Background subtraction, part 1: MATLAB models

Seth Benton - August 10, 2008

Part 2 uses Agility MCS to translate our MATLAB models to C tool. Also see MATLAB to C using MCS: Advanced topics, where we highlight advanced topics using the mixture-of-Gaussians background subtraction method.

Introduction
Many DSP designs start out in MATLAB and are then translated into C. Unfortunately, MATLAB has a number of quirks that make this translation a headache. MATLAB extends arrays without programmer intervention. It organizes matrices in column major order, while C uses row major order. It uses vector math which must be translated into loops. It indexes the first element of an array with '1' instead of '0'. And so on. All this leads to an error-prone, time-consuming translation process that typically doesn't begin until the MATLAB is 100% verified— lest the C programmers waste valuable time hunting MATLAB bugs.

So when Catalytic (now Agility), released their MATLAB to C tool (MCS) in early 2006, I was intrigued. As were the readers of the DSP DesignLine, where articles on automatic MATLAB to C conversion are some of the site's most popular.

Wanting to see what all the fuss was about, I contacted Agility and obtained an evaluation copy to try it out for myself.

Because MCS targets image/video processing, I chose to implement three types of background subtraction—a common first step in many video processing applications. The algorithms were implemented from scratch in MATLAB, then converted to C using MCS. The C code was then verified in MATLAB using mex files. (All m-files, generated C files, and the test video clip used—can be downloaded here).

In Part 1 of this 2-part series, I'll give a brief overview of background subtraction and go into detail on the three methods I chose to implement: frame differencing, approximate median, and mixture of Gaussians. Part 2 illustrates the MATLAB to C conversion process and offers impressions of the tool.

Background Subtraction
As the name suggests, background subtraction is the process of separating out foreground objects from the background in a sequence of video frames. Background subtraction is used in many emerging video applications, such as video surveillance (one of today's hottest applications), traffic monitoring, and gesture recognition for human-machine interfaces, to name a few.

Many methods exist for background subtraction, each with different strengths and weaknesses in terms of performance and computational requirements. Most were developed in university labs over the last couple decades (I'm guessing 99% of them in MATLAB). Unfortunately, as is often the case
in academia, many of these methods are currently impractical for commercial application. Eigen-
analysis on 10 seconds of video? May earn you a PhD, but it’ll probably never leave the lab. For this
evaluation, my goal was to implement three methods that were

1. Computationally efficient enough to make the leap from MATLAB to commercial application, and
2. A good representation of background subtraction implementations in today's video applications.

Since background subtraction is being implemented on a wide range of hardware—and thus within a
wide range of computational budgets—I chose to implement methods of varying complexity;

1. Low-complexity, using the frame difference method,
2. Medium complexity, using the approximate median method, and
3. High-complexity, using the Mixture of Gaussians method.

The rest of this article focuses on detailing these three methods. For an overview of other
background subtraction methods, this Power Point presentation gives a good high-level overview.
For a more in-depth review and comparison of techniques, this paper is fairly comprehensive and
was a good start for my MATLAB implementations.

Test video

Figure 1. Test video used for background subtraction

I shot the test video at an intersection near my house in San Francisco California. My aim was not to
obtain the most representative test video for a particular use case. Rather, I simply wanted to get
the most challenging video I could and see how the different methods handled it. As seen in Figure
1, the video contains many challenging elements, including traffic moving at fast and slow speeds,
pedestrians, bicyclists, changes in light levels, and a waving rainbow flag. The video was shot at VGA
resolution (640x480), 30 frames per second. Because background subtraction algorithms typically
process lower resolution grayscale video, I converted the video to grayscale and scaled it to QVGA
(320x240).

Frame Difference

Frame Difference

Frame difference is arguably the simplest form of background subtraction. The current frame is
simply subtracted from the previous frame, and if the difference in pixel values for a given pixel is
greater than a threshold $T_s$, the pixel is considered part of the foreground.

$$|frame_i - frame_{i-1}| > T_s$$

Figure 2 shows the frame difference method applied to the test video. Non-black pixels are
foreground pixels.

Figure 2. Frame difference output

As can be seen, a major (perhaps fatal) flaw of this method is that for objects with uniformly
distributed intensity values (such as the side of a car), the interior pixels are interpreted as part of
the background. Another problem is that objects must be continuously moving. If an object stays still
for more than a frame period (1/fps), it becomes part of the background.

This method does have two major advantages. One obvious advantage is the modest computational
load. Another is that the background model is highly adaptive. Since the background is based solely on the previous frame, it can adapt to changes in the background faster than any other method (at 1/fps to be precise). As we'll see later on, the frame difference method subtracts out extraneous background noise (such as waving trees), much better than the more complex approximate median and mixture of Gaussians methods.

A challenge with this method is determining the threshold value. (This is also a problem for the other methods.) The threshold is typically found empirically, which can be tricky. This is where MATLAB's powerful visualization tools come in handy.

Instead of running the m-file for a series of values and examining the output (a possibly time-consuming exercise), I decided to whip up a simple MATLAB gui (m-file). As shown in Figure 3, the GUI allows you to change the threshold value and see the resulting output. Using the GUI, I was able to quickly zero in on the threshold that gave the best looking output.

![MATLAB GUI for determining threshold value](image)

*Figure 3. MATLAB GUI for determining threshold value.*

For this example, a GUI probably didn't save too much time. But when more variables are involved (as is the case with other methods), doing quick and dirty optimization with a GUI (at least for a first order approximation) can save considerable time. In addition, a GUI can be a good way to “visually debug”. In my personal experience, looking at what's happening to the pixels can often tell you where your code is malfunctioning quicker than looking through the code.

If you'd like to experiment further with the frame difference method, the m-file can be found [here](#).

**Approximate median**

Approximate median

In median filtering, the previous $N$ frames of video are buffered, and the background is calculated as the median of buffered frames. Then (as with frame difference), the background is subtracted from the current frame and thresholded to determine the foreground pixels.

Median filtering has been shown to be very robust and to have performance comparable to higher complexity methods. However, storing and processing many frames of video (as is often required to track slower moving objects) requires an often prohibitively large amount of memory. This can be
alleviated somewhat by storing and processing frames at a rate lower than the frame rate—thereby lowering storage and computation requirements at the expense of a slower adapting background.

A more efficient compromise was devised back in 1995 by UK researchers N.J.B. McFarlane and C.P. Schofield. While doing government funded research on piglet tracking in large commercial farms, they came up with an efficient recursive approximation of the median filter. Their 'approximate median' method, presented in their seminal paper, 'Segmentation and tracking of piglets in images', has since seen wide implementation in the background subtraction literature, and been applied to a wide range of background subtraction scenarios.

The approximate median method works as such: if a pixel in the current frame has a value larger than the corresponding background pixel, the background pixel is incremented by 1. Likewise, if the current pixel is less than the background pixel, the background is decremented by one. In this way, the background eventually converges to an estimate where half the input pixels are greater than the background, and half are less than the background—approximately the median (convergence time will vary based on frame rate and amount movement in the scene.) Figure 4 shows the approximate median method at work on the test video.

Figure 4. Approximate median output

As you can see, the approximate median method does a much better job at separating the entire object from the background. This is because the more slowly adapting background incorporates a longer history of the visual scene, achieving about the same result as if we had buffered and processed N frames.

We do see some trails behind the larger objects (the cars). This is due to updating the background at a relatively high rate (30 fps). In a real application, the frame rate would likely be lower (say, 15 fps). If you'd like to tinker with the update rate and eliminate the trails, the m-file can be downloaded here.

To get a feel for how the background model works, sometimes it's useful to visualize it. Below is a video of the background model. Rather ghostlike if you ask me.

Figure 5. Approximate median background

This method is a very good compromise. It offers performance near what you can achieve with higher-complexity methods (according to my research and the academic literature), and it costs not much more in computation and storage than frame differencing. Mixture of Gaussians

* Mixture of Gaussians

Among the high-complexity methods, two methods dominate the literature; Kalman filtering and Mixture of Gaussians (MoG). Both have their advantages, but Kalman filtering gets slammed in every paper for leaving object trails that can't be eliminated. As this seems like a possible deal breaker for many applications, I went with MoG. Also, MoG is more robust, as it can handle multi-modal distributions. For instance, a leaf waving against a blue sky has two modes—leaf and sky. MoG can filter out both. Kalman filters effectively track a single Gaussian, and are therefore unimodal: they can filter out only leaf or sky, but usually not both.

In MoG, the background isn't a frame of values. Rather, the background model is parametric. Each pixel location is represented by a number (or mixture) of Gaussian functions that sum together to form a probability distribution function $F$. 

The mean \( u \) of each Gaussian function (or component as I'll call them), can be thought of as an educated guess of the pixel value in the next frame—we assume here that pixels are usually background. The weight and standard deviations of each component are measures of our confidence in that guess (higher weight & lower \( \sigma \) = higher confidence). There are typically 3-5 Gaussian components per pixel—the number typically depending on memory limitations.

To determine if a pixel is part of the background, we compare it to the Gaussian components tracking it. If the pixel value is within a scaling factor of a background component’s standard deviation \( \sigma \), it is considered part of the background. Otherwise, it's foreground.

For space constraints, I’ll refrain from giving a more detailed description here. A detailed description of the steps in the MoG algorithm can be found in Appendix A. The full details of my particular implementation (including initial parameter values, etc), can be found in the \texttt{m-file}.

Figure 6 shows the MoG method applied to the test video. Figure 7 shows a visualization of the background model. For this experiment, I used three Gaussian components per pixel.

\textit{Figure 6. Mixture of Gaussians output}

Note that the video of the background model is not a literal visualization. It's simply a weighted sum of all components, whether they're part of the background model or not.

\textit{Figure 7. Mixture of Gaussians background model}

As can be seen, MoG is very good at separating out objects and suppressing background noise such as waving trees. However, there are several points where the method breaks down, allowing most of the background to seep into the foreground. These points correspond to relatively rapid changes in illumination. If we go back to the approximate median output we can see similar hiccups, although less pronounced. This is because the background model isn’t adapting quickly enough.

This is not to say the MoG is less robust, necessarily. But the problem MoG had with illumination changes in the test video does point to one of its main challenges; parameter optimization. The MoG method has five parameters which must be tweaked (the background component weight threshold \( T_s \), the standard deviation scaling factor \( D \), the learning rate \( \rho \), the total number of Gaussian components, and the maximum number of components \( M \) in the background model)—all of which can have a significant impact on the performance of the algorithm (see Appendix A for more details).

Perhaps if I’d chosen better parameter values, say a faster learning rate, the method would've performed better. Alas, I didn't have time to solve a complex multivariate optimization problem.

One notable advantage of MoG is that it is straightforward to extend to color data. Simply perform the algorithm on all three channels (RGB, YCrCb, etc). For some applications color data significantly improves performance. For instance, without color information, it can be difficult to separate objects from their shadows. Since shadows typically have a different color as the objects casting them, shadow suppression is much easier with color data.

\textbf{Conclusion}

As is often the case, the simplest method is arguably the most robust. While it has major flaws, and
is probably not suitable for most applications, frame differencing does the best job of subtracting out extraneous background noise such as waving trees (at least in the test video.) The second most robust method, approximate median, gives us significantly increased accuracy for not much more computation. It had a little trouble with quickly changing light levels, but handled them better than mixture of Gaussians. And Mixture of Gaussians, the most complex of the methods, gives us good performance, but presents a tricky parameter optimization problem. All three methods have their place in the background subtraction application space and are amenable to MATLAB to C conversion.

In Part 2, we'll delve into Agility's MCS, and illustrate how we convert these MATLAB models to C. Also see MATLAB to C using MCS: Advanced topics, where we highlight advanced topics using the mixture-of-Gaussians background subtraction method.

Appendix A

The following steps should give you a good understanding of the basic operation of the Mixture of Gaussians (MoG) method. For the full details required for implementation (equations, etc.), there are several good papers online (such as this one). The full details of my particular implementation (including initial parameter values, etc), can be found in the m-file.

MoG algorithm steps

1. Compare the input pixels to the means \( u_i \) of their associated components. If a pixel value is close enough to a given component's mean, that component is considered a matched component. Specifically, to be a matched component, the absolute difference between the pixel and mean must be less than the component's standard deviation scaled by a factor \( D \).

   \[ | i_{i,t} - \mu_{i,t-1} | \leq D \cdot \sigma \]

2. Update the component variables (\( w, u, \) and \( \sigma \)) to reflect the new pixel value. For matched components, a set of equations increase our confidence in the component (\( w \) increases, \( \sigma \) decreases, and \( u \) is nudged towards the pixel value). For non-matched components, the weights decrease exponentially (\( u \) and \( \sigma \) stay the same). How fast these variables change is dependent on a learning factor \( p \) present in all the equations.

3. Determine which components are part of the background model
   - Order the components according to a confidence metric \( w/\sigma \), which rewards high \( w \)'s and low \( \sigma \). We do this because we want to keep only the \( M \) most confident guesses.
   - Apply a threshold to the component weights \( w \).
   - The background model is then the first \( M \) components (in order of highest to lowest \( w/\sigma \)), whose weight \( w \) is above the threshold. \( M \) is the maximum number of components in the background model, and reflects the number of modes we expect in the background probability distribution function \( f \) (or it may reflect our computational resource limitations).

4. Determine foreground pixels. Foreground pixels are those that don't match any components determined to be in the background model.