多媒體通訊

期末報告

快速動作估計演算法

Fast Block Motion Estimation

資工所 **P76924194** 蘇琬婷

Fast Block Motion Estimation Comparison

P76924194 蘇琬婷

Ⅰ. Introduction

動作估計**(Motion Estimation)**在任何的視訊處理系統**(video processing system)**中都是一個重要的議題。以二維的動作估計而言，其廣泛的應用，包括**visual compressing, sampling rate conversion, filtering** 等等。對每種不同的應用來說，動作估計的方法可以非常的不同。

以視訊壓縮**(visual compressing)**爲例，所估計出來的動作向量**(motion vectors)**被用在動作補償預測**(motion compensation prediction)**編碼上，可以使編碼動作向量的所有位元數縮到最小，並減低預測錯誤**(prediction error)**。所以，動作補償編碼的成功率，取決於動作估計。只要動作估計做的好，誤差影像就可以小到足以忽略。**(**如下圖 **1** 所示**)**

許多視訊壓縮標準**(such as MPEG-1/2/4 and ITU-T H.261/262/263/264)**都使用方塊動作估計**(block motion estimation)**來降低畫面間累贅**(temporal redundancy)**。而編碼器的複雜度大部分可說是由動作估計來決定。若是使用 **Full Search(FS)**，則在參考畫面**(reference frame)**中搜尋範圍內的每個方塊，都必須依序與目前畫面的候選方塊做比較，找出最小的**distortion**。雖然正確率較高，但相對的，搜尋時間與計算複雜度也跟著提升。而 **MPEG** 與 **ITU** 都沒有在編碼器中定義動作估計的演算法。所以後來有不少研究提出一些依經驗法則降低計算複雜度的動作預估演算法，可能是將比對的位置減少，或是在計算每二個巨方塊的 **distortion** 時，使用較少的像素差。在此，將簡介一些快速動作預估演算法，並實作部分演算法做比較。

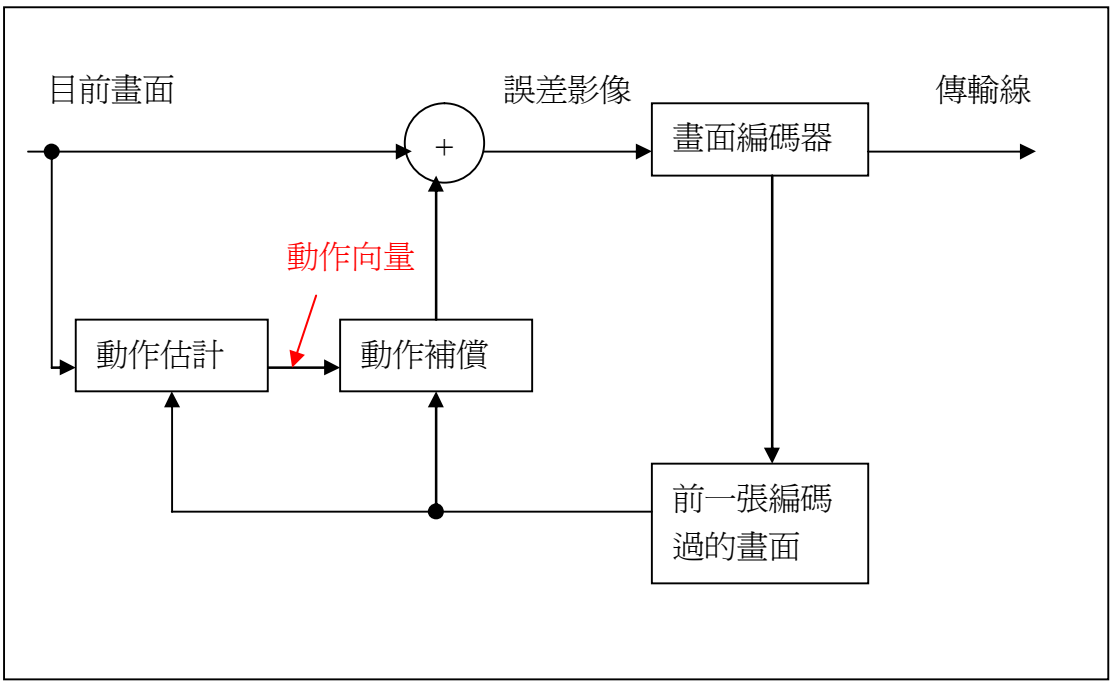


圖 **1.** 動作補償編碼方塊圖

Ⅱ. Motion Estimation

(i) Motion Models Definition and Notation

當場景中的攝影機或物體在移動時，**3D** 物體所投影到的影像也會跟著改變。**(**下圖 **2** 表示出其相對關係**)**

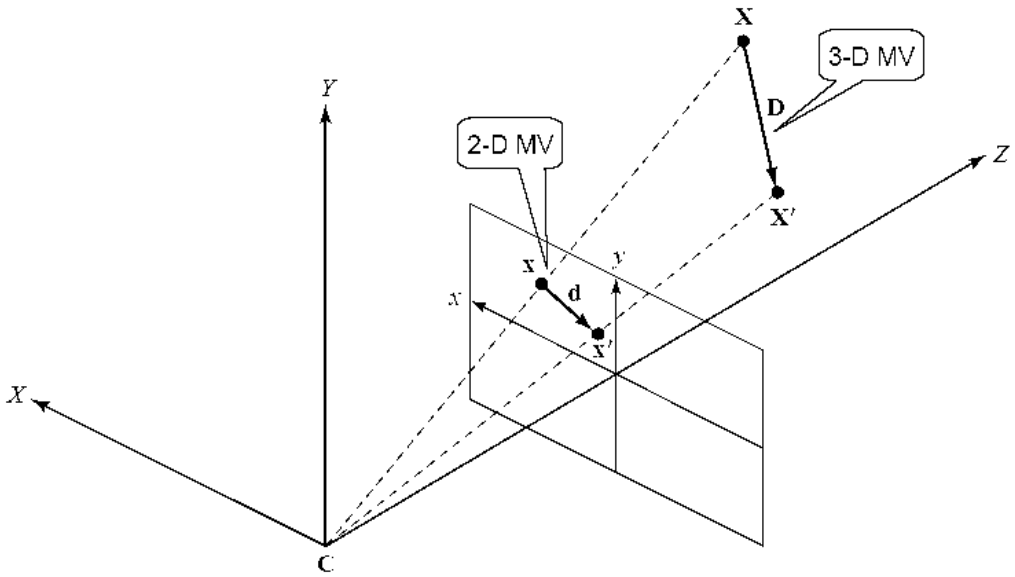


圖 **2. 3** 維和 **2** 維動作向量間的關係

當一物體在時間點 t_1 位於 $\mathbf{X} = [X, Y, Z]^T$ 的某一點於時間點 t_2 **(**其中 $t_2 = t_1 + d_t$**)**時移動至 $\mathbf{X}' = [X', Y', Z'] = [X+D_X, Y+D_Y, Z+D_Z]^T$，它所投影的 **2D** 影像則從 $\mathbf{x} = [x, y]^T$ 移動至點 $\mathbf{x}' = [x', y']^T = [x+d_x, y+d_y]$。在此，我們定義 **3D** 的位移 $\mathbf{D}(\mathbf{X}; t_1, t_2) = \mathbf{X}' - \mathbf{X} = [D_X, D_Y, D_Z]^T$ 爲在點 **X** 的 **3** 維動作向量，而 **2D** 的位移 $\mathbf{d}(\mathbf{x}; t_1, t_2) = \mathbf{x}' - \mathbf{x} = [d_x, d_y]^T$ 爲所對應平面影像點 **x** 的 **2** 維動作向量。我們以 $\mathbf{d}(\mathbf{x}; t_1, t_2)$表示影像中點 **x** 在時間 t_1 到 t_2 的 **2D motion field**。以下簡寫爲 **d(x)**。

爲了方便起見，時間點 t_1 的點 **x** 在時間點 t_2 的相對位置，我們以 $\mathbf{w}(\mathbf{x}; t_1, t_2) = \mathbf{x}'$ 表示，稱爲 **mapping function**，明顯的，**w(x) = x + d(x)**。以下

簡稱爲 **w(x)**。

在影片中，**x** 表示 **pixel index**，而基本的 **2D motion field** 通常以一張 **vector graph** 顯示**(**如下圖 **3)**，其中箭頭方向表示動作的方向，箭頭大小 **(magnitude)**表示動作的大小**(magnitude)**。

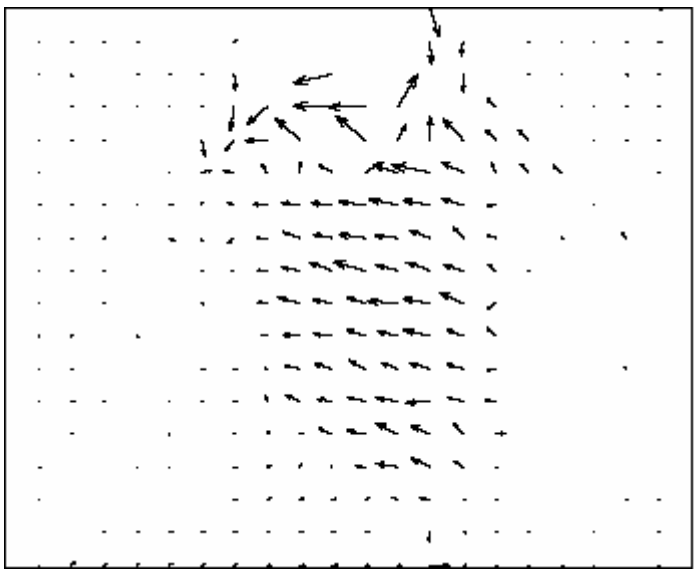


圖 **3. 2D motion field**

除了可以用在一段給定的時間間隔中實際的位移表示動作，我們通常用速度向量**(velocity vector)**來取代。一般速度向量被視爲 **flow vectors**，定義爲 $v = \frac{\partial d}{\partial t} = \left[\frac{\partial d_x}{\partial t}, \frac{\partial d_y}{\partial t}\right]^T$，當時間間隔 $\mathbf{d_t}$ 夠小，在這段時間間隔內的動作我們可視爲常數，也就是 $\mathbf{v = d/d_t}$。

在處理影片的動作向量時，通常將 $\mathbf{d_t}$ 定爲 **1**，所以 **flow vectors** 就是動作向量**(motion vectors)**，我們定義整張影像的 **flow field** 爲 $\mathbf{v(x;\ t_1,\ t_2)}$。

(ii) Optical Flow Equation in Motion Estimation

將一段影片的 **luminance** 變化用 Ψ**(x, y, t)**表示。假設在時間 **t** 影像中的

點**(x, y)**在時間 **t+d_t** 位移到點**(x+d_x, y+d_y)**。依照移動點的 **luminance** 不會改變的假設**(constant intensity assumption)**，則在不同時間，相同物體的點將會有相同的 **luminance** 值。

ð $\Psi(x+d_x, y+d_y, t+d_t) = \Psi(x, y, t)$

由 **Taylor's expansion** 且 **d_x, d_y, d_t** 爲極小值

ð $\Psi(x+d_x, y+d_y, t+d_t) = \Psi(x, y, t) + \frac{\partial y}{\partial x} d_x + \frac{\partial y}{\partial y} d_y + \frac{\partial y}{\partial t} d_t$

結合上述二式

ð $\frac{\partial y}{\partial x} d_x + \frac{\partial y}{\partial y} d_y + \frac{\partial y}{\partial t} d_t = 0$

同除以 **d_t**

ð $\frac{\partial y}{\partial x} v_x + \frac{\partial y}{\partial y} v_y + \frac{\partial y}{\partial t} = 0$

其中**(v_x, v_y)**表示速度向量**(**也叫 **flow vector)**，上式稱做 **optical flow equation**。

在影片中，我們所考慮的爲二張畫面**(frames)**的動作估計，分別記作 **Ψ(x, y, t_1)**和 **Ψ(x, y, t_2)**。先前我們定義點 **x** 在時間 **t_1** 到 **t_2** 的動作向量爲這點從 **t_1** 到 **t_2** 的位移 **d(x; t_1, t_2)**，**(**因爲 **v=d/d_t** 其中 **d_t=1)**，在此我們將在時間 **t_1** 的畫面稱爲 **anchor frame**，而在時間 **t_2** 的畫面稱爲 **target frame**。一般而言，**t_1<t_2**，但事實上可以擴充到 **t_1<>t_2(**例如 **B frames)**。當 **t_1<t_2** 時，稱爲向前動作估計**(forward motion estimation)**；當 **t_1>t_2** 時，稱爲向後動作估計**(backward motion estimation)(**如下圖 **4** 所示**)**。我們以 **Ψ_1(x)**和 **Ψ_2(x)**分別表示 **anchor frame** 和 **target frame**，而 **motion field** 表示爲 **d(x; a)**，其中 **a = [a_1, a_2, ..., a_L]^T**，表示所有動作參數的向量。

動作估計主要就是估計動作參數向量 **a** 的一個問題，總共分爲二類。一爲 **feature-based**，一爲 **intensity-based**。其中，**intensity-based** 假設每個像素**(pixel)**爲 **constant intensity**，在此，我們只考慮 **intensity-based**。

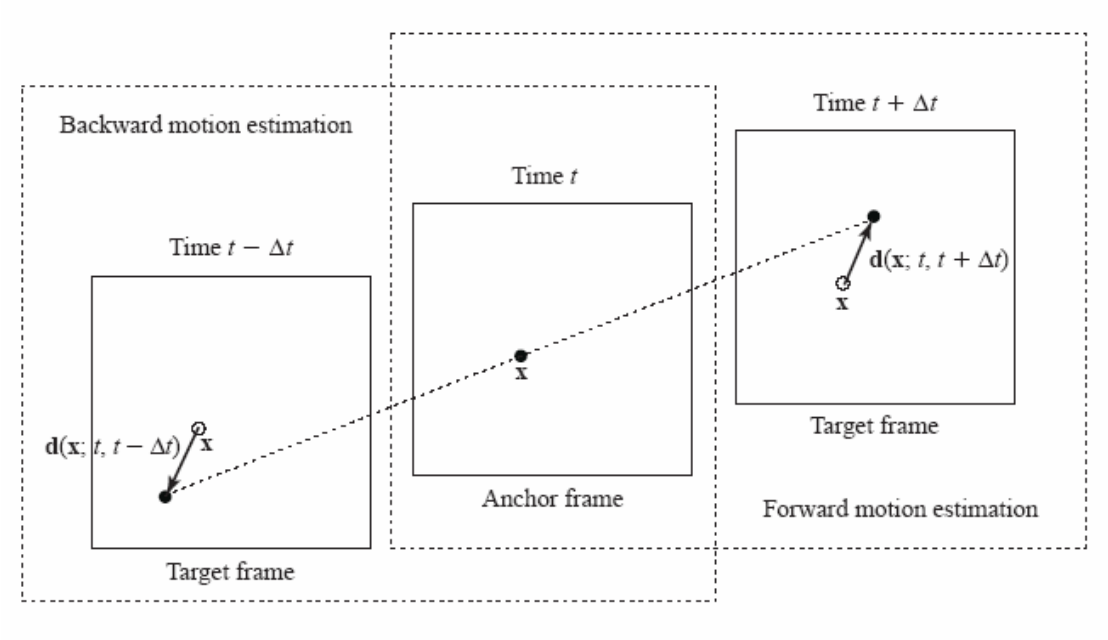


圖 **4.** 向前及向後動作估計

(iii) Motion Representation

- **global motion representation**

整張影像的 **motion field** 用一組動作參數來表示，適用於只有攝影機移動或整張影像的景只有單一個移動的物體。**(**如下圖 **5.a)**

- **pixel-based representation**

每個像素有一個動作向量，但很明顯的，**anchor frame** 中的每個像素都可以在 **target frame** 中找到很多與它相同的 **intensity** 值，所以不能單獨每個像素下去找，必須要有一些 **smoothness** 的限制，使一個新的像素的動作向量，只由周圍像素的動作向量求出。**(**如下圖 **5.b)**

- **block-based representation**

整張畫面切割成許多小方塊**(blocks)**，當這些方塊夠小時，每個方塊的動作變異值可以用一組動作參數來表示，而每個方塊的動作參數可以獨立估計，所以每個方塊會有一個動作向量，這是現行標準採用的方式。

用 **block-based representation** 的最大缺點，在於沒有在相鄰方塊間強調動作轉變的限制，所以在方塊的邊界**(boundary)**會產生不連續感**(discontinuous)**，而真實的 **motion field** 卻是非常流暢的。**(**如下圖 **5.c)**

- **region-based motion representation**

整張畫面切割成數個區域**(region)**，每個區域對應一個物體或一個子物體**(sub-object)**，每組動作參數對應到一個區域。而這個方法的困難點是在如何判斷哪些像素有相似的動作，所以若要得到較好的結果，必須將切割區域**(segmentation)**和估計動作向量二步驟反覆進行，但這需要較大的計算量，在實際上也許是不可行的。**(**如下圖 **5.d)**

- **mesh-based representation**

若使用 **block-based model**，每個方塊的動作參數都是分別獨立計算，所以在方塊邊緣通常會有不連續性產生，有時會產生混亂。**(**如下圖 **6.a)**解決的方法是改採用 **mesh-based model**。**(**如下圖 **6.b)anchor frame** 由一個 **mesh** 表示，**mesh** 上的每個點有一個動作向量，去 **target frame** 找對應的 **deformed element**，而每個點會影響周圍所有相鄰的區域的動作。**(**如下圖 **7)**

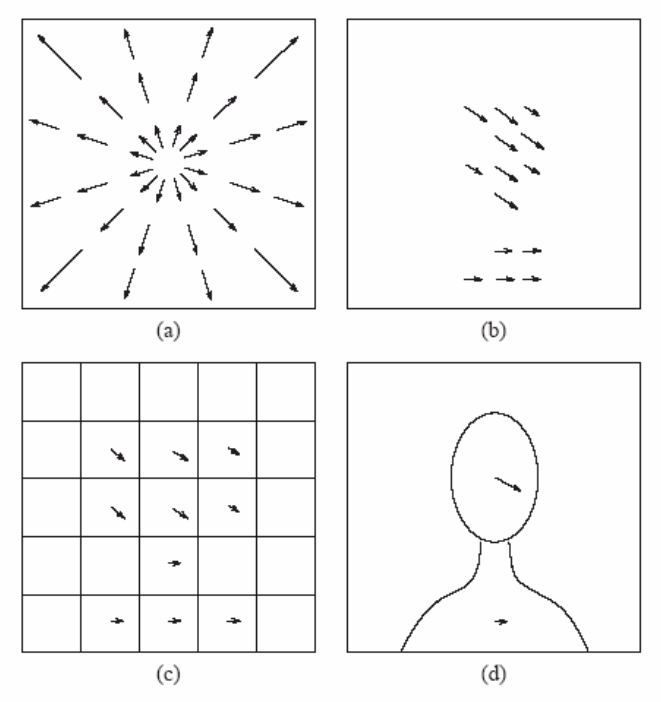


圖 **5.** 不同動作向量表示法

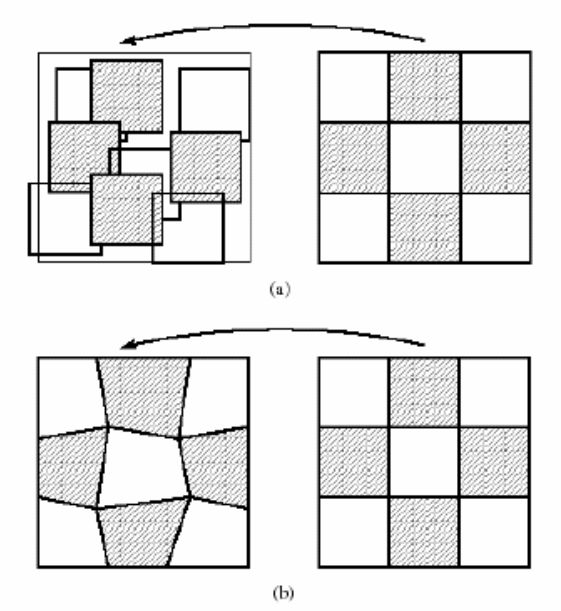


圖 **6. block-based** 和 **mesh-based** 動作表示法的比較

(a) block-based

(b) mesh-based

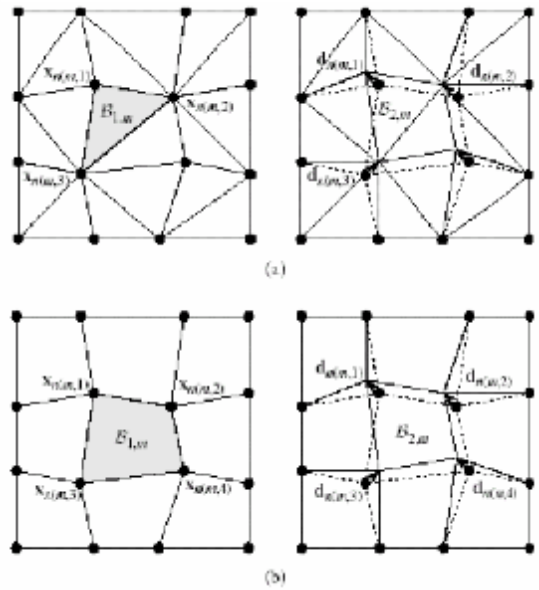


圖 **7. mesh-based** 動作表示法

(a) triangular mesh

(b) quadrilateral mesh

(iv) Motion Estimation Criteria

一般動作估計的標準是算 **anchor frame** 和 **target frame** 對應的每個點的 **luminance** 差值的總和。

當在 **anchor frame** Ψ_1 點 **x** 在 **target frame** Ψ_2 移到 **w(x; a)**，則估計的方程式可表示爲

$$E_{DFD}(a) = \sum_{x \in \Lambda} | y_2(w(x;a)) - y_1(x) |^p$$

其中 Λ 爲在 Ψ_1 中的所有像素集合，而 **p** 爲一正整數。

當 **p=1** 時，上式稱爲平均絕對差**(mean absolute difference(MAD))**。

當 **p=2** 時，上式稱爲最小平方差**(mean squared error(MSE))**。

Ⅲ. Block Motion Estimation

pixel-based 的動作估計最大的問題是必須強調相鄰像素的連續性 **(smoothness)**。而解決這個問題的一個方法就是將畫面切割成許多沒有重疊的小區域，在此我們稱爲方塊**(block)**，並假設每個方塊可以用一個簡單的動作參數表示。

令 $\mathbf{B_m}$ 表示第 **m** 個方塊，**M** 爲在畫面中的所有方塊數，$M = \{1, 2, \ldots, M\}$

$$\Rightarrow \bigcup_{m \in M} B_m = \Lambda \quad \text{且} \quad B_m \cap B_n = f, m \neq n$$

以最簡單的情況而言，每個方塊的動作向量假設爲常數，也就是整個方塊爲一個簡單的轉移**(translation)**。在此只考慮這種情況，每個方塊去找一個單獨的動作向量，這種方法稱爲 **block-matching algorithm(BMA)**。

(i) Exhaustive (Full) Block-Matching Algorithm

給定一個在 **anchor frame** $\mathbf{B_m}$ 的影像方塊做動作向量估計的問題，是去決定在 **target frame** 中所對應到的方塊 $\mathbf{B_m'}$，使得這二個方塊間的誤差值爲最小。在二方塊間的位移向量 $\mathbf{d_m}$ 就是這方塊的動作向量。其中 $\mathbf{w(x; a) = x + d_m}$，誤差值可表示如下：

$$E(d_m, \forall m \in M) = \sum_{m \in M} \sum_{x \in B_m} |y_2(x + d_m) - y_1(x)|^p$$

因爲對每個方塊所估計出來的動作向量都只會影響到該方塊的預估誤差值 **(prediction error)**，所以我們可以分別對每個方塊估計其動作向量，將每個方塊所累積的誤差值求最小：

$$E_m(d_m) = \sum_{x \in B_m} |y_2(x + d_m) - y_1(x)|^p$$

而求得有最小誤差值的 $\mathbf{d_m}$，其中一種算法爲完全搜尋**(exhaustive search / full search)**。**(**如下圖 **8)**每一個在 **anchor frame** 中的方塊 $\mathbf{B_m}$，與 **target frame** 中，每一個在事先定義的搜尋視窗內的方塊 $\mathbf{B_m'}$做比較，算出誤差值，找出最小誤差的對應方塊，在這二個方塊間的位移就是預估的動作向量。

一般而言，爲了減低運算量，通常採用平均絕對差**(p=1)**來做運算。搜尋視窗的大小通常以要做比對的方塊爲中心，對稱的往上下左右分別擴大 R_x, R_y 個像素，如果水平與垂直方向的搜尋範圍大小一致，則 $R_x=R_y=R$。

預估的正確率是由搜尋的 **step size** 來決定，最簡單的情況是採整數像素精確度搜尋**(integer-pel accuracy search)**，**step size** 爲 **1** 像素。

令方塊大小爲 **N*N** 個像素，搜尋範圍爲±**R** 個像素，當 **step size** 爲 **1** 像素時，每個方塊有**(2R+1)**個搜尋位置，每一個搜尋位置必須比對 **N*N** 個像素，也就是要做 **N*N** 個 **MAD** 運算**(**包含一個減法，一個絕對值，一個加法**)**，若影像大小爲 **M*M**，則有 M^2/N^2 個方塊，整張畫面全部的運算爲 $M^2(2R+1)$，與方塊大小並沒有關係。

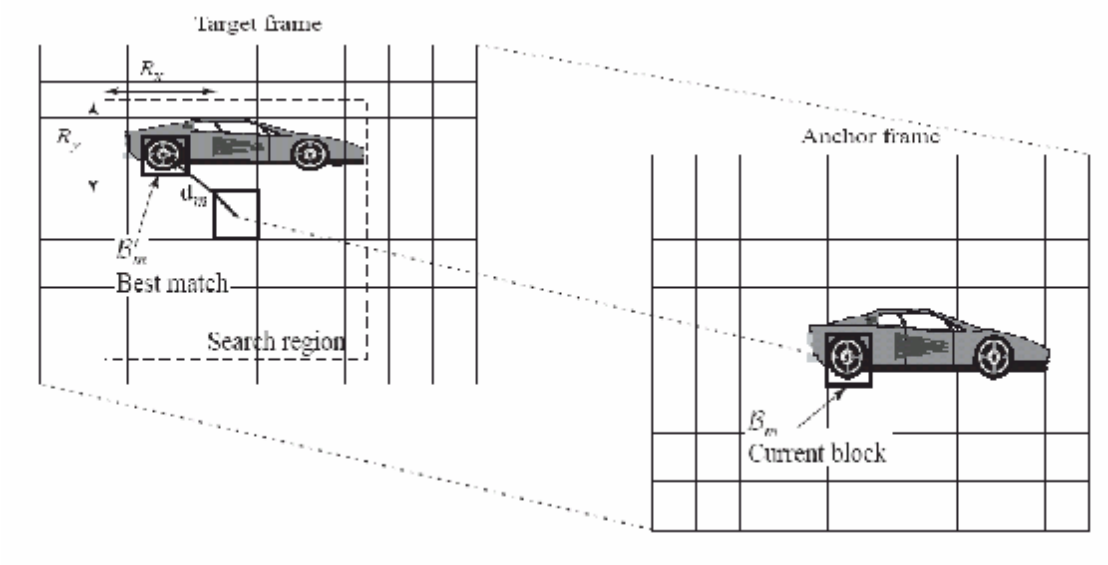


圖 8. 完全搜尋法的搜尋程序

(ii) Fractional Accuracy Search

實際上，畫面與畫面的位移跟像素的取樣位置並沒有關係，因此如果用更高解析度來做動作向量估計，應該會得到更好的預測，也就是使用分數或半像素精確度**(half-pel accuracy search)**來決定動作向量。

用半像素精確度做動作向量估計，必須先將 **anchor frame** 與 **target frame** 做內插以產生半像素，然後再以先前所介紹的方法求動作向量，而這樣

的缺點是需要大量的記憶體，所以一般將它分爲二個步驟。**(**如下圖 **9)**

首先在第一步的時候，先找出整數像素精確度的動作向量**(**如下圖 **9** 中填滿的點**)**，接著將第一步的結果做微調產生半像素精確度的動作向量。

在找出整數像素精確度的動作向量後，圍繞它的四圍，由內插定出八個新的搜尋位置，**(**如下圖 **9** 中空心的點**)**，計算 **anchor frame** 中的方塊與這八個半像素位置的方塊的 **MAD** 值，與原整數像素精確度的 **MAD** 值，找出最小 **MAD** 值的動作向量即爲所求。

不只限定在 **1/2** 像素，**MPEG4** 的標準更將其擴充到 **1/4** 像素。

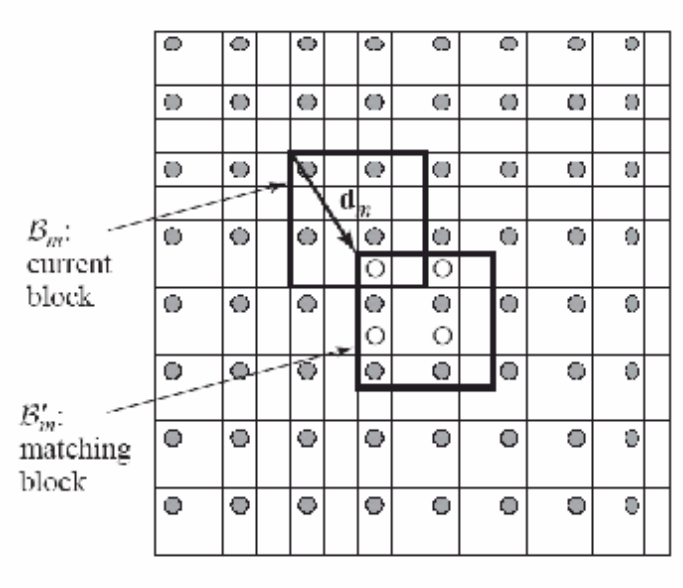


圖 **9.** 半像素精確度的動作估計

(iii) Fast Algorithm

(1) 2-D-Log Search

2-D-Log 在編號爲 **1** 的五個點代表第一次做比對的五個方塊位置，這些位置之間的間隔相同而且均勻的分佈於搜尋視窗內。假設第一次的五個位置在比對後，使 **MAD** 最小的位置是發生在**(i, j+2)**，那麼第二次的比對便是圍在這個點上方的三個鄰居位置**(**如圖 **10** 中編號爲 **2** 的點**)**。計算這三個編號 **2** 位置的 **MAD** 值，再與前一個步驟中 **MAD** 值最小的點做比較，找出最小的一個，假設爲編號 **2** 中上方的一點**(i, j+4)**。下一步的搜尋則是圍在

這個點位置上方的三個編號 **3** 的點，加上這個編號 **2** 的點本身。在每一步驟的搜尋，我們都將搜尋視窗內可能的範圍去掉一大塊。以圖 **10** 爲例，第一次搜尋確定了匹配不在下半平面，第二次搜尋則確定了匹配也不在上半平面之下半平面，如此則每個步驟都固定去掉所剩範圍之 **1/2**。重覆這個搜尋的過程直到所剩下之搜尋點已經很少，然後對這些搜尋點採取完全搜尋的方式找出最後的動作向量。

以圖 **10** 爲例，共需要五個步驟，也就是 **18** 次的比對。若以完全搜尋，則需 **13x13=169** 次的比對。由於它的搜尋範圍呈指數函數的縮小**(**第 **k** 步驟後只剩下 $1/2^k$ 的搜尋範圍**)**，只需對數函數個步驟便可完成搜尋的工作。

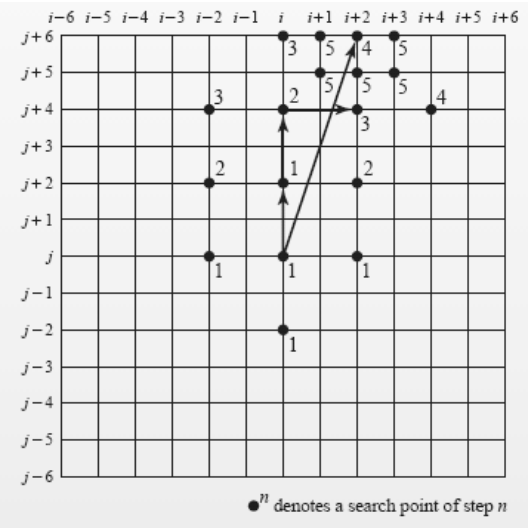


圖 **10. 2-D-Log Search**

(2) Three Step Search (TSS)

Three-Step 在編號爲 **1** 的 **9** 個點代表第一步驟做比對的九個方塊位置，這個位置之間的間隔爲 **3**，而且均勻的分佈在搜尋視窗內。假設第一步的比對結果，**MAD** 最小的點在**(i+3, i+3)**，則第二步的比對爲圍繞在這個點四周的八個編號 **2** 的點，如圖 **11** 所示，其間隔減爲 **2**。類似地，第三步之比對位置則是間隔爲 **1** 的編號 **3** 的點。整個演算法固定需要三個步驟，所以稱爲三步驟搜尋。和完全搜尋需要 **169** 次比對相較下，三步驟搜尋只需要 **25** 次比對。

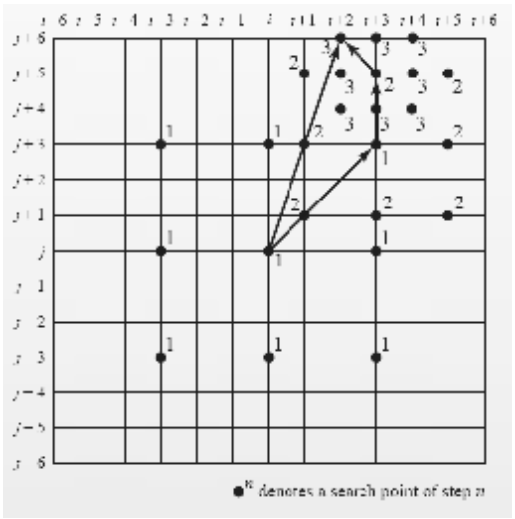


圖 **11. Three Step Search**

(3) Four Step Search (4SS)

Four-Step 是改良 **Three-Step** 的方法，在第一步用一個 **5*5** 的視窗取代在 **Three-Step** 中 **9*9** 的視窗，搜尋視窗的中心移到最小 **MAD** 值的方塊位置，下一步的視窗大小**(window size)**及增加的點的個數取決於有最小 **MAD** 值方塊的位置，若此方塊位於搜尋視窗的中心，則搜尋視窗大小縮減爲 **3*3**，此爲最後一次搜尋。

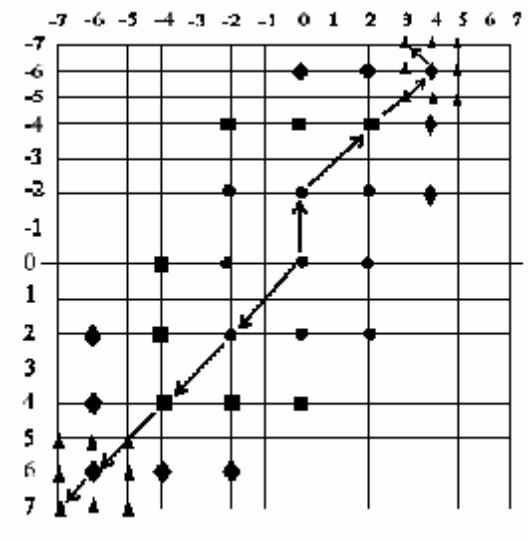


圖 **12. Four-Step Search** 的二種路徑

(4) Block-Based Gradient Descent Search (BBGDS)

BBGDS 用一個 **3*3** 的視窗，內含九個點。開始的視窗建構在中心爲原點的位置，找出差距最小的點，當下一個視窗的中心，找到差距最小的點

是現在視窗的中心時搜尋就停止了。

由實驗數據上來看，**FSS** 和 **BBGDS** 明顯在效能**(performance)**上比 **TSS** 高。而 **FSS** 的效能比 **BBGDS** 稍微高出一些，但計算也比 **BBGDS** 複雜許多。**FSS** 比較適用於找大的動作，而 **BBGDS** 比較適用於找小的動作。

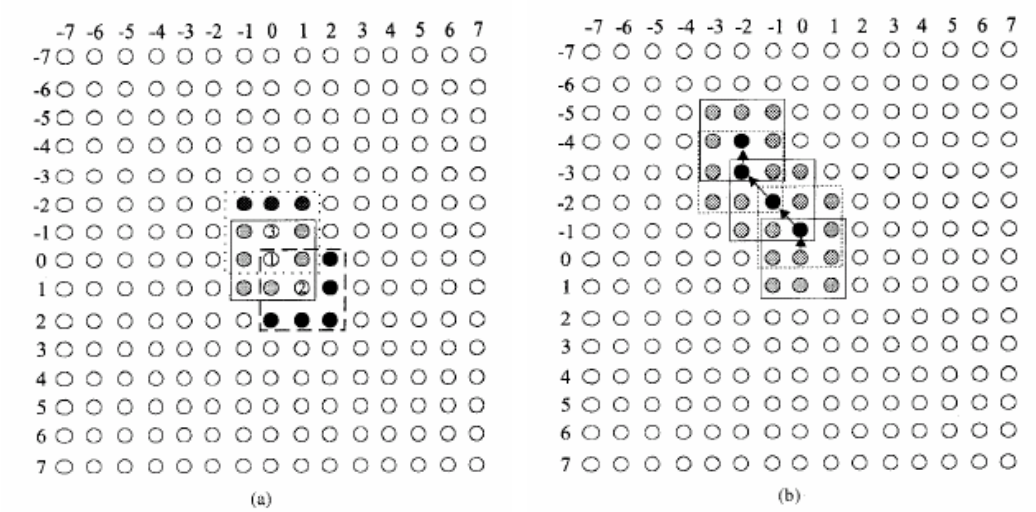


圖 **13. Block-Based Gradient Descent Search**

(5) Predictive Motion Vector Field Adaptive Search Technique (PMVFAST)

一般 **Block-matched Motion Estimation Algorithm** 假設方塊的動作集中在中心點**(0,0)**，造成某些圖片的估算失真。

PMVFAST 假設時間上、空間中鄰近方塊的動作向量有相關性，設計一預測器**(predictor)**，將這些鄰近點的動作向量與中心點**(0,0)**存入，作爲候選動作向量值，用以預測下一個方塊的動作向量。同時設計一門檻**(threshold)**值，依動作移動量分級，不同級用不同的動作向量當作跑 **Diamond Search Algorithm** 的中心點。**(**簡介於**(9))**

比較蒐集的論文顯示，此法有最佳的圖形表現，但由於實做步驟較爲複雜，故最後沒有採用。

(6) Hierarchical Block Motion Estimation (HBM)

HBM 是高解析度**(multi-resolution)**的一個特殊狀況，它將影像分層，由解析度最低的一層開始往高解析度的底層找尋動作向量，因此限制了各個動作向量的搜尋範圍來減少運算量。

HBM 的總運算次數大約爲 $\frac{1}{3}4^{-(L-2)}4M^2R^2$ ，**R** 是在最大解析度下的搜尋範圍，**L** 是分層的層數，而影像大小是 **MxM**。而 **EBMA** 的總運算量是 $M^2(2R+1)^2 \approx 4M^2R^2$ ，也就是說 **HBM** 比 **EBMA** 減少了 $3\times4^{L-2}$ 倍的運算量。通常取 **L** 爲 **2** 或 **3**。

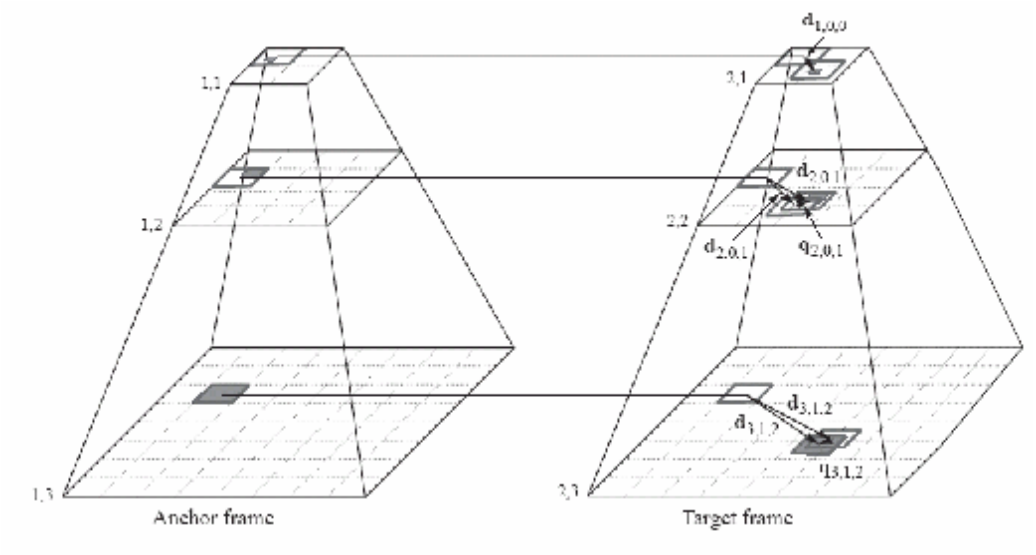


圖 **14. Hierarchical Block Motion Estimation**

(7) Adaptive Motion Tracking Search

AMT 爲一以中心爲基準的演算法**(central biased algorithm)**，實做上較 **TSS**、**4SS**、**N3SS** 等同爲以中心爲基準的演算法，有較佳的運算速度，但由於 **AMT** 爲以中心爲基準的演算法，當動作不集中在圖形中央時，動作預估的圖形表現會比較差。

(8) Diamond Search

Diamond Search 假設動作向量是大致上偏向中央的。因此這個演算法都從中央開始，找周圍九個點來比較差距。找到最相近的點以後再把菱形中心移到那一點。如果最相近的點就是最中央的點，則把菱形縮小找周圍四個點比較。大致上的步驟如下圖。

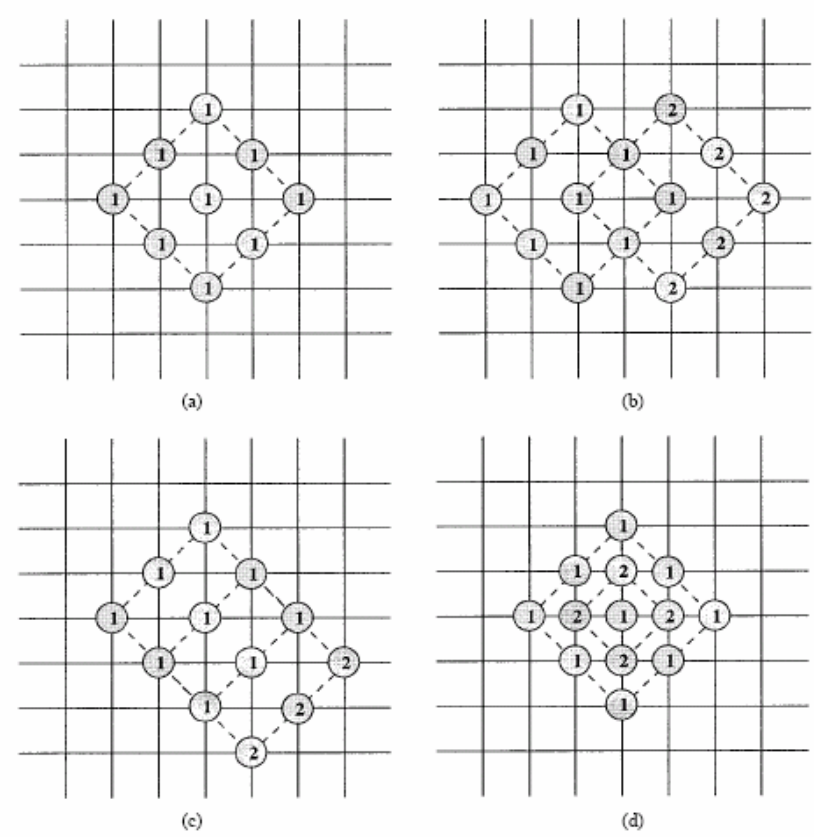


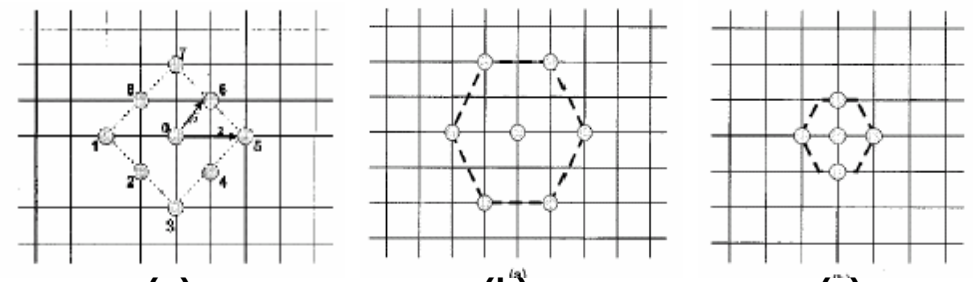
圖 **15. Diamond Search**

Diamond search 與 **EBMA** 相較之下大幅減少了運算量，同時維持了相當的效果。例如 **4SS** 中平均是 **21~19** 個搜尋點，**Diamond search** 可以達到平均 **13** 個搜尋點。

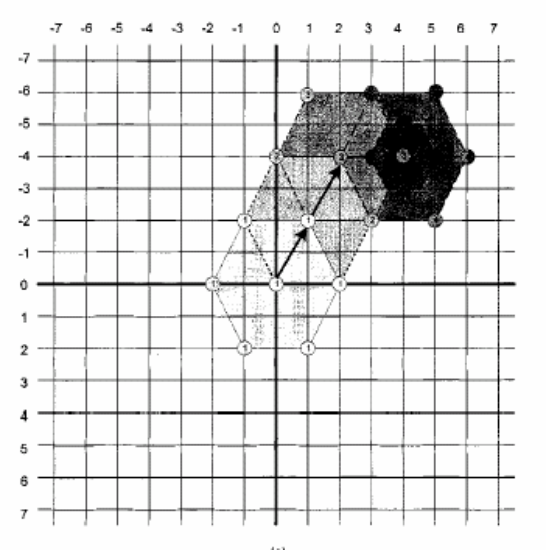
(9) Hexagon-Based Search(HEXBS)

Hexagon-based search 主要是改進 **diamond search**，因爲 **diamond search** 的每個點距離中心的位置不同，分別爲**(2, √2)**。並不

是逼近圓形，而越近的點算出來的誤差值可能會較小**(**如下圖 **16.a)**，因此這個演算法都從中央開始，找周圍七個點來比較差距，而水平二個點距中心爲 **2**，上下各四個點距中心爲√**5**，比 **diamond search** 更接近圓形。找到最相近的點以後再把六邊形中心移到那一點。如果最相近的點就是最中央的點，則把六邊形縮小找周圍四個點比較。大致上的步驟如下圖 **16.d**。



(a) **(b)** **(c)**



(d)

圖 **16.**
(a) diamond search 的範圍
(b) HEXBS 大的範圍
(c) HEXBS 小的範圍
(d) Hexagon-Based Search

每個方塊所需搜尋的點爲 $N_{HEXBS}(m_x, m_y) = 7 + 3 \times n + 4$，其中，**n** 爲執行大範圍的次數。

比較 **diamond search** 所需的點數 $N_{DS}(m_x, m_y) = 9 + M \times n' + 4$，其中 **M** 爲 **5** 或 **3**，依搜尋的方向而決定，**n′**依搜尋的距離而決定。

Ⅳ. Implementation

在此實作了 **Full Search**，**Diamond Search**，以及 **Hexagon-Based Search**。定搜尋視窗的大小爲 **31*31(R_x=R_y=15)**，巨方塊大小爲 **16*16**。

演算法及流程圖：

(a) FS

將初始的搜尋範圍中心放置在現在要找尋方塊 **B_m**，在 **target frame** 中相對應的位置，一次移一個像素，比對在搜尋範圍中每個方塊與現在方塊 **B_m** 的 **MAD** 值，找有最小 **MAD** 的方塊，二方塊的位移即爲動作向量。

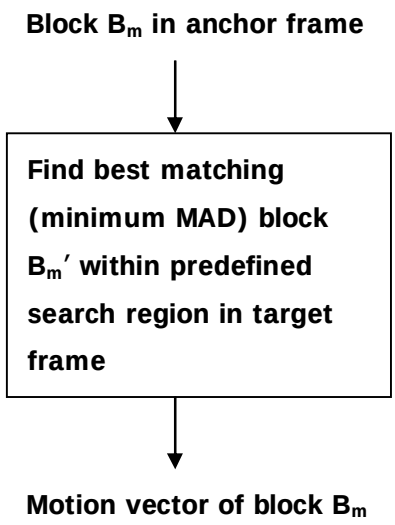


圖 **17. FS** 流程圖

(b) DS:

Step1: 將初始的菱形搜尋範圍放置到搜尋視窗的中心，計算 **9** 個候選的搜尋點與在 **anchor frame** 中的方塊 **B_m** 做比較，若最小 **MAD** 值的點位在菱形的中心點，則 **step3**，否則做 **step2**。

Step2: 如果最小 **MAD** 值的點落在菱形的四個角上**(vertices)**，則往下做 **Vertex Search(**圖 **15.b)**，如果落在菱形四個邊的中間點**(faces)**，則往下做 **Face Search(**圖 **15.c)**。

-Vertex Search: 以最小 **MAD** 值的點爲中心，爲一個新的菱形**(**更新中心點**)**，會有五個新的候選點產生。**(**圖 **15.b)**

-Face Search: 以最小 **MAD** 值的點爲中心，爲一個新的菱形**(**更新中心點，會有三個新的候選點產生。**(**圖 **15.c)**

[只要在先前算過的候選點都可以忽略。**]**重新算新的候選點和 **anchor frame** 中的方塊 $\mathbf{B_m}$ 的 **MAD** 值，如果最小 **MAD** 值落在新的菱形的中心點，則做 **Step3**，否則做 **Step2**。

Step3: 以最小 **MAD** 值的點爲中心，將菱形縮小爲 **step size** 等於 **1**，會有四個新的候選點**(**上圖 **15.d)**，**(**同樣的，在先前算過的候選點都可以忽略**)**，重新算新的候選點和 **anchor frame** 中的方塊 $\mathbf{B_m}$ 的 **MAD** 值，最小 **MAD** 值的方塊與 $\mathbf{B_m}$ 的距離就是估計出的動作向量。

回到 **Step1** 做下一個巨方塊的估計。

(c) HEXBS:

步驟與 **DS** 相同，只需將菱形改成六邊形，而 **step2** 中每次會新增 **3** 個候選點**(**不管最小 **MAD** 值的點發生在 **face** 或 **vertex)**，在 **step3** 中最後將六邊形縮成 **step size** 爲 **1**，會有四個新的候選點。**(**上圖 **16)**其餘步驟皆相同。

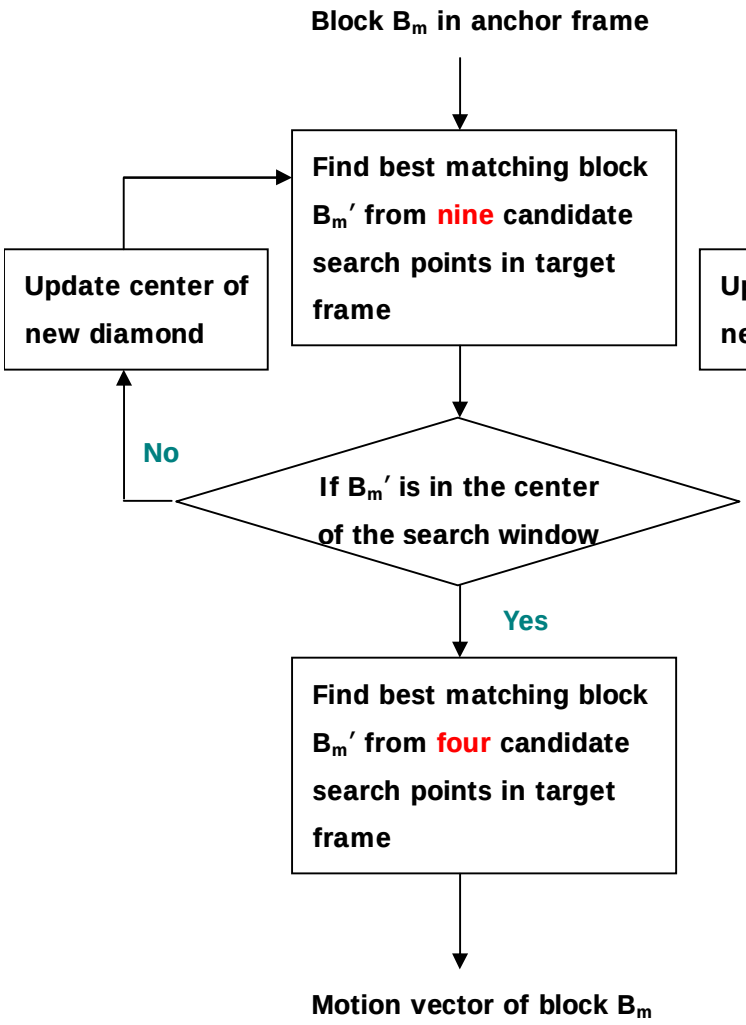


圖 **18. DS** 流程圖

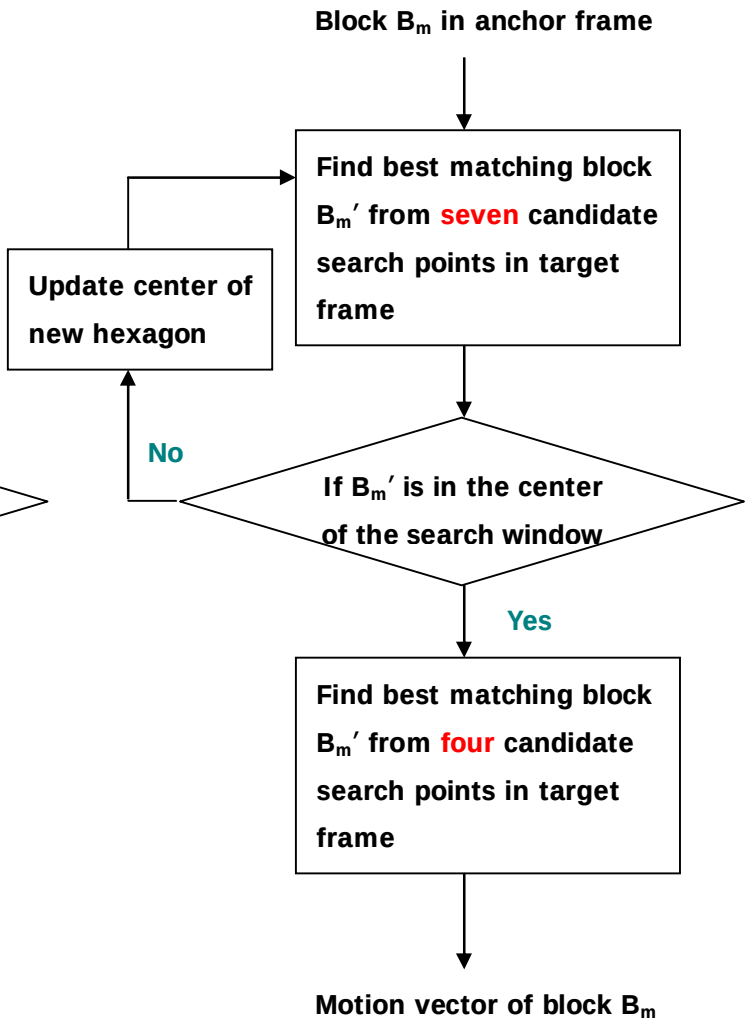
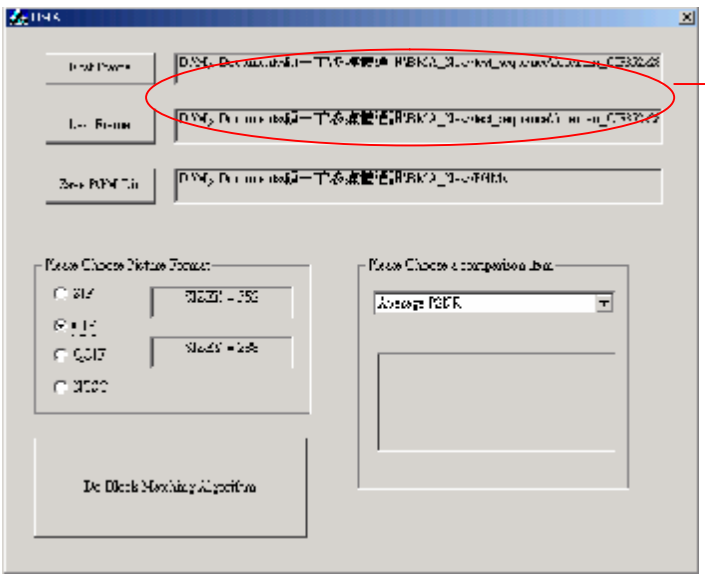
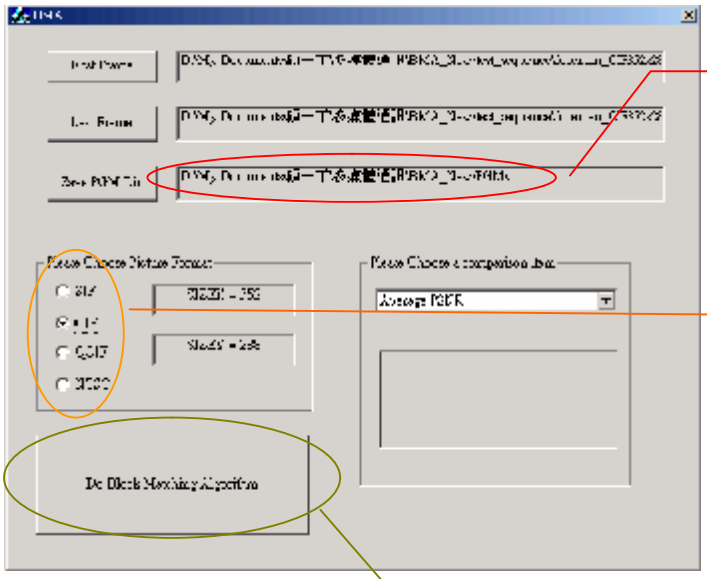


圖 **19. HEXBS** 流程圖

程式執行方式：



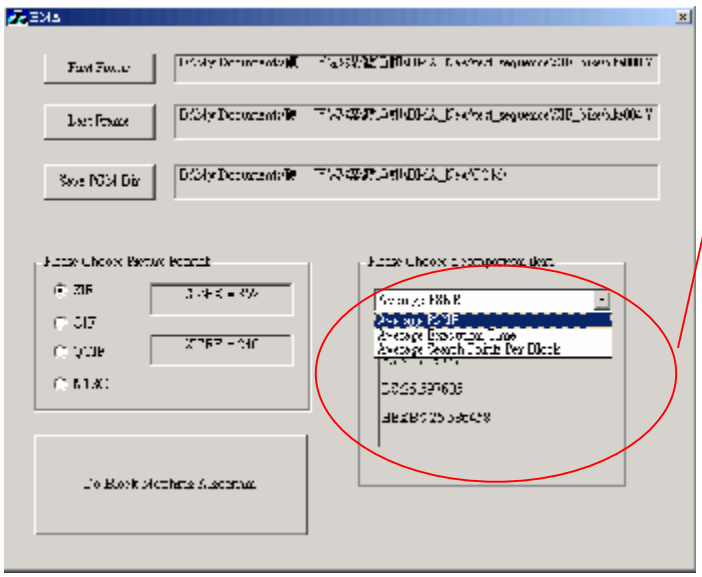
選要做動作向量估計的第一張和最後一張畫面



選最後預估完的畫面(*.PGM)存放的位置

選影片格式(以便知道畫面大小)

執行 **FS, DS, HEXBS**



可以選擇顯示三種演算法的結果比較(平均 PSNR 值，平均執行時間，平均一個巨方塊計算的方塊個數)

程式解說：

void OnFirstidx(), void CBMADlg::OnLastidx()

紀錄分別由使用者輸入第一張和最後一張做動作向量估計的畫面**(**例如：**bike000.Y... bike100.Y)**，檔案名稱爲有順序之**.Y** 檔。

OnSif(), OnCif(), OnQcif(),OnNtsc()

由使用者選擇畫面的格式，以便得知畫面的大小。**(SIF**：**SIZEX=352 SIZEY=240, CIF**：**SIZEX=352 SIZEY=288, QCIF**：**SIZEX=176**

SIZEY=144, NTSC：SIZEX=352 SIZEY=240)

void OnPGMDir()

選擇存放最後輸出檔的資料夾，在此存放預估的畫面影像檔**(*.PGM)**。

void OnDoBMA()

一開始先將第一張畫面讀進 **yuv_buffer**，將其複製到 **ref_buffer(reference buffer)**，再讀第二張畫面進 **yuv_buffer**，依序將 **ref_buffer** 和 **yuv_buffer** 丟進 **FullSearch function**、**DiamonSearch function** 和 **HexagonSearch function** 求出動作向量和 **enc_buffer**，並計算每張畫面的 **PSNR** 值、所花的執行時間及每個巨方塊所需計算的方塊數。

int read_YUV_frame(unsigned char *yuv_buffer, const char *Dir, const char *Prefix, const int frno)

將**.Y** 檔**(**在此只處理**.Y** 檔，而不處理 **chrominance UV)**讀到 **yuv_buffer**。

int YUV2PGM(unsigned char *yuv_buffer, char *filename)

將存有 **luminance(*.Y)**的 **buffer** 寫成影像檔**(*.PGM)**。

float framePSNR(unsigned char *pix,unsigned char *src)

計算影像的 **PSNR** 值。

int FullSearch(unsigned char *ref_buffer, unsigned char *yuv_buffer, unsigned char *enc_buffer, struct MV *mv)

搜尋範圍爲±**15** 個像素，將 **ref_buffer** 中的每個巨方塊依序與 **yuv_buffer** 中在搜尋範圍內的巨方塊做比較，算 **MSE** 值，找出最小 **MSE** 值的巨方塊，二個方塊的距離即爲動作向量。

將 **ref_buffer** 加上預估出來的動作向量，存進 **enc_buffer**，此爲預估的畫面。

int DiamondSearch(int bidx, unsigned char *ref_buffer, unsigned char *yuv_buffer, struct MV *mv)

一次算一個巨方塊，當最小 **MSE** 值不是出現在菱形中心時，維持 **step size** 等於 **2**，做 **CompLarge(...)**，迴圈一直持續到最小 **MSE** 出現在中心，**step size**

縮減爲 **1**，只找四個點，做 **CompFinal()**。

int HexagonSearch(int bidx, unsigned char *ref_buffer, unsigned char *yuv_buffer, struct MV *mv)

類同於 **DamondSearch**。

int caldist(int bx, int by, int x, int y, unsigned char *ref_buffer, unsigned char *yuv_buffer)

計算二方塊間的距離，在此採 **MSE** 值**(p=2)**。

int CompLarge(char PType, int bidx, int cmpidx, int *min_value, unsigned char *ref_buffer, unsigned char *yuv_buffer, BlockMatrix SP)

PType 表示 **pattern type**，若傳入**'D'**表示做 **Diamond Search**，若傳入**'H'**表示做 **Hexagon Search**。

比較在搜尋範圍內的候選點與現在在比較的方塊的 **MSE** 值，傳回最小 **MSE** 的方塊的位置。

int CompFinal(char PType, int bidx, int cmpidx, int *min_value, unsigned char *ref_buffer, unsigned char *yuv_buffer, struct MV *mv)

在 **DS** 和 **HEXBS** 的最後一個步驟，比較搜尋範圍內的四個候選點的 **MSE** 值，將二個方塊的距離存入動作向量。

Ⅴ. Experimental result

	bike	clarie	football	foreman	sales
Frame #	000-100	100-109	100-109	100-109	000-100
Size	SIF	SIF	SIF	CIF	QCIF

<附表**>** 各影片執行畫面數及影像大小

	bike	clarie	football	foreman	sales
FS	34.736622	41.759171	23.327349	35.028156	40.027382
DS	30.194513	37.691204	20.165565	30.832695	39.124695
HEXBS	30.160852	37.691204	20.019016	30.788321	39.124508

表 **1.** 平均 **PSNR** 值**(dB) (**搜尋視窗大小爲±**15)**

	bike	clarie	football	foreman	sales
DS	-4.542109	-4.067967	-3.161784	-4.195461	-0.902687
HEXBS	-4.57577	-4.067967	-3.308333	-4.239835	-0.902874

表 **2.** 與 **FS** 的平均 **PSNR** 值差**(**搜尋視窗大小爲±**15)**

	bike	clarie	football	foreman	sales
FS	4053.770020	4094.444336	4307.222168	4989.555664	1146.260010
DS	93.529999	88.000000	101.222221	106.777779	21.910000
HEXBS	81.510002	89.222221	113.666664	89.000000	20.030001

表 **3.** 平均執行時間**(ms) (**搜尋視窗大小爲±**15)**

	bike	clarie	football	foreman	sales
DS	0.02307	0.02149	0.02350	0.02140	0.01911
HEXBS	0.02011	0.02179	0.02638	0.01783	0.01747

表 **4.** 與 **FS** 的平均執行時間差**(**倍**)(**搜尋視窗大小爲±**15)**

	bike	clarie	football	foreman	sales
FS	1024	1024	1024	1024	1024
DS	13.742788	13.114141	16.268349	13.078002	12.020597
HEXBS	11.347814	10.906397	13.092930	11.034793	10.060607

表 **5.** 每個方塊平均搜尋點數**(blocks) (**搜尋視窗大小爲±**15)**

	bike	clarie	football	foreman	sales
DS	-1010.257212	-1010.885859	-1007.731651	-1010.921998	-1011.979403
HEXBS	-1012.652186	-1013.093603	-1010.90707	-1012.965207	-1013.939393

表 **6.** 與 **FS** 的平均搜尋點數差**(**搜尋視窗大小爲±**15)**

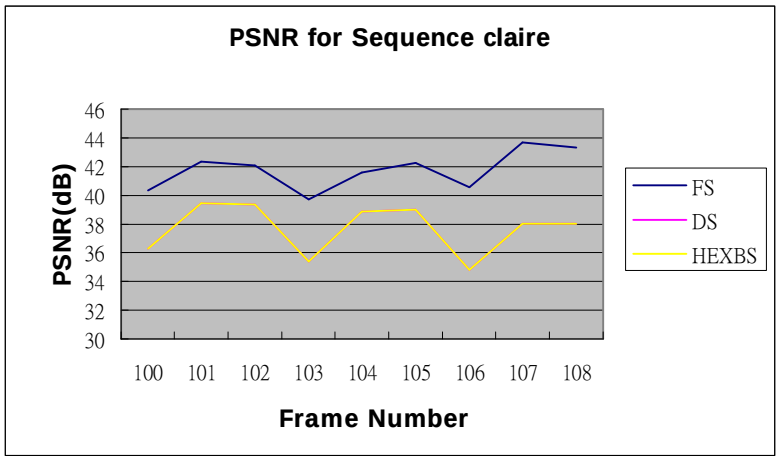


圖 **20. PSNR of claire**

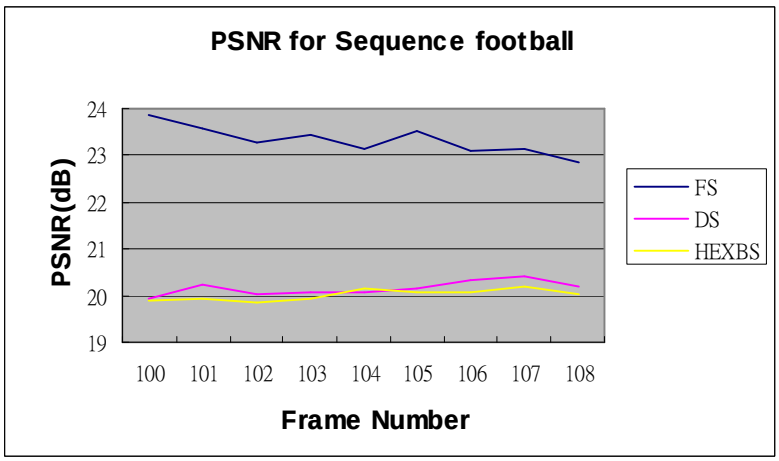


圖 **21. PSNR of football**

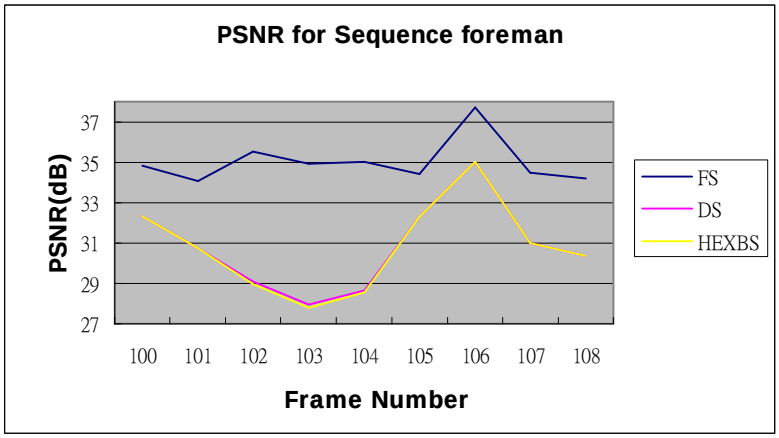


圖 22. PSNR of foreman

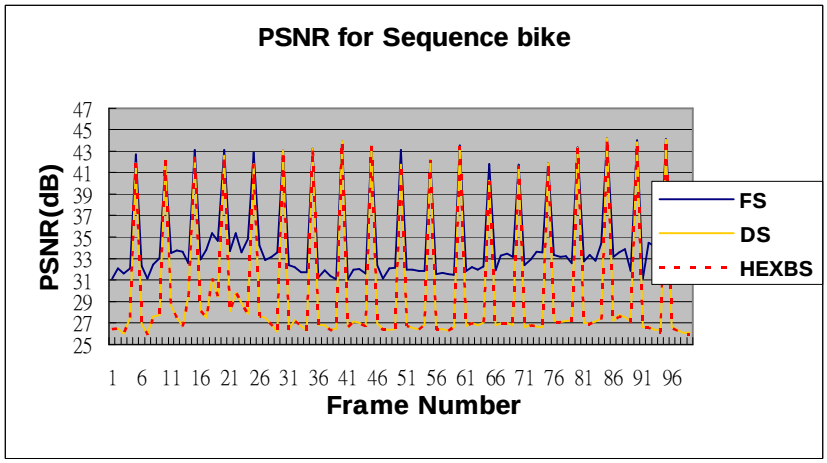


圖 23. PSNR of bike

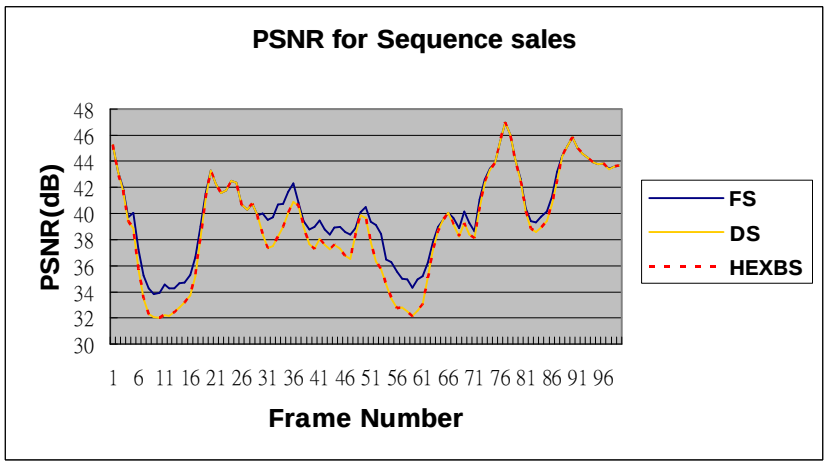


圖 24. PSNR of sales

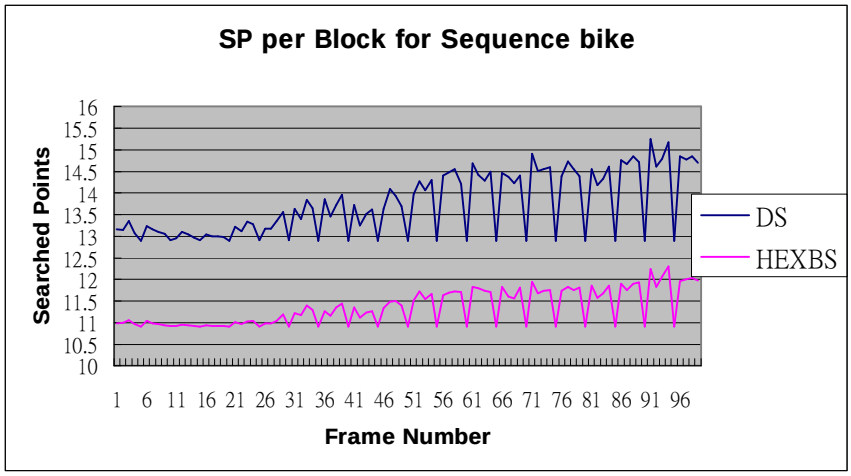


圖 **25. SPs/block of bike**

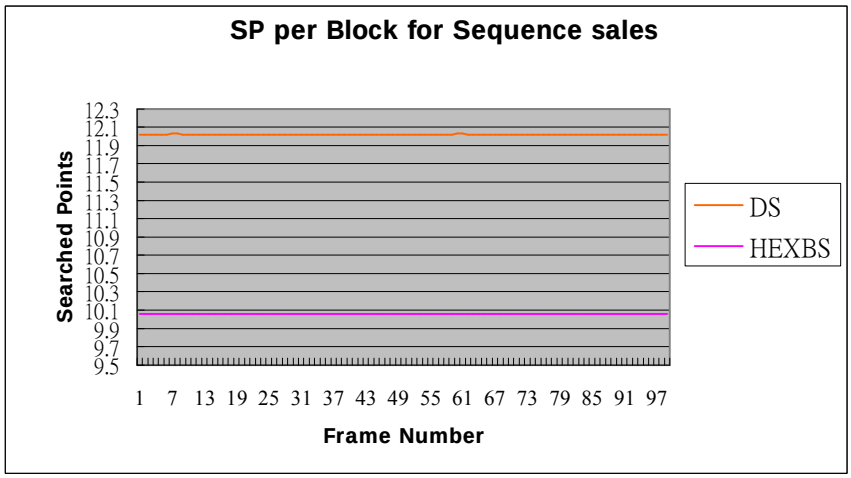


圖 **26. SPs/block of sales**

<註**>**在上圖表示每個方塊所需搜尋點數，僅繪出較長的二段影片做比較。

Ⅵ. Conclusion

由實驗結果可知，**Hexagon-Based Search** 在影像的精確度上**(PSNR** 值**)**，與 **Diamond Search** 結果差不多，而在 **HEXBS** 的論文中並沒有提出這方面的比較。但在執行時間上與每個方塊所需的搜尋點數上，**HEXBS** 平均比 **DS** 快且搜尋點數較少，**(**平均約少 **1.16-3.17 SP/block)**。

未來可以再加進其他快速計算方法，如 **Predictive Motion Vector**，或是更精簡的差值計算函數。

Ⅶ. Reference

1. 戴顯權，陳瀅如，王春清，〝多媒體通訊〞第二版
2. **Yao Wang, Jorn Ostermann, Ya-Qin Zhang, 〝Video processing and communications〞**
3. **R. Li, B. Zeng, and M. L. Liou, 〝A New Three-Step Search Algorithm for Block Motion Estimation,〞 *IEEE Trans. Circuits Syst. Video Technol.*, Vol. 4, No. 4, pp. 438-442, Aug. 1994.**
4. **L.-M. Po, and W.-C. Ma, 〝A Novel Four-Step Search Algorithm for Fast Block Motion Estimation,〞 *IEEE Trans. Circuits Syst. Video Technol.*, vol. 6, No. 3, pp. 313-317, Jun. 1996.**
5. **L.-K. Liu, and B. Feig, 〝A Block-Based Gradient Descent Search Algorithm for Block Motion Estimation in Video Coding,〞 *IEEE Trans. Circuits Syst. Video Technol*., vol. 6, No. 4, pp. 419-422, Aug. 1996.**
6. **J. Y. Tham, S. Ranganath, M. Ranganath, and A. A. Kassim, 〝A Novel Unrestricted Center-Biased Diamond Search Algorithm for Block Motion Estimation,〞 *IEEE Trans. Circuits Syst. Video Technol.*, vol. 8, No. 4, pp. 369-377, Aug. 1998.**
7. **A. M. Tourapis, O. C. Au, M. L. Liou, 〝Predictive Motion Vector Field Adaptive Search Technique (PMVFAST) – Enhancing Block Based Motion Estimation,〞 in *proceedings of Visual Communications and Image Processing 2001 (VCIP-2001)*, San Jose, CA, January 2001.**
8. **K. Cheung and L. M. Po, "A hierarchical block motion estimation algorithm using partial distortion measure," in *Proceedings of International Conference on Image Processing*, April 1997, vol. 3, pp. 606-609.**
9. **J. B. Xu, L. M. Po and C. K. Cheung, "Adaptive motion tracking block matching algorithms for video coding," *IEEE Trans. Circuits and Syst. Video Technol.*, vol. 97, pp. 1025-1029, Oct. 1999.**
10. **Ce Zhu, Xiao Lin, and Lap-Pui Chau, 〞Hexagon-Based Search Pattern for Fast Block Motion Estimation,〞 *IEEE Trans. Circuits and Syst. Video Techonol.,* vol. 12, NO. 5, May. 2002**