

國立交通大學

資訊學院 資訊學程

碩士論文

以去浮點數實現 GMM 之研究

Investigation of Fixed-Point Implementation of GMM

研究生：黃智偉

指導教授：莊仁輝 教授

中華民國 一 百 年 十 月

以去浮點數實現 GMM 之研究

Investigation of Fixed-Point Implementation of GMM

研究生：黃智偉

Student：Chih-Wei Huang

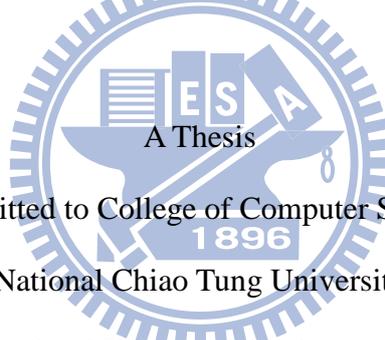
指導教授：莊仁輝

Advisor：Jen-Hui Chuang

國立交通大學

資訊學院 資訊學程

碩士論文



Submitted to College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Computer Science

October 2011

Hsinchu, Taiwan, Republic of China

中華民國 一 百 年 十 月

以去浮點數實現 GMM 之研究

研究生：黃智偉

指導教授：莊仁輝 教授

國立交通大學

資訊學院 資訊學程 碩士班



本論文的主要研究內容，為將 GMM(Gaussian Mixture Modeling)演算法中的參數及其運算過程中的變數去浮點數化，此為演算法硬體化之前或應用在嵌入式系統中所必須做。在 PC 上實作不同資料格式的 GMM 演算法，並將其參數去浮點數化後，比較其去浮點數化前後的影像處理速度、記憶體使用的大小及前景物件辨識的準確度。提出一個參數的資料格式，可以維持高準確度，並有效降低記憶體的使用量。將 GMM 演算法中的參數去浮點數化後，除了可以提高日後硬體化實作開發的速度，並可大量降低記憶體的使用量。

Investigation of Fixed-Point Implementation of GMM

Student : Chih-Wei Huang Advisor : Dr. Jen-Hui Chuang

Degree Program of Computer Science

National Chiao Tung University

Abstract

The goal of this thesis is investigation of Fixed-Point implementation of GMM (Gaussian Mixture Modeling). It must be done before hardware implementation or porting to embedded system. We implement it for several data types of GMM in PC and Fixed-Point parameters of GMM then compare its speed of image processing, memory size and recognition precision of foreground objects with the result after Fixed-Point. We provide a data type of parameters that can keep high precision and down size memory. After Fixed-Point implementation of GMM, it can save developing time and a lot memory when implementing the hardware of GMM or porting GMM to embedded system.

誌 謝

本論文能順利的完成，首先要感謝我的指導教授莊仁輝老師。因為有他這三年來不斷的指導解惑，適時的指導我的研究方向，並包容我在職的身分所造成的許多不便，讓我能利的朝研究目標前進，完成我的碩士論文。接著要感謝的是林泓宏博士，因為論文中主要實作的演算法即是他的研究成果，也就是說沒有他的研究成果，我的論文是必有更多的困難，也因為有他詳盡的解說，才能避免掉許多的錯誤。除此之外，還要感謝博士班的國華學長，以及實驗室的同學們，適時的給予我鼓勵與幫助。

因為我是一個在職進修的研究生，相對於還有需多工作上的壓力，在此非常感謝台灣超微股份有限公司的同仁們。在我進修的期間，給予我工作上的支持，包容我需要上課所造成的困擾，讓我在工作與進修兼顧的情況下，順利完成我的論文。

最後要感謝的是我的老婆貞妤，因為有她默默的在旁陪伴與支持，讓我能學習過程中，在家庭上能無後顧之憂。



目 錄

摘要.....	i
Abstract.....	ii
誌謝.....	iii
目錄.....	iv
圖目錄.....	vi
表格目錄.....	x
一、 緒論.....	1
1.1 研究動機.....	1
1.2 研究背景與相關研究.....	2
1.3 論文架構.....	3
二、 GMM 演算法.....	5
2.1 本論文選擇的 GMM 演算法.....	6
2.2 實作 GMM 演算法.....	9
三、 GMM 演算法實作及去浮點數化.....	11
3.1 實作.....	12
3.2 GMM 演算法驗證.....	13
3.3 GMM 之去浮點數化.....	22
3.3.1 Full 32-bit case.....	23
3.3.2 Partial 32-bit case.....	24
3.3.3 16-bit case.....	27

四、	GMM 演算法實驗結果.....	29
4.1	GMM 去浮點數化之影像分析.....	29
4.2	GMM 去浮點數化之處理速度分析.....	34
4.3	GMM 去浮點數化之記憶體分析.....	46
4.4	GMM 去浮點數化之準確度分析.....	48
五、	結論與未來展望.....	61
六、	參考文獻.....	62



圖 目 錄

圖 1-1(a)傳統監控系統示意圖。(b)連接許多攝影機的監控系統示意圖。	1
圖 2-1 規律晃動的小草，會慢慢更新的背景中，不會造成前景的誤判。	5
圖 2-2。(a)開始的場景圖 (b)物件移走了被偵測成前景 (c)更新後進入 背景，前景不會再偵測到.....	5
圖 2-3 GMM 背景維護更新流程圖。.....	8
圖 2-4 文獻[19]的原始演算法。.....	9
圖 2-5 本論文修改文獻[19]後之演算法。.....	10
圖 3-1 以 GMM 背景模型作前景偵測的基本流程.....	12
圖 3-2 四個相鄰點之 Erosion(兩個例子).....	13
圖 3-3 四個相鄰點之 Dilation.....	13
圖 3-4 Erosion + Dilation 之效果.....	13
圖 3-5 以單一像素作 GMM 程式之基本驗證.....	15
圖 3-6 用以驗證 GMM 程式之一些影像畫面.....	17
圖 3-7 用以驗證 GMM 程式之一些背景畫面.....	17
圖 3-8 Full 32-bit 格式中之 $I_{t,x}$ 、 $\mu_{t,x,n}$ 的資料格式.....	23
圖 3-9 Full 32-bit 格式中之 $Var_{t,x,n}$ 的資料格式.....	24
圖 3-10 Full 32-bit 格式中之 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的資料格式.....	24
圖 3-11 Partial 32-bit 格式中之 $I_{t,x}$ 、 $\mu_{t,x,n}$ 的資料格式.....	25
圖 3-12 Partial 32-bit 格式中之 $Var_{t,x,n}$ 的資料格式.....	25
圖 3-13 Partial 32-bit 格式中之 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的資料格式.....	25
圖 3-14 Partial 32-bit 格式中之 $Var_{t,x,n}$ 格式轉換流程.....	26

圖 3-15 16-bit 格式中之 $I_{t,x}$ 、 $\mu_{t,x,n}$ 的資料格式	27
圖 3-16 16-bit 格式中之 $Var_{t,x,n}$ 的資料格式	27
圖 3-17 16-bit 格式中之 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的資料格式	27
圖 4-1 乘法運算的基本測試迴圈程式碼	34
圖 4-2 各種資料型的乘法運算基本測試態運算程式碼	35
圖 4-3 Check matched Gaussian 函式的複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.	38
圖 4-4 Get weight learning rate 函式的指令複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.	40
圖 4-5 Update Gaussian 函式的指令複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.....	41
圖 4-6 Update background 函式的指令複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.	45
圖 4-7 GMM 程式中之全域變數	46
圖 4-8 GMM 程式中之區域變數	47
圖 4-9 GMM 演算法前景更新過程之示意圖	49
圖 4-10 Object base 演算法之示意圖	50
圖 4-11 以 Object based 作前景 Match Rate 之分析結果	51
圖 4-12 以 Object based 作前景 False Alarm 之分析結果	51
圖 4-13 Test Video 1: Float 格式程式之前景 Match Rate 及 False Alarm 分析結果.....	51
圖 4-14 Test Video 1: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	52
圖 4-15 Test Video 1: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	52

圖 4-16 Test Video 1: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果.....	52
圖 4-17 Test Video 2: Float 格式程式之前景 Match Rate 及 False Alarm 分 析結果.....	53
圖 4-18 Test Video 2: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	53
圖 4-19 Test Video 2: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	53
圖 4-20 Test Video 2: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果.....	54
圖 4-21 Test Video 3: Float 格式程式之前景 Match Rate 及 False Alarm 分 析結果.....	54
圖 4-22 Test Video 3: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	54
圖 4-23 Test Video 3: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	55
圖 4-24 Test Video 3: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果.....	55
圖 4-25 Test Video 4: Float 格式程式之前景 Match Rate 及 False Alarm 分 析結果.....	55
圖 4-26 Test Video 4: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	56
圖 4-27 Test Video 4: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果	56
圖 4-28 Test Video 4: 16-bit 格式程式之前景 Match Rate 及 False Alarm	

分析結果.....	56
圖 4-29 Test Video 使用 1:16-bit 格式(變異數為 10.6 格式)進行前景分析 之 Match Rate 及 False Alarm.....	58
圖 4-30 Test Video 1:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果	59
圖 4-31 Test Video 2:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果	59
圖 4-32 Test Video 3:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果	60
圖 4-33 Test Video 4:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果	60



表格目錄

表格 3-1 手算以單一像素作 GMM 之結果	15
表格 3-2 程式輸出以單一像素作 GMM 之結果	16
表格 3-3 實際影像經 GMM 分析之手算結果	19
表格 3-4 實際影像經 GMM 程式分析之輸出結果	20
表格 3-5 實際影像測試之手算結果與 GMM 程式輸出結果的差值表	21
表格 3-6 GMM 演算法中相關的運算式與 Partial 32-bit 格式中的運算格 式對照表 1.....	26
表格 3-7 GMM 演算法中相關的運算式與 Partial 32-bit 格式中的運算格 式對照表 2.....	26
表格 4-1 Test Video1 輸出影像比對表	30
表格 4-2 Test Video2 輸出影像比對表	31
表格 4-3 Test Video3 輸出影像比對表	32
表格 4-4 Test Video4 輸出影像比對表	33
表格 4-5 以基本乘法計算測試之速度表	36
表格 4-6 Test Video 以 GMM 程式作前景分析之處理速度表	36
表格 4-7 Test Video 以 GMM 程式作前景分析之速度比較表	36
表格 4-8 GMM 涵式複雜度之分析表	37
表格 4-9 GMM 程式中全域變數記憶體分析表	47
表格 4-10 GMM 程式中全域變數記憶體比例之對照表	48
表格 4-11 GMM 程式中記憶體使用之分析表	48
表格 4-12 GMM 演算法更新速度測試結果	49
表格 4-13 Partial 32-bit 格式與 16-bit 格式程式中變數精確度比較表..	58
表格 4-14 Test Video1 程式運算中變異數大於整數最大值次數比較表	58

一、緒論

1.1 研究動機

在監控系統中，許多的管理方式都是將攝影機連接至伺服器，作統一管理，所以伺服器必須負責許多影像處理及數據判斷。若將影像的演算法硬體化至每一台攝影機，則每一台攝影機可以獨立處理其所擷取的影像，甚至直接分析影像處理的結果，最後只需將處理後的結果傳送至伺服器即可，或是等到有需要應變的時候，再通知伺服器，這樣可以大幅的降低傳輸的資料量及伺服器運算的負擔，進而提升伺服器的反應效率，因此可以提升伺服器對攝影機的管控數量。

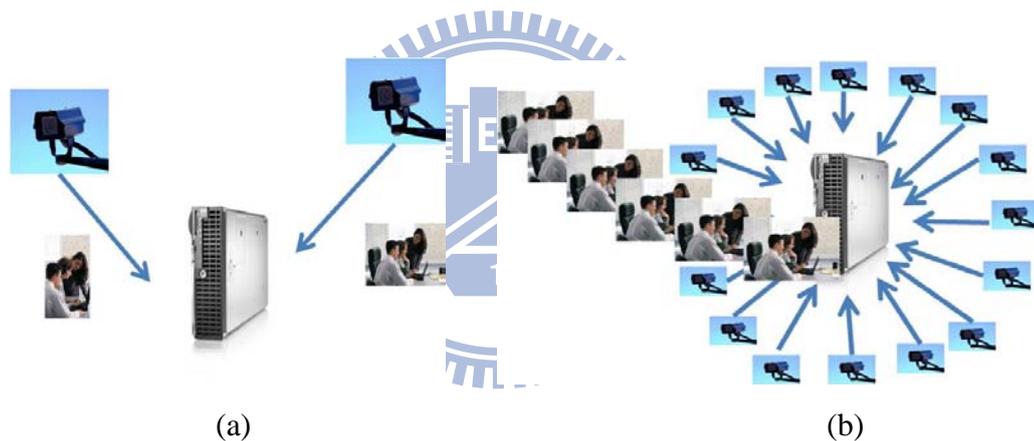


圖 1-1(a)傳統監控系統示意圖。(b)連接許多攝影機的監控系統示意圖。

而將演算法硬體化的過程中，必須做的是對所有參數的去浮點數化(Fixed-Point)。在一般的 PC 上，CPU(Central Processing Unit)中央處理器中通常會包含一個 FPU(Floating Point Unit)浮點數運算器，這個浮點數運算器會專門處理浮點數的資料型態，所以我們可以很輕鬆的在 PC 上使用整數及浮點數的資料型態作編譯及運算。但是在一般嵌入式系統的 CPU 上，通常不會包含有浮點數運算器，所以就能直接處理浮點數的資料型態。所以在硬體化的過程中，就必須將常用的浮點數資料型態轉換成整數的型態，這樣演算法才能在嵌入式系統上順利的運行。而將這個將浮點數轉換成整數資料型態的過程，及稱為去浮點數化。

如果選擇一個常用之的影像處理的演算法，如 GMM (Gaussian Mixture Modeling)，將其內建於攝影機中，將可達到提升伺服器的反應效率的效果。而 GMM 有很多優點，因為 GMM 為一個大家常用的演算法，如背景去除、物體移動偵測，而且可參考的文獻很多。如果要將 GMM 硬體化或實作至嵌入式系統，則必須對 GMM 演算法中的參數作去浮點數化。本論文則對此步驟進行相關研究及討論，以提供一個實作 GMM 去浮點數化的方法。

1.2 研究背景與相關研究

GMM 的相關研究非常的多，尤其是在背景去除、前景分析及物體移動偵測，甚至在膚色辨識以及人臉辨識的領域中，都有以 GMM 為基礎的相關研究。而許多 GMM 的研究文獻中，更是以 GMM 原理在進行改良，而達到更好的效果，應用面非常廣泛。

在背景去除部分，文獻[1]中是以 GMM 為基礎，再加上判斷是否為”有效點”的方法，改良背景去除的步驟，進而提高移動物體偵測的效果。文獻[2]則是以 GMM 結合”PixelMap”來增進背景維護的健全度。文獻[3]，以 GMM 在搭配上”Time Gap”的統計方式，來提高成都市交通影片中，去除背景的可靠度。文獻[4]則是針對 GMM 的”參數”調整做研究，找出何種參數可提提高物體偵測的效果。文獻[5]針對不同的場景如室內、室外，提出不同適應方式的 GMM，提高前景偵測準確率。

在偵測移動物體方面，文獻[6]中，則以 GMM 在加上”Neighborhood-based difference“以及”Overlapping-based Classification”提出以 GMM 為基礎的新演算法，來提高移動物體偵測的準確率。文獻[7]，更是以 GMM 為基礎演算法在再加上”Principal Component Analysis (PCA)”來分析移動物體軌跡，並提高偵測準確率。文獻[8]則是考慮光線變化的場景，以 GMM 為基本演算法進行修改，以提高移動物體偵測的準確率。因此 GMM 幾乎被視為偵測物體移動的中，最基本的一種演算法，也幾乎被視為偵測效果的比較基準。如在文獻[9] -[10]，GMM 皆被當

比較基準，與文獻中的研究方法作比較。

GMM 是偵測物體移動的常用的演算法，所以在其他應用上也常會使用 GMM。在文獻[11]中，作者使用 GMM 為基礎進行資料庫的訓練，找出膚色偵測的參數，進而發展出膚色偵測的演算法。文獻[12]進一步將 GMM 應用在上肢動作辨識。文獻[13]以 GMM 為基礎進行改良，提高多人時的人物追蹤精確度。文獻[14]- [17]則是結合了 GMM 與膚色的應用。文獻[14]中人臉辨識研究，使用 GMM 再搭配“PCA Based Face Recognition”來達到不同角度的人臉辨識效果。文獻[15]更以 GMM 為基礎開發出結合利用膚色偵測人臉的即時行動平台。文獻[16]中細緻的語言辨識系統，則是 GMM 結合語音辨識系統所開發出來的。文獻[17]的手勢辨識研究，也是以 GMM 為基礎所做的研究。

但是上面所提到的文獻大多是在 PC 開發，所以比較少遇到浮點數格式計算的問題，所以不需要作去浮點數化。文獻[18] 中，雖然有在 PDA 實作 GMM，但是卻是應用在語音辨識，與影像處理較不相關。而且在去浮點數化的部分，也只有簡單的描述使用 Q15.16 格式，也就是整數有 15 個位元，小數有 16 個位元，資料格式則是以 32 位元的整數作表示。其中並沒有針對不同的參數如：平均數、變異數或是加權比重等參數作相關的討論及最佳化。本論文是針對上述等各參數，進行精確度的討論，並針對不同的資料格式如 Double、Float、Integer 及 Short 進行比較。希望提供一個 GMM 去浮點數化的效能評估，期能在硬體化或嵌入式系統實作中提供一個參考。

1.3 論文架構

本論文對於實作文獻[19] “Regularized Background Adaptation: A Novel Learning Rate Control Scheme for Gaussian Mixture Modeling” 之 GMM 演算法，所需之去浮點數化，進行分析研究。主要作法是將一實用之 GMM 演算法，在 PC 實作去浮點數化後，對於不同的資料型態提供不同的參數精確度，以進行速度、記憶體大小及準確度的分析，並與未去浮點數化之前的結果作比較，提出一

個參數的資料格式，可以維持高準確度，並有效降低記憶體的使用量。

本論文架構如下:第二章是對 GMM 演算法的一般特性，進行簡單的描述說明，並針對論文中所將實作之上述 GMM 演算法，提供其優點及特性說明，以及演算法初步的分析，最後再提出論文中真正實作完成的部分。第三章則提出一驗證 GMM 演算法之”正確性”的方法，驗證後即根據其參數及運算時之最大值進行資料格式的規畫及去浮點數化，並對各種不同的資料格式進行”精確度”評估，最後將其實作於 PC 上，並以影片進行測試及分析其輸出結果。第四章為實驗結果比對，根據實作的各種資料格式進行處理速度、記憶體大小及”準確度”的比較。最後第五章為論文之總結及未來研究方向。



二、GMM 演算法

Gaussian Mixture Modeling(GMM) 是常用來做背景更新及維護的一種演算法，這個演算法對於複合式的背景，常態性隨風飄動的樹葉場景，或是規律噴水的場景等，均有相當不錯的維護效果。GMM 使用多個高斯分佈來記錄各個不同的背景場景，如在樹葉飄動的背景場景，就可以使用一個高斯分佈來維護有樹葉的背景，再用另一個高斯分佈來維護沒有樹葉的背景場景，這樣當樹葉飄動時，不論是有樹葉或是沒有樹葉的場景，皆可以被視為背景，而不是被誤判成前景。如圖 2-1(a)是第 520 個畫面，圖 2-1(b)是第 528 個畫面，雖然小草持續在晃動，但不會被偵測成前景移動的物體。而對於前景中不動的物體或是消失的物體，也可以隨著時間慢慢的被更新到背景中，不會因為一開始取景的不同，而造成一直的誤判。如圖 2-2(a)是第 1 個畫面，圖 2-2(b)是第 90 個畫面，當物件被移走時會被偵測成前景，圖 2-2(c)是第 506 個畫面，物件以更新至背景中，前景不會再偵測到。



圖 2-1 規律晃動的小草，會慢慢更新的背景中，不會造成前景的誤判。

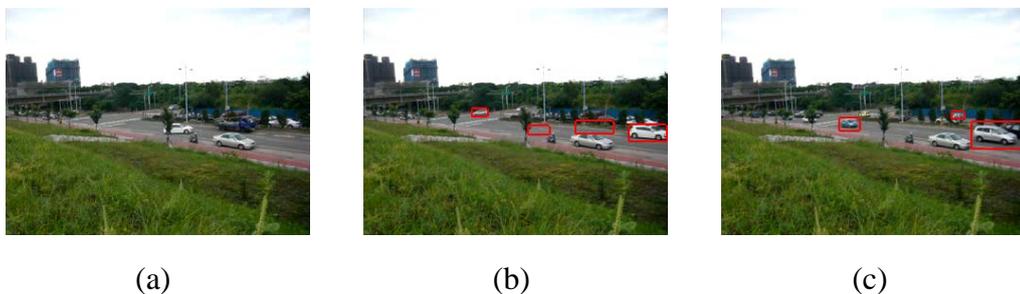


圖 2-2。(a)開始的場景圖 (b)物件移走了被偵測成前景 (c)更新後進入背景，前景不會再偵測到

2.1 本論文選擇的 GMM 演算法

GMM 的演算法非常的多，本論文所採用的演算法為文獻[19] “Regularized Background Adaptation: A Novel Learning Rate Control Scheme for Gaussian Mixture Modeling”。這個演算法對下面的狀況處理效果較好: (1)能夠適應光線快速的變化，(2)有效偵測消失的物件及殘像的抑制。而這個演算法有幾個特點:

(1) 兩種學習速率：一個針對平均數及變異數；另一個針對加權比重。

(2) 對加權比重的學習速率分為 4 種不同的狀況:

接下來說明一般的 GMM 的演算法為：

$$P(I_{t,x}) = \sum_{n=1}^N w_{t-1,x,n} N(I_{t,x}; \mu_{t-1,x,n}, \sigma_{t-1,x,n}^2) \quad (2.1)$$

其中, $w_{t-1,x,n}$ 是加權比重，

$$N(I; \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(I-\mu)^2}{2\sigma^2}\right) \quad (2.2)$$

(2.2)為高斯分佈，而其更新是由 $\rho(\alpha)$ 及 α 所控制。首先假設 $I_{t,x}$ 符合到第 n 個高斯分佈，而其中的平均數、變異數以及加權比重的更新方式為:

$$\mu_{t,x,n} = (1 - \rho_{t,x,n}(\alpha))\mu_{t-1,x,n} + \rho_{t,x,n}(\alpha) \cdot I_{t,x} \quad (2.3)$$

$$\sigma_{(t,x,n)}^2 = (1 - \rho_{t,x,n}(\alpha))\sigma_{(t-1,x,n)}^2 + \rho_{t,x,n}(\alpha) \cdot (I_{t,x} - \mu_{t-1,x,n})^2 \quad (2.4)$$

$$w_{t,x,n} = (1 - \rho_{t,x,n}(\alpha))w_{t-1,x,n} + \alpha \quad (2.5)$$

所有的參數更新都是由單一個學習速率 α 所控制。而本論文實作的文獻[19]GMM 演算法，則是使用了兩個學習速率 $\rho(\alpha)$ 和 $\eta(\beta)$ ：

$$\mu_{t,x,n} = (1 - \rho_{t,x,n}(\alpha))\mu_{t-1,x,n} + \rho_{t,x,n}(\alpha) \cdot I_{t,x} \quad (2.6)$$

$$\sigma_{(t,x,n)}^2 = (1 - \rho_{t,x,n}(\alpha))\sigma_{(t-1,x,n)}^2 + \rho_{t,x,n}(\alpha) \cdot (I_{t,x} - \mu_{t-1,x,n})^2 \quad (2.7)$$

$$w_{t,x,n} = (1 - \eta_{t,x}(\beta))w_{t-1,x,n} + \eta_{t,x}(\beta) \quad (2.8)$$

經由兩個學習速率的控制，可以將精確度跟可靠度及靈敏度分別交由這兩個學習速率控制更新:

- (1) 精確度：經由 $\rho(\alpha)$ 的更新來維護場景的 μ 和 σ^2 ，盡量提高場景的準確度。一般來說 $0.001 \leq \alpha \leq 0.1$ ，而 α 在這個範圍內越大越好。
- (2) 可靠度及靈敏度： $w_{t,x,n}$ 則用來判斷前景或背景的標準，如(2-9)，當 $w_{t,x,n}$ 大於 T_w 時，將這一點視為背景，否則即將其視為前景，而 $w_{t,x,n}$ 則由 $\eta(\beta)$ 來控制更新。

$$F_{t,x,n} = \begin{cases} 0 & w_{t,x,n} \geq T_w \\ 1 & \text{otherwise} \end{cases} \quad (2-9)$$

$w_{t,x,n}$ 的學習速率再根據不同的條件分成4種狀況，如(2-10)，對不同的狀況給予不同的學習速率 $\eta(\beta)$ 來控制，其特性如(2-11)。而GMM的背景及參數經由不斷的回饋更新如圖2-3，來提高辨識背景或前景可靠度及靈敏度。

$$\eta_{t,x}(\beta) = \begin{cases} (1 - \beta_b) \eta_{t-1,x} + \eta_b \beta_b & \leftarrow \text{Background} \\ \beta_d N(I_{t,x}; \mu_{t-1,x,n}, \sigma_{t-1,x,n}^2) & \leftarrow \text{Shadow} \\ \beta_s & \leftarrow \text{Still foreground} \\ \beta_m & \leftarrow \text{Moving foreground} \end{cases} \quad (2-10)$$

$$\eta(\beta_b) \geq \eta(\beta_d) \geq \eta(\beta_s) \geq \eta(\beta_m) \quad (2.11)$$

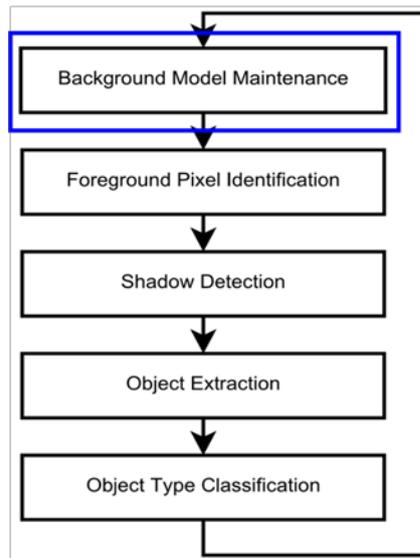


圖 2-3 GMM 背景維護更新流程圖。

接下來分析一下演算法，原始的演算法，如圖 2-4。在演算法中，大致分兩大部分：第一部分主要是判斷是哪一個高斯分佈是符合的，首先計算 σ 並找出符合 $|I_{t,x} - \mu_{t-1,x,n}| \leq T_\sigma \cdot \sigma_{t-1,x,n}$ ，也就是 σ 是落在 T_σ 倍數內的高斯分佈，最後在這些高斯分佈中再找出 w 最大的那一個 $l_{(t,x)}$ ，這一個高斯分佈也就是真正符合的那一個。第二部分則是對是否符合的情況，以不同的方式更新 μ 、 σ 以及 w 。這一個部分則分成兩個狀況，一個是有符合的，一個是不符合的。如果是有符合的，則會計算 μ, σ, w 相對應的學速率 $\rho(\alpha)$ 、 $\eta(\beta)$ ，並將 μ, σ, w 全部更新；如果是不符合的，則是找出 w 最小的高斯分佈，並更新 $\mu = I_{t,x}$ ，而 σ, w 設定為初始值。

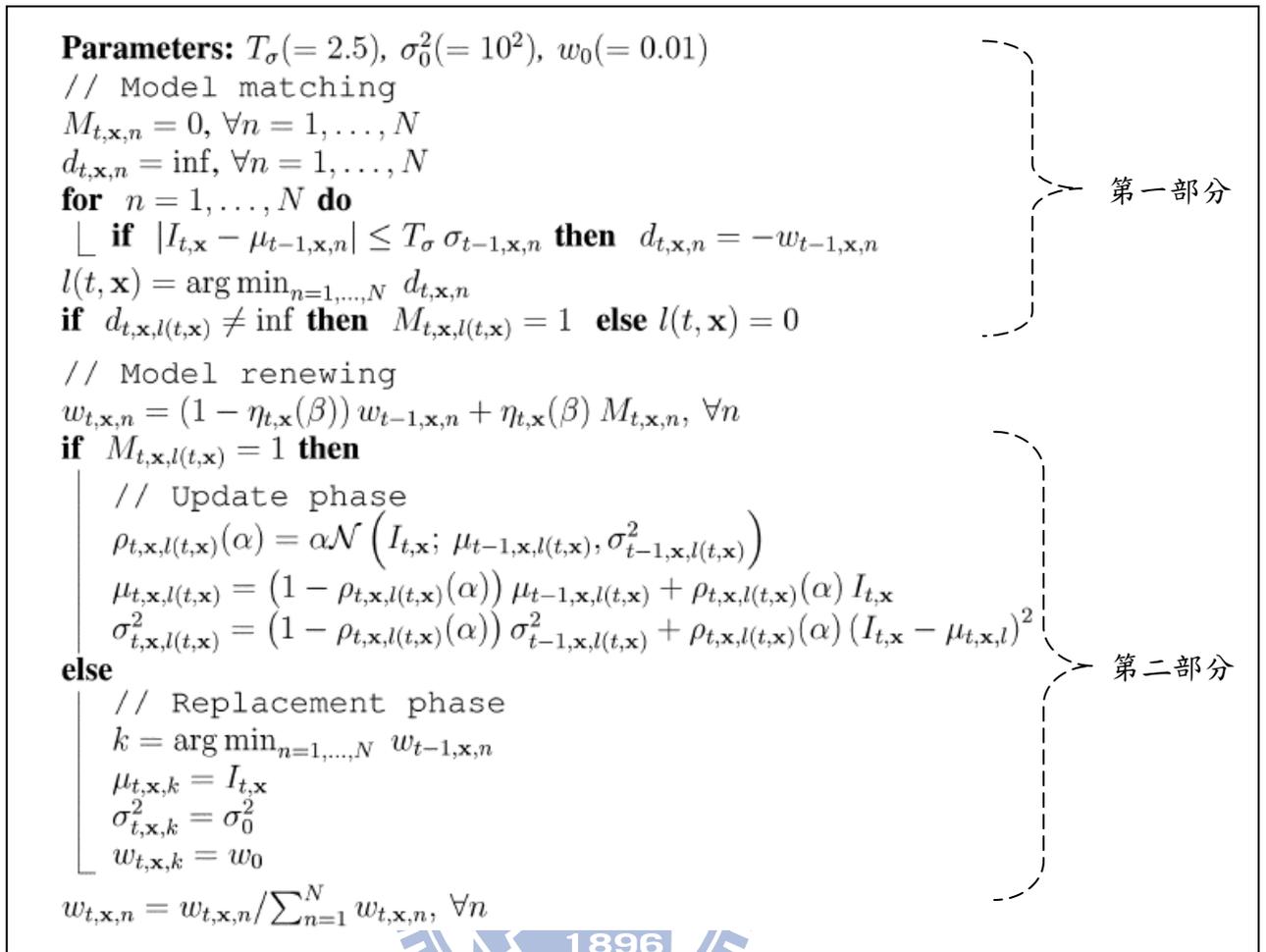


圖 2-4 文獻[19]的原始演算法。

2.2 實作 GMM 演算法

因為考慮到其他因素及複雜度，真正實作的部分為：

1. 兩種學習速率
2. 加權比重四種狀況的學習速率中，實作了三種狀況：
 - Background
 - Still Foreground
 - Moving Foreground

而未實作 Shadow 的狀況，主要原因是經過初步的評估，偵測 Shadow 會降低處理的速度約 20%，且記憶體的需求將增加 12%，所以基於效能的考量，暫時不實作 Shadow 的狀況。修改後之演算法，如圖 2-5。在第一部分，將 ρ 設定為

三、GMM 演算法實作及去浮點數化

利用 GMM 背景模型作前景偵測的基本流程，如圖 3-1。第一步驟為”Check Matched Gaussian”，主要目的再找出是否有高斯分佈是符合的，如果有，是哪一個高斯分佈；第二個步驟為”Update Foreground”，目的在更新前景影像，對於第一個步驟找出的高斯分佈判斷其的加權比重是否大於 $T_w = 0.24$ ，若大於 T_w 之像素材需要更新至前景；第三個步驟是”Erosion & Dilation”，其主要目的為對前景做基本的影像處理以消除微小的雜訊；第四個步驟是”Connected Component Analysis”，將前景的影像作連通單元分析，藉此比較出前景的物件是屬於哪一個型態的前景，如 Still Foreground 或是 Moving Foreground。透過此分析才能知道 w 的學習速率是使用哪一個；第五步驟是”Update Pixel Type”，根據前景型態的分析結果，更新每個像素的學習速率 $\eta(\beta)$ 型態；第六步驟是”Update Gaussian”，將第五個步驟所得知的學習速率對高斯分佈的參數進行更新；第七個步驟”Update Background”，則是找到加權比重最大的高斯分佈，並將他的 μ 值更新到背景的影像。這樣就完成了一個以 GMM 背景模型作前景偵測的循環。在本論文中所實作的高斯分佈個數為 3 ($N = 3$)。

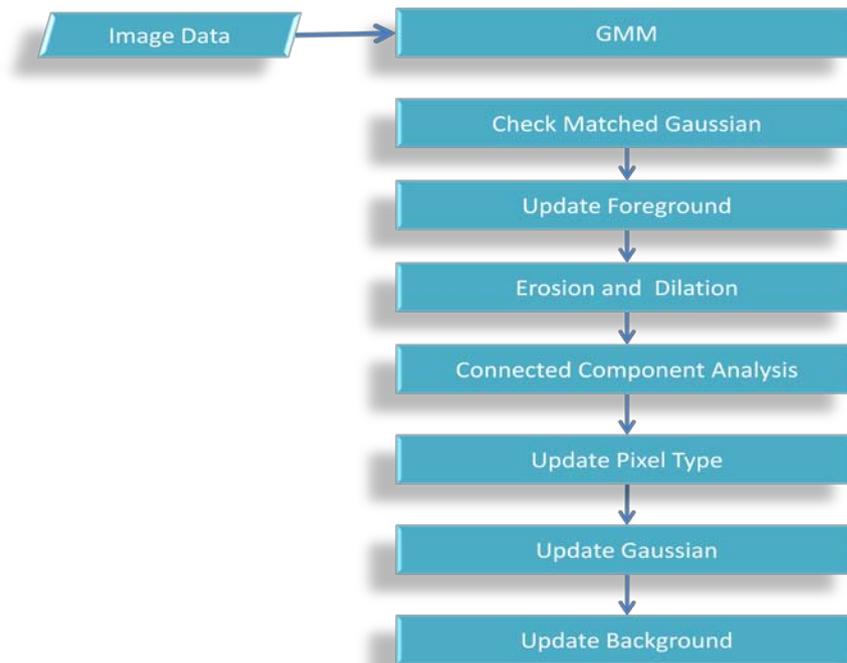


圖 3-1 以 GMM 背景模型作前景偵測的基本流程

3.1 實作

我們除了需要實作上一章所提出的 GMM 的演算法外，還需要實作 Erosion、Dilation 以及 Connected Component Analysis 等影像處理演算法，而這三個演算法皆為一般常用的演算法，所以在本論文中只作簡單的敘述。

對於 Erosion 及 Dilation，本論文採用簡單的四個相鄰點。Erosion 部分，如圖 3-2，先將影像二值化，若像素的值為 1 時，且上下左右的點皆為 1 時，則此點的值不變；若只要上下左右有一點不為 1 時，則此點的值改為 0。在 Dilation 部分，如圖 3-3，若像素的值為 1 時，則將上下左右四點的值改成 1。而 Erosion + Dilation 的效果可以消除一些細微的雜點，如圖 3-7。Connected Component Analysis 則使用文獻[20] Labeling 的演算法。利用影像 Labeling 來將物件分群。

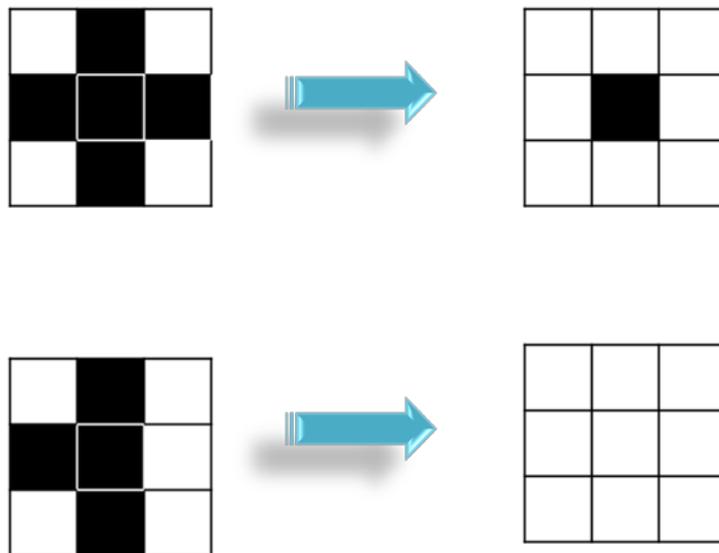


圖 3-2 四個相鄰點之 Erosion(兩個例子)

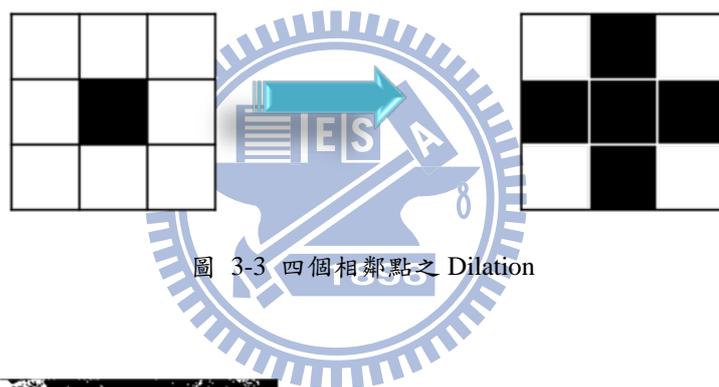


圖 3-3 四個相鄰點之 Dilation

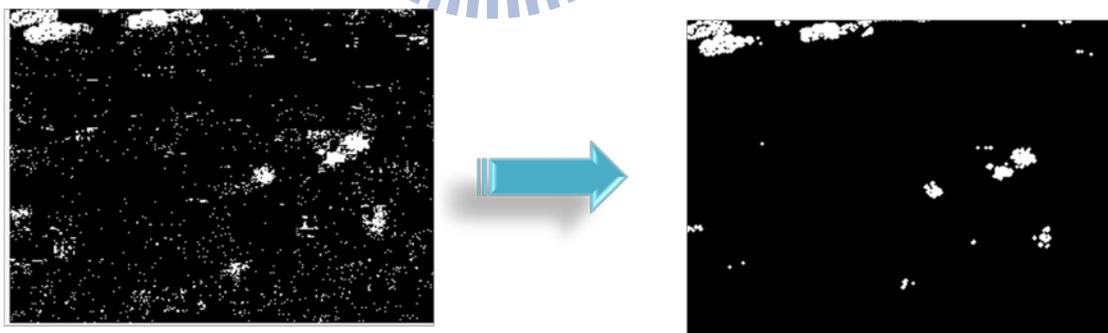


圖 3-4 Erosion + Dilation 之效果

3.2 GMM 演算法驗證

一開始我們先驗證實作的 GMM 是否正確。首先假設第 n 個高斯分佈是符合的，首先定義三個對像素驗證的檢查點：

- (1) 像素顏色變化時，是否被偵測為前景。
- (2) 像素不再被偵測成前景時間點是否正確($W_{t,n} \geq T_w$)。
- (3) 像素背景顏色更新的時間點是否正確(符合的高斯分佈之加權比重為最大值)。

我們使用單一的像素資料來作驗證，如圖 3-5，在第一個和第二個單位時間，先輸入紅色的像素資料，這時候紅色會進入背景，所以不會被偵測為前景。接下來再輸入藍色，這時像素會被偵測為前景，但是背景會繼續維持在紅色。持續輸入藍色，直到像素不再被偵測為前景，但是此時背景一樣維持在紅色，這個時間點為驗證($W_{t,n} \geq T_w$)。之後一直等到背景的颜色更新為藍色，這個時間點為驗證 $W_{t,n}$ 的值為所有高斯分佈中最大的。

我們先使用手算，計算出這幾個時間點，如表格 3-1。接下來再與 GMM 程式輸出的結果，如表格 3-2，作比較驗證。首先驗證第一個檢查點，手算的結果在第 3 個畫面時符合第 1 個高斯分佈，而程式輸出結果也是一樣的。第二個檢查點為像素不再被偵測為前景的時間點($W_{t,n} \geq 0.24$)，手算的結果為第 1590 個畫面，與程式輸出的結果一致。第三個檢查點為背景顏色更新為藍色， $W_{t,n}$ 為所有高斯分佈中的最大值，也就是 $Weight[1] > Weight[0] > Weight[2]$ ，在第 1653 個畫面得到了這個結果，與程式輸出的結果一致。除了在三個檢查點得到了正確的驗證外，我們還發現 GMM 背景更新過程中數據的誤差小於 10^{-7} ，所以手算的結果與程式輸出的結果是一致的，證明實作的程式基本上是正確的。

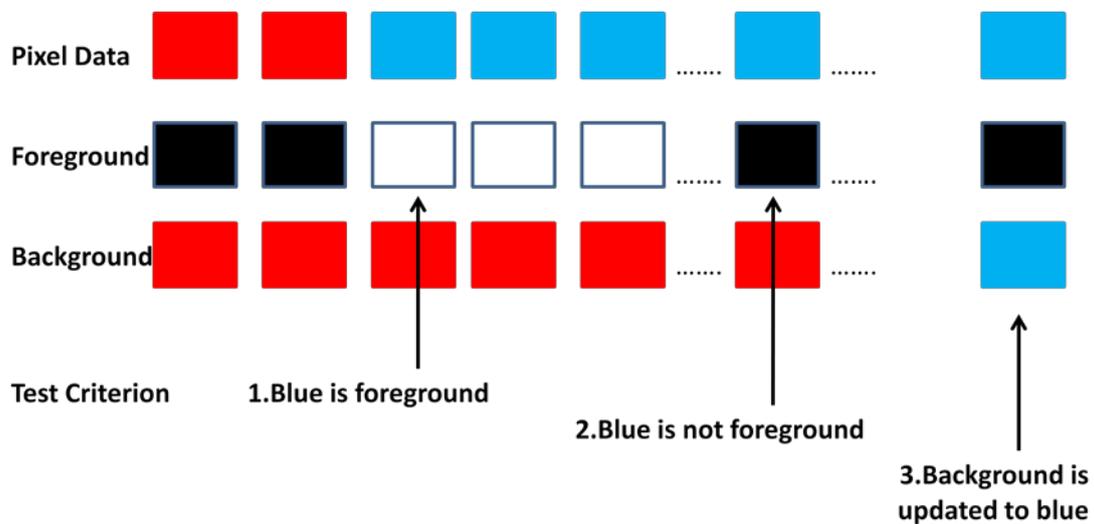


圖 3-5 以單一像素作 GMM 程式之基本驗證

表格 3-1 手算以單一像素作 GMM 之結果

Excel Simulation						
Frame	R	G	B	Weight[0]	Weight[1]	Weight[2]
1	255	0	0	1.0000000	0.0000000	0.0000000
2	255	0	0	1.0000000	0.0000000	0.0000000
3	0	0	255	0.9900974	0.0099026	0.0000000
4	0	0	255	0.9899324	0.0100676	0.0000000
5	0	0	255	0.9897674	0.0102326	0.0000000
1586	0	0	255	0.7604786	0.2395214	0.0000000
1587	0	0	255	0.7603519	0.2396481	0.0000000
1588	0	0	255	0.7602251	0.2397749	0.0000000
1589	0	0	255	0.7600984	0.2399016	0.0000000
1590	0	0	255	0.7599717	0.2400283	0.0000000
1591	0	0	255	0.7596564	0.2403436	0.0000000
1592	0	0	255	0.7591543	0.2408457	0.0000000
1650	0	0	255	0.5154448	0.4845552	0.0000000
1651	0	0	255	0.5094924	0.4905076	0.0000000
1652	0	0	255	0.5035402	0.4964598	0.0000000
1653	0	0	255	0.4975904	0.5024096	0.0000000
1654	0	0	255	0.4916454	0.5083546	0.0000000

表格 3-2 程式輸出以單一像素作 GMM 之結果

Program Output						
Frame	R	G	B	Weight[0]	Weight[1]	Weight[2]
1	255	0	0	1.0000000	0.0000000	0.0000000
2	255	0	0	1.0000000	0.0000000	0.0000000
3	0	0	255	0.9900974	0.0099026	0.0000000
4	0	0	255	0.9899324	0.0100676	0.0000000
5	0	0	255	0.9897674	0.0102326	0.0000000
1586	0	0	255	0.7604786	0.2395214	0.0000000
1587	0	0	255	0.7603519	0.2396481	0.0000000
1588	0	0	255	0.7602251	0.2397749	0.0000000
1589	0	0	255	0.7600984	0.2399016	0.0000000
1590	0	0	255	0.7599717	0.2400283	0.0000000
1591	0	0	255	0.7596564	0.2403436	0.0000000
1592	0	0	255	0.7591543	0.2408457	0.0000000
1650	0	0	255	0.5154448	0.4845552	0.0000000
1651	0	0	255	0.5094924	0.4905076	0.0000000
1652	0	0	255	0.5035402	0.4964598	0.0000000
1653	0	0	255	0.4975904	0.5024096	0.0000000
1654	0	0	255	0.4916454	0.5083546	0.0000000

經過這一個基本的驗證後，接下來再使用一段真正的影像來作驗證，如圖 3-6，而其相對之背景變化如圖 3-7。開始時先拍攝一個基本場景如圖 3-6(a)；第二步在此場景中央放入一個一小方盒，而後持續一段時間圖 3-6(b)-(c)，直到小方盒慢慢的完整呈現在背景中，如圖 3-7(b)-(c)。第三步為將小方盒移走然後持續一段時間，如圖 3-6(d)-(e)，直到小方盒慢慢消失於背景中，如圖 3-7(d)-(e)。第四步為在場景中放入另一個大方盒，如圖 3-6(f)-(g)，直到大方盒慢慢呈現於背景中如圖 3-7(f)-(g)。

透過這樣一個完整的影片，我們可以模擬(1)物體進入場景；(2)停留在場景內；(3)離開場景；(4)另一個物體進入場景同一個位置，並利用這樣較複雜的場景變化，來驗證 GMM 程式實作是否正確。

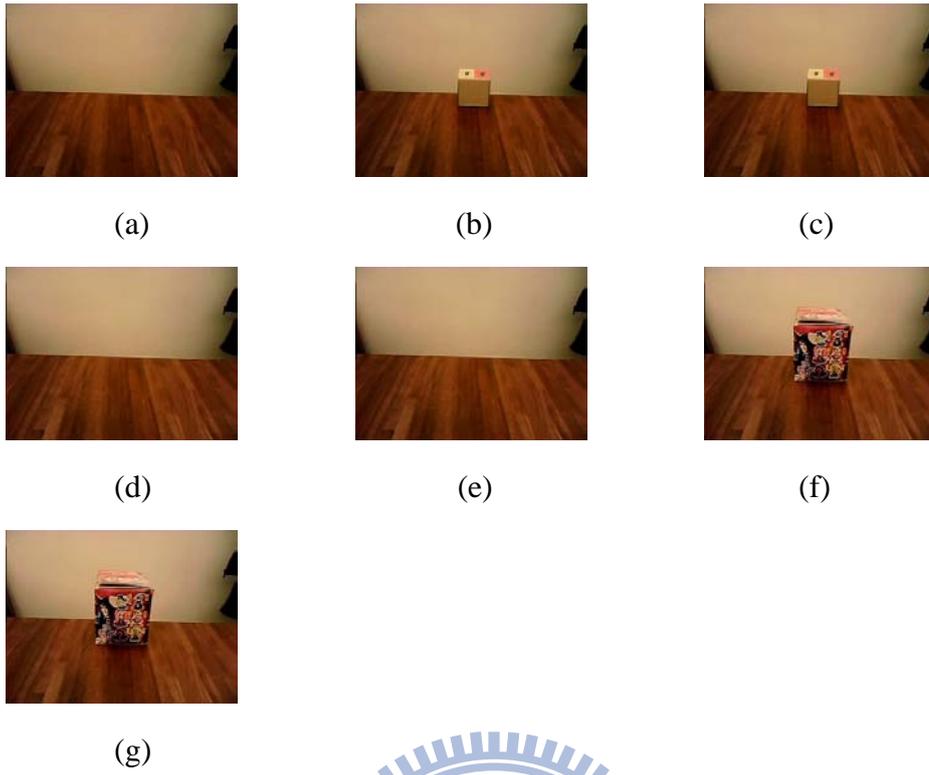


圖 3-6 用以驗證 GMM 程式之一些影像畫面

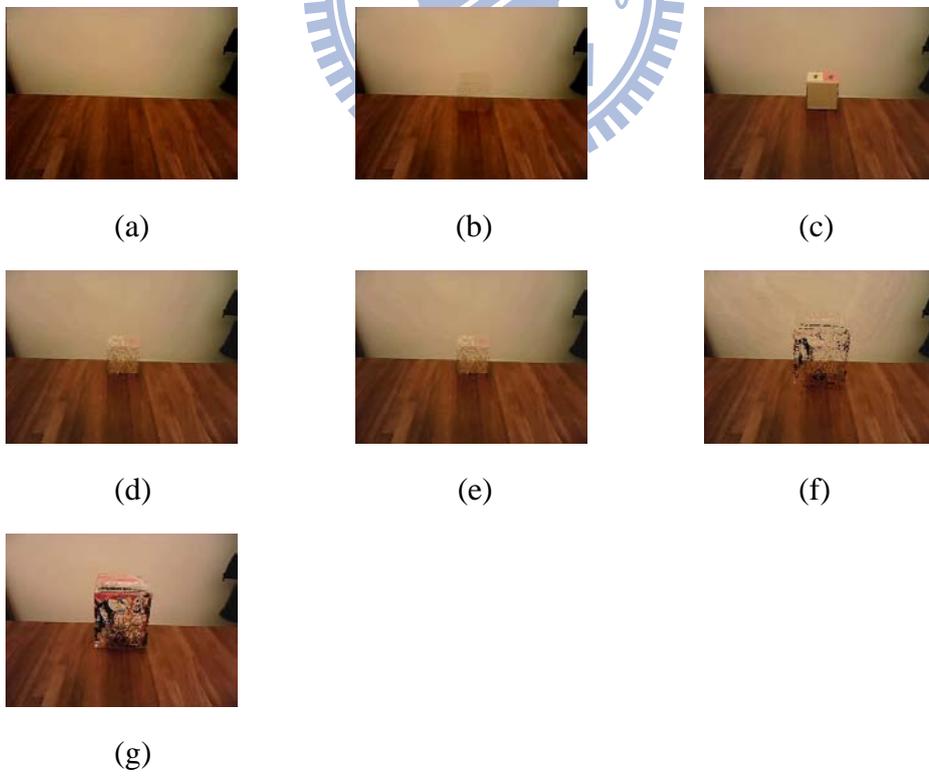


圖 3-7 用以驗證 GMM 程式之一些背景畫面

這次使用的測試影片大小為 320x240，以影片的中心點，也就是(160,120)位置的資料來當作驗證的參考點。經手算之結果，如表格 3-3，發現第 23 個畫面時，發生了不符合的狀況，也就是參考點被偵測為前景，接下來第 268、1014 個畫面，參考點不再被偵測為前景($W_{t,n} \geq 0.24$)；第 365、574 及 1094 個畫面，參考點的背景顏色切換。檢視 GMM 程式輸出的結果，如表格 3-4，在第 23 個畫面參考點也被偵測為前景，以及接下來的幾個檢查點，其結果皆與手算一致。之後更進一步比對手算與 GMM 程式輸出值的精確度，計算兩者之平均數、變異數以及加權比重的差值，如表格 3-5，其整體的差值約在均不超過 10^{-8} ，所以經過此兩步驟的驗證後，可以確認實作的 GMM 程式是正確的。



表格 3-3 實際影像經 GMM 分析之手算結果

Excel Simulation						
Frame	R	G	B	Weight[0]	Weight[1]	Weight[2]
1	85	141	168	1	0	0
2	83	141	168	1	0	0
21	82	135	163	1	0	0
22	80	135	166	1	0	0
23	36	96	145	0.990088106	0.00991189	0
24	31	89	136	0.988988008	0.011011992	0
266	37	101	153	0.749041365	0.239073084	0.011885551
267	46	101	151	0.748209096	0.239918559	0.011872345
268	44	100	147	0.747377753	0.24076309	0.011859154
269	41	103	147	0.746368793	0.241788063	0.011843144
270	41	103	147	0.745184679	0.242990967	0.011824355
360	39	99	148	0.498046780	0.491485684	0.010467536
361	39	99	148	0.493798527	0.495823223	0.010378250
362	41	101	152	0.489505180	0.500206800	0.010288016
363	41	101	152	0.485169348	0.504633763	0.010196889
364	34	97	143	0.484630271	0.504073059	0.011296670
570	107	149	192	0.492761459	0.477154042	0.030084498
571	103	148	190	0.498037292	0.472191121	0.029771587
572	107	147	188	0.503331530	0.467210885	0.029457584
573	101	148	188	0.508641715	0.46221565	0.029142635
574	104	147	185	0.513965430	0.457207686	0.028826883
1012	76	121	195	0.749767094	0.010601076	0.239631830
1013	94	143	217	0.750209682	0.010017033	0.239773285
1014	81	143	212	0.749376116	0.010005903	0.240617980
1015	81	138	202	0.748364458	0.009992395	0.241643147
1016	74	122	200	0.747177178	0.009976542	0.242846280
1092	70	130	199	0.488747442	0.012730337	0.498522221
1093	65	118	195	0.487521133	0.012698396	0.499780471
1094	66	123	188	0.486188253	0.012663679	0.501148070
1095	61	107	185	0.484750762	0.012626236	0.502623002
1252	79	125	202	0.322014451	0.014353538	0.663632011
1253	100	150	221	0.323418870	0.010054786	0.666526345

表格 3-4 實際影像經 GMM 程式分析之輸出結果

Program Output						
Frame	R	G	B	Weight[0]	Weight[1]	Weight[2]
1	85	141	168	1	0	0
2	83	141	168	1	0	0
21	82	135	163	1	0	0
22	80	135	166	1	0	0
23	36	96	145	0.9900881	0.0099119	0
24	31	89	136	0.9889880	0.0110120	0
266	37	101	153	0.7490414	0.2390731	0.0118856
267	46	101	151	0.7482091	0.2399186	0.0118723
268	44	100	147	0.7473778	0.2407631	0.0118592
269	41	103	147	0.7463688	0.2417881	0.0118431
270	41	103	147	0.7451847	0.2429910	0.0118244
360	39	99	148	0.4980468	0.4914857	0.0104675
361	39	99	148	0.4937985	0.4958232	0.0103782
362	41	101	152	0.4895052	0.5002068	0.0102880
363	41	101	152	0.4851693	0.5046338	0.0101969
364	34	97	143	0.4846303	0.5040731	0.0112967
570	107	149	192	0.4927615	0.4771540	0.0300845
571	103	148	190	0.4980373	0.4721911	0.0297716
572	107	147	188	0.5033315	0.4672109	0.0294576
573	101	148	188	0.5086417	0.4622156	0.0291426
574	104	147	185	0.5139654	0.4572077	0.0288269
1012	76	121	195	0.7497671	0.0106011	0.2396318
1013	94	143	217	0.7502097	0.0100170	0.2397733
1014	81	143	212	0.7493761	0.0100059	0.2406180
1015	81	138	202	0.7483645	0.0099924	0.2416431
1016	74	122	200	0.7471772	0.0099765	0.2428463
1092	70	130	199	0.4887474	0.0127303	0.4985222
1093	65	118	195	0.4875211	0.0126984	0.4997805
1094	66	123	188	0.4861883	0.0126637	0.5011481
1095	61	107	185	0.4847508	0.0126262	0.5026230
1252	79	125	202	0.2783973	0.0143517	0.7072510
1253	100	150	221	0.2796110	0.0100548	0.7103343

表格 3-5 實際影像測試之手算結果與 GMM 程式輸出結果的差值表

Mean[0] R	-1.578833E-08
Mean[0] G	-6.007646E-09
Mean[0] B	1.572694E-08
Mean[1] R	-2.887726E-09
Mean[1] G	-1.395733E-08
Mean[1] B	-1.099158E-08
Mean[2] R	3.392723E-10
Mean[2] G	-3.559222E-09
Mean[2] B	7.010194E-09
Var[0] R	-5.01746E-09
Var[0] G	-1.77466E-08
Var[0] B	-1.20599E-08
Var[1] R	9.80232E-09
Var[1] G	-8.57381E-09
Var[1] B	1.03828E-08
Var[2] R	2.64479E-10
Var[2] G	4.72581E-09
Var[2] B	9.15916E-12
Weight[0]	3.7221E-10
Weight[1]	1.05768E-09
Weight[2]	9.97634E-10
Average	-2.18577E-09

3.3 GMM 之去浮點數化

在驗證完實作 GMM 演算法的程式後，為了讓程式能硬體化或是在嵌入式系統中執行，我們開始對程式中的變數進行去浮點數化。在去浮點數化的過程中，必須注意的事項，(1)記憶體資料格式大小，(2)運算時所需使用到的臨時緩衝記憶體的大小。所以針對 GMM 演算法中主要的參數平均數(Mean)、變異數(Variance)及加權比重(Weight)，將其轉換成三種不同的格式：(1)**Full 32-bit**，在這個格式中利用整數格式中的 32 個位元，使變數達到最高的精確度，我們規劃的變數格式為，平均數：8.24 (格式為：整數.小數)，變異數：16.16，加權比重：1.31。使用這個資料格式，在程式計算過程中，會需要一個臨時的 64 位元緩衝記憶體，其結果的準確度可達到幾乎與 Double 格式一樣，但是計算的速度卻會慢很多。

(2)**Partial 32-bit**，在這個格式中，主要是降低變數的精確度，使所有程式中所有變數的計算，能在整數格式的 32 位元完成，所以規劃變數的格式不會使用到整數全部的 32 位元，規劃的格式為，平均數：8.8，變異數：10.8，加權比重：1.15。使用這個資料格式，在程式計算過程中，我們不需要一個臨時的 64 位元緩衝記憶體，準確度會降低一些，但是程式計算的速度幾乎可達到與 Double 格式一樣。

(3)**16-bit**，主要是使用 16 位元的整數格式，並使變數的精確度能達到最高，規劃的格式為，平均數：8.8，變異數：8.8，加權比重：1.15。使用這個資料格式，在程式計算過程中，我們會需要一個臨時的 32 位元緩衝記憶體，計算速度幾乎可達到與 Double 格式一樣，但是準確度會降低一些，所需要記憶體使用量最少。

在 GMM 演算法的程式中，需要考慮的參數如下：

- $I_{t,x}$: Pixel Intensity
- $\mu_{t,x,n}$: Mean
- $Var_{t,x,n}$: Variance
- $w_{t,x,n}$: Gaussian weight
- $\eta_{t,x,n}$: Learning rate of weight
- ρ : Learning rate of μ and Var

因為 $I_{t,x}$ 和 $\mu_{t,x,n}$ 的最大值是一樣的，所以放在一起考慮。而 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的最大值皆是 1，將其放在一起討論。以下我們將對 GMM 演算法程式中的變數，所規劃之去浮點化後的格式，於 3.3.1~3.3.3 做詳細的說明。

3.3.1 Full 32-bit case

在 Full 32-bit 的格式中，定義資料格式時，以精確度為最高考量。 $I_{t,x}$ 和 $\mu_{t,x,n}$ 的最大值為 255，所以我們保留整數的位元數為 8 ($2^8 = 256$)。其餘的 24 個位元都給小數用，如圖 3-8 所示； $Var_{t,x,n}$ 的最大值為 $(I_{t,x} - \mu_{t-1,x,l(t,x)})^2 = (255)^2 = 65025$ ， $65025 < 65535$ (2^{16})，所以我們以 16 個位元表示整數部分，其餘 16 個位元來表示小部分，如圖 3-9 所示； $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 其最大值為 1，所以僅需用 1 個位元來表示整數部分，所以其餘的 31 個位元全部可用來表示小數部分，如圖 3-10 所示。

Data Size 32bits : 8.24



圖 3-8 Full 32-bit 格式中之 $I_{t,x}$ 、 $\mu_{t,x,n}$ 的資料格式

Data Size 32bits : 16.16



圖 3-9 Full 32-bit 格式中之 $Var_{t,x,n}$ 的資料格式

Data Size 32bits : 1.31



圖 3-10 Full 32-bit 格式中之 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的資料格式

定義完資料格式後，接下來必須檢查溢位(Overflow)的問題。因為在 Full 32-bit 資料格式裡，我們使用了完全的 32 位元，所以在作乘除的運算時，必須先將運算元的數值，轉移至 64 位元的緩衝記憶體，等運算完之後，再將結果的數值回存至原來 32 位元的資料格式內，以避免發生溢位的錯誤。

3.3.2 Partial 32-bit case

在 Partial 32-bit 的格式中，定義資料格式時，我們以速度為最高考量，所以避免使用到 64 位元的緩衝記憶體，因為程式運算時，若使用到 64 位元的緩衝記憶體，會需要更多處理的時間。由於 $I_{t,x}$ 和 $\mu_{t,x,n}$ 的最大值為 255，所以我們保留整數的位元數為 $8(2^8 = 256)$ ，小數則使用 8 個位元，如圖 3-11 所示；因為 $Var_{t,x,n}$ 的值最大值為 65025，但是為了將使運算能在 32 位元格式的範圍內完成，所以我們將整數最大值限定在 10 個位元($2^{10} = 1024 - 1 = 1023$)，因此若 $Var_{t,x,n}$ 的值大於 1023，將會被截位成 1023，整數的精確度會因此降低。小數的部分則保留 8 個位元來表示，如圖 3-12 所示； $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 其最大值為 1，所以僅需用 1 個位元來表示整數部分，小數部分則使用 15 個位元表示，如圖 3-13 所示。

Data Size 32bits : 8.8



圖 3-11 Partial 32-bit 格式中之 $I_{t,x}$ 、 $\mu_{t,x,n}$ 的資料格式

Data Size 32bits : 10.8



圖 3-12 Partial 32-bit 格式中之 $Var_{t,x,n}$ 的資料格式

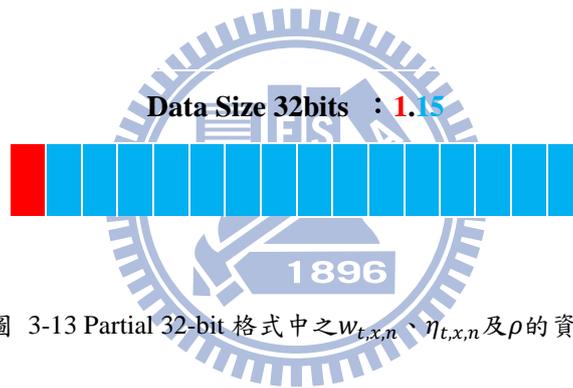


圖 3-13 Partial 32-bit 格式中之 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的資料格式

定義完資料格式後，接下來必須檢查溢位的問題，在此我們需要檢查的乘法運算可分為兩個部分。第一部分，如表格 3-6 所示，因為乘數與被乘數皆在 16 個位元內，相乘後的結果不會超過 32 位元。所以這些運算皆可以在 32 位元的格式中完成運算，不會有溢位的問題。但是第二個部分，如表格 3-7 所示，因為被乘數為 16 個位元，乘數為 18 個位元，所以運算的結果會達到 34 個位元，可能無法在 32 位元的格式中完成運算。前半部的 $(1 - \rho) \cdot Var_{t-1,x,n}$ 的計算，其中 ρ 的小數有 15 個位元，為了使運算能在 32 位元的格式中完成，所以我們將 $(1 - \rho)$ 的值右移 2 個位元，將其格式暫時轉換成 1.13 的格式。如此運算將可轉換成 14 位元乘上 18 個位元，即可在 32 位元的格式中完成運算，其計算轉換流程如圖 3-14

所示。後半部的計算為 $\rho \cdot (I_{t,x} - \mu_{t-1,x,n})^2 = \rho \cdot Var_{t,x,n}$ ，其中 $\rho = 0.025$ ， $Var_{t,x,n}$ 容許的對大值為 1023，將這兩個數轉換成相對應的格式 1.15 及 10.8，所以 $\rho \cdot Var_{t,x,n}$ 的最大值等於 $(0.025 \times 2^{15}) \times (1024 \times 2^8) = 214696936 < 2^{32}$ ，其結果小於 32 位元的最大值，所以 $\rho \cdot (I_{t,x} - \mu_{t-1,x,n})^2$ 的結果不可能會超過 32 位元的格式，經過這些轉換與檢查後，可以確定在 Partial 32-bit 的格式中不會發生溢位的問題。

表格 3-6 GMM 演算法中相關的運算式與 Partial 32-bit 格式中的運算格式對照表 1

運算式	運算格式
$\mu_{t,x,n} = (1 - \rho) \cdot \mu_{t-1,x,n} + \rho \cdot I_{t,x}$	8.8 × 8.8
$Var_{t,x,n} = (I - \mu) \times (I - \mu)$	8.8 × 8.8
$w_{t,x,n} = (1 - \eta_{t,x}(\beta)) \cdot w_{t-1,x,n}$	1.15 × 8.8

表格 3-7 GMM 演算法中相關的運算式與 Partial 32-bit 格式中的運算格式對照表 2

運算式	運算格式
$Var_{t,x,n} = (1 - \rho) \cdot Var_{t-1,x,n} + \rho \cdot (I_{t,x} - \mu_{t-1,x,n})^2$	1.15 × 10.8

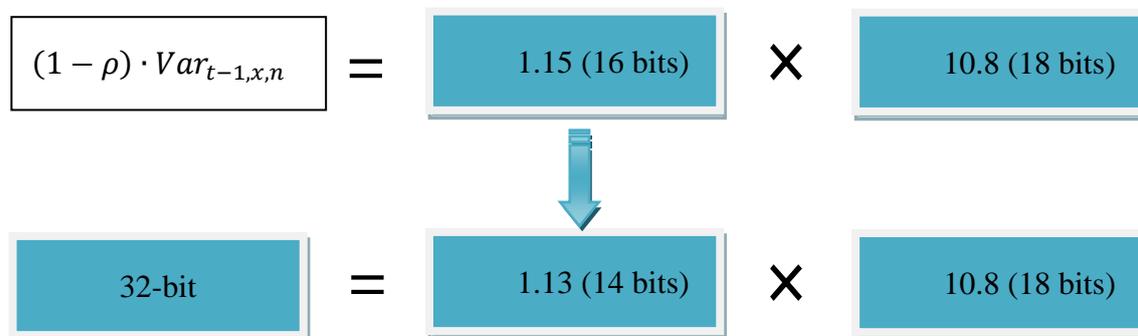


圖 3-14 Partial 32-bit 格式中之 $Var_{t,x,n}$ 格式轉換流程

3.3.3 16-bit case

在 16-bit 的格式中，定義資料格式時，我們以在 16 位元內能達到最高精確度為考量，可使用 32 位元的緩衝記憶體。由於 $I_{t,x}$ 和 $\mu_{t,x,n}$ 的最大值為 255，所以我們保留整數的位元數為 8 ($2^8 = 256$)，其餘 8 個位元給小數用，如圖 3-15 所示；考慮到 16 位元的數值限制，所以我們將 $Var_{t,x,n}$ 整數最大值限定在 8 個位元，超過 255 的值將會被視為 255，整數的精確度會因此降低。小數的部分則保留 8 個位元來表示，如圖 3-16 所示； $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 其最大值為 1，所以僅需用 1 個位元來表示是整數部分，小數部分則使用 15 個位元表示，如圖 3-17 所示。

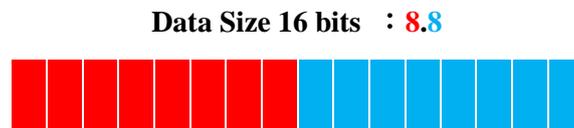


圖 3-15 16-bit 格式中之 $I_{t,x}$ 、 $\mu_{t,x,n}$ 的資料格式



圖 3-16 16-bit 格式中之 $Var_{t,x,n}$ 的資料格式

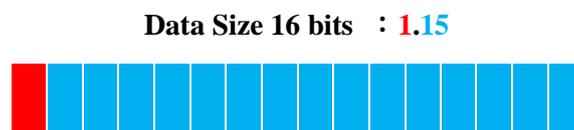


圖 3-17 16-bit 格式中之 $w_{t,x,n}$ 、 $\eta_{t,x,n}$ 及 ρ 的資料格式

定義完資料格式後，接下來必須檢查溢位的問題。因為在 16-bit 資料格式裡，我們使用了完全的 16 位元，所以在作乘除的運算時，必須先將運算元的數值，轉移至 32 位元的緩衝記憶體，等運算完之後，再將結果的數值回存至原來 16 位元的資料格式內，以避免發生溢位的錯誤。

以上我們所規劃的三種去浮點數化後的資料格式，在經過詳細分析後，皆不會發生溢位的問題，所以將這些去浮點數化後的資料格式，實作至 GMM 演算法的程式中，再將視訊輸入至 GMM 演算法的程式中做影像分析。接下來在第四章中，將對使用去浮點數化前後的各種資料格式之 GMM 演算法程式，所作之影像分析的輸出結果，作分析比較。



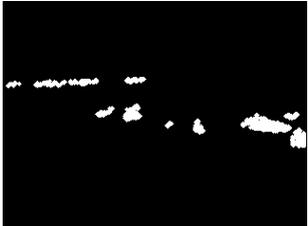
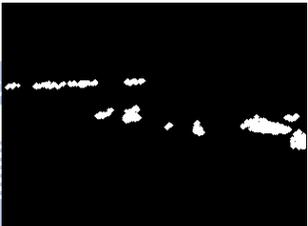
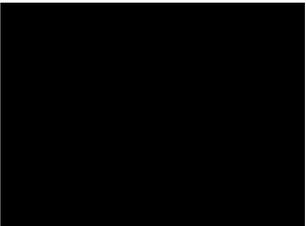
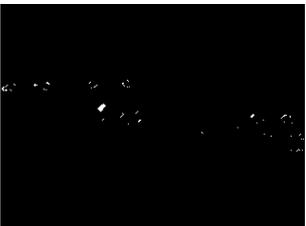
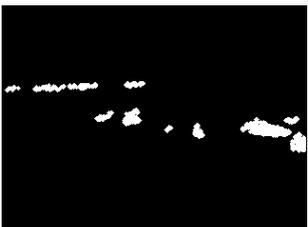
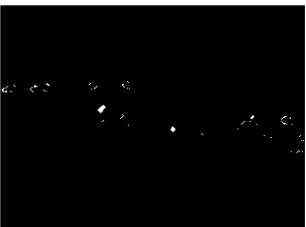
四、GMM 演算法實驗結果

在規劃好去浮點數化的資料格式後，將去浮點數化的資料格式實作至 GMM 演算法的程式中。以標準 C 語言的 Double 資料格式為基準，將影像處理的輸出結果與 Float、Full 32-bit、Partial 32-bit 與 16-bit 的輸出結果作比較，比較的项目為各種資料格式在 GMM 程式中的處理速度、整體記憶體使用量以及輸出結果的準確度。

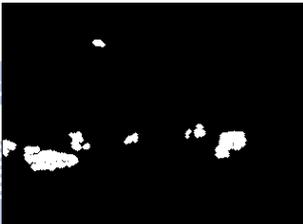
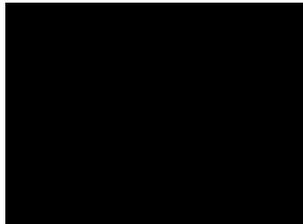
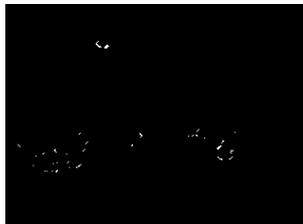
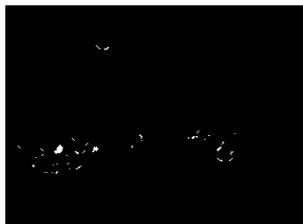
4.1 GMM 去浮點數化之影像分析

以 Test Video 1~Test Video 4 作為測試的視訊來源，將各種資料格的程式之 GMM 前景分析的輸出結果，與 Double 格式的程式之 GMM 前景分析輸出結果作比對，比對結果如表格 4-1~表格 4-4。在 Float 和 Full 32-bit 格式之程式輸出的影像結果，與 Double 格式程式輸出之前景作比對，幾乎是與 Double 格式一致。而 Partial 32-bit 格式與 16-bit 格式程式輸出之前景的影像結果，與 Double 格式程式輸出之前景作比對時，物件的邊緣有一些差異，但是比對物件數量的偵測，得到的結果是一致的。

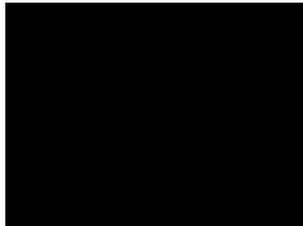
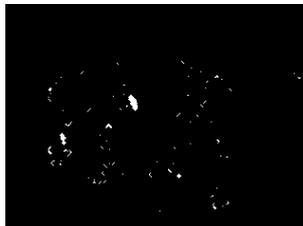
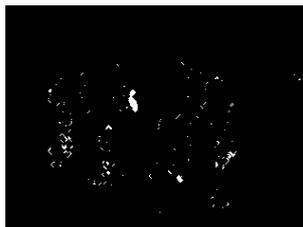
表格 4-1 Test Video1 輸出影像比對表

Data type	Marked Source	Detected Foreground	Foreground difference compares with Double Type
Double			
Float			
Full 32-bit			
Partial 32-bit			
16-bit			

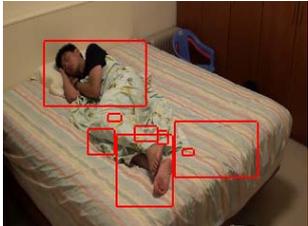
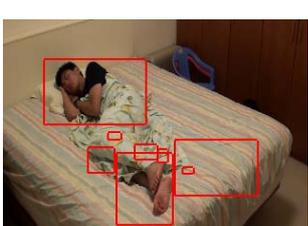
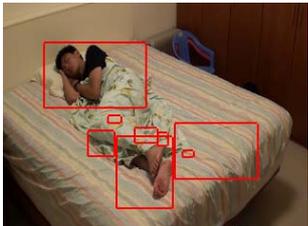
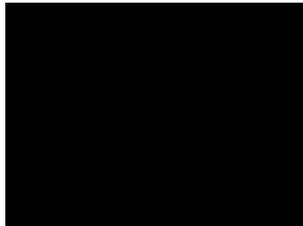
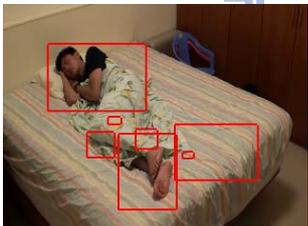
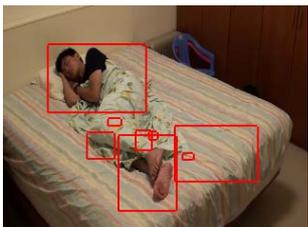
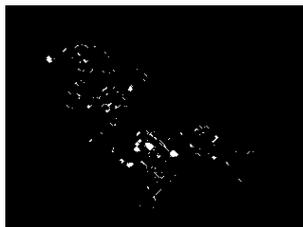
表格 4-2 Test Video2 輸出影像比對表

Data type	Marked Source	Detected Foreground	Foreground difference compares with Double Type
Double			
Float			
Full 32-bit			
Partial 32-bit			
16-bit			

表格 4-3 Test Video3 輸出影像比對表

Data type	Marked Source	Detected Foreground	Foreground difference compares with Double Type
Double			
Float			
Full 32-bit			
Partial 32-bit			
16-bit			

表格 4-4 Test Video4 輸出影像比對表

Data type	Marked Source	Detected Foreground	Foreground difference compares with Double Type
Double			
Float			
Full 32-bit			
Partial 32-bit			
16-bit			

4.2 GMM 去浮點數化之處理速度分析

因為不同的資料型態在 PC 上處理會有不同的處理速度，我們以一個乘法運算的程式，計算以 Double、Float 及去浮點數化的三種格式作乘法運算所需要的時間。以 Double 格式為例，如圖 4-1 所示，計算處理所需的時間，然後將其中乘法運算的資料格式更換成其他去浮點數化資料格式，並加入包含去浮點數化所需的資料位移步驟，以及將資料搬移至暫存的緩衝記憶體的過程來做測試，如圖 4-2 所示。

```
count = 5000;
for(i = 0;i<count;i++)
{
    for(j=0;j < count;j++)
    {
        a_double = b_double*c_double;
    }
}
```

圖 4-1 乘法運算的基本測試迴圈程式碼

```
Double:
    a_double = b_double*c_double;
Float:
    a_float = b_float*c_float;
Full 32-bit:
    a_long = b_int32;
    a_long = c_int32;
    a_long = b_long*c_long;
    b_int32 = a_long >> 16;
Partial32-bit
    a_int32 = b_int16*c_int16;
    b_int16 = a_int32>>8;
16-bit
    b_int32 = b_short
    c_int32 = c_short
    a_int32 = b_int32 * c_int32;
    a_short = a_int32>>8;
```

圖 4-2 各種資料型的乘法運算基本測試態運算程式碼

經過基本的乘法運算試測後，我們得到了上述各種資料格式乘法運算的執行時間，如表格 4-5。然後以 Double 的格式乘法運算的執行時間為基底，計算與各種資料格式乘法運算的執行時間的比例，這樣可得到 Float 格式以及各種去浮點數化格式相對於 Double 格式的相對時間。再將時間取倒數，即得到了對 Double 格式乘法運算的相對速度。我們發現 Full 32-bit 格式(0.352)的速度最慢，因為計算時須先將資料暫存到 64 位元的緩衝記憶體，除了搬移資料的動作外，使用 64 位元的緩衝記憶體也導致速度下降。Partial 32-bit 格式(0.998)的速度與 Double 格式(1.0)的速度相近，因為他在計算完結果後需要再增加一個位移的動作，否則理論上的整數格式的处理速度應該會比 Double 的格式較快。而 16-bit 格式(0.694)的速度，並沒有想像中的更快，因為計算的過程中增加了將資料搬移至 32 位元緩衝記憶體的動作，以及將最後計算結果位移的動作。

將 GMM 程式的視訊處理的測試時間，如表格 4-6 所示，與基本乘法的測試結果做比對，如表格 4-7 所示。除了 16-bit 格式結果差距較大外，其餘的格式接與基本乘法的測試結果相近，其主要原因為在 16-bit 格式的運算過程中，使用了許多標準的 32 位元暫存的緩衝記憶體，除了最後的計算結果外，運算的過程中並不需要每次再做資料搬移到 32 位元緩衝記憶體的動作，因此節省了許多將資料搬移至暫存的緩衝記憶體的時間。

表格 4-5 以基本乘法計算測試之速度表

Data Type	Double	Float	Full 32-bit	Partial 32-bit	16-bit
Time (ms)	56.064	56.144	159.179	56.180	80.763
Normal	1.000	1.001	2.839	1.002	1.441
Speed	1.000	0.999	0.352	0.998	0.694

表格 4-6 Test Video 以 GMM 程式作前景分析之處理速度表

Data Type	Double	Float	Full 32-bit	Partial 32-bit	16-bit
TestVideo1	25.952	28.187	9.295	25.133	25.421
TestVideo2	25.757	28.185	8.835	24.179	24.980
TestVideo3	26.177	28.187	9.495	25.533	25.921
TestVideo4	25.585	28.807	9.095	23.956	24.676

單位: fps

表格 4-7 Test Video 以 GMM 程式作前景分析之速度比較表

Data Type	Double	Float	Full 32-bit	Partial 32-bit	16-bit
TestVideo1	1.000	1.086	0.358	0.968	0.980
TestVideo2	1.000	1.094	0.343	0.939	0.970
TestVideo3	1.000	1.077	0.363	0.975	0.990
TestVideo4	1.000	1.126	0.355	0.936	0.964

除了利用乘法運算的測試方法，我們更進一步分析了在去浮點數化的資料格式之 GMM 演算法程式中的指令複雜度，程式碼如圖 4-3~圖 4-6 所以，最後將其匯整至表格 4-8。從指令複雜度計算的結果，可以看出使用去浮點數化的格式程式碼之指令複雜度，明顯都比使用 Double 格式、Float 格式之程式碼的指令複雜度高出許多。所以雖然在理論上，使用整數運算的處理速度應該會比使用浮點數格式運算的處理速度快很多，但是卻無法明顯反應在去浮點數化後的 GMM 程式之影像處理的輸出結果。不過至少在 Partial 32-bit 格式及 16-bit 格式，雖然其指令複雜度高出 Double 格式許多，但是因為未使用到 64 位元的緩衝記憶體，所以其處理速度可以維持接近 Double 格式的速度。

表格 4-8 GMM 涵式複雜度之分析表

Data Type	Double	Float	Full 32-bit	Partial 32-bit	16-bit
Check matched Gaussian	83	83	155	137	146
Update foreground	14	14	14	14	14
Update Gaussian	70	70	124	106	113
Update background	21	21	24	24	24
Sum	188	188	317	281	297

單位：Statement

Double, Float case :

1. $temp_1_{double} = (I_{t,x} - \mu_{t-1,x,n})$
2. $temp_1_{double} = temp_1_{double} \times temp_1_{double}$
3. $temp_2_{double} = Var_{t-1,x,n} \times T_{\sigma}^2$
4. **If** ($temp_1_{double} < temp_2_{double}$)
- ...

Sum = 4

(a)

Full 32-bit case:

```
#define MEAN_SHIFT_BITS 24
#define MEAN_2_VAR_BITS 8
#define VAR_SHIFT_BITS 16
```

1. $I_{t,x} = I_{t,x} \ll MEAN_SHIFT_BITS$
2. **If** ($I_{t,x} > \mu_{t-1,x,n}$)
3. $temp_1_{64bit} = I_{t,x} - \mu_{t-1,x,n}$
4. **else**
5. $temp_1_{64bit} = \mu_{t-1,x,n} - I_{t,x}$
6. $temp_1_{64bit} = temp_1_{64bit} \times temp_1_{64bit}$
7. $temp_1_{64bit} = temp_1_{64bit} \gg MEAN_2_VAR_BITS$
8. $temp_2_{64bit} = Var_{t-1,x,n}$
9. $temp_{64bit} = T_{\sigma}^2$
10. $temp_2_{64bit} = temp_2_{64bit} \times temp_{64bit}$
11. $temp_2_{64bit} = temp_2_{64bit} \gg VAR_SHIFT_BITS$
12. **If** ($temp_1_{64bit} < temp_2_{64bit}$)
- ...

Difference : (12 - 4) × (Gaussian number) × (Channels) = 8 × 3 × 3 = 72

(b)

圖 4-3 Check matched Gaussian 函式的複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.

Partial 32-bit case:

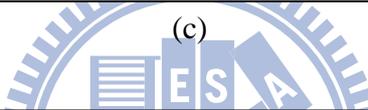
```

#define MEAN_SHIFT_BITS 8
#define MEAN_2_VAR_BITS 0
#define VAR_SHIFT_BITS 8

1.  $I_{t,x} = I_{t,x} \ll \text{MEAN\_SHIFT\_BITS}$ 
2. If ( $I_{t,x} > \mu_{t-1,x,n}$ )
3.      $\text{temp\_1}_{32it} = I_{t,x} - \mu_{t-1,x,n}$ 
4. else
5.      $\text{temp\_1}_{32bit} = \mu_{t-1,x,n} - I_{t,x}$ 
6.  $\text{temp\_1}_{32bit} = \text{temp\_1}_{32bit} \times \text{temp\_1}_{32bit}$ 
7.  $\text{temp\_1}_{32bit} = \text{temp\_1}_{32bit} \gg \text{MEAN\_2\_VAR\_BITS}$ 
8.  $\text{temp\_2}_{32bit} = \text{Var}_{t-1,x,n} \times T\sigma^2$ 
9.  $\text{temp\_2}_{32bit} = \text{temp\_2}_{32bit} \gg \text{VAR\_SHIFT\_BITS}$ 
10. If ( $\text{temp\_1}_{32it} < \text{temp\_2}_{32it}$ )
...

```

Difference : $(10 - 4) \times (\text{Gaussian number}) \times (\text{Channels}) = 6 \times 3 \times 3 = 54$



16-bit case:

```

#define MEAN_SHIFT_BITS 8
#define MEAN_2_VAR_BITS 0
#define VAR_SHIFT_BITS 8

1.  $I_{t,x} = I_{t,x} \ll \text{MEAN\_SHIFT\_BITS}$ 
2. If ( $I_{t,x} > \mu_{t-1,x,n}$ )
3.      $\text{temp\_1}_{32bit} = I_{t,x} - \mu_{t-1,x,n}$ 
4. else
5.      $\text{temp\_1}_{32bit} = \mu_{t-1,x,n} - I_{t,x}$ 
6.  $\text{temp\_1}_{32bit} = \text{temp\_1}_{32bit} \times \text{temp\_1}_{32bit}$ 
7.  $\text{temp\_1}_{32bit} = \text{temp\_1}_{32bit} \gg \text{MEAN\_2\_VAR\_BITS}$ 
8.  $\text{temp\_2}_{32bit} = \text{Var}_{t-1,x,n}$ 
9.  $\text{temp\_2}_{32bit} = \text{temp\_2}_{32bit} \times T\sigma^2$ 
10.  $\text{temp\_2}_{32bit} = \text{temp\_2}_{32bit} \gg \text{VAR\_SHIFT\_BITS}$ 
11. If ( $\text{temp\_1}_{32bit} < \text{temp\_2}_{32bit}$ )...

```

Difference : $(11 - 4) \times (\text{Gaussian number}) \times (\text{Channels}) = 7 \times 3 \times 3 = 63$

(d)

圖 4-3 (續)

Double, Float case :

1. $temp_I_{double} = (1 - \beta_b)$
2. $temp_I_{double} = temp_I_{double} \times \eta_{t-1,x}(\beta)$
3. $\eta_{t,x} = temp_I_{double} + \eta_b \beta_b$

Sum = 2

(a)

Full 32-bit case:

- ```
#define WEIGHT_SHIFT_BITS 15
```
1.  $temp\_I_{64\ bit} = (1 - \beta_b)$
  2.  $temp\_I_{64\ bit} = \eta_{t-1,x}(\beta)$
  3.  $temp\_I_{64\ bit} = temp\_I_{64\ bit} \times temp\_2_{64\ bit}$
  4.  $temp\_I_{64\ bit} = temp\_I_{64\ bit} + \eta_b \beta_b$
  5.  $\eta_{t,x} = temp\_I_{64\ bit} \gg WEIGHT\_SHIFT\_BITS$

**Difference : (5 - 3) = 2**

(b)

**Partial 32-bit case:**

- ```
#define WEIGHT_SHIFT_BITS 15
```
1. $temp_I_{32bit} = (1 - \beta_b)$
 2. $temp_I_{32\ bit} = temp_I_{32\ bit} \times \eta_{t-1,x}(\beta)$
 3. $temp_I_{32bit} = temp_I_{32bit} + \eta_b \beta_b$
 4. $\eta_{t,x} = temp_I_{32bit} \gg WEIGHT_SHIFT_BITS$

Difference : (4 - 3) = 1

(c)

16-bit case:

- ```
#define WEIGHT_SHIFT_BITS 15
```
1.  $temp\_I_{32bit} = (1 - \beta_b)$
  2.  $temp\_I_{32bit} = \eta_{t-1,x}(\beta)$
  3.  $temp\_I_{32bit} = temp\_I_{32bit} \times temp\_2_{32bit}$
  4.  $temp\_I_{32bit} = temp\_I_{32bit} + \eta_b \beta_b$
  5.  $\eta_{t,x} = temp\_I_{32bit} \gg WEIGHT\_SHIFT\_BITS$

**Difference : (5 - 3) = 2**

(d)

圖 4-4 Get weight learning rate 函式的指令複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.

**Double, Float case :**

$$1. \quad \mathbf{temp\_I}_{double} = (1 - \eta_{t,x}(\beta)) \quad // \eta_{t,x}(\beta) \text{ statements} = 3$$

$$2. \quad \mathbf{W}_{t,x,n} = \mathbf{temp\_I}_{double} \times \mathbf{W}_{t-1,x,n}$$

1. **If Gaussian matched**

$$2. \quad \mathbf{temp\_I}_{double} = (1 - \rho)$$

$$3. \quad \mathbf{temp\_I}_{double} = \mathbf{temp\_I}_{double}$$

$$1. \quad \mathbf{temp\_I}_{double} \times \mu_{t-1,x,n}$$

$$2. \quad \mathbf{temp\_2}_{double} = \rho \times \mathbf{I}_{t,x}$$

$$3. \quad \mu_{t,x,n} = \mathbf{temp\_I}_{double} + \mathbf{temp\_2}_{double}$$

$$4. \quad \mathbf{temp\_I}_{double} = (1 - \rho)$$

$$5. \quad \mathbf{temp\_I}_{double} = \mathbf{temp\_I}_{double} \times \mathbf{Var}_{t-1,x,n}$$

$$6. \quad \mathbf{temp\_2}_{double} = \rho \times \mathbf{Var}_n$$

$$7. \quad \mathbf{Var}_{t,x,n} = \mathbf{temp\_I}_{double} + \mathbf{temp\_2}_{double}$$

$$8. \quad \mathbf{W}_{t,x,n} = \mathbf{W}_{t,x,n} + \eta_{t,x}(\beta)$$

9. **else**

$$10. \quad \mu_{t,x,n} = \mathbf{I}_{t,x}$$

$$11. \quad \mathbf{Var}_{t,x,n} = \mathbf{Var}_0$$

$$12. \quad \mathbf{W}_{t,x,n} = \mathbf{W}_0$$

$$1. \quad \mathbf{Weight}_{sum} = \Sigma \mathbf{W} \quad // \text{Sum weight}$$

$$1. \quad \mathbf{W}_{t,x,n} = \mathbf{W}_{t,x,n} / \mathbf{Weight}_{sum} \quad // \text{Normal weight}$$

**Sum = 12+1+1 = 14**

(a)

圖 4-5 Update Gaussian 函式的指令複雜度分析(a)Double and Float cases. (b)Full 32-bit case.

(c)Partial 32-bit case. (d)16-bit case.

Full 32-bit case:

$$1. \quad temp\_1_{64\text{ bit}} = (1 - \eta_{t,x}(\beta)) \quad // \eta_{t,x}(\beta): \text{statements} = 5$$

$$2. \quad temp\_2_{64\text{ bit}} = W_{t-1,x,n}$$

$$3. \quad temp\_1_{64\text{ bit}} = temp\_1_{64\text{ bit}} \times temp\_2_{64\text{ bit}}$$

$$4. \quad W_{t,x,n} = temp\_1_{64\text{ bit}} \gg \text{WEIGHT\_SHIFT\_BITS}$$

1. If Gaussian matched

$$2. \quad temp\_1_{64\text{ bit}} = (1 - \rho)$$

$$3. \quad temp\_2_{64\text{ bit}} = \mu_{t-1,x,n}$$

$$4. \quad temp_{64\text{ bit}} = temp\_1_{64\text{ bit}} \times temp\_2_{64\text{ bit}}$$

$$5. \quad temp\_1_{64\text{ bit}} = \rho$$

$$6. \quad temp\_2_{64\text{ bit}} = I_{t,x}$$

$$7. \quad temp\_2_{64\text{ bit}} = temp\_1_{64\text{ bit}} \times temp\_2_{64\text{ bit}}$$

$$8. \quad temp\_2_{64\text{ bit}} = temp_{64\text{ bit}} + temp\_2_{64\text{ bit}}$$

$$9. \quad \mu_{t,x,n} = temp\_2_{64\text{ bit}} \gg \text{WEIGHT\_SHIFT\_BITS}$$

$$10. \quad temp\_1_{64\text{ bit}} = (1 - \rho)$$

$$11. \quad temp\_2_{64\text{ bit}} = \text{Var}_{t-1,x,n}$$

$$12. \quad temp_{64\text{ bit}} = temp\_1_{64\text{ bit}} \times temp\_2_{64\text{ bit}}$$

$$13. \quad temp\_1_{64\text{ bit}} = \rho$$

$$14. \quad temp\_2_{64\text{ bit}} = \text{Var}_n$$

$$15. \quad temp\_2_{64\text{ bit}} = temp\_1_{64\text{ bit}} \times temp\_2_{64\text{ bit}}$$

$$16. \quad temp\_2_{64\text{ bit}} = temp_{64\text{ bit}} + temp\_2_{64\text{ bit}}$$

$$17. \quad \text{Var}_{t,x,n} = temp\_2_{64\text{ bit}} \gg \text{WEIGHT\_SHIFT\_BITS}$$

$$18. \quad W_{t,x,n} = W_{t,x,n} + \eta_{t,x}(\beta)$$

19. else

$$20. \quad \mu_{t,x,n} = I_{t,x}$$

$$21. \quad \text{Var}_{t,x,n} = \text{Var}_0$$

$$22. \quad W_{t,x,n} = W_0$$

$$1. \quad \text{Weight}_{sum,64\text{ bit}} = \sum W \quad // \text{Sum weight}$$

$$1. \quad temp\_1_{64\text{ bit}} = W_{t,x,n} \quad // \text{Normal weight}$$

$$2. \quad temp\_1_{64\text{ bit}} = temp\_1_{64\text{ bit}} \ll \text{WEIGHT\_SHIFT\_BITS}$$

$$3. \quad temp_{64\text{ bit}} = temp\_1_{64\text{ bit}} / \text{Weight}_{sum,64\text{ bit}}$$

$$4. \quad W_{t,x,n} = temp_{64\text{ bit}}$$

Difference :  $((9-5)+(4-1)) \times (\text{Gaussian number}) + (22-11) \times (\text{Channels}) = 7 \times 3 + 11 \times 3 = 54$

(b)

圖 4-5 (續)

**Partial 32-bit case:**

1.  $temp\_1_{32\ bit} = (1 - \eta_{t,x}(\beta))$  //  $\eta_{t,x}(\beta)$  statements = 4
2.  $temp\_1_{32\ bit} = temp\_1_{32\ bit} \times W_{t-1,x,n}$
3.  $W_{t,x,n} = temp\_1_{32\ bit} \gg \text{WEIGHT\_SHIFT\_BITS}$

**1. If Gaussian matched**

2.  $temp\_1_{32\ bit} = (1 - \rho)$
3.  $temp\_2_{32\ bit} = temp\_1_{32\ bit} \times \mu_{t-1,x,n}$
4.  $temp\_1_{32\ bit} = \rho$
5.  $temp\_2_{32\ bit} = temp\_1_{32\ bit} \times I_{t,x}$
6.  $temp\_2_{32\ bit} = temp\_2_{32\ bit} + temp\_2_{32\ bit}$
7.  $\mu_{t,x,n} = temp\_2_{32\ bit} \gg \text{WEIGHT\_SHIFT\_BITS}$
8.  $temp\_1_{32\ bit} = (1 - \rho)$
9.  $temp\_1_{32\ bit} = temp\_1_{32\ bit} \gg 2$
10.  $temp\_2_{32\ bit} = temp\_1_{32\ bit} \times Var_{t-1,x,n}$
11.  $temp\_1_{32\ bit} = \rho$
12.  $temp\_1_{32\ bit} = temp\_1_{32\ bit} \gg 2$
13.  $temp\_2_{32\ bit} = temp\_1_{32\ bit}$
14.  $temp\_2_{32\ bit} = temp\_2_{32\ bit} + temp\_2_{32\ bit}$
15.  $Var_{t,x,n} = temp\_2_{32\ bit} \gg \text{VAR\_WEIGHT\_SHIFT\_BITS}$
16.  $W_{t,x,n} = W_{t,x,n} + \eta_{t,x}(\beta)$
17. **else**
18.  $\mu_{t,x,n} = I_{t,x}$
19.  $Var_{t,x,n} = Var_0$
20.  $W_{t,x,n} = W_0$

1.  $Weight_{sum,32\ bit} = \Sigma W$

1.  $W_{t,x,n} = W_{t,x,n} \ll \text{WEIGHT\_SHIFT\_BITS}$

2.  $W_{t,x,n} = W_{t,x,n} / Weight_{sum,32\ bit}$

**Difference : ((7-5)+(2-1))× (Gaussian number)+ (20-11) × (Channels)**

**= 3 × 3 + 9 × 3 = 36**

(c)

圖 4-5 (續)

16-bit case:

1.  $temp\_I_{32\ bit} = (I - \eta_{t,x}(\beta))$  //  $\eta_{t,x}(\beta)$  statements = 5
2.  $temp\_I_{32\ bit} = temp\_I_{32\ bit} \times W_{t-1,x,n}$
3.  $W_{t,x,n} = temp\_I_{32\ bit} \gg \text{WEIGHT\_SHIFT\_BITS}$

1. **If Gaussian matched**

2.  $temp\_I_{32\ bit} = (I - \rho)$
3.  $temp_{32\ bit} = temp\_I_{32\ bit} \times \mu_{t-1,x,n}$
4.  $temp\_2_{32\ bit} = \rho \times I_{t,x}$
5.  $temp\_2_{32\ bit} = temp_{32\ bit} + temp\_2_{32\ bit}$
6.  $\mu_{t,x,n} = temp\_2_{32\ bit} \gg \text{WEIGHT\_SHIFT\_BITS}$
7.  $temp\_I_{32\ bit} = (I - \rho)$
8.  $temp_{32\ bit} = temp\_I_{32\ bit} \times Var_{t-1,x,n}$
9.  $temp\_2_{32\ bit} = \rho \times Var_n$
10.  $temp\_2_{32\ bit} = temp_{32\ bit} + temp\_2_{32\ bit}$
11.  $Var_{t,x,n} = temp\_2_{32\ bit} \gg \text{WEIGHT\_SHIFT\_BITS}$
12.  $W_{t,x,n} = W_{t,x,n} + \eta_{t,x}(\beta)$
13. **else**
14.  $\mu_{t,x,n} = I_{t,x}$
15.  $Var_{t,x,n} = Var_0$
16.  $W_{t,x,n} = W_0$

1.  $Weight_{sum,32\ bit} = \Sigma W$

1.  $temp\_I_{32\ bit} = W_{t,x,n}$
2.  $temp\_I_{32\ bit} = temp\_I_{32\ bit} \ll \text{WEIGHT\_SHIFT\_BITS}$
3.  $temp_{32\ bit} = temp\_I_{32\ bit} / Weight_{sum,32\ bit}$
4.  $W_{t,x,n} = temp_{32\ bit}$

**Difference : ((8-5)+(4-1)) × (Gaussian number)+ (19-11) × (Channels)**  
**= 6 × 3 + 8 × 3 = 42**

(d)

圖 4-5 (續)

**Double, Float case :**

1.  $\mathbf{Bg\_Data} = \mu_{max\ weight}$

**Sum = 1**

(a)

**Full 32-bit case:**

1.  $\mathbf{Bg\_Data} = \mu_{max\ weight}$

2.  $\mathbf{Bg\_Data} = \mathbf{Bg\_Data} \gg \mathbf{MEAN\_SHIFT\_BITS}$

**Difference :  $(2 - 1) \times (\mathbf{Channels}) = 3$**

(b)

**Partial 32-bit case:**

1.  $\mathbf{Bg\_Data} = \mu_{max\ weight}$

2.  $\mathbf{Bg\_Data} = \mathbf{Bg\_Data} \gg \mathbf{MEAN\_SHIFT\_BITS}$

**Difference :  $(2 - 1) \times (\mathbf{Channels}) = 3$**

(c)

**16-bit case:**

1.  $\mathbf{Bg\_Data} = \mu_{max\ weight}$

2.  $\mathbf{Bg\_Data} = \mathbf{Bg\_Data} \gg \mathbf{MEAN\_SHIFT\_BITS}$

**Difference :  $(2 - 1) \times (\mathbf{Channels}) = 3$**

(d)

圖 4-6 Update background 函式的指令複雜度分析(a)Double and Float cases. (b)Full 32-bit case. (c)Partial 32-bit case. (d)16-bit case.

### 4.3 GMM 去浮點數化之記憶體分析

為統計 GMM 程式中的記憶體，我們先將 GMM 程式中所有的變數分成全域變數及區域變數，如圖 4-7、圖 4-8，再分別計算所有使用的記憶體數量。先分析全域變數的部分，如表格 4-9、表格 4-10 所示，在統計的數量中，最大的記憶體使用量為維護高斯分佈的變數：平均數、變異數及加權比重，幾乎佔了全域變數的 80%。所以如果能有效減少這些變數的資料格式大小，即可有效降低全域變數之記憶體的使用量，之後再將區域變數與全域變數合併作比較，如表格 4-11 所示。由比較結果發現區域變數在 GMM 程式中所佔的比例非常的微小，而全域變數在 GMM 程式中所佔的比例高達整體記憶體的 99%。我們進而確定如果能降低平均數、變異數及加權比重的資料格式大小，即可使 GMM 程式的記憶體需求降低。在 GMM 程式中使用去浮點數化的格式，如使用 16-bit 格式所使用的記憶體已降低至原來 Double 格式的 1/4。

```
#define GAUSS_NUM 3
#define MAGE_HEIGHT 240
#define IMAGE_WIDTH 320
#define IMAGE_CHANNELS 3
#define MAX_OBJ_NUM 4800

double GaussPointWeight[GAUSS_NUM][IMAGE_HEIGHT][IMAGE_WIDTH];
double GaussPointMean[GAUSS_NUM][IMAGE_HEIGHT][IMAGE_WIDTH][IMAGE_CHANNELS];
double GaussPointVar[GAUSS_NUM][IMAGE_HEIGHT][IMAGE_WIDTH][IMAGE_CHANNELS];
double GaussPointEata[IMAGE_HEIGHT][IMAGE_WIDTH];
unsigned char GaussPointWeightLearningMode[IMAGE_HEIGHT][IMAGE_WIDTH];
unsigned int Match_Index[IMAGE_HEIGHT][IMAGE_WIDTH];
double Match_Var[IMAGE_HEIGHT][IMAGE_WIDTH][IMAGE_CHANNELS];
CC_Obj CC_Obj_buffer[2][MAX_OBJ_NUM];
CC_Obj *CC_Obj_Pre_Double;
CC_Obj *CC_Obj_Curr_Double;
unsigned int MaxObj_Curr_Double;
unsigned int MaxObj_Pre_Double;
unsigned int Obj_Num_Double;
```

圖 4-7 GMM 程式中之全域變數

```

double GetWeightLearningRate(unsigned char mode,double eata_t_1)
 double eata

void InitGMMGauss(unsigned char *data_ptr)
 short x,y,ch,g;

unsigned int UpdateBGModel(unsigned char *data_ptr, unsigned char *BG_ptr,unsigned char *FR_ptr)
 unsigned int x,y,ch;
 unsigned int g,max_weight_index,min_weight_index,g_match_index;
 double d[GAUSS_NUM][IMAGE_CHANNELS];
 double var[GAUSS_NUM][IMAGE_CHANNELS];
 double mean[GAUSS_NUM][IMAGE_CHANNELS];
 double input[IMAGE_CHANNELS];
 double min_weight,max_weight,sum_weight;
 double t_val_2;
 double learning_rate;
 double weight_learning_rate;
 unsigned int color;
 double w_min;
 unsigned char *data_head;
 unsigned char* fr_head;
 unsigned char* mode_ptr

```

圖 4-8 GMM 程式中之區域變數

表格 4-9 GMM 程式中全域變數記憶體分析表

| Data Type                    | Double | Float | Full 32-bit | Partial 32-bit | 16-bit |
|------------------------------|--------|-------|-------------|----------------|--------|
| Weight                       | 1800   | 900   | 900         | 900            | 450    |
| Mean                         | 5400   | 2700  | 2700        | 2700           | 1350   |
| Variance                     | 5400   | 2700  | 2700        | 2700           | 1350   |
| Eata                         | 600    | 300   | 300         | 300            | 150    |
| GaussPointWeightLearningMode | 75     | 75    | 75          | 75             | 75     |
| Match Index                  | 75     | 75    | 75          | 75             | 75     |
| Match Variance               | 1800   | 900   | 900         | 900            | 450    |

單位：KByte

表格 4-10 GMM 程式中全域變數記憶體比例之對照表

| Data Type                    | Double | Float  | Full 32-bit | Partial 32-bit | 16-bit |
|------------------------------|--------|--------|-------------|----------------|--------|
| Weight                       | 11.88% | 11.76% | 11.76%      | 11.76%         | 11.54% |
| Mean                         | 35.64% | 35.29% | 35.29%      | 35.29%         | 34.62% |
| Variance                     | 35.64% | 35.29% | 35.29%      | 35.29%         | 34.62% |
| Eata                         | 3.96%  | 3.92%  | 3.92%       | 3.92%          | 3.85%  |
| GaussPointWeightLearningMode | 0.50%  | 0.98%  | 0.98%       | 0.98%          | 1.92%  |
| Match Index                  | 0.50%  | 0.98%  | 0.98%       | 0.98%          | 1.92%  |
| Match Variance               | 11.88% | 11.76% | 11.76%      | 11.76%         | 11.54% |

單位：KByte

表格 4-11 GMM 程式中記憶體使用之分析表

| Data Type             | Double  | Float   | Full 32-bit | Partial 32-bit | 16-bit  |
|-----------------------|---------|---------|-------------|----------------|---------|
| Global Variables      | 16350.0 | 8250.00 | 8250.000    | 8250.000       | 4200.00 |
| Connected Component   | 159.375 | 159.375 | 159.375     | 159.375        | 159.375 |
| GetLearningRate       | 0.008   | 0.004   | 0.004       | 0.004          | 0.002   |
| GetWeightLearningRate | 0.008   | 0.004   | 0.004       | 0.004          | 0.002   |
| InitGMMGauss          | 0.008   | 0.008   | 0.008       | 0.008          | 0.008   |
| UpdateBGModel         | 0.360   | 0.192   | 0.185       | 0.192          | 0.144   |
| Sum                   | 16509.9 | 8409.58 | 8409.575    | 8409.583       | 4359.53 |

單位：KByte

#### 4.4 GMM 去浮點數化之準確度分析

GMM 演算法的程式在去浮點數化後，因為每一種資料格式的小數精確度不一樣，所以在 GMM 模型中的更新速度也有所差異。因此使用一個的測試圖檔來測試當前景物離開後，每一種資料格式的 GMM 程式作前景分析之輸出結果，不再偵測成前景所需要的時間(畫面數)，如圖 4-9 所示。其測試結果如表 4-12 所示，可以發現 Partial 32-bit 格式(1998 個畫面)與 16-bit 格式(1998 個畫面)的程式之輸出結果，更新前景的速度明顯比 Double 格式(1590 個畫面)、Float 格式(1590 個畫面)及 Full 32-bit 格式(1590 個畫面)慢很多，所以無法直接以前景重疊的點數來比較輸出結果的準確度。

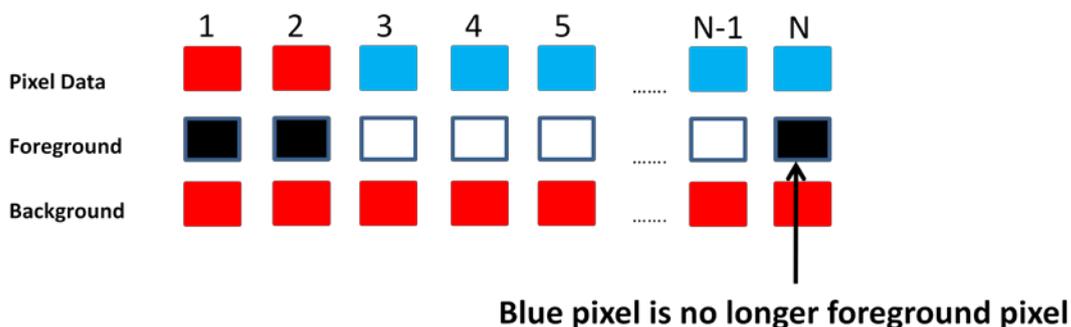


圖 4-9 GMM 演算法前景更新過程之示意圖

表格 4-12 GMM 演算法更新速度測試結果

| Data Type | Double | Float | Full 32-bit | Partial 32-bit | 16-bit |
|-----------|--------|-------|-------------|----------------|--------|
| 前景過程的時間   | 1590   | 1590  | 1590        | 1998           | 1998   |

單位：Frame

由於無法直接以前景重疊的點數來比較輸出結果的準確度，所以我們提出以 object based 來計算準確度，其演算法如圖 4-10 所示。先找出 Double 格式程式輸出結果之前景中所有的物體(object)，並以其為基準，與其他去浮點數化格式程式的輸出結果之前景中的每一個物體作比對，若前景重疊比例大於等於  $T_m$ ，即為符合(Matched)，否則為不符合(Unmatched)。而為了找出在去浮點數化格式程式輸出結果之前景中比 Double 格式程式輸出結果之前景中所多偵測到的物體，我們再以其他去浮點數化格式程式的輸出結果之前景中的每一個物體作基準，與 Double 格式程式輸出結果之前景中所有的物體作比對，若前景重疊比例小於  $T_f$ ，即為不符合(Unmatched)。最後，我們將兩種不符合的個數之總和稱為失敗數。

所以先以 Test Video 1 為測試視訊，找出較佳之  $T_m$  及  $T_f$ ，其測試結果如圖 4-11、圖 4-12 所示。 $T_m$  在 0.5~0.8 之間的前景符合率(Match Rate)較穩定，所以我們取  $T_m = 0.7$ ，而  $T_f$  小於 0.3 之後，對前景失敗率(False Alarm)影響較不大，且 0.3 之後的前景失敗率趨近更穩定，所以取  $T_f = 0.3$ 。

以此方法，分別計算出 Float、Full 32-bit、Partial 32-bit 及 16-bit 格式程式的

前景分析輸出結果的符合率及失敗率，其結果如圖 4-13~圖 4-28 所示。檢視輸出結果，Float 格式程式之前景分析的輸出結果的符合率幾乎與 Double 一致；而 Full 32-bit 格式程式的前景分析輸出結果的前景符合率幾乎為 100%，失敗率極低，效果與 Float 格式程式非常接近；而 Partial 32-bit 格式與 16-bit 格式程式的的前景分析輸出結果，前景符合率大致可維持在 70% 以上。但是前景失敗率只可維持在 40% 以下。由於每一種格式程式的前景更新過程的時間不一樣，所以我們可以看到，當前景物進入場景時的一段時間，會導致符合率下降，經過一段時間的前景更新後，符合率即會上升並趨近穩定。

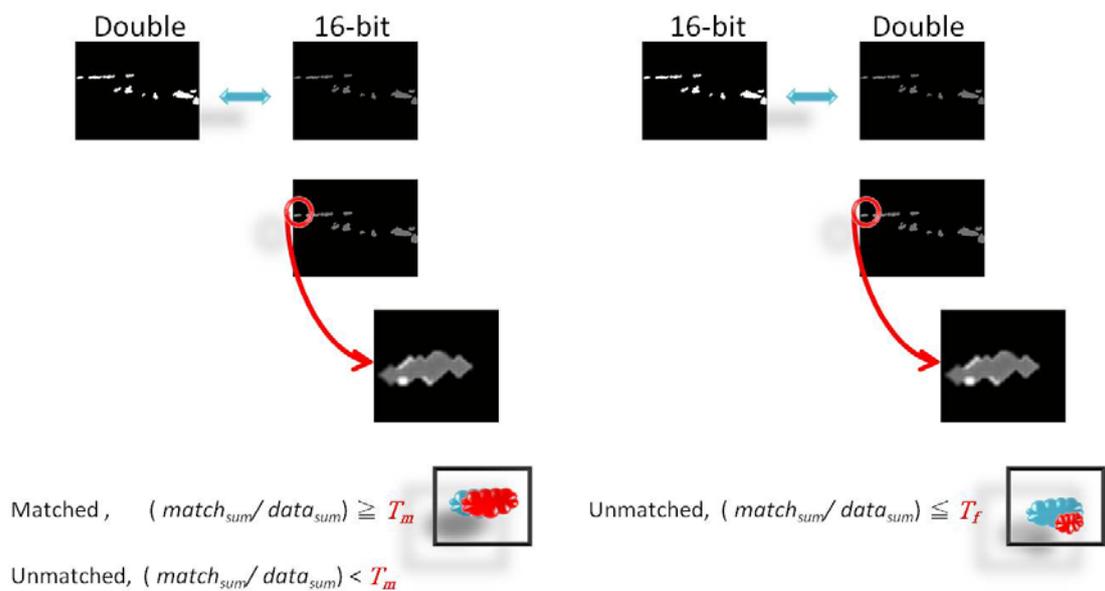


圖 4-10 Object base 演算法之示意圖

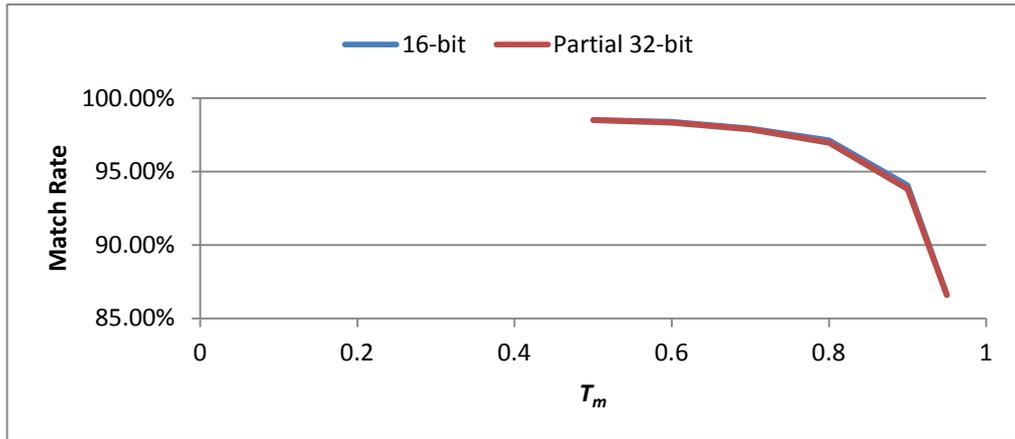


圖 4-11 以 Object based 作前景 Match Rate 之分析結果

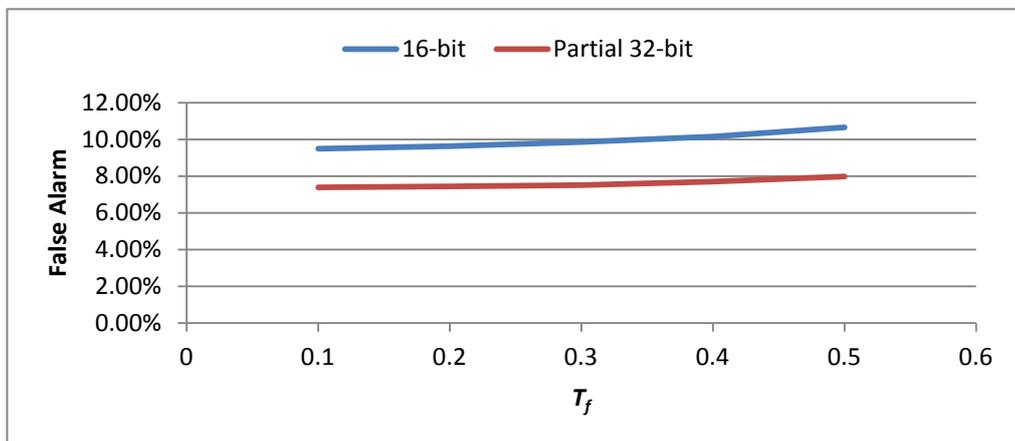


圖 4-12 以 Object based 作前景 False Alarm 之分析結果

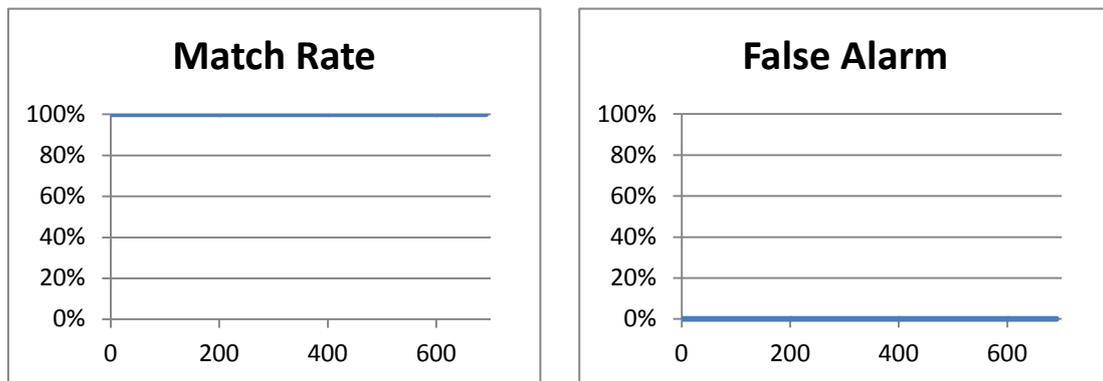


圖 4-13 Test Video 1: Float 格式程式之前景 Match Rate 及 False Alarm 分析結果

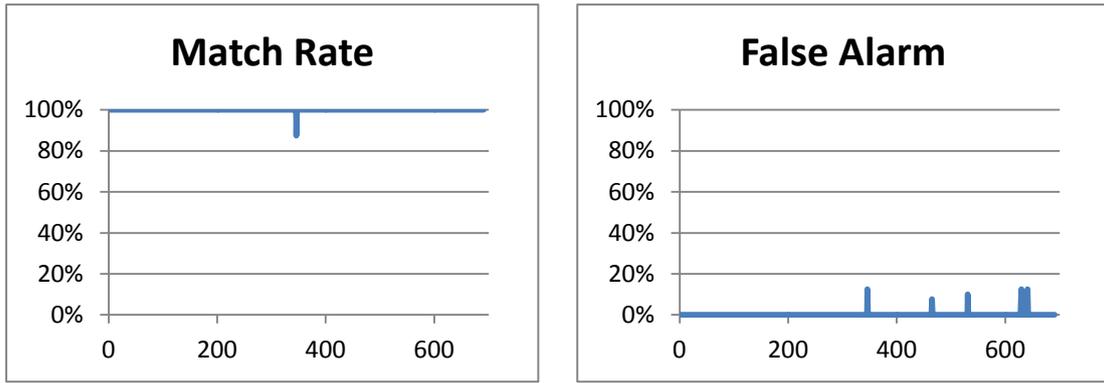


圖 4-14 Test Video 1: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

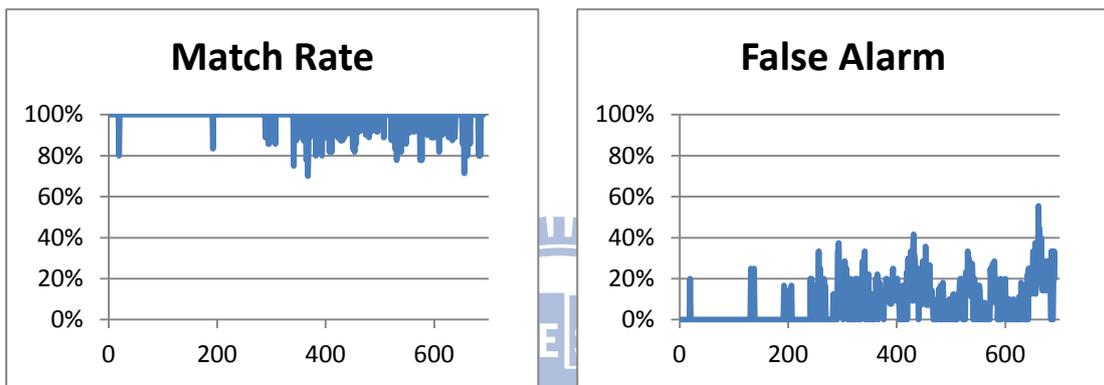


圖 4-15 Test Video 1: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

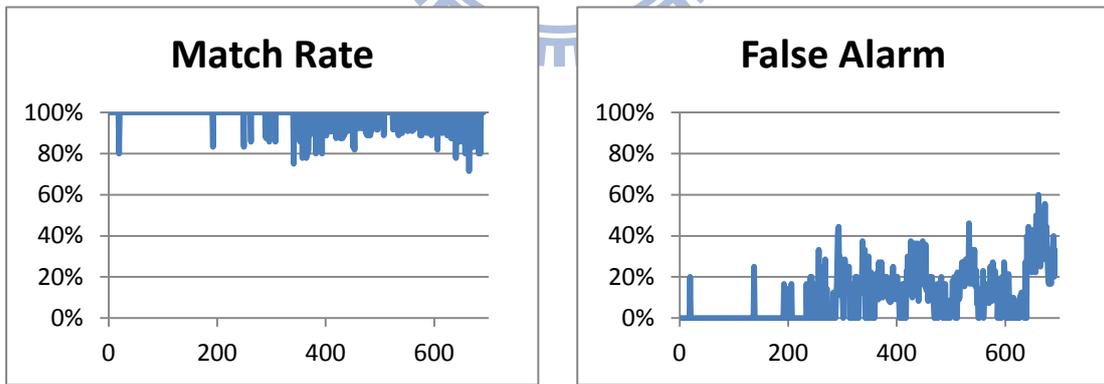


圖 4-16 Test Video 1: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

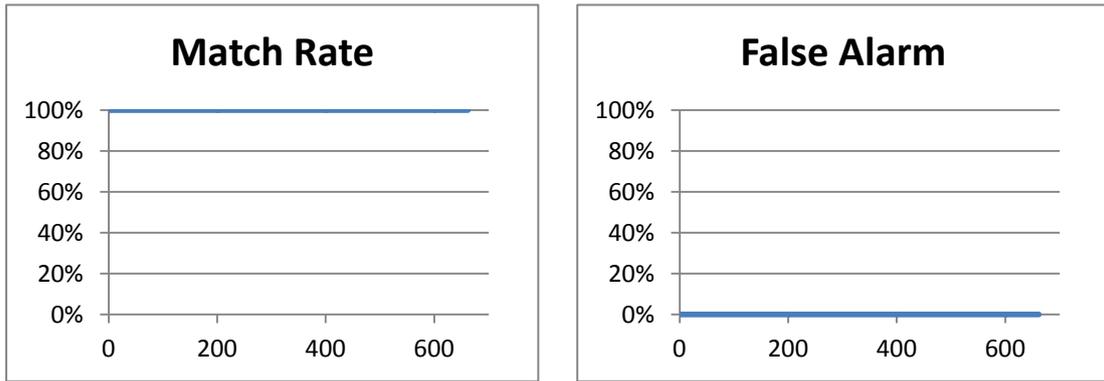


圖 4-17 Test Video 2: Float 格式程式之前景 Match Rate 及 False Alarm 分析結果

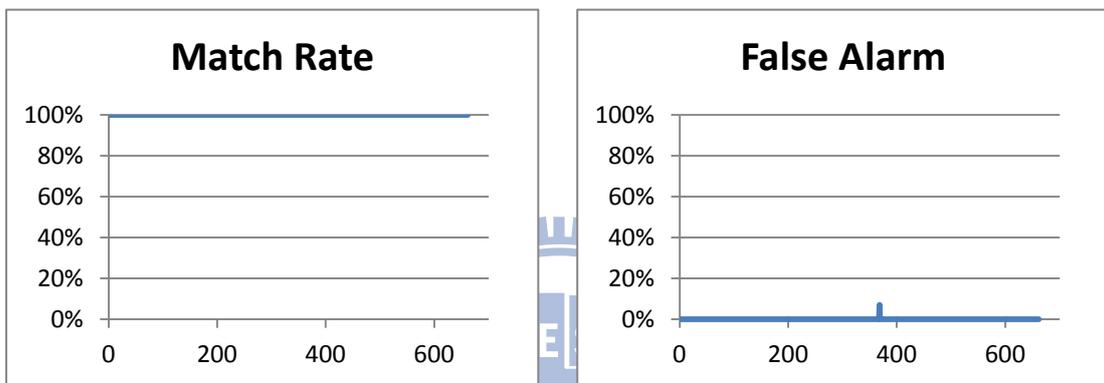


圖 4-18 Test Video 2: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

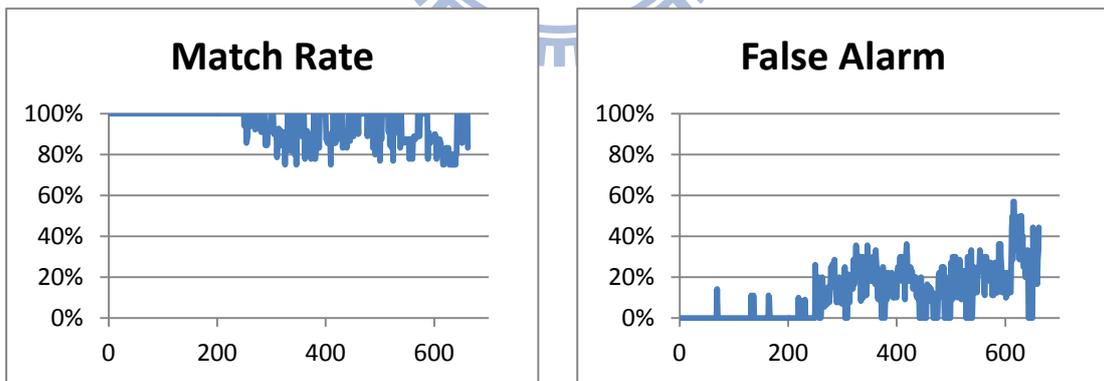


圖 4-19 Test Video 2: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

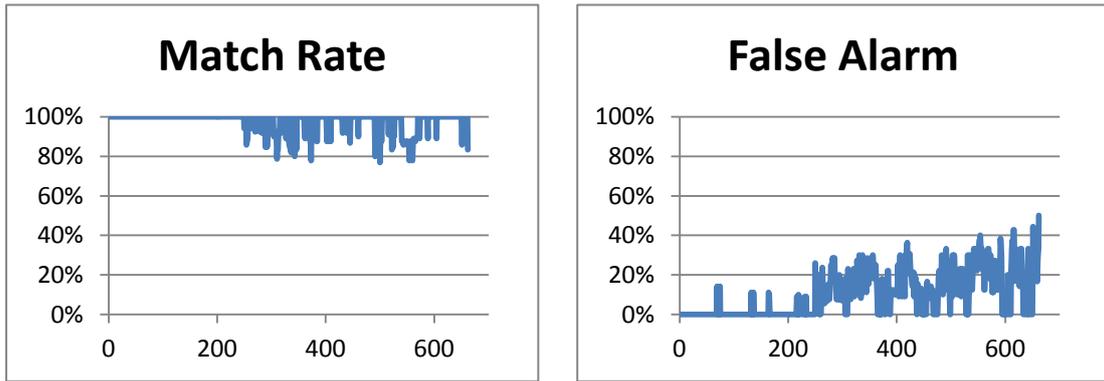


圖 4-20 Test Video 2: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

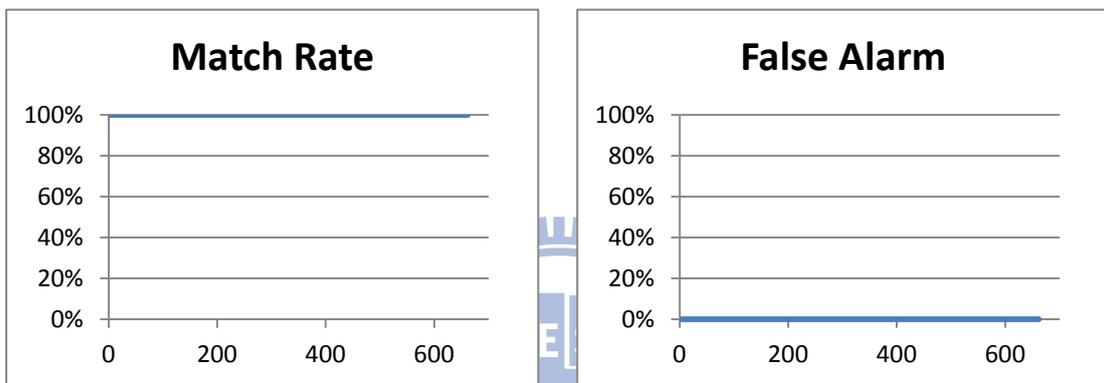


圖 4-21 Test Video 3: Float 格式程式之前景 Match Rate 及 False Alarm 分析結果

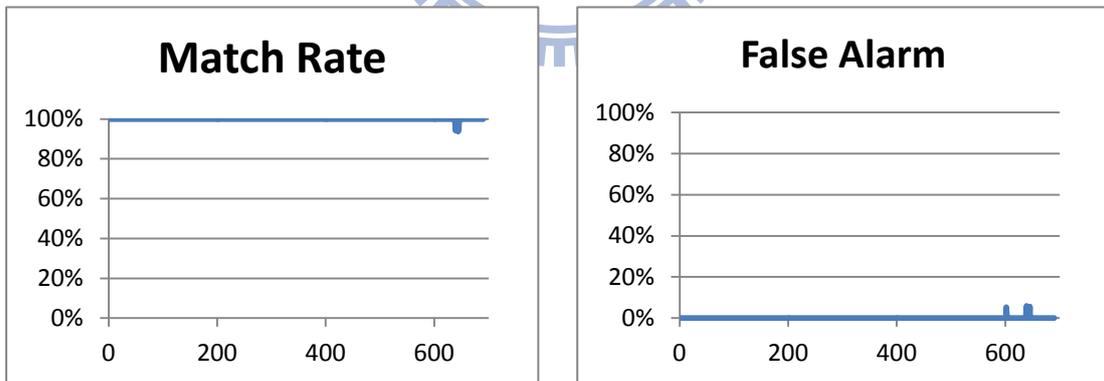


圖 4-22 Test Video 3: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

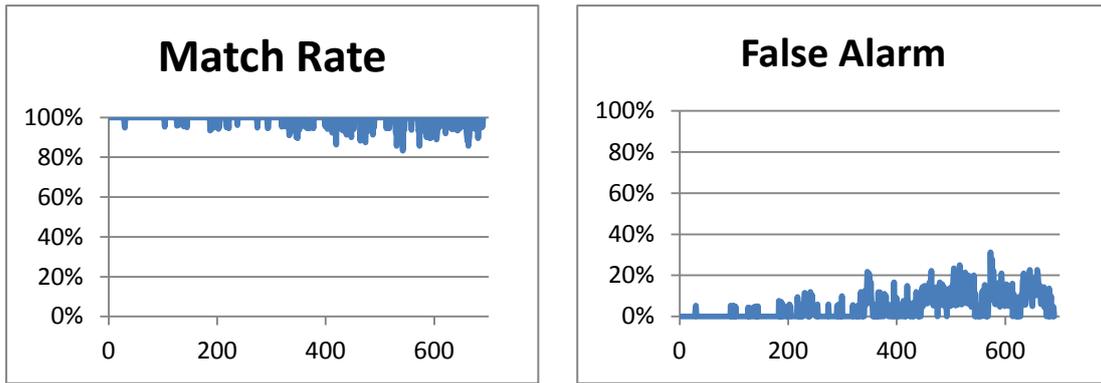


圖 4-23 Test Video 3: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

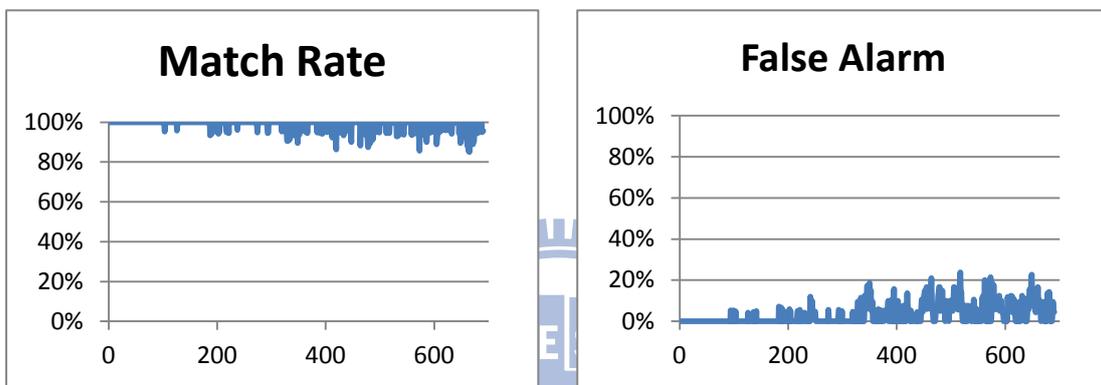


圖 4-24 Test Video 3: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

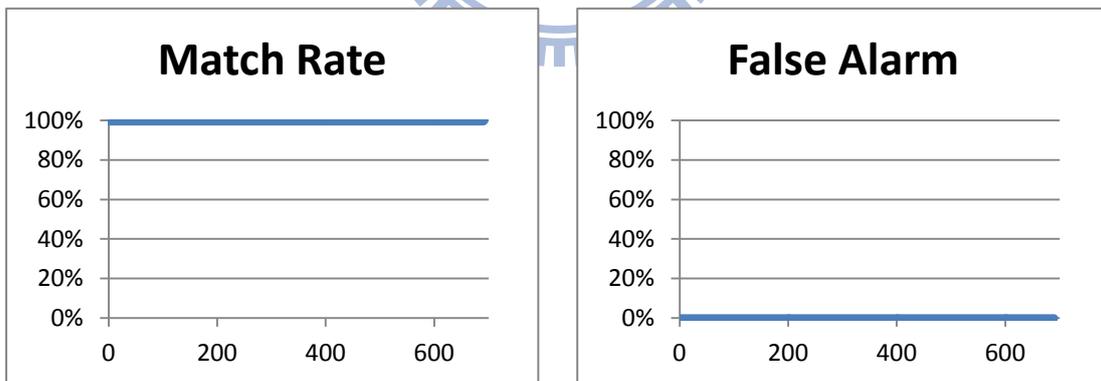


圖 4-25 Test Video 4: Float 格式程式之前景 Match Rate 及 False Alarm 分析結果

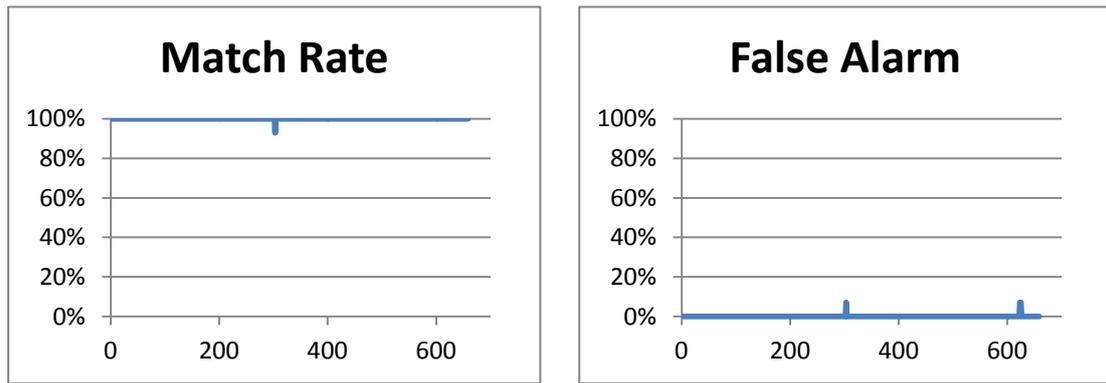


圖 4-26 Test Video 4: Full 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

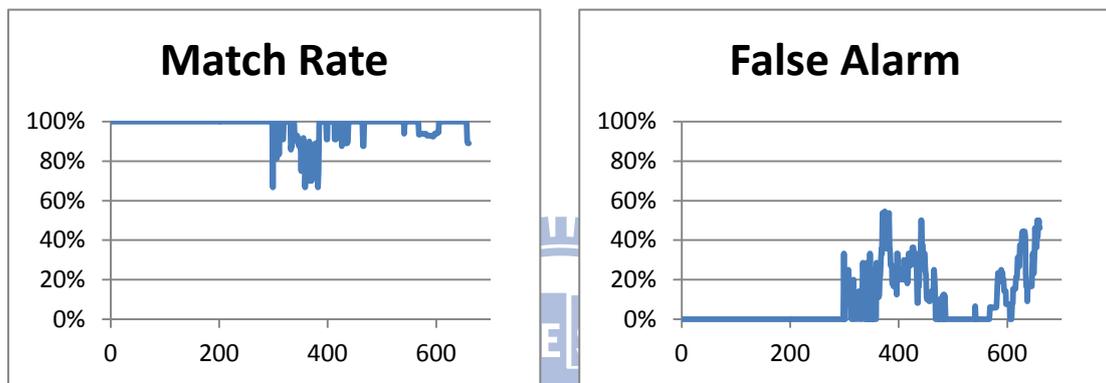


圖 4-27 Test Video 4: Partial 32-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

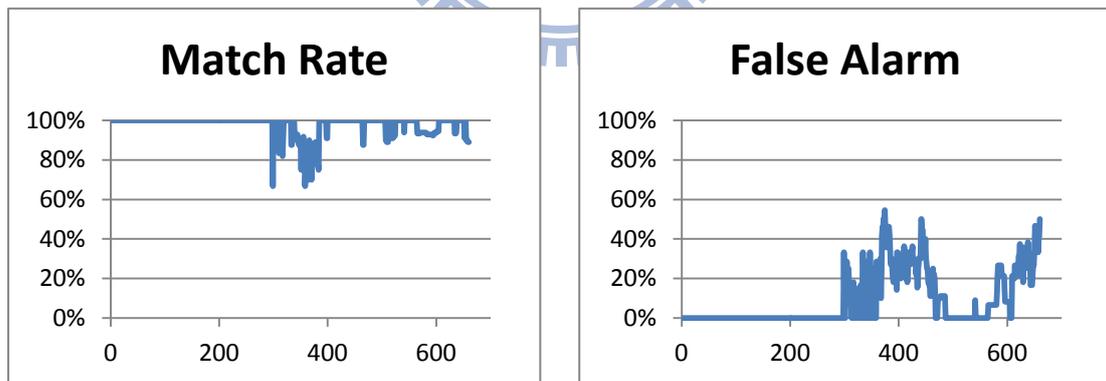


圖 4-28 Test Video 4: 16-bit 格式程式之前景 Match Rate 及 False Alarm 分析結果

因為 Partial 32-bit 格式與 16-bit 格式在程式中的變數精確度極為相近，所以在此進一步比較其格式精確度的差異性，如表格 4-13 所示。在 Partial 32-bit 格式與 16-bit 格式程式中，除了變異數的精確度不同之外，平均數與加權比重的精確

度，兩者是一樣的。所以觀察以 Test Video 1~ Test Video 4 測試之結果的差異主要由變異數的精確度所影響。除此之外，我們在使用 Partial 32-bit 格式時，因為運算的需求，我們在更新變異數時，須將 Learning Rate 右移 2 個位元，是否因為更新之後的變異數之精確度下降，導致其結果接近為變異數為 10.6 格式的程式之前景分析輸出結果，我們將作進一步的探討。

對於上述以 16-bit 格式程式為基礎，先維持平均數及加權比重格式的精確度，來比較變異數小數精確度的差異性，並將變異數的精確度修改為 10.6 的格式並以 Test Video 1 進行測試分析，其結果如圖 4-29 所示。與 Partial 32-bit 格式程式之前景分析輸出結果比較，10.6 格式的前景分析結果並沒有較佳。所以小數點的精確度使用 8 個位元還是有其必要性。

接著比較整數的部分，因為在 Partial 32-bit 格式與 16-bit 格式中，對整數皆有截位的運算，會降低整數部分的精確度，在 Partial 32-bit 格式中變異數的最大值為 1023，而 16-bit 格式中變異數的最大值為 255。若以 Test Video 1 作測試，分別統計出兩種格式程式在運算過程中，超出最大值的次數，也就是受到整數精確度影響的次數，結果如表格 4-14 所示。雖然 16-bit 格式運算超出整數最大值的次數為 Partial 32-bit 格式程式運算中的 11 倍，但是相對於程式對整個視訊分析的運算結果，僅佔了不到 0.5% 的次數。所以由此看來，變異數的整數部分使用 8 個位元的精確度是足夠的。

另一方面，在 GMM 演算法中，在判斷高斯分佈是否為符合時，是以計算輸入像素的變異數是否落在此高斯分佈之變異數的  $T_\sigma^2 (2.5^2 = 6.25)$  倍數內，若以 16-bit 格式中變異數的容許最大值為 255 為例，乘上  $T_\sigma^2$  後為 1593.75，也就是說當像素之變異數值大於 1593.75 後才會受到影響，皆被判斷成不符合而偵測成前景。計算出其相對的  $\sigma$  值為  $\sqrt{1593.75} \cong \pm 40$ ，表示  $\sigma$  值為  $\pm 40$  內，前景背景更新是不受到整數的精確度所限制，若  $\sigma$  值超出此範圍的像素，則會被判斷成前景。而當  $\sigma$  值大於 40，表示像素的值與背景值差異很大，本應該被視為前景，對 GMM

的更新不會造成太大的影響。在 Partial 32-bit 格式中變異數的容許最大值為 1023，所以影響更為微小。

表格 4-13 Partial 32-bit 格式與 16-bit 格式程式中變數精確度比較表

| Data Type | Partial 32-bit | 16-bit     |
|-----------|----------------|------------|
| Mean      | 8.8            | 8.8        |
| Variance  | <b>10.8</b>    | <b>8.8</b> |
| Weight    | 1.15           | 1.15       |

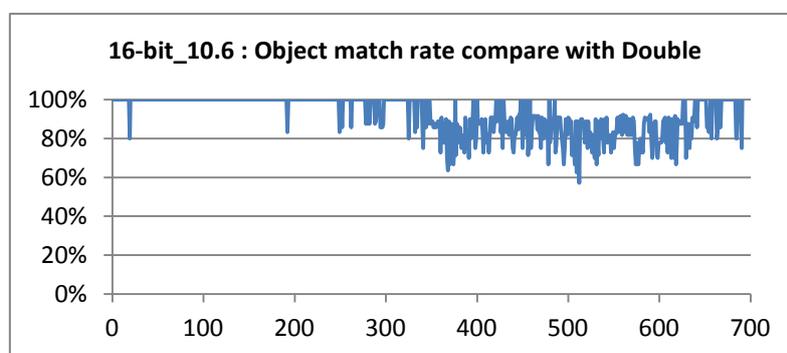


圖 4-29 Test Video 使用 1:16-bit 格式(變異數為 10.6 格式)進行前景分析之 Match Rate 及 False Alarm

表格 4-14 Test Video1 程式運算中變異數大於整數最大值次數比較表

| Data Type           | Partial 32-bit | 16-bit |
|---------------------|----------------|--------|
| 次數                  | 70573          | 794422 |
| %(of 692×320×240×3) | 0.044%         | 0.496% |

以下我們進一步分析單獨提高變異數小數的精確度，與前景符合率之關係。首先我們維持平均數(8.8)以及加權比重(1.15)的格式精確度，並將變異數之整數精確度提高至 16 個位元，以避免因為整數位數不足而導致準確度下降，影響分析結果。接下來將變異數小數部分從 6 個位元開始，不斷增加變異數小數的位元數，直到小數部分到達 16 個位元。也就是調整變異數的資料格式從 16.6 增加到

16.16，利用 Test Video 1~4 作為測試視訊，其結果如圖 4-30~圖 4-33 所示。由程式前景分析輸出結果來看，對於 Test Video 3 而言，變異數部分的精確度並未對前景分析的輸出結果有影響，但是 Test Video 1、2 及 4 之前景符合率會因該精確度之提升而改善，並在小數為 8 個位元之後趨近於穩定，不再有上升的趨勢，前景失敗率也在 8 個位元之後，不再有大幅的下降。所以可以確認，在固定平均數(8.8)以及加權比重(1.15)的格式下，變異數精確度只需要使用 8 個位元作為小數部分。若要再提高前景分析的前景符合率與降低前景失敗率，可能必須另行提高平均數及加權比重格式的精確度。

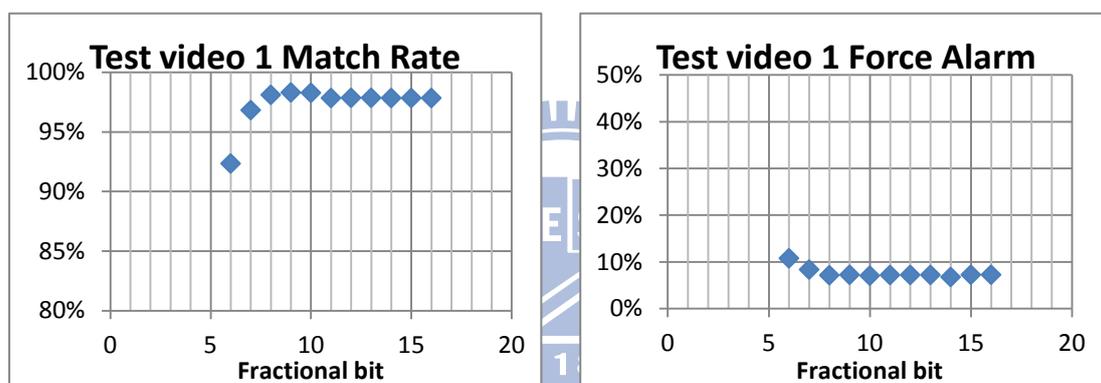


圖 4-30 Test Video 1:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果

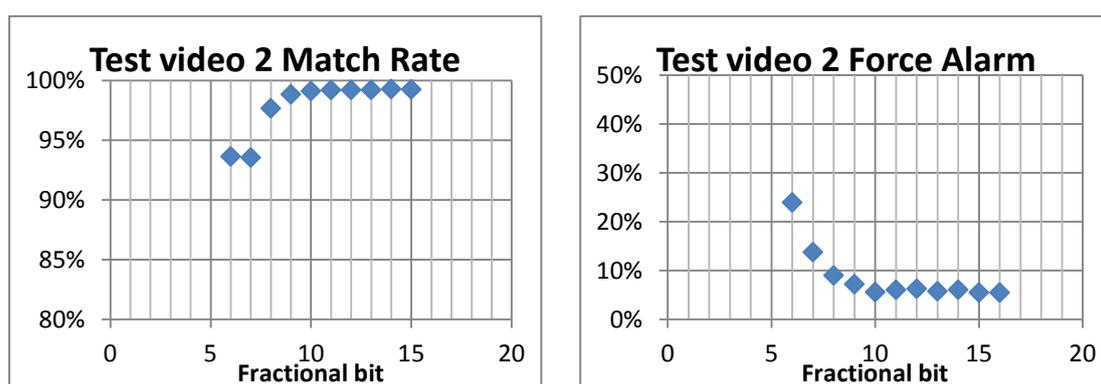


圖 4-31 Test Video 2:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果

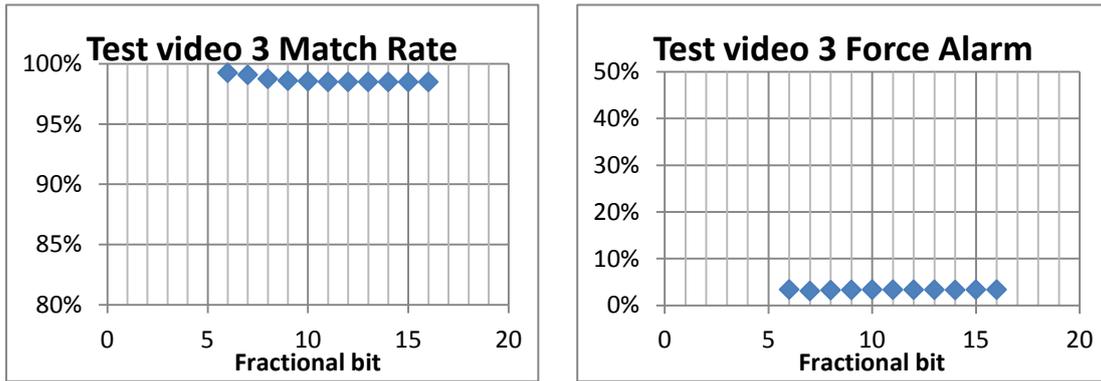


圖 4-32 Test Video 3:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果

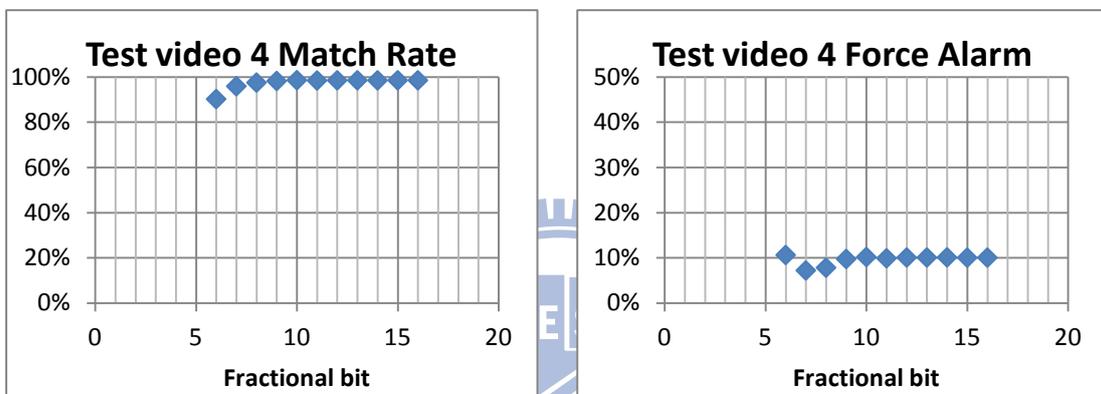


圖 4-33 Test Video 4:變異數格式為 10.6~16.16 之前景 Match Rate 及 False Alarm 分析結果

## 五、結論與未來展望

在本論文中，將 GMM 演算法的程式去浮點數化並在 PC 上實作，使用了四種不同資料格式的程式以產生前景分析之結果，包括(1) Float、(2) Full 32-bit、(3) Partial 32-bit 與(4) 16-bit 等格式，最後提出建議使用之去浮點數化的 16-bit 格式。使用 16-bit 格式程式作前景分析可降低記憶體使用量，僅需使用 Double 格式 1/4 的記憶體，而其前景符合率可達到 Double 格式之前景分析結果的 70%，惟其運算速度也幾乎與 Double 格式一樣。除此之外，透過深入的探討，我們並了解到對變數作去浮點數化時，並不是單一增加變異數的精確度即可提升整體前景符合率，可能同時提高平均數、加權比重格式的精確度，才能對前景符合率的提升有所幫助。而經過去浮點數化後的 GMM 演算法程式，可望能夠提供硬體化或嵌入式系統的快速開發，節省許多開發時人力與時間的成本。

監控系統已經與我們的生活息息相關，希望本論文的研究可以真正應用到監控系統上，再納入增加其他相關應用的演算法，如人臉辨識、手勢辨識等，進行硬體化或與其他各種影像辨識相關的嵌入式系統做結合，以組成一個功能強大的產品。

## 六、參考文獻

- [1] Lianqiang Niu, “A Moving Objects Detection Algorithm Based on Improved Background Subtraction,” in *Proceedings of International Conference on Intelligent Systems Design and Applications*, Volume 3, pp. 604 – 607, 2008.
- [2] Qi Zang and Reinhard Klette, “Robust Background Subtraction and Maintenance,” in *Proceedings of the 17th International Conference on Pattern Recognition*, Volume 2, pp. 90 – 93, 2004.
- [3] Pyung-Soo Hwang, Ki-Yeol Eom, Jae-Young Jung, and Moon-Hyun Kim, “A Statistical Approach to Robust Background Subtraction for Urban Traffic Video,” in *Proceedings of International Workshop on Computer Science and Engineering*, Volume 3, pp. 177 – 181, 2009.
- [4] Shahrizat Shaik Mohamed, Nooritawati Md Tahir, and Ramli Adnan, “Background Modeling and Background Subtraction Performance for Object Detection,” in *Proceedings of International Colloquium on Signal Processing and Its Applications*, pp. 1 – 6, 2010.
- [5] Jian Cheng, Jie Yang, Yue Zhou, and Yingying Cui, “Flexible Background Mixture Models for Foreground Segmentation,” *Image and Vision Computing*, Volume 24, pp. 473 – 482, 2006.
- [6] Saeid Fazli, Hamed Moradi Pour, and Hamed Bouzari, “A Novel GMM-Based Motion Segmentation Method for Complex Background,” in *Proceedings of IEEE GCC Conference & Exhibition*, pp 1 – 5, 2009.
- [7] Faisal Bashir, Ashfaq Khokha, and Dan Schonfeld, “Automatic Object Trajectory -Based Motion Recognition Using Gaussian Mixture Models,” in *Proceedings of IEEE International Conference on Multimedia and Expo.*, pp. 1532 – 1535, 2005.
- [8] Dongxiang Zhou and Hong Zhang, “Modified GMM Background Modeling and Optical Flow for Detection of Moving Objects,” in *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, Volume 3, pp. 2224 – 2229, 2005.
- [9] Guo Jing, Deepu Rajan, and Chng Eng Siong, “Motion Detection With Adaptive Background And Dynamic Thresholds,” in *Proceedings of International Conference on Information, Communications and Signal Processing*, pp. 41 – 45, 2005.

- [10] Yongquan Xia, Shaohui Ning, and Han Shen, "Moving Targets Detection Algorithm Based on Background Subtraction and Frames Subtraction," in *Proceedings of International Conference on Industrial Mechatronics and Automation*, Volume 1, pp. 122 – 125, 2010.
- [11] Ming-Hsuan Yang and Narendra Ahuja, "Gaussian Mixture Model for Human Skin Color and Its Application in Image and Video Databases," in *Proceedings of Conference on International Society for Optical Engineering*, Volume 3656, pp. 458 – 466, 1998.
- [12] Y. Huang, K.B. Englehart, B. Hudgins, and A.D.C. Chan, "Optimized Gaussian Mixture Models for Upper Limb Motion Classification," in *Proceedings of IEEE International Conference on Engineering in Medicine and Biology Society*, Volume 1, pp. 72 – 75, 2004.
- [13] Saeid Fazli, Hamed Moradi Pour, and Hamed Bouzari, "Multiple Object Tracking Using Improved GMM Based Motion Segmentation," in *Proceedings of International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*, Volume 2, pp. 1130 – 1133, 2009.
- [14] Ralph Gross, Jie Yang, and Alex Waibel, "Growing Gaussian Mixture Models for Pose Invariant Face Recognition," in *Proceedings of International Conference on Pattern Recognition*, Volume 1, pp. 1088 – 1091, 2000.
- [15] M. Rahman, Jianfeng Ren, and N. Kehtarnavaz, "Real-Time Implementation of Robust Face Detection on Mobile Platforms," in *Proceedings of IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 1353 – 1356, 2009.
- [16] Dereje Teferi, Maycel I. Faraj, and Josef Bigun, "Text Driven Face-Video Synthesis Using GMM and Spatial Correlation," in *Proceedings of Scandinavian Conference on Image Analysis*, pp. 572 – 580, 2007.
- [17] Hebert Luchetti Ribeiro and Adilson Gonzaga, "Hand Image Segmentation in Video Sequence by GMM: A Comparative Analysis," in *Proceedings of Brazilian Symposium on Computer Graphics and Image Processing*, pp. 357 – 364, 2006.
- [18] Yan Chen, Qingyang Hong, XiaoYang Chen, and Caihong Zhang, "Real-time Speaker Verification Based on GMM-UBM for PDA," in *Proceedings of IEEE International Symposium on Embedded Computing*, pp. 243 – 246, 2008.
- [19] Horng-Horng Lin, Jen-Hui Chuang, and Tyng-Luh Liu, "Regularized Background Adaptation: A Novel Learning Rate Control Scheme for Gaussian Mixture Modeling," *IEEE Transactions on Image Processing*, Volume 20, pp. 822 – 836, 2011.

- [20] Luigi Di Stefano and Andrea Bulgarelli, “A Simple and Efficient Connected Components Labeling Algorithm,” in *Proceedings of International Conference on Image Analysis and Processing*, pp. 322 – 327, 1999.

