Motion Compensated SNR and Dynamic Range Enhancement with Motion Blur Prevention using Multicapture

Ali Ercan & Ulrich Barnhoefer

ABSTRACT

Most conventional digital cameras use single capture to get an image of the scene. If long exposure time is used, not only bright regions in the scene saturate, but also motion blur causes further degradation in the image quality. If short exposure time is used, motion blur problem is not that severe, but the SNR of the final image is poor. Recent work has shown the availability of very high frame rate CMOS image sensors are available today [1] and SNR, dynamic range (DR) and motion blur properties still images can be enhanced using these cameras [2]. In this work, we explain how to incorporate motion estimation to further reduce the motion blur of a still image while getting better SNR and DR performance.

I. INTRODUCTION

Today's conventional cameras use a single capture to get an image of the scene. This results in a trade-off, in the sense that the camera has to use a single exposure time for the capture. That means, one has to choose between either a short or long exposure time. A short exposure time means you get relatively motion-blur free image, but your image will end up very noisy. However, a long exposure time means you get a good SNR. But if there is motion in the scene, you will get motion blur.

To illustrate this effect, please refer to figure 1.

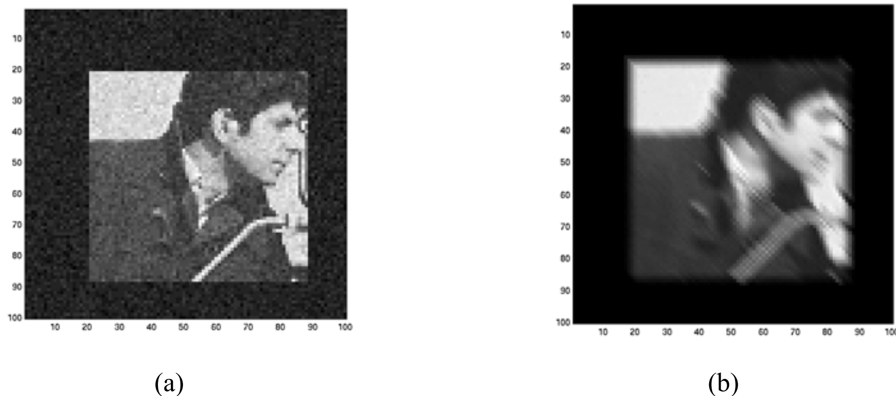


Figure 1. Two images illustrating the artifacts of single capture of a scene with motion. (a) Short exposure time. (b) Long exposure time.

Both images are simulated images from a noisy digital still camera, when the scene moves globally. Both readout and shot noise of a typical CMOS imager is simulated. No fixed pattern noise or reset noise is added; because these noise components are successfully cancelled with a technique called correlated double sampling (CDS). The image on the left is a single capture with a short exposure time. As you see, it almost has no motion blur, however, it is disturbingly noisy. The image on the right has been taken

by exposing the camera for a much longer time, which gives a better SNR, but leads to blur due to motion. Ideally what we want is an image with both high SNR and low motion blur.

What is equally important also is, dynamic range (DR) of the camera, even when there is no motion in the scene. When you have a short exposure time, which means you can see the bright parts of the scene well in the final image, but the dark parts are underexposed and cannot be seen in detail due to noise. If you have long exposure times, you can capture the dark parts nice, but then the bright parts of the scene will saturate the pixels that they shine on. Ideally, one would want to capture both dark and bright parts well. When there is both motion in the scene and the scene is high DR, things get even worse, in terms of final image quality, when a single capture is used. You either get a noisy image with less motion blur but non-visible dark areas, or a less noisy image but with high motion blur and with bright parts of the scene saturated.

The goal of this project is to propose and simulate an algorithm that improves motion blur, SNR and DR problems of conventional digital cameras with the use of high -speed CMOS cameras, which are available today.

II. PROBLEMS ADDRESSED AND ALGORITHMS

Our approach to solve the above-explained problem is, in general, to use high speed capabilities of CMOS image sensors, run them in a video or multicapture mode, get many images of the scene with varying exposure times, and incorporate motion estimation together with a noise estimation algorithm that fits into a CMOS imager to get motion-blur free, high SNR and high DR images.

By multicapture, we refer to a technique, where you continuously read all pixels values at very high speed without destroying the charge collected at each pixel while integrating the light signal. So, although you have a long exposure time for the last image you read, the intermediate images have less and less exposure time, thus less and less motion blur and SNR. Also, if you read the reset image, that is the image right after you reset the sensor, you can subtract the reset image from all other captures to get rid of reset noise and offset fixed pattern noise from your images. We are proposing a method, where one can use a point's pixel values at different captures along its motion trajectory to estimate for a final pixel value that is less noisy, not blurred by motion and higher DR.

While multicapture is only available to CMOS imagers, one can in theory apply our proposed algorithm to normal video mode CCD imagers where you destroy the charge when you do a read, and having differential images each with small exposure time, and adding them up to get higher and higher exposure time images (with the expense that you have to increase read noise as total exposure time increases, because each frame's read noise adds when you sum them up). However, the key thing is the speed of the imager. CCD's are not that fast to take short exposure time images. The minimum motion blur you can get in the final image will be as little as the motion blur of the minimum amount exposure time image you have. So, the algorithm depends on taking images very fast, such that some have minimum motion blur, but low SNR, some have lots of motion blur, but high SNR. The algorithm will combine those images to get a better final one. However, the readout speeds of CCD's are very slow, such that you cannot get lots of frequently spaced minimum motion blur pictures with a CCD camera. This is only possible with the CMOS imagers [1]. Additionally, with multicapture, you

have less read noise for each frame you capture compared to destructive read case (normal video-mode operation), we focused on a CMOS imager used in a multicapture mode to apply our algorithm.

So, the idea is to do multicapture at high speeds, and get N images of the scene each with longer exposure time. Then do motion estimation on the captured images. Then follow each pixel on the first image on its trajectory at the later frames and collect data to estimate the photocurrent that point in the scene induces. Then use these photocurrent estimates to construct a motion blur free, high SNR and DR image. Previous work [2] has used data within one pixel to achieve the above goals. We tried to incorporate motion estimation into it to get better results. But first, to understand what estimator we should use to estimate the photocurrent, we should model a CMOS imager.

1. CMOS Image Sensor Model

We are not going to go into a detailed model of CMOS image sensors here. We are going to use a simplified model that allows us to derive an estimator we can use.

A CMOS imager is basically a reverse biased photodiode. When light shines upon it, some of the photons create electron-hole pairs and they are swept away from each other at the reverse biased junction, and collected on the intrinsic capacitance of the diode. As the charge collects on the intrinsic capacitance of the diode, the reverse bias voltage across it drops. There is a minimum value that this voltage is allowed to drop, in order to avoid blooming of the charges to other pixels. That is why, a pixel has a maximum amount of charge it can collect. This is called well capacity. At low light, not the full capacity is used, but at high light, the pixel saturates (figure 2).

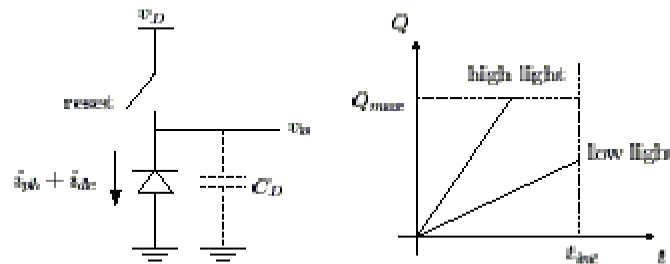


Figure 2. An illustration of a CMOS imager pixel and the charge it collects

The charge that collects on the capacitor consists of photocurrent (i_{ph}) charge, dark current (i_{dc}) charge, shot noise, reset noise, readout noise, gain FPN and offset FPN. (FPN is fixed pattern noise, which is caused by variations of CMOS processing, across the pixels) The offset FPN and reset noise are cancelled by correlated double sampling (CDS) where you read out the pixel values right after you reset them and subtract this reset image from the image after exposure. Although this would double the readout noise because reset frame has its own readout noise, which is independent from that of exposed image's, because readout noise is very small compared to reset noise and offset FPN generally, we prefer to do that. Gain FPN and dark current are also negligible for the high speed CMOS imager we have in our lab. So, taking that as a reference, we do not include dark current, offset and gain FPN and reset noise in our model. So, for a CDS'ed pixel, the charge that collects can be modeled as:

$$Q = i_{ph}T + U_T + V_T - V_0 \qquad (2.1.1)$$

where, T is the exposure time, U_T is the shot noise with zero mean and variance $qi_{ph}T$ (q is electron charge) and V_T and V_o are read noises of image with exposure time T and reset image (zero mean, variance σ_v^2). Similarly, the charge collected after kth capture in a multicapture scheme can be modeled as

$$Q_k = i_{ph}k\tau + \sum_{l=1}^{k} U_l + V_k - V_o \qquad (2.1.2)$$

where τ is the exposure time for single capture, and U_j is the shot noise collected during jth exposure.

Using this sensor model, we can derive the estimator we are going to use.

2. Photo-Current Estimation:

Suppose we have the trajectory of a point in the scene on the pixel plane, and the transition times (in integer multiples of τ) of that point to each pixel location. The algorithm estimates these motion trajectories from the multiple capture images from the camera with a motion estimation algorithm we will describe in the following pages. We want to use the integrated photo-charge values at correct pixel locations on the sensor plane during correct times to estimate for the photocurrent that a moving surface at the scene induces in the sensor.

To make everything easier, let us give an example. Suppose we have a 3x3 sensor plane as shown below:

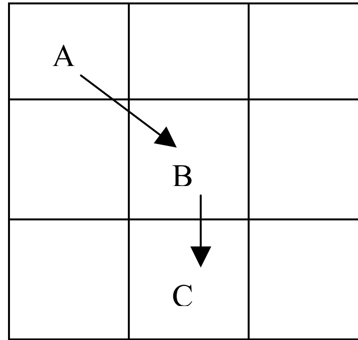


Figure 3: The Sensor Plane

Suppose we have a moving surface that initially shines on pixel A. Say after 3τ, it moved to pixel B on the pixel plane. And 2τ after that, it moved to pixel C. Suppose this surface was a bright surface and all the other things in the scene have darker surfaces.

The integrated charge and the charge values that are read out will be as follows:

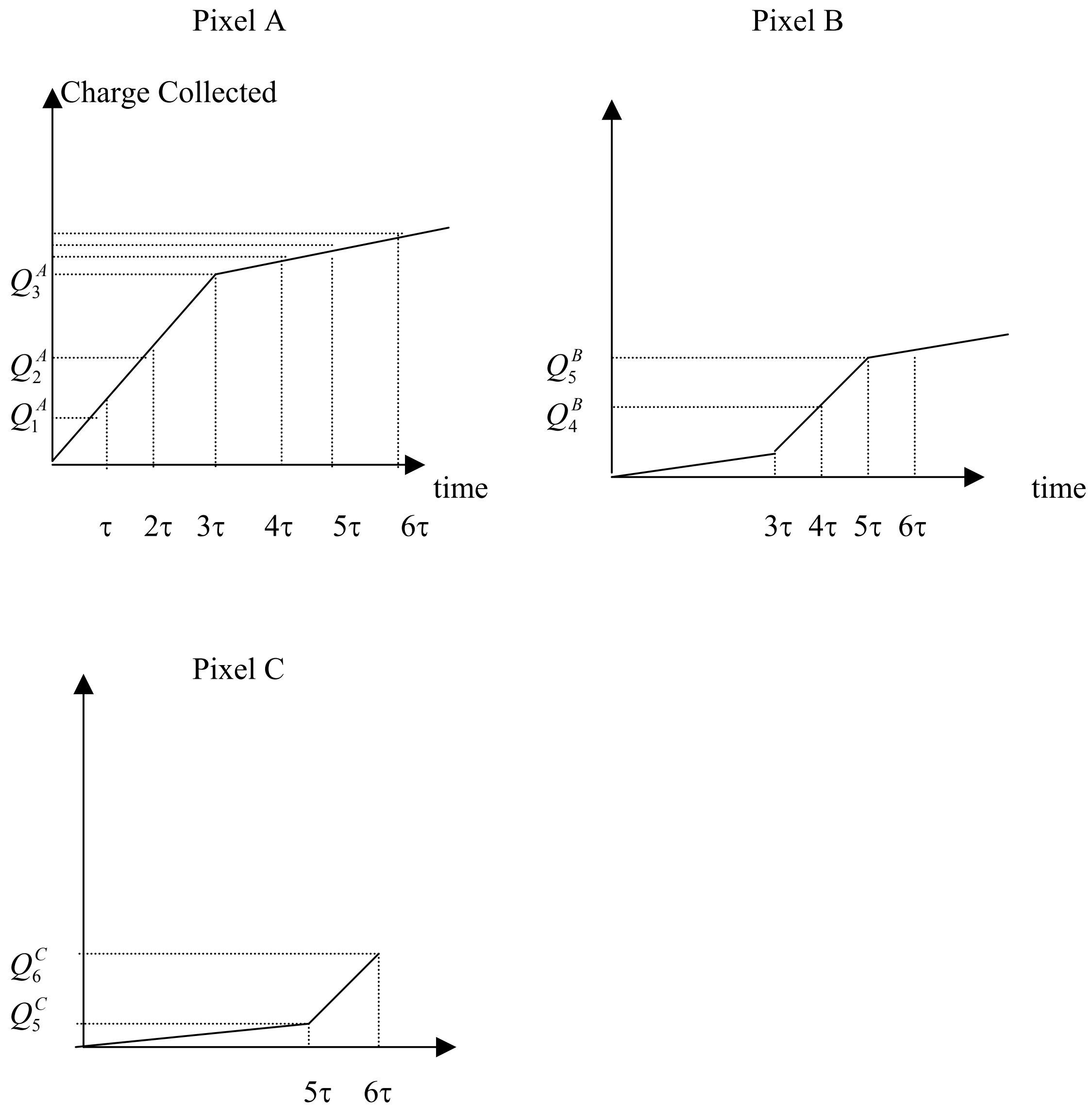


Figure 4: The collected charge values versus time at pixels A,B and C.

For the sake of simplicity, assume that we have a total of 6 captures. As seen from the charge-time graphs for these 3 pixels, the slope of the graph (the photo-current that the moving surface induces at a pixel) is equal during when the bright point in the scene shines on them. These times are from 0 to 3τ for pixel A, from 3τ to 5τ for pixel B, from 5τ to 6τ for pixel C. These transition times and the locations of the pixels A,B and C are found by the motion detection and estimation algorithm.

Now it comes to decide how to use the charge values that are read out to estimate for the correct photocurrent that the surface induces. We have:

$$Q_k^A = ik\tau + \sum_{l=1}^{k} U_l + V_k^A - V_o^A$$
$$Q_k^B = ik\tau + \sum_{l=1}^{k} U_l + V_k^B - V_o^B \qquad (2.2.1)$$
$$Q_k^C = ik\tau + \sum_{l=1}^{k} U_l + V_k^C - V_o^C$$

The i's in the above formulae are the photocurrents (i_{ph}) induced at pixels A, B and C, and they change with time, as different lights shine on them. But they are constant and same across all three pixels during the time segments of interest for each pixel.

The previous work [2] uses only the first 3 charge values read from pixel A to estimate for the photocurrent. Then it detects motion and stops here. We will use the relevant charge values from other pixels (B and C) also, to estimate photocurrent.

Define:

$$Q_1^{B'} = Q_4^B - Q_3^B$$
$$Q_1^{C'} = Q_6^C - Q_5^C \qquad (2.2.2)$$

We will use the charge collected at pixel A at times τ and 2τ. Then at 3τ that pixel moves to B. We are going to use the differential charge accumulated at pixel B from 3τ to 4τ because that differential charge is induced by the same point in the scene and so on. Now we have:

$$Q_1^{B'} = i\tau + U_4^B + V_4^B - V_3^B$$
$$Q_1^{C'} = i\tau + U_6^C + V_6^C - V_5^C \qquad (2.2.3)$$

On the other hand, if you look at the charge terms that are read from pixel A:

$$Q_1^A = i\tau + U_1^A + V_1^A - V_o^A$$
$$Q_2^A = i2\tau + U_1^A + U_2^A + V_2^A - V_o^A \qquad (2.2.4)$$

We want to calculate a linear estimate of $i,$ which is a weighted mean of the read out charge values, actually. But, as seen from above equations, $Q_1^A, Q_1^{B'}$ and $Q_1^{C'}$ all have the same signal ($i\tau$) and same amount of noise which are independent from each other. So, if one gives a weight w to one of these charges in the estimate, than all the others should get the same weight because none of them is more important than the other. So, one good strategy would be averaging them, and treating them as a single point. Averaging would decrease their variance, so that the final estimator will give them a bigger weight.

So, define:

$$\tilde{Q}_1 = \frac{1}{3}(Q_1^A + Q_1^{B'} + Q_1^{C'})$$
$$\tilde{Q}_2 = Q_2^A \qquad (2.2.5)$$

and use these $\tilde{Q}_j$'s to estimate for the photocurrent. Now because the $\tilde{Q}_j$'s are averaged versions of Q's, they have less noise. So, for each $\tilde{Q}_j$, if n_j is the number of Q's we are averaging, that is:

$$\tilde{Q}_j = \frac{1}{n_j}\sum_{l=1}^{n_j} Q_j^{X_l} \tag{2.2.6}$$

where, X_l is the name of the pixel (like A,B,C). The effective shot noise and readout noise in $\tilde{Q}_j$ will decrease by $1/\sqrt{n_j}$. So, we what we have actually is:

$$\tilde{Q}_k = ik\tau + \frac{1}{\sqrt{n_k}}\sum_{l=1}^{k} U_l + \frac{V_k - V_o}{\sqrt{n_k}} \tag{2.2.7}$$

Define:

$$\tilde{I}_k = \frac{\tilde{Q}_k}{k\tau} = i + \frac{\sum_{l=1}^{k} U_l}{\sqrt{n_k}\,k\tau} + \frac{V_k - V_o}{\sqrt{n_k}\,k\tau} \tag{2.2.8}$$

We have N captures. Given $\tilde{I}_1, \tilde{I}_2, \ldots \tilde{I}_N$ find coefficients a_j such that

$$\hat{I} = \sum_{j=1}^{N} a_j \tilde{I}_j \tag{2.2.9}$$

That minimizes the MSE:

$$\phi^2 = E(\hat{I} - i)^2 \tag{2.2.10}$$

Subject to unbiased estimate condition:

$$E(\hat{I}) = i \tag{2.2.11}$$

which is equivalent to:

$$\sum_{j=1}^{N} a_j = 1 \tag{2.2.12}$$

It can be found that the MSE is:

$$\phi^2 = \sum_{j=1}^{N}\left[\sigma_u^2\left(\sum_{l=j}^{N}\frac{a_l}{\sqrt{n_l}\,l\tau}\right)^2 + \left(\frac{a_j}{j\tau}\right)^2\frac{2\sigma_v^2}{n_j}\right] \tag{2.2.13}$$

Where, $\sigma_u^2 = qi\tau$, q is the electron charge.

When one writes the Lagrange multipliers and takes the derivatives to get:

$$F = \phi^2 + \lambda\left(\sum_{j=1}^{N} a_j - 1\right)$$

$$\frac{\partial F}{\partial \lambda} = 0 = \sum_{j=1}^{N} a_j - 1 \tag{2.2.14}$$

$$\frac{\partial F}{\partial a_s} = 0 = \frac{\lambda s}{2} + \frac{2\sigma_v^2}{s n_s \tau^2} a_{s-1} + \frac{\sigma_u^2}{\tau^2} \sum_{j=1}^{s} \sum_{l=j}^{N} \frac{a_l}{l\sqrt{n_l}}$$

One can re-arrange the above formulae and come up with the below equations to calculate the best weights as:

$$b_1 = 1$$

$$b_j = j\frac{n_j}{n_1} + \frac{j}{j-1}\frac{n_j}{n_{j-1}} b_{j-1} + j n_j \frac{\sigma_u^2}{2\sigma_v^2} \sum_{l=1}^{j-1} \frac{b_l}{l\sqrt{n_l}} \tag{2.2.15}$$

$$a_j = \frac{b_j}{\sum_{l=1}^{N} b_l}$$

The above scheme is done in an iterative way. That is, $\tilde{I}_1$ is first set to $\hat{I}$, then the shot noise variance for the next iteration is estimated using this estimated photocurrent, then using this shot noise variance, b_2 is estimated, using b_2, a_2 is found. Then $\hat{I}$ is updated. This goes on like this until you use all the Q values for that pixel.

Estimated photocurrent of each pixel will be used to generate the final image. This can simply be done as follows. If we consider the above example again, we had 3 modified Q values for a pixel (after averaging the ones from three pixels A, B and C). $\hat{I}$ gives the estimated slope of the charge vs. time graph. We can simply multiply this slope with 6τ and get the final what-it-should-be charge value for that pixel. This number can turn out to be larger than 255 (which would be the largest number from an 8 bit imager you can get) which means DR of the final image is larger than that of the imager.

How we get improvement in DR this way can be intuitively explained as follows. Suppose we had a very bright point in the scene that is surrounded by darker regions. If there was no motion this point would saturate the pixel it was shining on. But if it moves around, it does not saturate any of the pixels because it distributes the charge it induces to multiple pixels. So, if you can track this pixel right and do the above algorithm, you can recover all of the charge it induces and get total effective well capacity for that pixel larger than that of the imager. So, if there is motion, DR is extended at high illumination edge. Also, we did not implement the technique where you use the pixel values until the pixel saturates to do estimation in our code, which is described in [2] and effectively increases DR in the high illumination end. To add this feature to our algorithm is relatively easy, but as we focused more on motion blur and noise reduction in this project, we simply did not add that feature yet. Our algorithm also increases the DR in the low illumination end by decreasing noise, also. The algorithm effectively decreases camera noise, and reduces motion blur because tracks the points in the scene during their motion and estimates their photocurrent and constructs the final image using those photocurrent estimates.

3. Algorithm

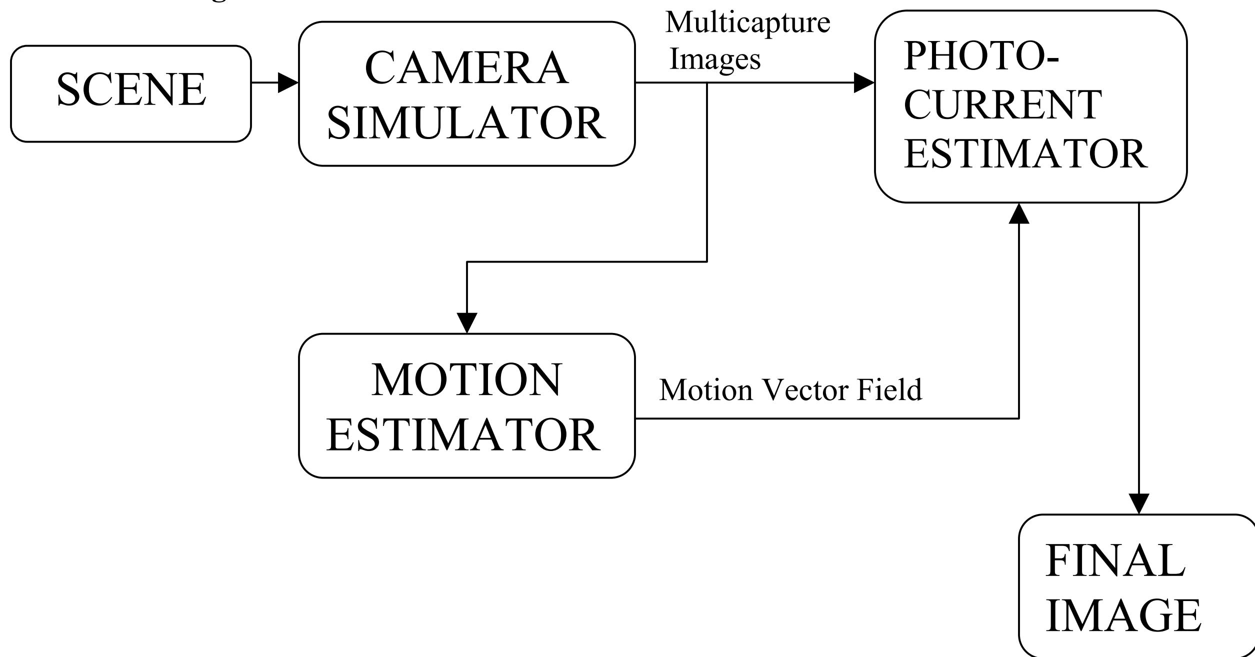


Figure 5: Block diagram for the algorithm

The scene is just a synthetic scene we construct in Matlab. It has a patch of a real image in the center with black area around (as can be seen in figure 1). The camera simulator simulates two things: 1) how many charges are collected at each pixel 2) the shot noise and read noise at each pixel. In order to have a realistic simulation we used experimentally obtained values of a typical setup for a) the read noise and b) the number of electrons created by the photoelectric effect per pixel per second.
While the camera is shooting this scene a defined motion is simulated so that motion blur results in the images formed. We had black regions around the patch shot by the camera because of two reasons. The first is, that we wanted to see the effect of read noise to our motion estimation in the absence of light. Second reason is, for the time being we did not want to deal with occlusions, so that the patch stays in the view of camera during the whole exposure.

The camera simulator shoots the scene and collects the charge at a position for a single capture time adding the right amount of noise, and moves to its next position and continues charging pixels from there. The total number of captures, single exposure time, scene, movement of the camera, the noise values, lightning conditions, the quantum efficiency of the camera are all user defined. A simple ideal ADC with an automatic gain control is also implemented and the conversion gain is supplied to the photocurrent estimator.

After the camera simulator, we have the multiple capture images. These images have an increasing level of brightness and blur. It is obvious that in order to calculate motion vectors these images cannot be used directly. It is necessary to calculate differential images with constant small blur and brightness, however with much higher noise. From these differential images, we estimate the motion vectors that describe the motion in the scene during the exposure time. We use block based motion estimation,

with a block size of 3 and search range of ±1 minimizing the sum of squared differences (SSD) between blocks. In order to estimate motion vectors in a robust way in the presence of noise we employ the following technique that Sebe suggested in his project presentation. Since we know the characteristics of noise of our sensor we can precisely say what SSD value could result purely from noise. In order to avoid wrongly calculated motion vectors therefore we add this known value to the SSD values of all search vectors that are not the zero vector, and thus favoring the zero vector by not more than just the appropriate amount. The search range and block size can be increased in the expense of computational load, but we assumed we did not need big block sizes because the imager we use is fast. (in [1], a 10,000 fps CMOS imager is described)

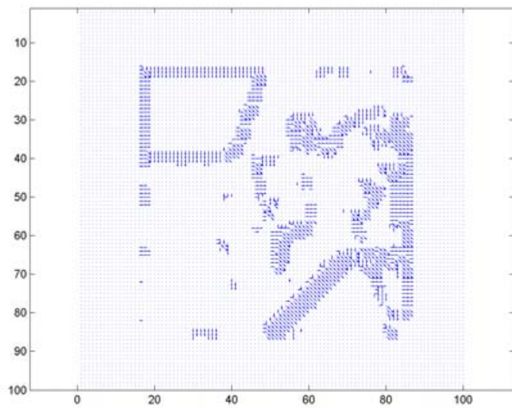
Then, the multicapture images and the estimated motion vectors are used to estimate for the photocurrent at each point in the scene. The multicapture images are in digital values, so conversion gain from the camera simulator is passed to the estimator to calculate for the corresponding charge values.

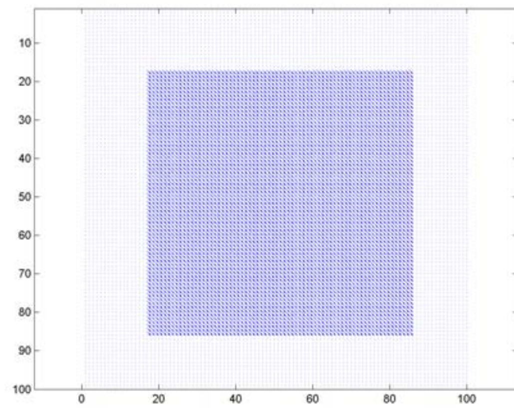
III. RESULTS

We have simulated the above algorithms in Matlab. We used 16 captures (17 with the reset one) each with 10 ms exposure time. A read noise level of 60e-, illumination level of 30 photons per pixel per brightness level of the scene per 10 ms is used, which are typical values.

We have 4 pixel motion blur in our simulated scene during the entire exposure time. Note that this imposes that the scene moves after only 4 captures. Previous work [2] has to stop at pixels where it detects motion and this would be only after 4 captures for some pixels. But this work relies on much more capture values for a pixel to have good estimation of its photocurrent. Under these conditions, for those pixels where there is motion, previous work would do a bad job. We illustrate in following lines that our algorithm achieves good results.

Following is a portion of the cameraman image used as a patch in our scene, and it moves globally. The single exposure (10 ms) and last capture (160 ms) are shown in figure 1 of introduction. The estimated and the perfect motion vectors from first frame to the second are shown in figure 6. The estimated vectors are estimated using the above algorithm. The perfect ones are constructed by us, for comparison, as we know the motion in the scene a priori.





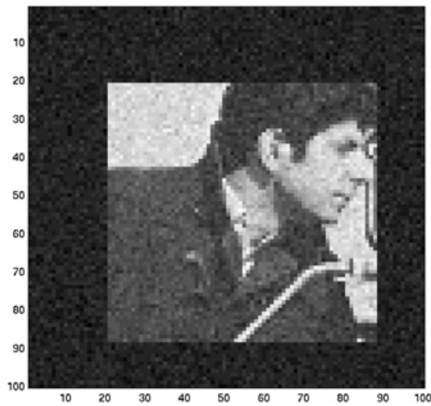
(a) (b)

Figure 6: The motion vectors: (a) Estimated (b) Perfect

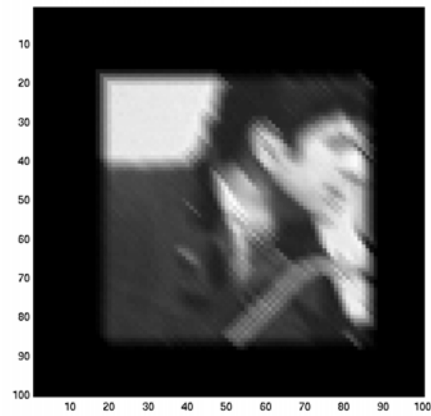
As you can see, where there is ambiguity, the motion vectors are not perfect. However, our photocurrent estimation algorithm is tolerant to this, because when there is motion for a pixel, if the point replacing it has the same brightness level, then the effect of that would be as if motion has not occurred, because the pixel would continue integrating with same amount of charge.

Figure 7 shows the results of our photocurrent estimation, together with the ones in figure 1 repeated. (7.c) is the one that uses estimated motion vectors. (7.d) is produced using perfect motion vectors, for comparison. As you can see, although (7.d) looks better, both images have almost no motion blur and very reduced noise, compared to ones in (7.a) and (7.b). The image with estimated motion vectors have lost some parts in the face of cameraman, due to imperfectness of the motion estimation algorithm.

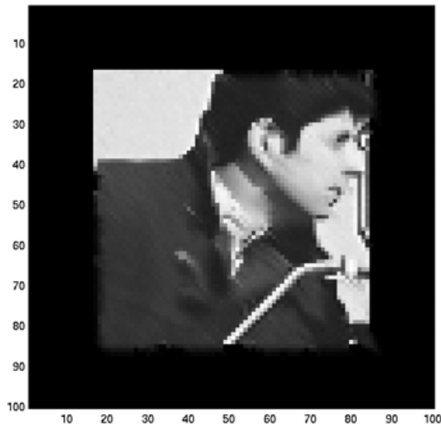
The std of the difference of the images at figure 7 from the original scene (which is a measure of how much noise and motion blur you have in the scene) are 71.31 (in digital counts) for 10 ms image, 37.41 for 160ms image, 12.05 for the image constructed using estimated motion vectors and 3.092 for the image constructed using real motion vectors.



(a)



(b)

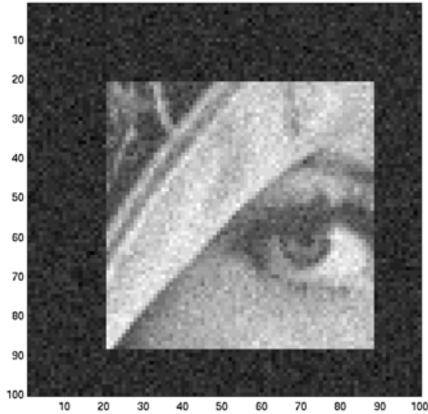


(c)

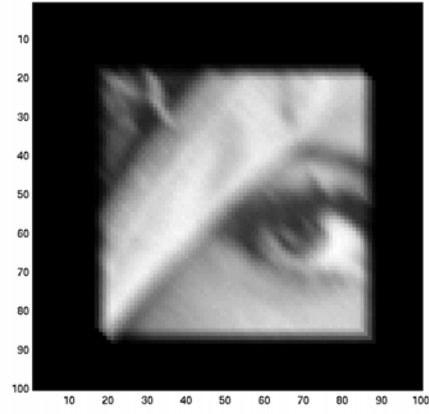
(d)

Figure 7: Cameraman test case. (a) 10 ms exposure image (b) 160 ms exposure image (c) Final Image using the estimated motion vectors (d) Final image using the perfect ones

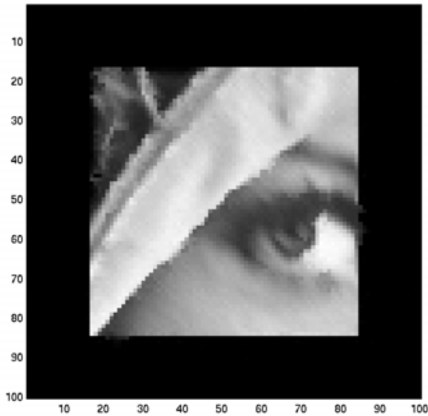
You can find the results to other test scenes in the following figures and the errors (std's) of the images in the following table.



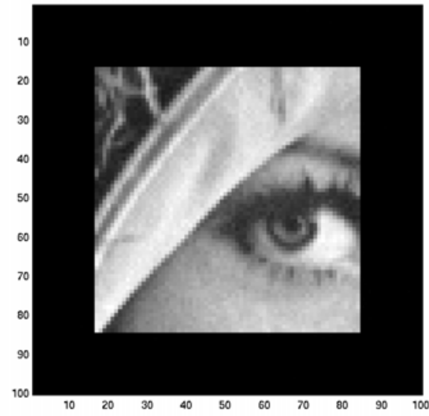
(a)



(b)

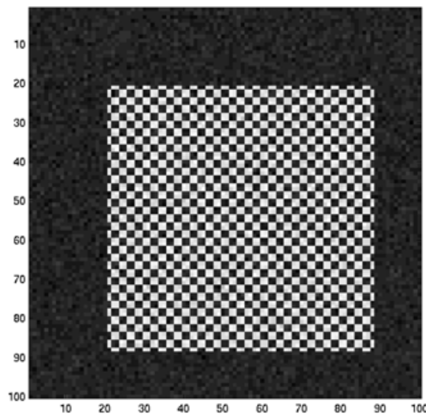


(c)

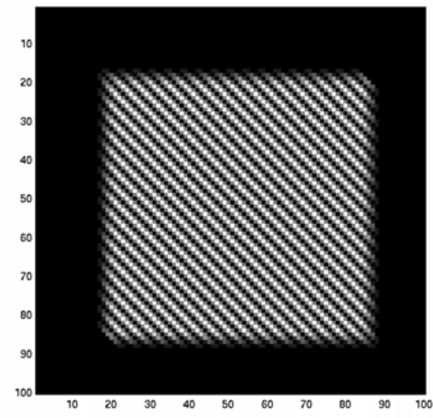


(d)

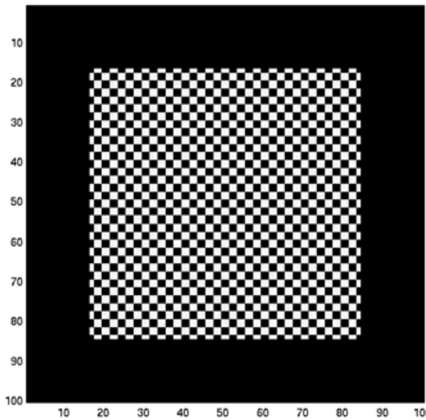
Figure 8: Lena test case. (a) 10 ms exposure image (b) 160 ms exposure image (c) Final Image using the estimated motion vectors (d) Final image using the perfect ones



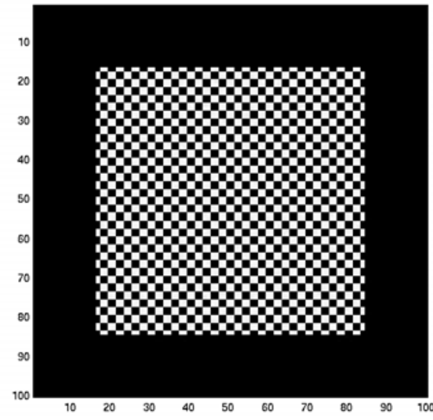
(a)



(b)



(c)



(d)

Figure 9: Checkerboard test case. (a) 10 ms exposure image (b) 160 ms exposure image (c) Final Image using the estimated motion vectors (d) Final image using the perfect ones

IMAGE ERRORS	CHECKER	LENA	CAMERAMAN
10 ms image	100.9	69.43	71.31
160 ms image	70.79	33.84	37.41
Const. with est. motion vectors	2.587	21.28	12.05
Const. with perfect motion vectors	2.576	17.22	3.092

Table 1. The comparison of errors of different test scenes

CONCLUSION

Within the limited time, we believe we achieved good results. The preliminary runs with simulated scenes gave both visually and in terms of error satisfactory outcomes. Now, there still are a lot of things that we need to do to make this algorithm complete. Some of the most important ones are:

- Use better motion estimation algorithms than block based
- Extend the algorithm to include the extension of DR with sensor saturation.
- Extend the algorithm to be able to handle occlusions.
- Generalize the algorithm to deal with sub-pixel motions.

REFERENCES

[1] S. Kleinfelder, S.H. Lim, X.Q. Liu and A. El Gamal A 10,000 Frames/s CMOS Digital Pixel Sensor, In *IEEE Journal of Solid State Circuits*, Vol.36, No.12, Pages 2049-2059, December 2001.

[2] X.Q. Liu and A. El Gamal, Simultaneous Image Formation and Motion Blur Restoration via Multiple Capture, In *ICASSP'2001 conference*, Salt Lake City, Utah, May 2001.

WORK DISTRIBUTION

It is hard to say "who did what" for our project, because we did everything together. Nobody basically worked independent of the other; we always came together, discussed what to do, how the algorithms should be, etc. Although we can say that the coding of the camera simulator and motion estimator were done by Ulrich, and codes that convert motion vectors and multicapture data to useful forms for the estimator and the code for the estimator were written by Ali, how to do these were discussed in detail before and during the coding process by both of us.

APPENDIX – M FILES

main.m

```
clear all, close all;

% Display Stuff
mygray = repmat(linspace(0, 1, 256)',1,3);
fps    = 5;
Nplay  = 1;


% 1) Define Sensor Characteristics
R = 100; C = R; N = 16;
std_v = 60; % Read noise (in e-)

% 2) Define Exposure Characteristics
dExpTime     = 10e-3;
ph_10ms_pix  = 30; % photons per 10ms of exposure per pixel


% 3) Definition of the scene
    scene1 = double(imread('lena.tif','tif')); %scene =
double(imread('testpat1.tif','tif'));
    imshow(scene,[]),axis image

    % Just a different scene
    AA     = ones(2,2)*255; BB = zeros(2,2); CC = [AA,BB;BB,AA];
    scene2 = repmat(C,128,128);

    % More different scenes
    scene3 = makePatternREC; %makePatternSMOOTH;
    figure, colormap(mygray), image(255*scene3/max(pat(:))),axis image;

scene = scene1;
pos0  = [210 200];
off = 16;
mask  = zeros(size(scene)); mask( pos0(1)+off : pos0(1)+R-off-1, pos0(2)+off :
pos0(2)+C-off-1 ) = 1;
scene = scene.*mask;
figure, imshow(scene,[]),axis image;
originalImage = scene(pos0(1):pos0(1)+R-1, pos0(2):pos0(2)+C-1);
figure, imagesc(originalImage), colormap(mygray), axis image;


% 4) Define Motion
TotalShift = 4;
positions  = createPositionsLin(N, TotalShift);
            %createPositionsCirc(TimeSteps, TotalAngle, Radius)

% END OF DEFINITIONS
% BEGIN OF MAIN ROUTINES

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create the Charge Distribution in the sensor's spacetime, with the following
parameters
% Sensor parameters (as described above)
% Exposure parameters
% The scene
% Motion parameters
noiseflag    = 1; adcflag = 1; % If 1, adds noise/does quantization
[Q,convgain] = createChargeDistribution(R,C,N, scene, positions, pos0,
dExpTime, noiseflag, adcflag, std_v, ph_10ms_pix);
Qmax         = max(Q(:));
```

```
% Play as movie
figure, colormap(mygray)
for tp = 1:N+1
    image(255*Q(:,:,tp)/Qmax),axis image;
    F(tp) = getframe;
end
movie(F,Nplay,fps)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the Differential Images
% from the discrete Charge Distribution

dQ = calculateDifferentialImages(Q);
dQmax = max(dQ(:));

% Play as movie
figure, colormap(mygray)
for tp = 1:N
    imagesc(dQ(:,:,tp)),axis image;
    F2(tp) = getframe;
end
movie(F2,Nplay,fps)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate the Motion Vector Field
% using
% Differential Images
% Noise level lamda used to reject false motion vectors because of noise
% Half block size hbs and search range sr
hbs = 1; sr = 1;
lambda = 2*sqrt(2)*std_v/(2*hbs+1); % Noise suppression factor for  motion
vector estimation

MVF = calculateMVFpix(dQ, hbs, sr, lambda);
% Play as movie
figure
for tp = 1:N-1
    quiver(MVF(:,:,tp,2), MVF(:,:,tp,1)),  axis ij, axis equal,
    F3(tp) = getframe;
end
movie(F3,Nplay,fps)


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create the datastructure 'motion' from the MVF
% This datastructure contains for each pixel in frame 1 a history of
% where it is later at which point in time.
motion = convert(MVF);


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create the datastructures 'modQ' and 'num' from the charge distribution Q
% and the array num.
% 'Qmod' contains a new charge distribution that contains
% for each pixel the amounts of charge that it created during
% its path over different pixels of the sensor surface
% The array 'num' keeps track of the numer of pixels per timestep over which
each point
% of the scene has moved on its trajectory. This information is
% needed for correct statistical evaluation in the routine 'blurFreeImConst'.
[modQ,num] = qModifier(Q,motion);
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Construct the blur free image with higher SNR and dynamic range
% from the information in modQ, and num that was gathered over the
% spacetime trajectory
im = blurFreeImConst(modQ, num, dExpTime, std_v, convgain);
figure,imagesc(im),colormap(mygray),axis image;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Do the same but use a perfect MVF (calculated from the original motion)
% as opposed to the MVF estimted from the image data
rMVF = realMVF(off, off, positions, [R,C]);
rmotion = convert(rMVF);
[rmodQ, rnum] = qModifier(Q, rmotion);
rim = blurFreeImConst(rmodQ,rnum,dExpTime,std_v,convgain);
figure,imagesc(rim),colormap(mygray),axis image;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Calculate some figures of merit
stdOneCap = std2(Q(:,:,2)-originalImage)
stdBlurred = std2(Q(:,:,16)-originalImage)
stdIm = std2(im - originalImage)
stdrIm = std2(rim - originalImage)
```

createChargeDistribution.m

```
function [Q,convgain] = createChargeDistribution(R, C, N, scene, positions,
support, dt, noiseflag, adcflag, std_v, poiss)

% sensor description
q       = 1.6*10^(-19);
qe      = 0.25;
epps1   = qe*poiss*100;  %q/sec  %total range epps1*(0 to 255)
sensorQ = zeros(R,C,N+1);


% go through all timesteps
for tp = 1:N

    % calculate the position of the scene on the imager
    pos = support - positions(tp,:) ;

    % calculate photon number given to a sensor pixel in a timestep
    epps = epps1*scene(pos(1):pos(1)+R-1, pos(2):pos(2)+C-1);  %electrons per
pixel and per second
    epp  = epps*dt;                                            %electrons per
pixel

    Mp = 10^12; eppn = reshape(Mp*imnoise(epp/Mp, 'poisson'), R, C);
    if noiseflag == 0
        eppn = epp;
    end
    sensorQ(:,:,tp+1) = sensorQ(:,:,tp) + eppn;

    % pixelwise and timestepwise add poisson noise
    %for r = 1:R
    %    for c = 1:C
```

```
    %        sensorQ(r,c,tp+1) = sensorQ(r,c,tp+1) +
noiseflag*poissrnd(epp(r,c));
    %    end
    %end
    %Mp = 10^12; sensorQ(:,:,tp+1) = reshape(Mp*imnoise(noiseflag*epp/Mp,
'poisson'), R, C);


end
sensorQp = sensorQ;



% => add poisson noise for frames independently
%Mp = 10^12; sensorQp = reshape(Mp*imnoise( sensorQ/Mp,'poisson'), R, C, N+1);

% => readout happens now
sensorQpr = sensorQp + noiseflag*std_v*randn(R,C,N+1);
%sensorQpr = sensorQp;


% Use Correlated Double Sampling
Q = sensorQpr - repmat(sensorQpr(:,:,1), [1 1 N+1]);

% convert from electron units to charge units
Q = Q*q;
convgain = 255/max(Q(:));

if adcflag == 0
    Q = (255*Q/max(Q(:))); % don't quantize Q (ADC)
else
    Q = floor(255*Q/max(Q(:))); % quantize Q (ADC)
end
```

calculateDifferentialImages.m

```
function dQ = calculateDifferentialImages(Q);

    sizeQ = size(Q); N = sizeQ(3);
    dQ    = zeros(sizeQ(1), sizeQ(2), N-1);

    for tp = N:-1:3
        dQ(:,:,tp-1) = Q(:,:,tp) - Q(:,:,tp-1);
    end

    dQ(:,:,1) = Q(:,:,2);
```

createPositionsLin.m

```
function positions = createPositionsLin(TimeSteps, TotalShift)
step       = ((TimeSteps)/TotalShift)^-1;
positions2 = round([0:step:TotalShift; 0:step:TotalShift]');
positions  = positions2(1:TimeSteps, 1:2);
```

calculateMVFpix.m

```
function MVF = calculateMVFpix(dQ, hbs, sr, lambda)

gridloc = zeros(1,1);
```

```
size_dQ = size(dQ); N = size_dQ(3);
MVF     = zeros(size_dQ(1), size_dQ(2), N-1, 2);


for tp = 2:N


    current  = dQ(:,:,tp);
    next     = dQ(:,:,tp-1);
    C        = double(current);
    NX       = double(next);

    % Determine the motion vector field (MVF) for one timestep
    marg = hbs+sr;

    % Go through all pixels
    for r = marg+1:size_dQ(1)-marg;
        for c = marg+1:size_dQ(2)-marg;

            loc = [r, c];

            % Cut a block from Current
            Cblock  = C( loc(1)-hbs:loc(1)+hbs, loc(2)-hbs:loc(2)+hbs );

            % Find a corresponding block in the Previous Frame and the
Translation-vector
            V = FindV(Cblock, loc, NX, sr, lambda);

            % Save this vector in MVF
            MVF(r, c, tp-1, :) = V;

        end
    end


end
```

findV.m

```
function V = FindV(Cblock, gridloc, NX, sr, lamda);
% This function finds a matching block in the next frame NX
% to the block Cblock in its vicinity -sr/+sr around its original
% location 'gridloc'
% The output is 'bestV' the corrsponding translation

bs = size(Cblock); hbs = (bs(1)-1)/2;

bestV   = [0,0];
bestSSD = inf;

% Search for the translation 'bestV' of the block Cblock witch yields
% the lowest SSD.
SSD = zeros(2*sr+1,2*sr+1);

for r = -sr:sr
    for c = -sr:sr

        % Get a block from P cut at the location gridloc + [r,c]
        loc     = gridloc + [r c];
        NXblock = NX(loc(1)-hbs:loc(1)+hbs, loc(2)-hbs:loc(2)+hbs);

        % Calc figure of merit SSD
```

```
        SSD(r+sr+1,c+sr+1) = sum(sum( abs(Cblock - NXblock).^2 )) + lamda*(r^2
+ c^2);
        %lamda+mat =

        % Calc figure of merit SSD
        %currentSSD = sum(sum(abs(Cblock - NXblock)));
        % Store global minimum
        %if currentSSD < bestSSD
        %    bestSSD = currentSSD;
        %    bestV = [r,c];
        %end

    end
end

%NXblockC = NX(gridloc(1)-hbs-1:gridloc(1)+hbs+1, gridloc(2)-hbs-
1:gridloc(2)+hbs+1);
%Cblock, NXblockC,
%SSD

% Search for the Vector
% How? Which one is the right one if there are several which yield the same
global minimum?
% 1) go circular

point = [];
[sizex sizey] = size(SSD); midx = (sizex-1)/2+1; midy = (sizey-1)/2+1;
%midx = hbs+1; midy = hbs+1;

[indexx indexy] = find(SSD == min(SSD(:)));
index = [indexx indexy]; normindex = index - midx*ones(size(index));

dist = [];
%[sizeindx sizeindy] = size(indexx);
for s = 1:size(indexx)
    dist(s) = sqrt(normindex(s,1)^2 + normindex(s,2)^2);
end

bestindrow = find(dist == min(dist(:)));
point = index(bestindrow(1),:);

V = -(point-[midx, midy]);
```

convert.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%
% function motion = convert(vectors,flag)
%
% This function converts the motion vector fields provided by the motion
estimation
% algorithm to motion trajectory matrices as required by the "qModifier"
function
%
% inputs  : vectors : The motion vector field (4 dimensional, vectors(i,j,t,:))
is
%                     the motion vector at point in the scene i,j at time t
%         : flag    : flsg(i,j) tells which frame is the last reliable frame
%                     for point (i,j) in the scene
% outputs : motion  : The motion trajectory matrices for each pixel
%                     motion{i,j} is a matrix that describes the motion of a
point
```

```
%                      in the scene
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%

function motion = convert(vectors);

[m n T dummy] = size(vectors);

for i = 1:m
    for j = 1:n
        motion{i,j}=[i j 1];
        for time = 1:T
            cur_point = motion{i,j}(end,1:2);
            v = vectors(cur_point(1),cur_point(2),time,:);
            v=reshape(v,1,2);
            if (v(1)~=0)|(v(2)~=0)
                %cur_point = motion{i,j}(end,1:2);
                nextpoint = cur_point+v;
                nextrow = [nextpoint(1),nextpoint(2),time+2];
                motion{i,j} = [motion{i,j};nextrow];
            end
        end
        lastrow = motion{i,j}(end,:);
        if lastrow(3) ~= (T+2)
            lastrow(3) = T+2;
            motion{i,j}=[motion{i,j};lastrow];
        end
    end
end

return;
```

qModifier.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% function [Q,num] = qModifier(inQ,motion)
% This function gets the camera charge as a function of pixel positions and
% frame numbers, and the trajectory of each point in the scene on pixel plane
% as input. It generates the averaged charge values of each pixel as a function
% of time as described in the project update report. It also outputs how many
% values were averaged per pixel during the process
%
% Inputs: inQ       : The charge collected at each pixel as a fn of no of
frames
%         motion    : The motion trajectory (cell array) last row of each
pixel's
%                     trajectory tells where to stop for that pixel
% Outputs: Q        : The averaged pixel charge values (zero padded to non-
valid
%                     charge values
%          num      : The number of averaged values per pixel
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%

function [Q,num] = qModifier(inQ,motion)

[m n T] = size(inQ);
Q = zeros(m,n,T);
num = zeros(m,n,T);
for i = 1:m
    for j=1:n
        traj = motion{i,j};
```

```
        initPoint = traj(1,1:2);
        [noPoints,dummy] = size(traj);
        noPoints = noPoints - 1;

        if noPoints<=0
            error('The motion trajectory is defined falsely - the number of 
rows in each cell has to be at least 2');
        end

        frames = traj(:,3);

        for k = 1:noPoints
            %k,pause;
            point = traj(k,1:2);
            frameStart = frames(k)+1;
            frameEnd =frames(k+1)-1;
            if k == noPoints
                frameEnd = frameEnd+1;
            end
            pointx = point(1);
            pointy = point(2);

            originCharge = inQ(pointx,pointy,frameStart-1);
            QCoorx = initPoint(1);
            QCoory = initPoint(2);

            for frame = frameStart:frameEnd
                QCoort = frame-frameStart+2;
                inQCoorx = point(1);
                inQCoory = point(2);
                inQCoort = frame;

Q(QCoorx,QCoory,QCoort)=Q(QCoorx,QCoory,QCoort)+inQ(inQCoorx,inQCoory,inQCoort)
-originCharge;
                num(QCoorx,QCoory,QCoort)=num(QCoorx,QCoory,QCoort)+1;
            end
            %Q10= squeeze(Q(10,10,:))'
            %n10=squeeze(num(10,10,:))'
        end
        thisNum = squeeze(num(QCoorx,QCoory,:));
        nonzeroInd = find(thisNum~=0);
        Q(QCoorx,QCoory,nonzeroInd) = 
Q(QCoorx,QCoory,nonzeroInd)./num(QCoorx,QCoory,nonzeroInd);
    end
end

return;
```

blurFreeImConst.m

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
% function im = blurFreeImConst(modQ,num,singleExpTime,sigma_v,sensorGain)
%
% This function gets the modified (averaged) Q values across pixel positions 
and
% time and consructs the estimated photo-current matrix for each pixel. Then
% calculates the final HDR image using estimated photocurrent.
%
% Inputs: modQ      : The averaged Q values (in DV's)(generated by 
"qModifier.m")
%         num       : The number of averaged values per pixel.
%         singleExpTime: Exposure time for a single exposure
```

```
%         sigma_v   : Read noise std dev. in e-
%         sensorGain: Gain of the sensor in DV/e;
% Outputs: im       : The final motion blur free low noise high DR image
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%

function im = blurFreeImConst(modQ,num,singleExpTime,sigma_v,sensorGain)

[m n T] = size(modQ);
modQ = modQ/sensorGain; % convert DV's to e-'s

for i = 1:m
    for j=1:n
        qPix = squeeze(modQ(i,j,:));
        nPix = squeeze(num(i,j,:));
        qPix = qPix(2:end); % The first value is zero, for reset value, after
CDS
        nPix = nPix(2:end); % Get rid of the reset charge and num for that

        % Check the modQ and num entrees were constructed right.

        zeroIndQ = find(qPix == 0);
        zeroIndn = find(nPix == 0);

%         if ~(zeroIndQ == zeroIndn)
%             error(strcat('The zero entrees in Q and num for
pixel',int2str(i),',',int2str(j),'do not match'));
%         end
        if (~(isempty(zeroIndQ))) & (~(isempty(zeroIndn)))
            temp1 = diff(zeroIndQ);
            temp2 = diff(zeroIndn);
            if (~(isempty(find(temp1>1)))) |  (~(isempty(find(temp2>1))))
           %     error(strcat('The zero entrees in Q and num for
pixel',int2str(i),',',int2str(j),'are not proper'));
            end
        end


        nPix = nPix(find(nPix ~= 0)); % The non-zero entrees of nPix are only
meaningful
        qPix = qPix(find(nPix ~= 0));

        % Calculate the coefficients for estimator and estimate of the
photocurrent
        % As the estimate for the photocurrent is recursive, each step we
estimate
        % a value for the photocurrent and use its estimate as an estimate for
the
        % shot noise for the next step

        s = length(nPix);

        time = singleExpTime*(1:s);
        I_tilda = qPix'./time;

        b = [];
        g = [];
        h = [];
        I_hat = [];
        b(1) = 1;
        g(1) = 1;
        h(1) = 1;
        I_hat(1) = I_tilda(1);
```

```
        for k = 2:s
            sigma_uSq = I_hat(k-1)*singleExpTime;
            divisor = (1:k-1).*sqrt(nPix(1:k-1))';
            b(k) = k*nPix(k)/nPix(1)+k/(k-1)*nPix(k)/nPix(k-1)*b(k-1)+...
                k*nPix(k)*sigma_uSq/2/(sigma_v^2)*sum(b(1:k-1)./divisor);
            g(k) = sum(b(1:k));
            h(k) = b(k)/g(k);
            I_hat(k) = I_hat(k-1)+h(k)*(I_tilda(k)-I_hat(k-1));
        end
        % Form the final photocurrent estimate matrix
        I(i,j) = I_hat(s);
    end
end

% Now the final HDR image can be constructed;

im = (T-1)*singleExpTime*I*sensorGain;
im(find(im<0)) = 0;
return;
```

realMVF.m

```
% Function that gives the real motion vectors

function MVF = realMVF(offsetx,offsety,positions,sizeImager);
% offsetx,offsety = offset of the patch in the camera
% positions = position shifts of the camera
[N dum] = size(positions);
MVF = zeros(sizeImager(1),sizeImager(2),N-2,2);
pos = [0 0];
for t = 1:N-1 % The MVF will be N-1 sized
    mvf =positions(t+1,:)-positions(t,:);
    pos = pos+mvf;
    for i=offsetx+pos(1):sizeImager(1)-offsetx+pos(1)
        for j = offsety+pos(2):sizeImager(2)-offsety+pos(2)
            MVF(i,j,t,:) = mvf;
        end
    end
end
return
```

makePatternREC,m

```
function pat = makePatternREC;

pat   = zeros(10,256);
for k = 1:100
    pat(k,:) = round((128 + 127*sin(2*pi*k^1.5/20*[1:256]/256))/255)*255;
end
```

makePatternSMOOTH.m

```
function pat = makePatternSMOOTH;

pat   = zeros(10,256);
for k = 1:100
    pat(k,:) = 128 + 127*sin(2*pi*k^1.5/20*[1:256]/256);
end
```