ordinate for ref block
            refBlkHor = x + SCSP(k,1);   % col/Horizontal co-ordinate
            if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
                 || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
               continue;   % since outside image boundary
            end

            if (k == 3)
               continue;   % since centre point has already been calculated
            elseif (checkMatrix(y-i+SCSP(k,2)+p+1 , x-j+SCSP(k,1)+p+1) == 1)% y-i=0 and x-j=0 in step 1
               continue;   % needed since step 2 will have some points where checking has already been
%done
            else
            costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
                 imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
            computations =  computations + 1;
            checkMatrix(y-i+SCSP(k,2) + p+1,x-j+SCSP(k,1) + p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
            end
         end % end for

         [cost, point] = min(costs);
         if (point == 3)
            MVfoundFlag = 1;        % first/second step stop
         else
            x = x + SCSP(point, 1); % shift centre to min BDM location for next step
            y = y + SCSP(point, 2);
            costs = ones(1,5) * 65537;
            costs(3) = cost;        % retain the cost so as not to calculate it again
         end

         step = step + 1;
        end % end while
```

```
      % **********Step 3 Uses a LCSP to guide the centre for the following LDSP step**********
      % find the minimum BDM between the 3 outer points of the LCSP and the mincost in step 2

      if (MVfoundFlag == 0)

       % The index points for the Large Cross Shape pattern
       LCSP(1,:) = [ 0 -2];
       LCSP(2,:) = [-2  0];
       LCSP(3,:) = [ 0  0];
       LCSP(4,:) = [ 2  0];
       LCSP(5,:) = [ 0  2];

       OuterCosts = ones(1,5) * 65537; % intialise a new cost matrix to store costs for the 3 outer points
       % original centre located at (xStart,yStart) - could also have used (j,i)here as well

       for k = 1:5
          refBlkVer = yStart + LCSP(k,2);   % row/Vert co-ordinate for ref block
          refBlkHor = xStart + LCSP(k,1);   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
             continue;   % since outside image boundary
          end

       % no need to add code referring to p parameter just yet as search is still within the search
%window
       % but it will be needed for the unrestricted LDSP search

          if (k == 3)     % since this is the original centre point and has already been calculated
             continue;   % in any case min cost from step 2 is < cost at the original centre point
          elseif (checkMatrix(LCSP(k,2)+p+1,LCSP(k,1)+p+1) == 1)% will avoid 1 of the 4 pts already
%calculated
             continue;
          else
          OuterCosts(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
               imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(LCSP(k,2)+ p+1,LCSP(k,1)+p+1) = 1; % row/Vert co-ord first, then col/Horiz
          end
       end % end for

       [Outercost, Outerpoint] = min(OuterCosts);

       if (Outercost < cost)                  % compare with min cost from step 2
          x = xStart + LCSP(Outerpoint, 1);    % min BDM location is at one of the 3 outer points of the
%LCSP
          y = yStart + LCSP(Outerpoint, 2);    % shift centre here for the upcoming LDSP
          mincost = Outercost;
       else
          % x,y are already pointing to mincost location from step 2
          mincost = cost;
       end
       costs = ones(1,9) * 65537; % initialise a new cost matrix for the upcoming LDSP search
       costs(5) = mincost;       % retain the cost so as not to calculate it again
      end % end if from start of step 3

      %****** Step 4 An unrestricted Large Diamond Search Pattern is applied until the Min BDM
%occurs at the centre**
      while (MVfoundFlag == 0 &&  SDSPFlag == 0)
```

```
    % The index points for Large Diamond search pattern
    LDSP(1,:) = [ 0 -2];
    LDSP(2,:) = [-1 -1];
    LDSP(3,:) = [ 1 -1];
    LDSP(4,:) = [-2  0];
    LDSP(5,:) = [ 0  0];
    LDSP(6,:) = [ 2  0];
    LDSP(7,:) = [-1  1];
    LDSP(8,:) = [ 1  1];
    LDSP(9,:) = [ 0  2];

    for k = 1:9
      refBlkVer = y + LDSP(k,2);   % row/Vert co-ordinate for ref block
      refBlkHor = x + LDSP(k,1);   % col/Horizontal co-ordinate
      if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
          || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
        continue;   % since outside image boundary
      end

      if (k == 5)
        continue;   % since centre point has already been calculated
      elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                || refBlkVer > i+p)
        continue;       % since outside of search window
      elseif (checkMatrix(y-i+LDSP(k,2)+p+1 , x-j+LDSP(k,1)+p+1) == 1)
        continue;
      else
      costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
          imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
      computations =  computations + 1;
      checkMatrix(y-i+LDSP(k,2)+p+1,x-j+LDSP(k,1)+p+1) = 1; % row/Vert co-ord first, then col/Horiz
      end
    end % end for

    [cost, point] = min(costs);
    if (point == 5)           % The SDSPFlag is set to 1 when the minimum
      SDSPFlag = 1;           % cost occurs at the centre of the diamond
    else
      x = x + LDSP(point, 1); % shift centre to min BDM location for next step
      y = y + LDSP(point, 2);
      costs = ones(1,9) * 65537; % reset cost matrix for another LDSP loop
      costs(5) = cost;        % retain the cost so as not to calculate it again
    end

  end % end while

% ****** Step 5  A final Small Diamond Search Pattern is applied*********
  if (SDSPFlag == 1)

    % The index points for the Small Diamond search pattern
    SDSP(1,:) = [ 0 -1];
    SDSP(2,:) = [-1  0];
    SDSP(3,:) = [ 0  0];
    SDSP(4,:) = [ 1  0];
    SDSP(5,:) = [ 0  1];

    costs = ones(1,5) * 65537;
```

```
        costs(3) = cost;           % value of cost comes from final LDSP loop

        for k = 1:5
          refBlkVer = y + SDSP(k,2);   % row/Vert co-ordinate for ref block
          refBlkHor = x + SDSP(k,1);   % col/Horizontal co-ordinate
          if ( refBlkVer < 1 || refBlkVer+mbSize-1 > row ...
               || refBlkHor < 1 || refBlkHor+mbSize-1 > col)
             continue;   % do nothing, outside image boundary
          elseif (refBlkHor < j-p || refBlkHor > j+p || refBlkVer < i-p ...
                      || refBlkVer > i+p)
             continue;   % do nothing, outside search window
          end

          if (k == 3)
             continue;   % since centre point has already been calculated
          elseif (checkMatrix(y-i+SDSP(k,2)+p+1 , x-j+SDSP(k,1)+p+1) == 1)
             continue;
          else
          costs(k) = costFuncMAD(imgP(i:i+mbSize-1,j:j+mbSize-1), ...
               imgI(refBlkVer:refBlkVer+mbSize-1, refBlkHor:refBlkHor+mbSize-1), mbSize);
          computations =  computations + 1;
          checkMatrix(y-i+SDSP(k,2)+ p+1,x-j+SDSP(k,1)+ p+1) = 1; % row/Vert co-ord first, then
%col/Horiz
          end
        end %end for

        [cost, point] = min(costs);
        x = x + SDSP(point, 1);
        y = y + SDSP(point, 2);
       end %end if from start of step 5

       vectors(1,mbCount) = y - i;   % row co-ordinate for the motion vector
       vectors(2,mbCount) = x - j;   % col co-ordinate for the motion vector
       mbCount = mbCount + 1;
       costs = ones(1, 5) * 65537;   % reset cost matrix for next MV search i.e. 5 point Small Cross
%Shaped pattern
       checkMatrix = zeros(2*p+1,2*p+1); % reset checkMatrix for next MV search
    end
end

motionVect = vectors;
NCDScomputations = computations/(mbCount - 1);
```

plots.m - Produces frame-wise plots of Average Searching Points and Average PSNR and saves the corresponding images.

```
%**************Plot Search Points per MacroBlock vs Frame Number**********
    % Line Style Specifiers: Specifier  Line Style
    % - Solid line (default) -- Dashed line;
    % : Dotted line   -. Dash-dot line

    % Specifier and Marker Type
    % + Plus sign;o Circle;* Asterisk;. Point;x Cross;'square' or s Square;
    %'diamond' or d Diamond;^ Upward-pointing triangle;v Downward-pointing triangle
    % > Right-pointing triangle; < Left-pointing triangle
    %'pentagram' or p Five-pointed star (pentagram)
    %'hexagram' or h Six-pointed star (hexagram)

    % Color and Specifiers
    % r Red;g Green;b Blue;c Cyan;m Magenta;y Yellow;k Black;w White
figure(1)
    plot (NTSScomputations,'-.b');
    hold all;
    grid on;
    plot (SS4computations,':g');
    plot (SESTSScomputations,'-g');
    plot (TSScomputations,'-k');
    plot (ARPScomputations,'-r');
    plot (DScomputations,'-db');
    plot (CDScomputations,'-dy');
    plot (SCDScomputations,'-dm');
    plot (NCDScomputations,'-dc');

    title([imageName,' sequence    Block Size ',num2str(mbSize),'x',num2str(mbSize),'   Search Parameter p =
',num2str(p)]);
    xlabel([imageName,' Frame Number']);
    ylabel('Search Points per MacroBlock');

legend({'NTSScomputations','SS4computations','SESTSScomputations','TSScomputations','ARPScomputations','
DScomputations','CDScomputations','SCDScomputations','NCDScomputations'},'FontSize',7);

    hold off;
    saveas(gcf,[imageName,'NumSearchPtsperMacroBlock.jpg'])

%*****Plot PSNR of the motion compensated image w.r.t. original image vs Frame Number******
    figure(2)
    plot (ESpsnr,'-.r');
    hold all;
    grid on;
    plot (NTSSpsnr,'-.b');
    plot (SS4psnr,':g');
    plot (SESTSSpsnr,'-g');
    plot (TSSpsnr,'-k');
    plot (ARPSpsnr,'-r');
    plot (DSpsnr,'-db');
    plot (CDSpsnr,'-dy');
    plot (SCDSpsnr,'-dm');
    plot (NCDSpsnr,'-dc');

    title([imageName,' sequence    Block Size ',num2str(mbSize),'x',num2str(mbSize),'   Search Parameter p =
',num2str(p)]);
    xlabel([imageName,' Frame Number']);
    ylabel('PSNR (dB)');

legend({'ESpsnr','NTSSpsnr','SS4psnr','SESTSScomputations','TSSpsnr','ARPSpsnr','DSpsnr','CDSpsnr','SCDSps
nr','NCDSpsnr'},'FontSize',7);

    hold off;
    saveas(gcf,[imageName,' PSNR of the motcomp image w.r.t. original.jpg'])
     %*************************************************************************
```

stats.m - code to calculate some statistics such as *Average Searching Points*, *Average PSNR Speed Improvement Ratio*, and the *PSNR difference* (w.r.t. Diamond Search) for the sequence overall.

```
%**********Stats: Average Searching Points, Average PSNR, Speed Improvement Ratio********
%%%% Average Searching Points for selected Fast BMAs %%%%
ES_Av_Comp = mean(EScomputations)                %1.
TSS_Av_Comp = mean(TSScomputations)              %2.
NTSS_Av_Comp = mean(NTSScomputations)            %3.
SS4_Av_Comp = mean(SS4computations)              %4.
SESTSS_Av_Comp = mean(SESTSScomputations)        %5.
DS_Av_Comp = mean(DScomputations)                %6.
CDS_Av_Comp = mean(CDScomputations)              %7.
SCDS_Av_Comp = mean(SCDScomputations)            %8.
NCDS_Av_Comp = mean(NCDScomputations)            %9.
ARPS_Av_Comp = mean(ARPScomputations)            %10.

%%%% Average PSNR for selected Fast BMAs %%%%
ES_Av_PSNR = mean(ESpsnr)                        %1.
TSS_Av_PSNR = mean(TSSpsnr)                      %2.
NTSS_Av_PSNR = mean(NTSSpsnr)                    %3.
SS4_Av_PSNR = mean(SS4psnr)                      %4.
SESTSS_Av_PSNR = mean(SESTSSpsnr)                %5.
DS_Av_PSNR = mean(DSpsnr)                        %6.
CDS_Av_PSNR = mean(CDSpsnr)                      %7.
SCDS_Av_PSNR = mean(SCDSpsnr)                    %8.
NCDS_Av_PSNR = mean(NCDSpsnr)                    %9.
ARPS_Av_PSNR = mean(ARPSpsnr)                    %10.

%%%% Average Speed Improvement Ratio (%) over ES %%%%
SirCDS_ES = (ES_Av_Comp - CDS_Av_Comp)*100/ ES_Av_Comp
SirSCDS_ES = (ES_Av_Comp - SCDS_Av_Comp)*100/ ES_Av_Comp
SirNCDS_ES = (ES_Av_Comp - NCDS_Av_Comp)*100/ ES_Av_Comp
SirARPS_ES = (ES_Av_Comp - ARPS_Av_Comp)*100/ ES_Av_Comp

%%%% Average Speed Improvement Ratio (%) over DS %%%%
SirCDS_DS = (DS_Av_Comp - CDS_Av_Comp)*100/ DS_Av_Comp
SirSCDS_DS = (DS_Av_Comp - SCDS_Av_Comp)*100/ DS_Av_Comp
SirNCDS_DS = (DS_Av_Comp - NCDS_Av_Comp)*100/ DS_Av_Comp
SirARPS_DS = (DS_Av_Comp - ARPS_Av_Comp)*100/ DS_Av_Comp

%%%% Difference in Average PSNR over DS %%%%
PSNR_Diff_CDS_DS = DS_Av_PSNR - CDS_Av_PSNR
PSNR_Diff_SCDS_DS = DS_Av_PSNR - SCDS_Av_PSNR
PSNR_Diff_NCDS_DS = DS_Av_PSNR - NCDS_Av_PSNR
PSNR_Diff_ARPS_DS = DS_Av_PSNR - ARPS_Av_PSNR
```

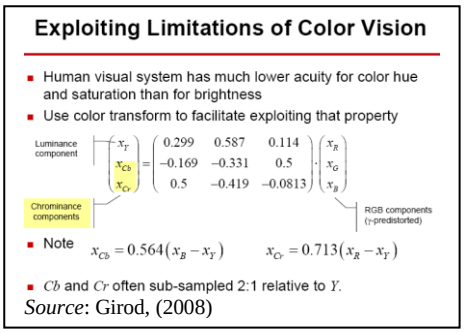
Appendix E: M-Code to convert CIF & QCIF files

Some of the standard video sequences used for algorithm analysis are saved as CIF (Common Intermediate Format) or QCIF (Quarter CIF format) which are in the *.yuv* file format. These need to be converted to usable images for input to the various block search algorithms. In the conversion the C_b and C_r components are suppressed while the Y (luma) component is retained. The motion vectors are typically estimated from the luma component only. The frames are saved to disk in Sun Rasterfile format (.ras) which is an uncompressed greyscale format. These are later called for processing by the *motionsEstAnalysis.m* main script. The MATLAB code used is adapted from Amer (2008) and is as follows:

readYUV.m Conversion of CIF/QCIF Files to Usable Images

```
%% This script reads a 4:2:0 yuv video file and converts it into RGB frames
% The tunable parameters are:
%
% * |*filename*|: a string specifying the yuv filename;
%
% * |*width*|: specifies the width of the frame;
%
% * |*height*|: specifies the height of the frame;
%
% * |*num*|: the number of frames to be read;
%
% * |*start*|: the number of frame at which we start reading from the file
%               (assuming the first frame is 0)

clear, close all;

filename = 'mobile_qcif.yuv';
width = 352/2;
height = 288/2;
num = 32;
start = 0;

%% First, open the video file for binary reading
fid=fopen(filename,'r');
if (fid < 0)
   error('File does not exist!');
end

%% Pre-allocate temp variables for performance boost
tmpY = zeros(width, height); % rows then columns
tmpUV = zeros(width/2, height/2); % 1/4 the number of elements in Y
frmSize = numel(tmpY) + 2*numel(tmpUV); % frame size equals the total number of elements


%% Seek the video file till we reach the starting frame
fseek(fid, start * frmSize , 'bof');

%% Define the output YUV components
Y=cell(num,1); % a vector of arrays, each array corresponds to one frame of Y component
U=cell(num,1); % a vector of arrays, each array corresponds to one frame of U component
V=cell(num,1); % a vector of arrays, each array corresponds to one frame of V component

%% Read the binary values from the file into the vectors of frames
```

```
% The assumption here is that the file format is 4:2:0
for i=1:num
    % this automatically reads the 8-bit binary values into the matrix
    % starting with filling the first column then continues in a column
    % order. That's why transposing the matrix is necessary.

    tmpY = fread(fid,[width height],'uint8'); % the final dimensions of tmpY are width(rows) x
height(columns)
    Y{i} = tmpY'; % transposing and casting to uint8 for imshow() to work correctly

    tmpUV = fread(fid,[width/2 height/2],'uint8');
    U{i} = tmpUV'; % transposing and casting to uint8 for imshow() to work correctly

    tmpUV = fread(fid,[width/2 height/2],'uint8');
    V{i} = tmpUV'; % transposing and casting to uint8 for imshow() to work correctly
end

% we're done reading, so close the file
fclose(fid);


%%% Perform the YUV-to-RGB transfromation
%%% Define the output RGB cells
vRGB=cell(num,1);

%%% Define the forward (RGB-to-YUV) transformation matrix as for YPrPb
% (see http://en.wikipedia.org/wiki/YCbCr)
rgb2yuvT = [0.299 0.587 0.114; -0.168736 -0.331264 0.5; 0.5 -0.418688 -0.081312];

%%% Get the inverse (YUV-to-RGB) transormation matrix
yuv2rgbT = inv(rgb2yuvT);

%%% Pre-allocate the temp variables
dY = zeros(height, width);
dU = zeros(height, width);
dV = zeros(height, width);

% the variable to hold the final rgb values for the frame
% rgb(:,:,1) will hold the Red component
% rgb(:,:,2) will hold the Green component
% rgb(:,:,3) will hold the Blue component
rgb = zeros(height, width, 3);

%%% Iterate through all the frames
% The calculations will be done using double float numbers. After the
% transformation is done, the results will be scaled and quantized to 8-bit
% unsigned format again.

for i=1:num;
    % Convert the class of Y{i} to double instead of uint8 for better
    % precision during conversion
    dY = double(Y{i});

    % Convert the class of U{i} to double instead of uint8 for better
    % precision during conversion, and then perform bilinear interpolation
    %dU = imresize(double(U{i}), 2, 'bilinear');    % i suppress the color component here
```



Source: Girod, (2008)

```
    % Convert the class of U{i} to double instead of uint8 for better
    % precision during conversion, and then perform bilinear interpolation
    %dV = imresize(double(V{i}), 2, 'bilinear');    % i suppress the color component here

    % Shift the values of U and V down by 128 since we do not use uint8 anymore
    dY = dY -16; %dU=dU-128; %dV=dV-128;

    % Perform the transformation
    rgb(:,:,1) = yuv2rgbT(1,1) * dY + yuv2rgbT(1,2) * dU + yuv2rgbT(1,3) * dV; % Red
    rgb(:,:,2) = yuv2rgbT(2,1) * dY + yuv2rgbT(2,2) * dU + yuv2rgbT(2,3) * dV; % Green
    rgb(:,:,3) = yuv2rgbT(3,1) * dY + yuv2rgbT(3,2) * dU + yuv2rgbT(3,3) * dV; % Blue

    % Return to the unit8 class
    rgb = uint8(rgb);

    % Assign the frame to the vector of frames
    vRGB{i} = rgb;
end



%% Save the images into ras files, if desired
% The frames will be saved to ras files if the |*sv*| flag is set to one

% (save the images)-flag
sv=1;

% desired images filename_prefix (assumed to be the same as the video filename
fp = filename;

if(sv==1)
    for j=1:num
        fname=strcat(fp,'_','frame_',num2str(start+j),'.ras');
        imwrite(vRGB{j},fname); % the format is determined to be ras from the filename extension
    end
end



%% Plot the images, if desired
% The frames will be plotted in MATLAB if the |*pt*| flag is set to one

% (plot the images)-flag
pt=0;

if(pt==1)
    for k=1:num
        figure,
        figTitle=strcat('File:', fp,' - ','Frame No.',num2str(start+k));
        imshow(vRGB{k}), title(figTitle),
    end
end
```