18-551, Spring 2003

Group #12, Final Report

Shake that thing! – Video Stabilization

Jeff Bloomfield (jrb2@andrew.cmu.edu)

Ed Kao (kkao@andrew.cmu.edu)

Scott Marmer (smarmer@andrew.cmu.edu)

The Problem

Video quality is often hindered by shaky hands of the cameraman or by recording from a mobile platform (i.e. recording from a car). This project strives to improve video quality by stabilization through image processing. We will input short video clips with a resolution of 320x240 pixels and a frame rate of sixteen frames per second then output improved, stabilized video clips while retaining as much resolution as possible and the same frame rate.

Prior Relevant Work in 551

We looked at past projects in 551, but we did not find any that were relevant to our project.

Method

The overall algorithm we chose to approach our design with was the rectangular component of T. Chen's design. From the most basic perspective, the algorithm removes shakes from a video stream by analyzing the motion vectors of each frame, filtering those vectors, and ensuring that the output contains a video with only the *low-frequency* global motion in it. This is much more difficult than it first appears.

The algorithm applies filtering in advance – that is, it determines the original vectors through frame *i*, then filters them to obtain the smoothed vector, then applies a correction of that direction and magnitude to either frame *i-1* of the input video, or frame *i-1*of the output video (the last output frame). The borders are filled with other frames shifted and applied as background in attempt to best match the output frame, but depending on the exact combination of smoothed/original/final vector orientations, this isn't always possible and black borders are used instead. The most sensitive part of the logic is determining whether to resynchronize, meaning to use the previous input frame instead of the last output frame.

Resynchronization is meant to occur when the sum of the smoothed and original vectors up to the point of the previous frame are close in size; this way when there is an extremely large shake in the video, those frame's don't see the output. This prevents large borders in output frames and recognizes that motion estimation isn't accurate enough to correct the worst of movements with enough accuracy that original vectors (applied in reverse) could generate a clean, smooth output. Thus, if a stream is resynchronized too often, the output will be very jerky since large original motion vectors in the previous actual frame will propagate to the output (Adding the smoothed vector to the previous actual frame assumes that the previous frame had a similar sum-of-smoothed-vectors to sum-of-original-vectors). If it's resynchronized too rarely, resolution in the time-domain in lost because information from only a fraction of the input frames are used in the output, and the rest are simply old frames being shifted around in space. It appears that the solution would be to counter the smoothed motion vector during resynchronization with the difference between the sum-of-smoothed and sum-of-original vectors, but we will explain why our attempt at this improvement failed later.

When resynchronization occurs, the original algorithm would note the error between sum-of-original and sum-of-smoothed vectors and add it into "final" motion vectors, which would be what are later compared to the sum-of-original vectors to determine whether to resynchronize at future times. Since resynchronization can also occur when a limit to the number of non-resynchronized frames is reached, the theory is that this will cause future frames to be resynchronized when they're close to the *new actual* output, which is now slightly different than just the sum of the smoothed vector. To resynchronize the following condition must be met:

```
sum(abs(sum(originalMVs(1:i-1,:))-sum(finalMVs(1:i-1,:))).^2) < Threshold^2 ||
Count>Max_Count
```

Theoretical Problems

The above approach seemed logistically sound and passed preliminary tests we fed it in Matlab. When we ported the code to C, the same tests ran successfully as well. The algorithm worked perfectly on our simple movie with limited internal motion, and with movies we manually inserted rectangular motion into. However with most other real-world videos we fed into it completely broke down.

The problem was in the most fundamental and necessary part of the algorithm, resynchronization. All of the video being fed into the algorithm would reach the resynchronization frame limit before it would reach the threshold of vector similarity. In fact the difference between final and original vectors would grow inordinately so large that resynchronization would never take place except when forced to by the frame limit.

Upon resynchronization the following occurs:

```
finalMVs(i,:) = smoothedMVs(i,:) + sum(originalMVs(1:i-1,:)) - sum(smoothedMVs(1:i-1,:));
```

The sum of the final MVs is what is of necessary interest in determining resynchronization, and the affect this has on that is the following:

```
Sum_FinalsMvs + =      Frames since last resynchronization
                       * [ (Sum of original vectors up to point of last resynchronization)
                       -(Sum of smoothed vectors up to point of last resynchronization) ]
                       +(Sum of smoothed vectors since last resynchronization)
```

This is the equivalent of saying that to resynchronize at a future point, the total original vectors at that point must match closely with a video in which each frame since the last resynchronization exhibited the *same imperfection that we were forced to accept during that resynchronization.* Since lack of similarity to this final vector will still leave the output frames with the same unwanted translation (from not desynchronizing), this is a valid point.

However, in real-world video it prevents the resynchronization threshold from ever being reached again. If the sum of original and sum of smoothed vectors are very different, as is often the case when resynchronization is forced, this difference is multiplied larger the longer it goes before resynchronization is possible again, which is all the more likely to be *a long time* after a poor resynchronization choice. **In short, the more the system is in need of error control, the worse it handles the error.** Indeed, in Chen's own report, his tests resynchronized every 5-15 frames, and he only used video of still objects.

Theoretical Improvements

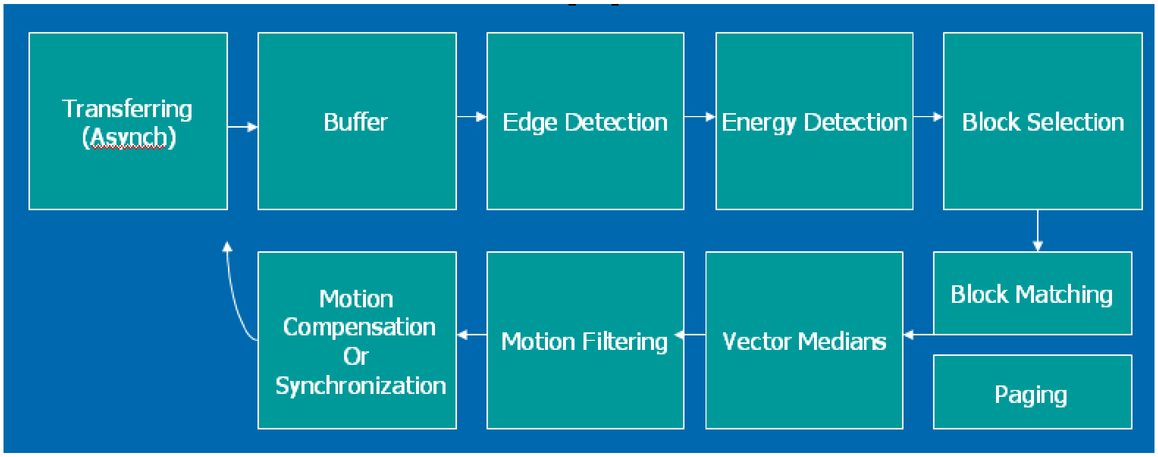
To counter this we tried a new algorithm entirely: applying the reverse of the original minus smoothed vectors in reverse to the same frame as was being inputted.

These results were equally interesting, and unacceptable. Correction to the output video provided an apparent proof of concept as large motions could be filtered out in a single frame and without losing time-domain resolution. However the visual quality was extremely jerky due to the fact that the edges changed rapidly, and vector determination isn't perfect - a smooth but motion-ridden input video produced an output with a smaller but more ugly and pronounced 16hz vibration as it was shifted around. The conclusion we reached was that limitations on the accuracy of vector determination and the adherence of real video to purely rectangular shifts made this a poor approach, and if motion vectors were to be used they had to be averaged together over several frames, as in the first algorithm.

Acceptable results were obtained by altering the resynchronization component of Chen's algorithm. We changed the logic that whenever resynchronization. occurs, the sum-of-final vectors is set to equal the sum-of-original vectors plus the current smoothed vector (and thus, errors don't accumulate past a resynchronization). This essentially accepts the fact that there will be error in the output video, and that such an error, in a worst case scenario, will be present for as long as is the limit to the number of frames between resynchronizations. In the worst case, it becomes pronounced because after the unwanted motion is forced to the output, it continues to be shifted in the undesired direction because its own motion is averaged into the filter and applied to itself when not resynchronizing. However, unlike in Chen's algorithm, use of this frame does not preclude future resynchronizing, and it does not break the system. The system outputs a bad frame but then, after one to several frames later, completely recovers. Also the likelihood of this situation occurring in the first place is much lower because the resynchronization logic is always meaningful.

Another modification allows the threshold to depend on how long it has been since the last resynchronization. This improved results by preventing the algorithm from crashing but allowing smaller, optimal value to be used in normal circumstances.

Implementation: Components



The algorithm infers three main pieces:

I) **Motion Determination**

II) **Motion Filtering**

III) **Motion Compensation**

The Matlab code we had was not easily portable to C, and ran unrealistically slow (several minutes per frame). To solve this we implemented a custom solution to these parts.

Motion Determination

We accepted Chen's approach of dividing the frames into blocks, and finding the median of the vector in each of those blocks between frames. This provides "global" motion and is meant to work even while there are objects in motion inside the video. However we don't do a search on every block, we use a limited search range, and we downsample the resolution. Block selection is done using edge and energy detection, and block matching by a custom algorithm.

Edge & Energy Detection

To select the blocks containing the most information for effective motion determination, we employed edge & energy detection. This way, the blocks being searched are the blocks which are most interesting and representational of the entire picture.

Edge & energy detection is accomplished by running a Sobel filter then obtain the sum of all pixels on each 16 by 16 block. The Sobel filter has two components to detect and vertical and horizontal contrasts in the picture. Essentially, these components can be represented by two 3 by 3 matrices:

$$h_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad h_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

We can perform convolution of the picture on both matrices by simply applying the following formula on each pixel:

xvector =   top right pixel+ top left pixel + 2 * right pixel – 2 * left pixel + bottom right pixel – bottom left pixel

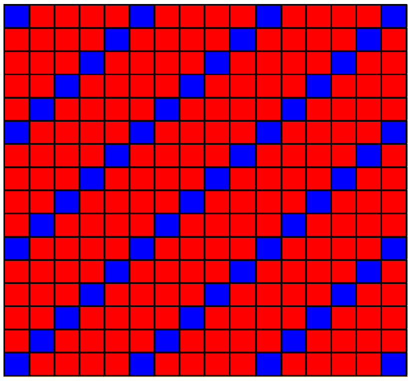
yvector =   - top right pixel - top left pixel - 2 * top pixel + 2 * bottom pixel + bottom right pixel + bottom left pixel

new pixel value = square_root(abs(vectorx/4)^2 + abs(vectory/4)^2)

We simply calculate each x and y vector then take the magnitude of the resulting normalized vector. A sample picture will be shown in the result section.

Block Matching

The block matching portion of our algorithm compares a 16 by 16 pixel block of the current frame to all of the 16 by 16 pixel blocks of the previous frame within a 16 pixel search radius of the corresponding block of the previous frame (making the search area in the previous frame 48 by 48 pixels). The goal of the search is to find the most similar nearby block in the previous frame, which should allow one to determine how much the image has shifted. To determine the similarity between two blocks, we subtract corresponding pixels and then sum the square of all of the differences. In order to decrease the number of computations and increase the speed of the algorithm, we only compare every fifth pixel of the blocks. We found that comparing every fifth pixel greatly increased the speed without significantly affecting the results.



An illustration of one of the 16x16 blocks.
The blue squares are the pixels that will be compared.

Motion Filtering

This is the simplest piece of the approach, and involves a simple, moving average filter. The length is parameterized but the best value we've found is a kernel length of 9. This worked well enough and was easy to implement using single array. Extensive testing and customization of the algorithm might do well to substitute more advanced or adaptable filters, but for our purposes this was not the bottleneck of the system in terms of result quality.

Motion Compensation

This portion of the code involves superimposing a foreground a background image of different rectangular shifts on top of each other. The Matlab was straight-forward:

```
if uf >= 0 & vf >= 0,
outFrame(1:nRows-vf,1:nCols-uf) = inFrame2(1+vf:nRows,1+uf:nCols);
elseif uf >=0 & vf < 0,
outFrame(1-vf:nRows,1:nCols-uf) = inFrame2(1:nRows+vf,1+uf:nCols);
elseif uf < 0 & vf >= 0,
outFrame(1:nRows-vf,1-uf:nCols) = inFrame2(1+vf:nRows,1:nCols+uf);
else
outFrame(1-vf:nRows,1-uf:nCols) = inFrame2(1:nRows+vf,1:nCols+uf);
end
if u >= 0 & v >= 0,
outFrame(1:nRows-v,1:nCols-u) = inFrame1(1+v:nRows,1+u:nCols);
elseif u >=0 & v < 0,
outFrame(1-v:nRows,1:nCols-u) = inFrame1(1:nRows+v,1+u:nCols);
elseif u < 0 & v >= 0,
outFrame(1:nRows-v,1-u:nCols) = inFrame1(1+v:nRows,1:nCols+u);
else
outFrame(1-v:nRows,1-u:nCols) = inFrame1(1:nRows+v,1:nCols+u);
end
```

We decided to improve this though by using pointer arithmetic instead of actually accessing the data. This would allow us to avoid having to perform reads to slow SDRAM memory, and we could generate pointers that could be fed into Asynchronous transfer registers to control a background transfer back to the computer. This wasn’t a bottleneck of our system and we would have been fine with a simpler implementation, but our approach would be useful if scaling the system to higher resolution, color-depth, and framerate, in which the simple approach would be too slow.

The initial logic in our code determines for each image and for each row what the offset is, and what the address is of memory where the first read in that row would be. A loop of other logic runs once per row and uses that data to classify the row into one of a number of conditions based on the overlap, and then stores up to three “start” and “stop” pointers in the array, which at completion of the function represents the new image in address-form.

Training and Test Sets

Due to the nature of our project, we did not need to use any training sets, but we did record and utilize several input videos to test our project. We recorded many videos and selected the ones that had the most useful properties to test. The names of the videos that we tested are: car, cat, park, truck, and walk. All of these videos were between five and fifteen seconds because we wanted to use videos that were long enough to demonstrate whether or not stabilization was working, but short enough that they did not take up too much disk space or too much time to process. Also, each video is an example of a different type of situation. In *Car*, there is no motion of the camera other than the shaking nor is there any motion in the scene being recorded. In *Truck*, there is no motion of the camera besides the shaking, but several automobiles quickly drive across the scene. *Park* has a small amount of motion within the scene and the camera pans across the scene. The camera is not panned in *Cat,* but the cat in the recording moves around in the video. In *Walk*, the cameraman walks down a sidewalk while videotaping.

Formats and Conversions

The camera we used outputted video in MPEG format. The video was recorded at 16 frames per second and had 320x240 resolution. We used Blaze Media Pro 2002 [http://www.blazemp.com], a shareware program, to convert the MPEG movies to 320x240 bitmap frames. We then used Photoshop 6.0 [http://www.adobe.com/products/photoshop/main.html] to convert the bitmap images into RAW images. After running our program to stabilize the video, we took the outputted RAW frames and converted them into bitmaps. Then we took the bitmap frames and converted them into an AVI movie that displayed 16 frames per second using Blaze Media Pro 2002.

Demo

The demo that we had planned was to take a few of our test videos, run each of them through our program on the EVM, and show the resulting movie along with the associated input movie for comparison. The movies that we were going to do this with were cat, park, and truck. However, in order to save time and since Prof. Casasent said that he did not need to see them run through the EVM, we simply showed the input and output videos of *Cat*, *Park*, and *Truck* for comparison. The results of the demo will be discussed in the results analysis section.

Results

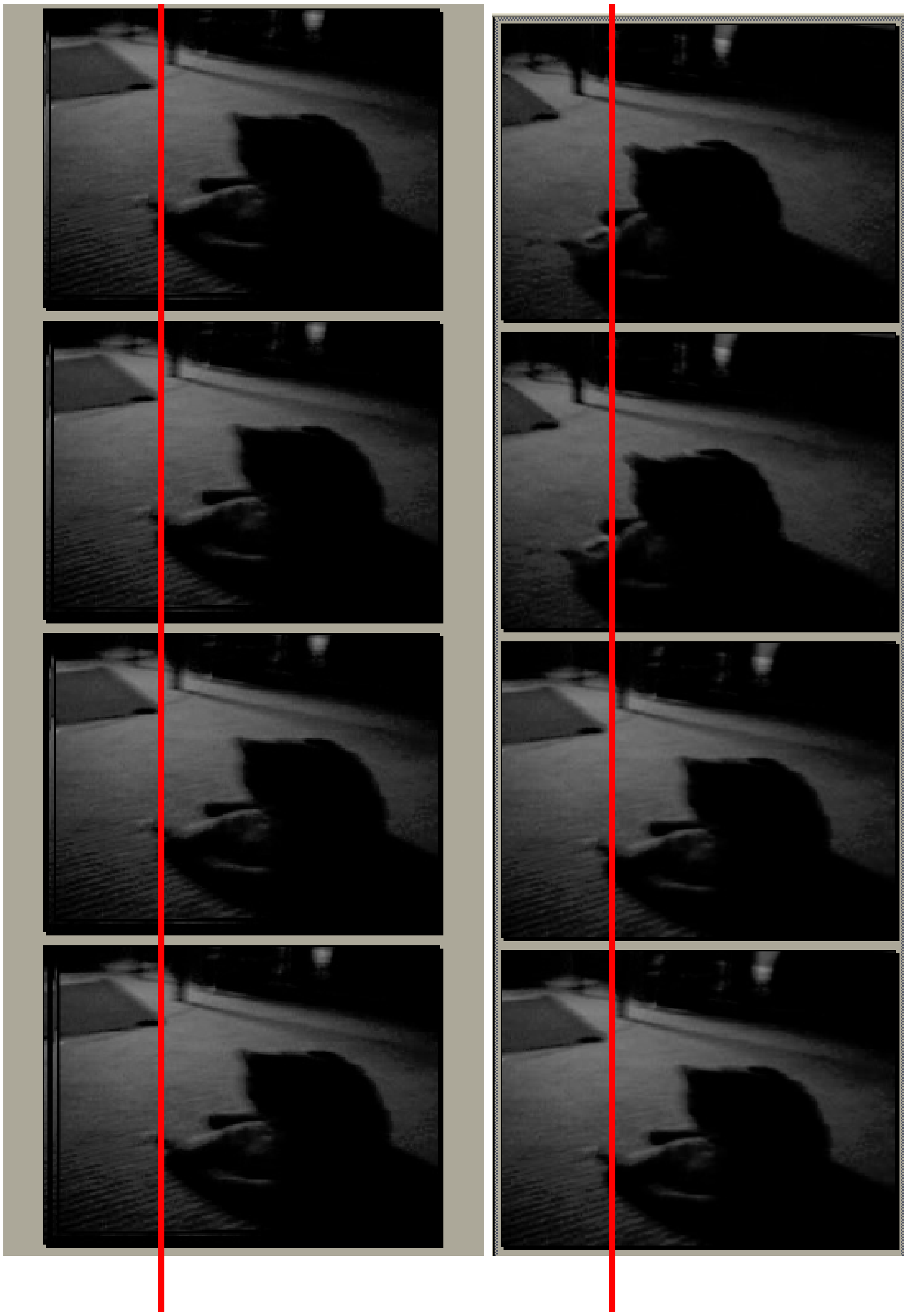
Results were obtained that indicate success to a point. Videos were stabilized, and the algorithm successfully runs on the EVM. The inherent algorithm we're using is not perfect, though, and the output videos show the weaknesses in it

<u>Ideal Video</u>
Smooth, oscillating, high-frequency movement
Purely Rectangular distortion
Little movement of objects.

<u>Non-Ideal Video</u>
Rotational Movement
Zooming / Forward Movement
Internal Movement

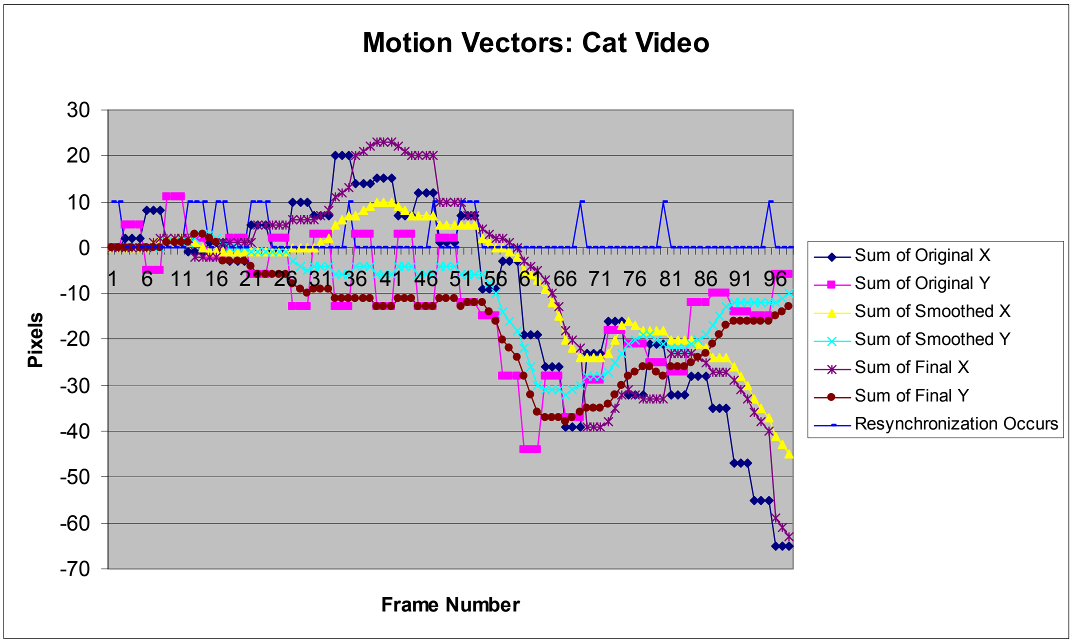
Cat Video

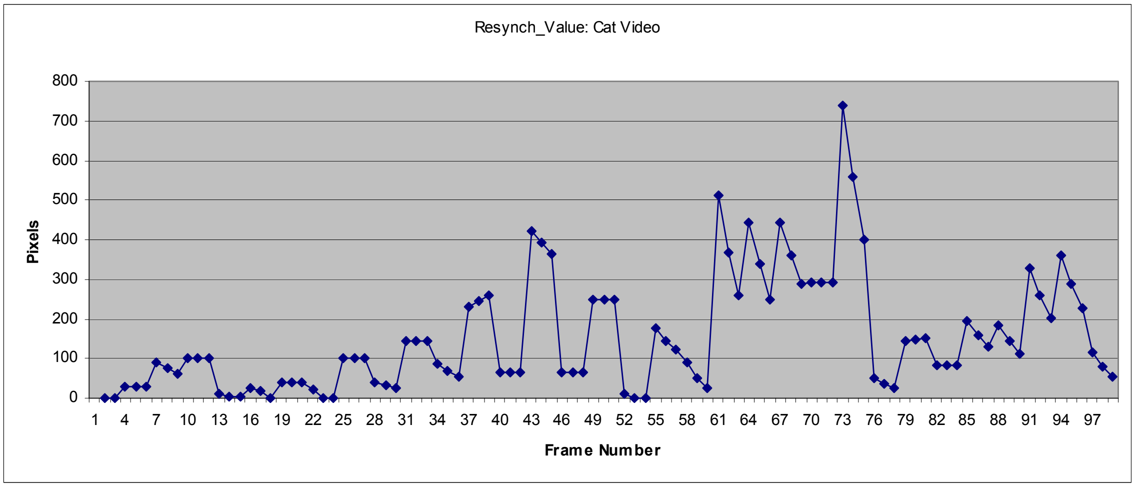
Example Strip



Stabilized **Original**

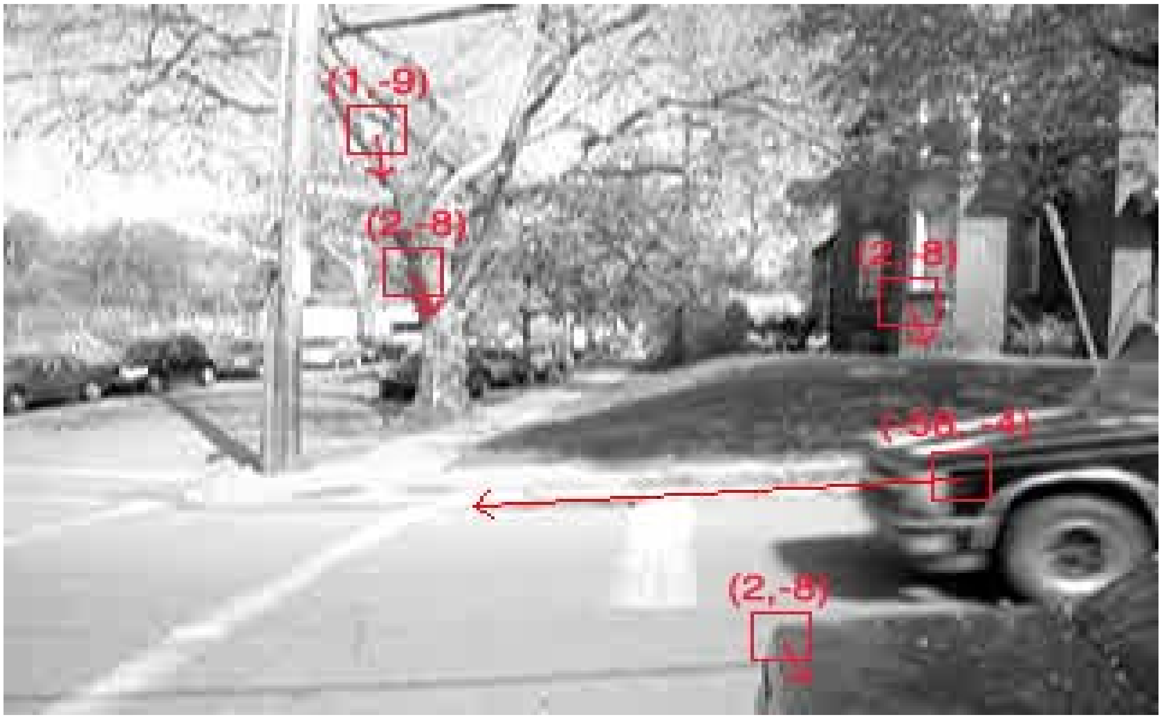
Above shows an example set of frames from this video. Much of the low-frequency oscillation in the movement is removed. The effects on the border show the new frame being superimposed in attempt to match the image up (with somewhat limited success.) This data also shows how the less regular the movement becomes, the more infrequently it resynchronizes. The bottom graph below shows, though, how our algorithm recovers and is able to be compared to a fixed threshold to determine when this should happen.





Truck Video

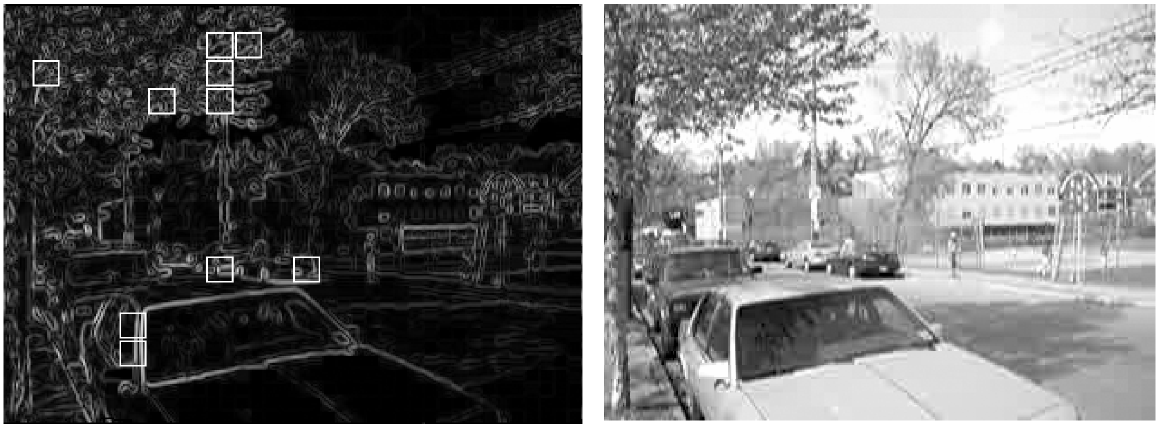
This video proved that it is possible to have internal motion and still have successful stabilization. Finding the median of the blocks ignores local motion. However, with only using 10 blocks its possible that if most of the energy in the picture is covered by the image, and the image covers most of the frame, that distortion would be present. When this is the case this motion is filtered out, and the frame doesn't propagate through resynchronization logic to the output.



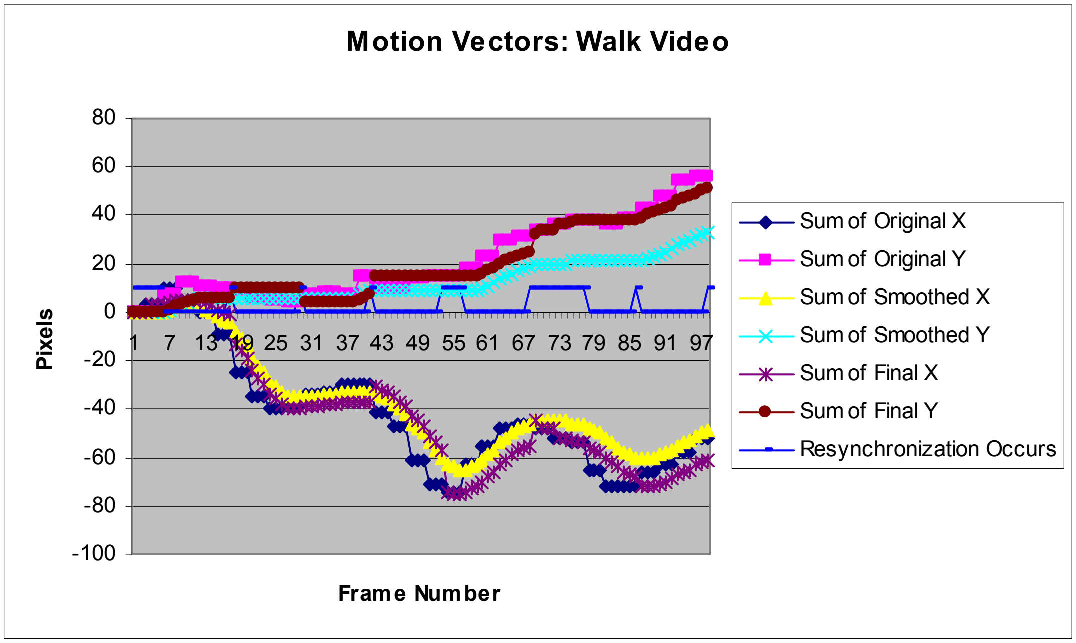
Here are some vectors as results of block matching. By taking the median of the x and y vector, we determined the global motion to be (2,-8). This way, we correctly eliminated the drastic internal movement of the truck. See next picture.

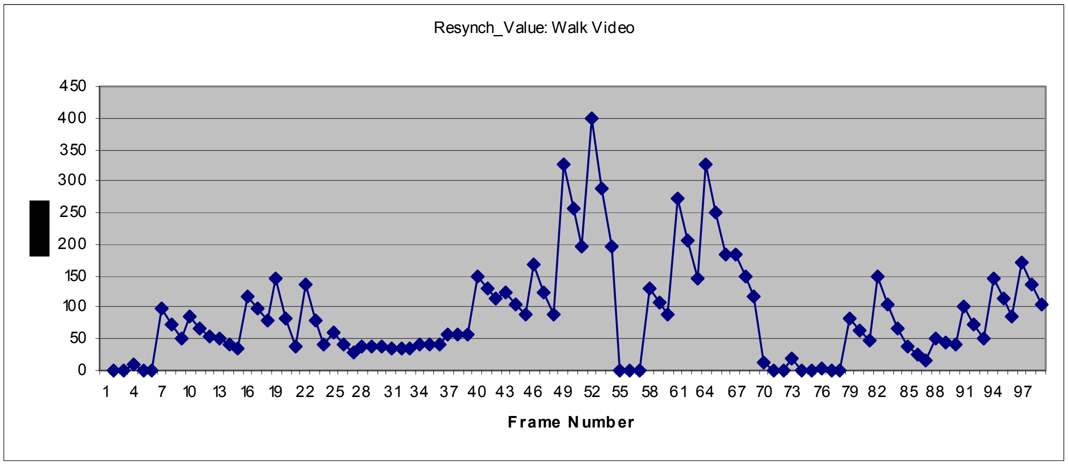
Walk Video

This video was perhaps the poorest performer. The movement patterns were very strange. Resynchronization was made all the more visible by the fact that there was forward motion. Every time it resynchronized there was a jerk, and every time it didn't resynchronize it created the effect of the motion stopping. This is clearly a limitation of the algorithm we used, and a 3D based design would have been better suited here.



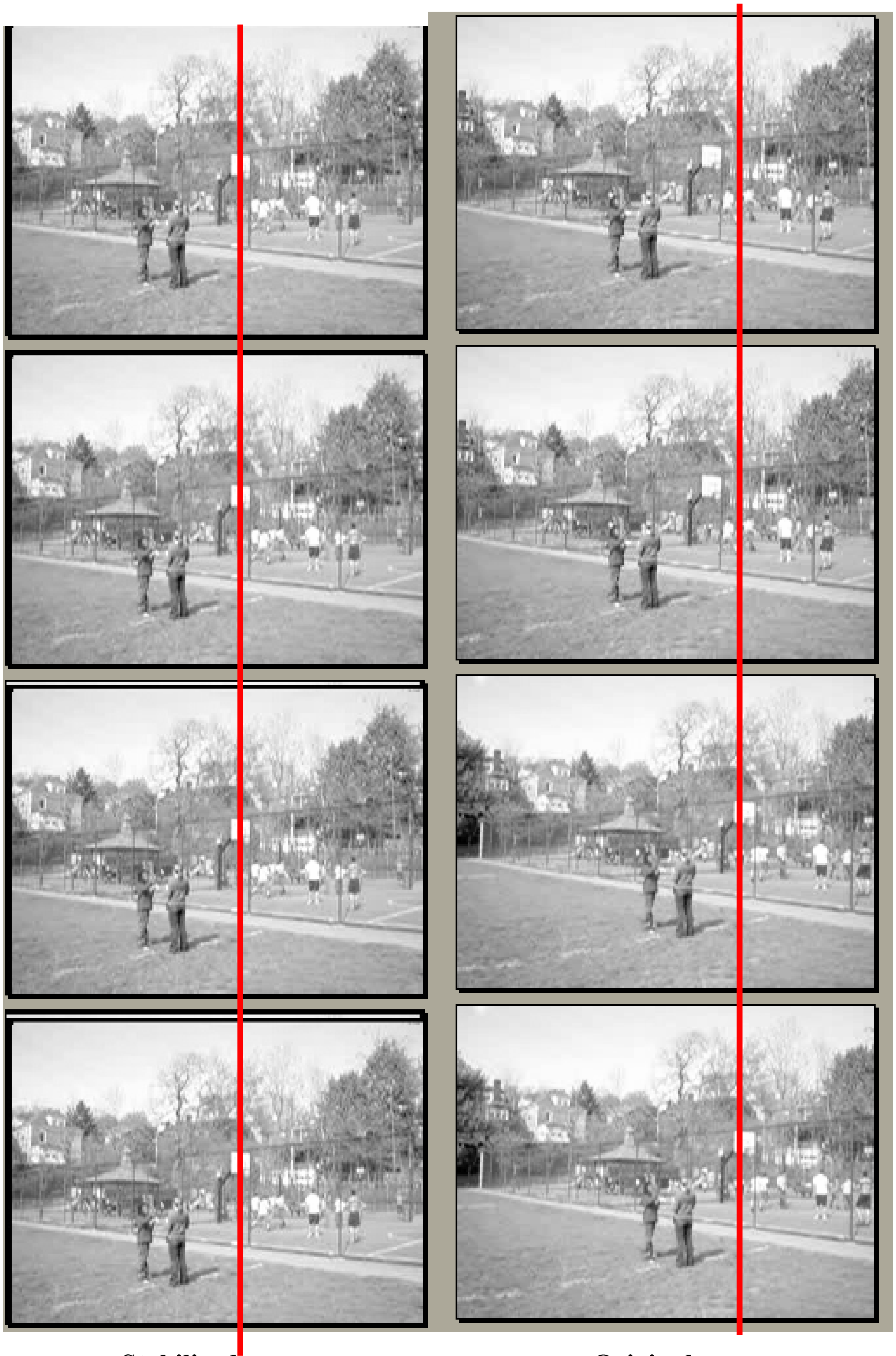
An example of the result of edge & energy detection: The most characteristic parts of the frame are used for motion determination.


Motion Vectors: Walk Video
80
60
40
20
0
-20
-40
-60
-80
-100
Pixels
1 7 13 19 25 31 37 43 49 55 61 67 73 79 85 91 97
Frame Number
Sum of Original X
Sum of Original Y
Sum of Smoothed X
Sum of Smoothed Y
Sum of Final X
Sum of Final Y
Resynchronization Occurs


Resynch_Value: Walk Video
450
400
350
300
250
200
150
100
50
0
[redacted]
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97
Frame Number

Park Video

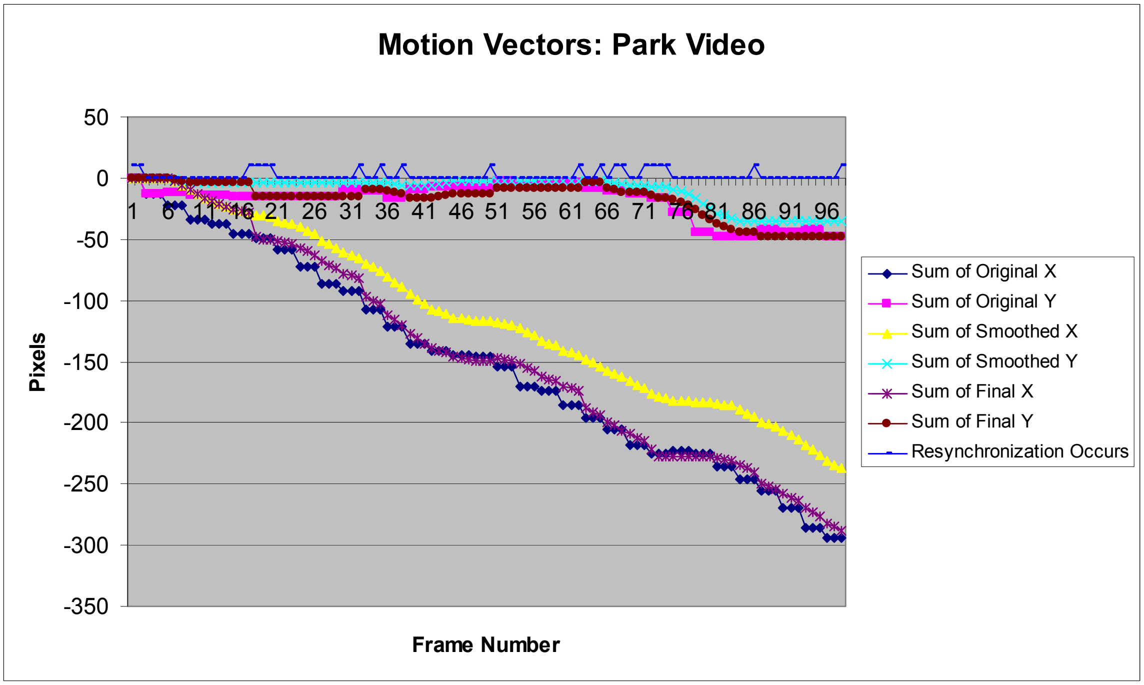
Example Strip

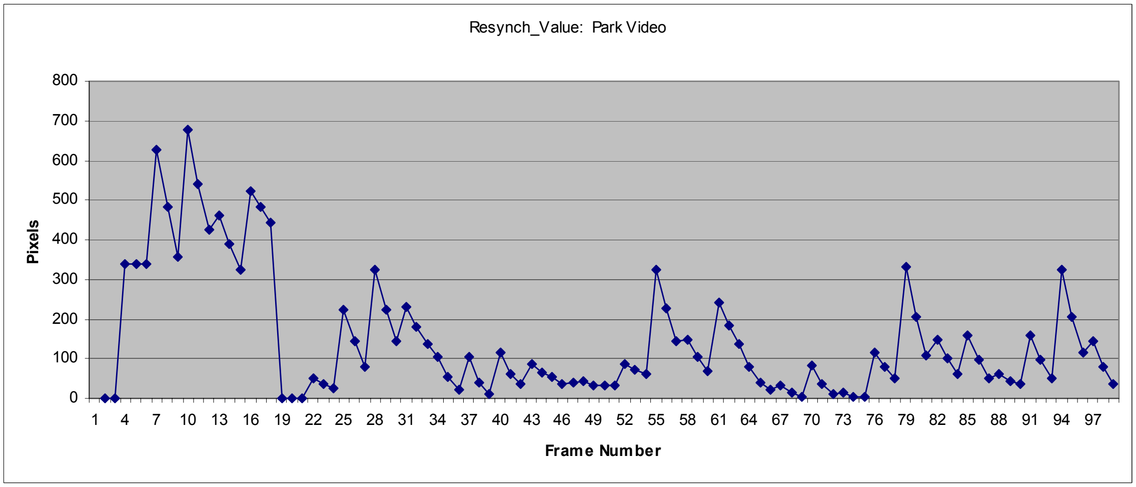


Stabilized **Original**

This video showed how a large DC movement will still propagate to the output. However, because the video was already stabilized, the algorithm introduced more distortion than it fixed, mostly because of the delay present in this movement, due to the moving average filter we used (see graph below).

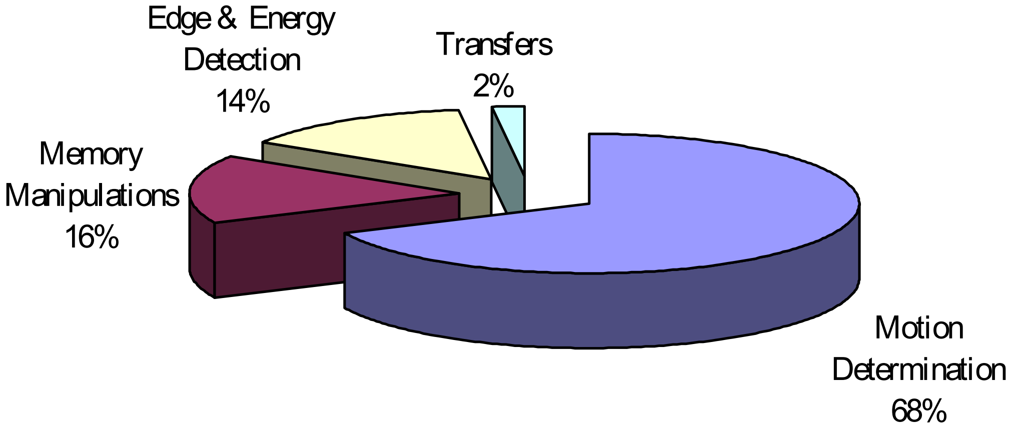
This data seems to suggest that a more advanced feature of the algorithm should be to turn itself off when it isn't needed, or to automatically scale resynchronization threshold to effectively accomplish the same thing.





EVM Speed

Although we improved the speed by a factor of 50, it turns out that our timing estimation still was off by a factor of 16. While real time performance processes 16 frames a second, the program is running at a speed of slightly less than one second per frame. Here is the breakdown of how the processing cycles are allocated for each stage of calculation:



In this section, we will first talk about what caused our estimation to be off then what we did to improve the performance by a factor of 50.

Why slower than real-time:

- Misestimated the amount of calculation needed for block search: Originally, we estimated only 288,000 cycles needed to match 10 blocks for each frame. However, the actual implementation after paging and unrolling still takes about 30,000,000 cycles per frame. Such big difference (about 100 times) is caused by paging and more logic in the block matching function than we expected.

- Edge detection taking much longer than expected: Edge enhancement from lab 3 took less than 1/100 of a second so we didn't expect this part to be a problem. However, the difference is that when lab3 had blocks paged from continuous memory space, our motion blocks are composed of disjoint part of the memory. In other words, we had to piece together the 16 by 16 block from 16 different segments in the original frame. This turned out to slow the edge detection down by a factor of 16.
- Unaccounted memory manipulation: We missed this part completely in estimation. This turned out to be necessary to keep the data flow going. As new frames come in and resulting frames being sent back. We need to move the frames around on the SDRAM. This turned out to be quite costly as well.

How we got 50 times faster

We still improved he speed from the original algorithm in Matlab by 50 times. This is done by:

- Fast algorithm: We modified the algorithm to pick the top 10 most interesting blocks for motion estimation instead of trying to match the entire 300 blocks. We also introduced down-sampling when matching blocks.
- Pointer Manipulations: We chose to manipulate our memory in the most efficient way to minimize movement of data on the SDRAM. Instead of moving these data around, we assigned pointers to each fragment and stored

these pointers on chip. This is done painstakingly but proved to be quite an improvement. However, there are still a lot of memory manipulation that can not be done by pointers.

- Paging: We paged relevant data to temporary work space on chip before calculating these data. The efficiency of this method is limited because of the disjoint memory segments.
- Unrolling: We unrolled some of the most calculation intensive part of the code to enhance pipeline effect.
- Asynchronous Transfers: Since processing time takes the majority of the processing life cycle, we tried to transfer as much data as possible in the background. This cuts the total transfer time (back and forth) to 1/50 of a second.

EVM Memory

Memory allocation effects the speed of this project tremendously. However, most of the data are so big that they can only fit on the SDRAM. Here are a list of how the memory are allocated:

Onchip Cache: Stack and global variables. (local variables such as counters, pointers, smaller arrays.)

SBSRAM: Program body.

SDRAM: Variables declared with far. (all the frames including new frame from PC and resulting frame sent back to PC as well as a couple other intermediate results.)

EVM <-> PC

Here is a graph to demonstrate the interaction between PC and EVM:

Time

EVM	PC
1. Receiving two frames	1. Sending two frames
2. Process frame while receiving one more frame in the background	2. Sending next frame
- Edge & energy detection - Calculating motion vectors - Determining global motion - Smoothening global motions - Correcting global motions	Idle . . . .
3. Send back resulting frame	3. Receiving frame
(repeat step 2 and 3)	(repeat step 2 and 3)

Conclusion

Implementing video stabilization on EVM with real-time performance turns out to be much more challenging than we expected. We were overly optimistic in both the algorithmic as well as the run time aspects. While we feel much of the misleading was done by Ting's paper as he claimed to have achieved good results as well as real-time performance, much has been done to make his algorithm more robust to the noise present in real life data. Also, much has been done in making the algorithm more efficient for a faster runtime.

We feel this has been a good and fulfilling learning experience although the goals we set out in the beginning is not met at the end. We believe with more design changes to the algorithm and more effort in optimization will continue to make this project closer to the goal while we are not entirely sure if real-time can be achieved given the existing hardware in the lab.

References

Chen, Ting (2000). Video Stabilization Algorithm Using a Block-Based Parametric Motion Model. *EE392J Project Report*. http://www.stanford.edu/~tingchen/report.pdf

Generation5.org: AISolutions: An Introduction to Edge Detection: The Sobel Edge Detector. http://www.generation5.org/aisolutions/im01.shtml

Sobel Edge Detection Algorithm (C code). http://www.visc.vt.edu/armstrong/ee2984/sobel.html