Low cost imaging and image processing with the Raspberry Pi

by

Jamie Magee

Fourth-year undergraduate project
in Group F, 2013/2014

DECLARATION

I hereby declare that, except where specifically indicated, the work submitted herein is my own original work.

Jamie Magee
2014

ABSTRACT

Optical flow is a critical tool in many experimental methods such as Particle Image Velocimetry for flow analysis and soil mechanics, speckle tracking echocardiography in medical imaging, as well as mechanical testing. During this project Lucas-Kanade, Farnebäck, and SimpleFlow optical flow methods were compared using artificial and real world datasets on the limited hardware of the Raspberry Pi. For dense optical flow Farnebäck was found to have the lowest processing time, SimpleFlow was found to have the highest accuracy, and Lucas-Kanade was found to lie somewhere in between. However, when using Lucas-Kanade as a sparse algorithm it was found to have the lowest processing time. Instructions, aimed at undergraduate level, on how to install the tools used in this project are provided, along with the robust toolset used in testing. Practical applications were also demonstrated by integrating optical flow methods into a tensile testing machine developed as part of the OpenLabTools initiative. Further work investigating performance under arbitrary affine transformations, or projective transformations using the toolset developed as part of this project is possible. Visualisation of vector field showing the curl or the divergence of the vector field may also be a useful tool.

CONTENTS

INTRODUCTION

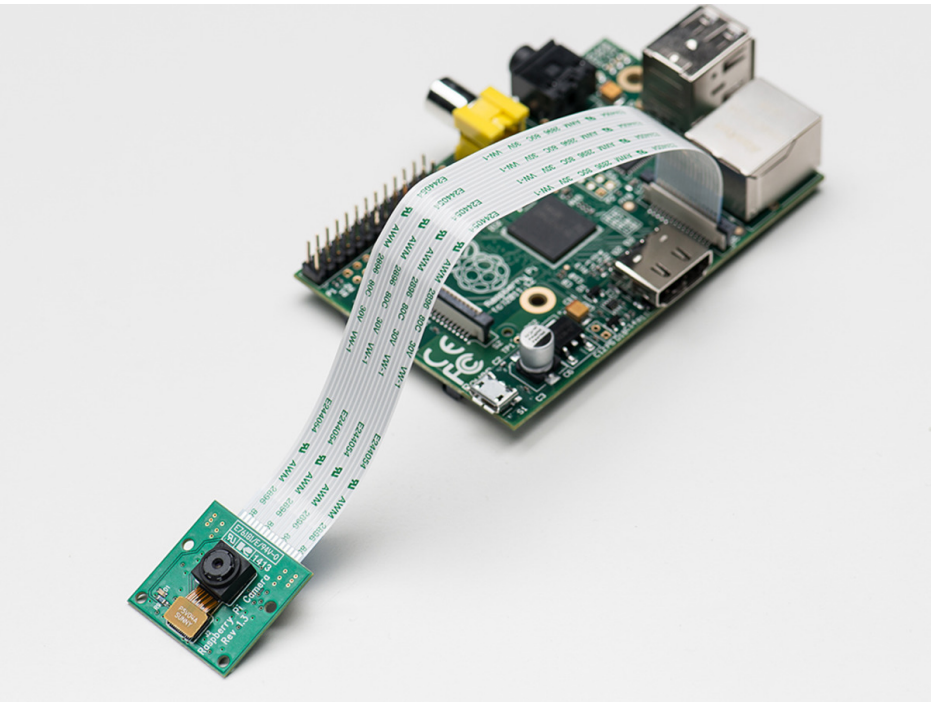


Fig. 1.1 Raspberry Pi with camera module attached

The Raspberry Pi is a low cost, credit-card sized computer developed in the UK by the Raspberry Pi Foundation. The primary objective of the Raspberry Pi is to increase, and encourage the teaching of computer science in schools. In this regard it has been highly successful and since its launch in February 2012 it has sold over 2 million units [16], however its success has also been, in part, due to the hobbyist and scientific markets. The Raspberry Pi boasts a 700MHz ARMv6k, 512 MB of RAM and a full Linux operating system for $35 [1]. In addition, the Raspberry Pi foundation have released a camera module that supports up to 1080p30 video capture, and can be seen in Figure 1.1. All of these features make the Raspberry Pi an ideal candidate for imaging and image processing on a low budget.

Optical flow is the apparent motion of objects in a scene due to relative motion between an observer and the objects being observed as seen in Figure 1.2. It is a critical tool in many experimental methods such as Particle Image Velocimetry [13] for flow analysis and and soil mechanics, speckle tracking echocardiography [6] in medical imaging, as well as mechanical testing [9]. There already exist commercial packages for executing optical flow algorithms, however licenses can often be costly and the software can require high end hardware. This project, as part of the OpenLabTools initiative, aims to provide access to optical flow methods on low cost and low powered hardware. The main objectives for the project are to:

- Provide access to optical flow methods on cheap and readily available hardware
- Perform online and offline processing using the Raspberry Pi
- Capture data using the Raspberry Pi Camera
- Integrate with existing experiments and other OpenLabTools projects
- Provide a clear set of instructions aimed at undergraduate level

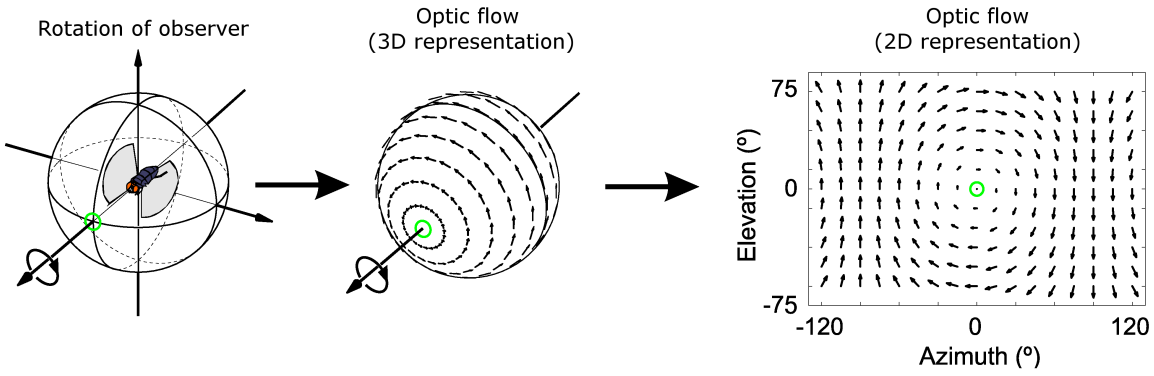


Fig. 1.2 Optical flow caused by a rotating observer [10]

In section 2, Theory and design of experiment, I will explain the assumptions behind the theoretical development and the application of the theory. Section 3, Apparatus and experimental techniques, describes the tools used in the running of the experiments and discusses the experimental accuracy. Section 4, Results and discussion, presents the results obtained from the various different experiments and analyses the raw data. Finally, section 5, Conclusions, highlights the main results and outlines avenues for further work and development.

2.1 Optical Flow

Optical flow methods attempt to calculate the motion between two images, or video frames, taken at time t and $t+\Delta t$ for every pixel in the images. For a 2D case a pixel at location (x,y,t) with intensity $I(x,y,t)$ will have moved by $\Delta x, \Delta y$ and Δt between the two image frames and therefore we can impose the constraint on the intensity:

$$I(x,y,t) = I(\Delta x, \Delta y, \Delta t)$$

If we assume that the movement is small, we can perform a Taylor series expansion on the constraint:

$$I(\Delta x, \Delta y, \Delta t) = I(x,y,t) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t + \ldots$$

Ignoring higher order terms, and rearranging, it follows that

$$\frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y + \frac{\partial I}{\partial t}\Delta t = 0$$

and dividing through by Δt we obtain

$$\frac{\partial I}{\partial x}\frac{\Delta x}{\Delta t} + \frac{\partial I}{\partial y}\frac{\Delta y}{\Delta t} + \frac{\partial I}{\partial t}\frac{\Delta t}{\Delta t} = 0$$

which simplifies to

$$\frac{\partial I}{\partial x}V_x + \frac{\partial I}{\partial y}V_y + \frac{\partial I}{\partial t} = 0$$

where V_x, V_y are the x and y components of the velocity or optical flow of $I(x,y,t)$ and $\frac{\partial I}{\partial x}$, $\frac{\partial I}{\partial y}$ and $\frac{\partial I}{\partial t}$ are the derivatives of the image at (x,y,t) in the corresponding directions. We can rearrange this to

$$I_x V_x + I_y V_y = -I_t$$

where I_x, I_y and I_t are $\frac{\partial I}{\partial x}$, $\frac{\partial I}{\partial y}$ and $\frac{\partial I}{\partial t}$ respectively. This is an equation in two unknowns, I and V, and therefore we cannot solve it in its current form. We require additional equations which introduce an additional constraint. All optical flow methods introduce an additional constrain to enable estimation of the optical flow.

2.1.1 Lucas-Kanade

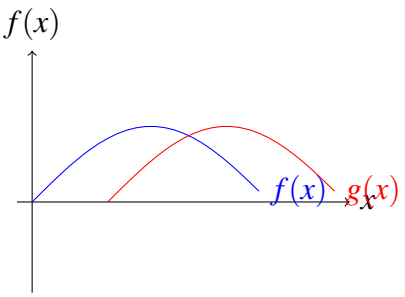


Fig. 2.1 The 1-D case

The Lucas-Kanade [11] method is a well established, sparse optical flow method. In the one dimensional case, illustrated in Figure 2.1, it can be thought of as finding the horizontal displacement, δ, between the curves $F(x)$ and $G(x) = F(x+\delta)$. If we assume that δ is small and that $F(x)$ is approximately linear in the area of x then we can say:

$$F'(x) \approx \frac{F(x+\delta) - F(x)}{\delta}$$
$$\therefore F(x+\delta) \approx F(x) + \delta F'(x)$$

Therefore, to find the δ which minimises the L_2 norm we measure the difference between the curves as

$$E = \sum_x [F(x+\delta) - G(x)]^2$$

and then to minimise the error with respect to δ, we set

$$\begin{aligned} 0 &= \frac{\partial E}{\partial \delta} \\ &\approx \frac{\partial}{\partial \delta} \sum_x \left[F(x) + \delta F'(x) - G(x)\right]^2 \\ &= \sum_x 2F'(x) \left[F(x) + \delta F'(x) - G(x)\right] \end{aligned}$$

from which

$$\delta \approx \frac{\sum_x F'(x)\,[G(x) - F(x)]}{\sum_x F'(x)^2}$$

We can generalise this to n-dimensions as follows. We still wish to minimise the L_2 norm as a measure of error

$$E = \sum_{x \in \mathbb{R}} [F(x+\delta) - G(x)]^2$$

where x and δ are n-dimensional row vectors. We make a linear approximation as before

$$F(x+\delta) \approx F(x) + \delta \frac{\partial}{\partial x} F(x)$$

Where

$$\frac{\partial}{\partial x} = \left[\frac{\partial}{\partial x_1} \frac{\partial}{\partial x_2} \cdots \frac{\partial}{\partial x_n}\right]^T$$

and as before to minimise the error with respect to δ, we set

$$\begin{aligned} 0 &= \frac{\partial E}{\partial \delta} \\ &\approx \frac{\partial}{\partial \delta} \sum_x \left[F(x) + \delta \frac{\partial F}{\partial x} - G(x) \right]^2 \\ &= \sum_x 2 \frac{\partial F}{\partial x} \left[F(x) + \delta \frac{\partial F}{\partial x} - G(x) \right] \end{aligned}$$

from which

$$\delta = \left[\sum_x \left(\frac{\partial F}{\partial x} \right)^T [G(x) - F(x)] \right] \left[\sum_x \left(\frac{\partial F}{\partial x} \right)^T \left(\frac{\partial F}{\partial x} \right) \right]^{-1}$$

2.1.2 Farnebäck

Farnebäck optical flow [8] is classified as a dense optical flow method. Using this method each neighbourhood of pixels is approximated by a quadratic polynomial of the form:

$$f(x) \approx x^T A x + b^t x + c$$

In the one dimensional case is shown in figure 2.2.

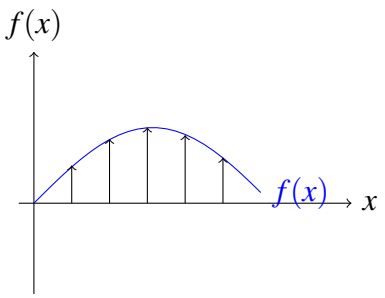


Fig. 2.2 The 1-D case

For the general case, if we consider the polynomial $f_1(x) \approx x^T A_1 x + b_1^T x + c_1$ which is shifted by a displacement δ

$$\begin{aligned} f_2(x) &= f_1(x-\delta) \\ &= (x-\delta)^T A_1 (x-\delta) + b_1^T (x-\delta) + c_1 \\ &= x^T A_1 x + (b_1 - 2A_1\delta)^T x + \delta^T A_1 \delta - b_1^T \delta + c_1 \\ &\equiv x^T A_2 x + b_2^T x + c_2 \end{aligned}$$

We can therefore equate the coefficients of $f_1(x)$ and $f_2(x)$ which yields

$$\begin{aligned} A_2 &= A_1 \\ b_2 &= b_1 - 2A_1\delta \\ c_2 &= \delta^T A_1 \delta - b_1^T \delta + c_1 \end{aligned}$$

Thus if A_1 is non-singular we can solve for the translation δ

$$\begin{aligned} b_2 &= b_1 - 2A_1\delta \\ 2A_1\delta &= -(b_2 - b_1) \\ \delta &= -\frac{1}{2} A_1^{-1} (b_2 - b_1) \end{aligned}$$

2.1.3 SimpleFlow

SimpleFlow is a combination of both dense and sparse optical flow, it only runs a full flow estimation at a few key pixels, and linearly interpolates the rest of the flow, thus it is classified as *semi-dense* optical flow [15]. At the key pixels a likelihood model is used, that is, we seek flow vectors (u,v) such that $F_t(x,y)$ and $F_{t+1}(x+u,y+v)$ are similar. This is modelled by the energy term:

$$e(x,y,u,v) = \|F_t(x,y) + F_{t+1}(x+u,y+v)\|^2$$

that corresponds to the likelihood probability $p \propto \exp(-e)$ [14]

SimpleFlow assumes that the flow is locally smooth, however it does not rely on a simple pairwise term such as $||(u_1,v_1)-(u_2,v_2)||^2$ as is often used as pairwise terms link flow

estimates at several pixels therefore making optimization problem harder to solve. Instead SimpleFlow requires the flow vector (u_0, v_0) at pixel (x_0, y_0) to not only be a good estimate for the optical flow of (x_0, y_0), but at other pixels in the local neighbourhood, $\mathcal{N}_0$. This can be expressed as

$$(u_0, v_0) = \arg \min_{(u,v)\in\Omega} \prod_{(x,y)\in\mathcal{N}_0} p(x, y, u, v)$$

where Ω is the set of all possible (u, v) vectors considered. This produces a smoother estimate of the optical flow of (x, y). We can better understand this behaviour in the negative log-likelihood (or energy cost) domain where we try to find the lowest cost flow vector for the pixel (x_0, y_0)

$$(u_0, v_0) = \arg \min_{(u,v)\in\Omega} \sum_{(x,y)\in\mathcal{N}_0} e(x, y, u, v)$$

The smoothness prior amounts to smoothing the the likelihood term instead of adding a pairwise term. As a consequence, finding a minimizer remains simple since there is no interaction between the unknowns, the solution at a pixel does not depend on the solution at its neighbours.

Smoothing with a box filter would not differentiate between pixels within $\mathcal{N}_0$, e.g. it would ignore edges. Therefore weights must be added to account for how pixels relate to each other. Two weighting functions are required, w_d for the distance between pixels and w_c for the colour difference. This is represented as:

$$\begin{aligned} E(x_0, y_0, u, v) &= \sum_{(x,y)\in\mathcal{N}_0} w_d w_c e(x, y, u, v) \\ \text{with } w_d &= \exp\left(-||(x_0, y_0) - (x, y)||^2 / 2\sigma_d\right) \\ \text{and } w_c &= \exp\left(-||F_t(x_0, y_0) - F_t(x, y)||^2 / 2\sigma_c\right) \end{aligned}$$

3.1 OpenCV

To aid in implementing image processing algorithms, including optical flow methods, OpenCV [2] was used. OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. A precompiled binary does not exist for the Raspberry Pi, so it was necessary to compile it from source and write instructions on how to do so. On the Raspberry Pi the compilation time is around 10-12 hours depending on the overclock used.

Also included in code listing A.1, Appendix A are instructions on how to allow the Raspberry Pi camera to interface with the operating system. An extra driver (UV4l-raspi [5]) is required, however a precompiled binary is available in this case. Also included in written up instructions on how to install SimpleCV [4], a Python wrapper for OpenCV. SimpleCV was not used in the following experiments due to the performance benefits of C/C++ over Python, however the instructions are included as they will be provided on the OpenLabTools website.

In order to use the library to perform a comparison, an object oriented program which contained implementations of the Lucas-Kanade, Farnebäck, and SimpleFlow algorithms. The program was designed to be run from the command line, so that it's operation can be scripted using BASH or any other scripting language, and also so as not to waste unnecessary resources on drawing a GUI. The full code listing for the program can be seen online [12]. Listed below is the help message for the `optical-flow` program. Note that multiple optical flow algorithms can be applied to the same input file at once i.e. `./optical-flow -fls -o output.avi input.avi` will run Lucas-Kanade, Farnebäck and SimpleFlow optical flow algorithms on the file `input.avi` and will output `output.lucas-kanade.avi` and `output.simpleflow.avi`

```
Usage: ./optical-flow [options] file...
Options:
  -h, --help          Display this help message
  -v, --version       Display the program version
  -o, --output        The file you wish to write to
  -f, --farneback     Use the Farneback optical flow method
  -l, --lucas-kanade  Use the Lucas-Kanade optical flow method
  -s, --simpleflow    Use the Simpleflow optical flow method
```

3.2 Comparison

The first area investigated was a comparison between the three optical flow algorithms previously listed for speed and accuracy on the Raspberry Pi. Each algorithm will be tested using an artificial dataset, and a real world dataset. Accuracy is only able to be measured quantitatively for the artificial dataset, and it is measured by taking the L_2 norm of the optical flow output from each algorithm and comparing it to the optical flow calculated. This technique is explained further in the section beneath. The speed of each algorithm is measured for both the artificial dataset and the real world dataset, and it is measured by calculating the mean processing time per pair of video frames processed.

Timing data was output from the `optical-flow` program to stdout, however when the comparison process was automated, this was redirected to a text file. A typical output can be seen beneath

```
Output file fracture.lucas-kanade.avi using Lucas-Kanade optical flow

Frame 1 took 0.24s to process
...
Frame 11 took 75.38s to process

Processing complete for fracture.lucas-kanade.avi
```

In order to process the large amount of timing information in an accurate and repeatable manner, a python script was authored. The full script can be seen in code listing A.2 of Appendix A. A typical output from the script can be seen beneath

```
Output file fracture.lucas-kanade.avi using Lucas-Kanade optical flow
74.60s average per frame
```

In order to process the large amount of optical flow information, and compare it to the known vector flow field, the output from `optical-flow` first has to be converted into

a form that can be read by MATLAB. `optical-flow` outputs vector field information in the YAML (Yet Another Markup Language) format, with Farnebäck and SimpleFlow directly outputting the vector field, whereas Lucas-Kanade outputs two arrays, `prevPts` and `nextPts`, which hold the coordinate pairs of the pixels being tracked in frame F_t and followed by their position in frame F_{t+1}. An example of the vector field data output from the Farnebäck algorithm is listed below

```
%YAML:1.0
FlowField : !!opencv-matrix
rows: 240
cols: 240
dt: "2f"
data: [ 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
...
       -4.66855984e-34, -1.97913447e-33, -3.70633920e-34 ]
```

Therefore a Python script was authored in order to convert this output into something which could be more easily processed. The full script can be seen in code listing A.3 of Appendix A.

3.2.1 Artificial Data

The artificial dataset consists of the MATLAB `checkerboard` pattern, which has five different affine transformations repeatedly applied to it in order to generate five different artificial data sequences. The sequences include: translation, rotation, scaling, shearing and fracture. Frames from each of the sample sequences can be seen in Figure 3.1 below.

(a) Original (b) Translate (c) Rotate (d) Scale (e) Shear (f) Fracture

Fig. 3.1 Artificial dataset

Each of the above sequences was generated by the following affine transformation matrices respectively:

$$\begin{bmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ S & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

From left to right, the above transformation matrices represent: a translation, of T_x pixels in the x-direction and T_y pixels in the y direction; a rotation of θ degrees about the centroid of the `checkerboard` pattern; a scaling of S_x in the x-direction and S_y in the y direction; A shear parallel to the x-axis of S pixels i.e. $x' = x + ky$. The fracture is represented in MATLAB as two separate translations applied to two halves of the original `checkerboard` pattern. For this dataset $T_x = -5i, T_y = -5i, \theta = 1^\circ, S_y = (i+1)/2, S_x = (i+1)/2$ and $S = .5$ where i is the frame of the video being generated. The fracture had transformation equivalent to a translation with $T_y = sgn(y)2i$ applied to it, where the centroid of the `checkerboard` pattern is coincident with the origin. Each transformation is applied repeatedly in order to generate sample sequences ten frames long. The MATLAB code used to generate the rotation sample sequence can be seen in code listing 3.1. The complete MATLAB code listing used to generate all the sample sequences can be seen in code listing A.4 in Appendix A.

```
theta=1;
tform=affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0;
  0 0 1]);
writerObj = VideoWriter('rotate.avi');
open(writerObj);
frame=uint8(padarray(orig,[200 200],255).*255);
writeVideo(writerObj,frame);
r=imwarp(orig,tform,'FillValue',255);
for i=1:10
  dr=size(r,1)-size(orig,1);
  dc=size(r,2)-size(orig,2);
  frame=uint8(padarray(r,[200-dr/2 200-dc/2],255).*255);
  writeVideo(writerObj,frame);
  r=imwarp(r,tform,'FillValue',255);
end
close(writerObj);
```

Listing 3.1 MATLAB code for creating rotation sample sequence

In order to evaluate the algorithms more rigorously, the five sample sequences were sub sampled by $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and $\frac{1}{16}$ of the original size. In addition, as the Lucas-Kanade algorithm is a sparse optical flow algorithm, in order to make the comparison more rigorous when selecting points to track a set of all the pixels in the image was passed to the algorithm. This effectively makes the Lucas-Kanade algorithm a dense optical flow algorithm.

In order to measure the accuracy of the optical flow algorithm, as stated previously, it is required to know the optical flow that each of the transformations cause in the sample sequences. Therefore it was necessary to calculate the velocity field of each of the transformations as they were applied to the `checkerboard` pattern. The MATLAB code written to calculate the velocity field for an arbitrary image which has an arbitrary transformation matrix applied to it can be seen in full in code listing A.5 in Appendix A, however the key parts of the algorithm can be seen below, in code listing 3.2.

```
% Set up matrix of points r0.
r0 = ones(n*m, 3);
n_ = -n/2;
m_ = -m/2;
for i = 1:n
  r0(i:m:(n*m),2) = n_;
  n_ = n_ + 1;
end
for j = 1:m
  r0((n*(j-1) + 1):n*j,1) = m_;
  m_ = m_ + 1;
end

% Calculate new positions for all points r0 after transformation under
%   tform, which are r1.
r1 = r0*tform.T;
r1(:,1) = r1(:,1)./r1(:,3);
r1(:,2) = r1(:,2)./r1(:,3);

% Eliminate third column (now unneeded)
r0_ = r0(:,1:2);
r1_ = r1(:,1:2);

% Calculate stacked matrix of velocities, v.
v = r1_ - r0_;

% Reshape v into V
V = reshape(v, [n m 2]);
r0 = reshape(r0_, [n m 2]);
r1 = reshape(r1_, [n m 2]);
```

Listing 3.2 MATLAB code to calculate the velocity field of a transformation matrix

It should be noted that while we are only investigating individual affine transformations, the code listed in code listing 3.2 and the full code in code listing A.5 in Appendix A also work for arbitrary affine transformation matrices, in addition to projective transformation matrices.

Figure 3.2 shows the output from the velocity field calculation plotted using the MATLAB `quiver` function. While at this scale is is impossible to discern the magnitude, or the

direction to any degree of accuracy, it can be seen that the magnitude of the velocity grows in proportion to the absolute value of the y-coordinate.

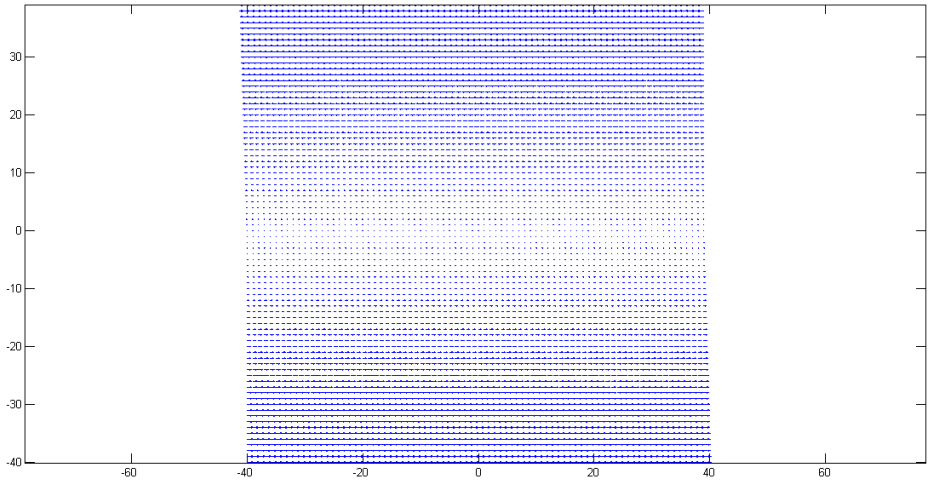
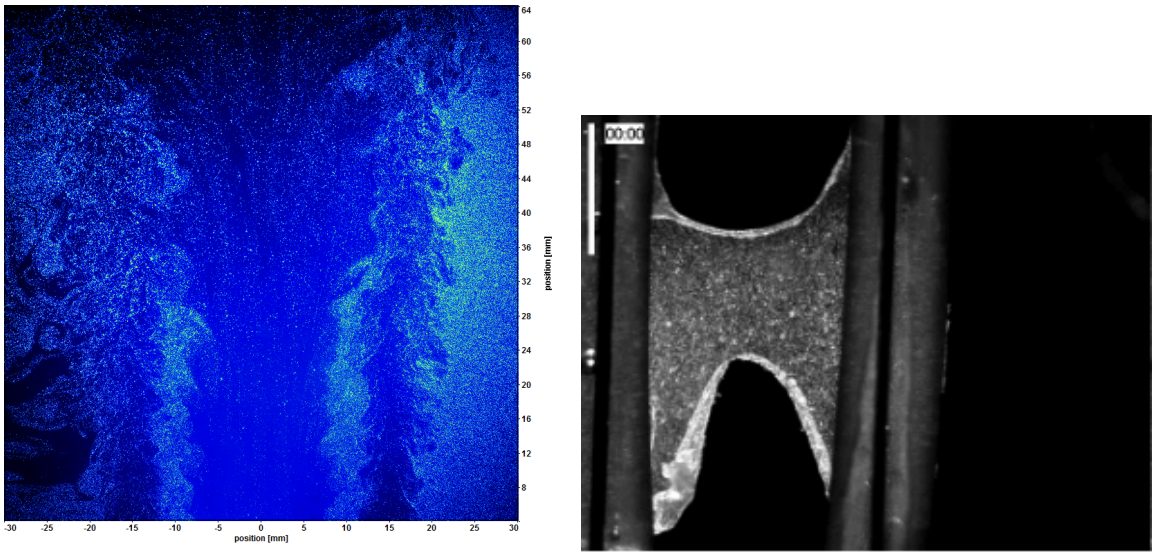


Fig. 3.2 Velocity field calculate for a shear

3.2.2 Real World Data

In order to examine the algorithms in a real world scenario I was graciously provided with data sets from Dr Alexandre Kabla [9], a lecturer and researcher in CUED, and Mustafa Kamal, A PhD student in the Hopkinson Lab. Figure 3.3a shows flow-flame interactions and comes from a high speed PIV dataset. Figure 3.3b shows a cultured cell monolayer undergoing tensile testing.

As there is no accurate, prior knowledge of the vector flow field, these datasets can only be used to compare the three optical flow algorithms for speed quantitatively, and for accuracy qualitatively. Also, for testing of real world data the Lucas-Kanade algorithm was used as a true sparse optical flow algorithm as opposed to its use in testing the artificial data where it was forced to track all pixels. It selected which pixels to track by calculating the minimal eigenvalue of gradient matrices for corner detection.



(a) Provided by Mustafa Kamal (b) Provided by Dr. Alexandre Kabla

Fig. 3.3 Real world data sets used

4.1 Comparison

4.1.1 Artificial Data

Plotted in figures 4.1 to 4.4, and listed in tables C.1 to 4.1 of Appendix C, are the timing and error (per pixel) values measured and calculated, respectively, averaged over three runs of each optical flow algorithm, so as to eliminate non-deterministic effects, for each of the affine transformations. The video sequences used varied from 480x480 pixels for a subsampling of 1, to 30x30 pixels for a subsampling of 16. The results shown in tables C.1 to 4.1 in Appendix C are rounded to two decimal places, however the plots found in figures 4.1 to 4.4 show the true values. The author was unable to construct the space dependent flow field required to calculate the accuracy of the fracture transformation, therefore only the timing information is included in table 4.1. However, it should be noted that the fracture transformation is effectively a translation.

Firstly taking the results for the rotation transformation, plotted in figure 4.1 and listed in table C.1 of Appendix C, it is plain to see that the error per pixel, and the processing time increases as the subsampling decreases, and conversely as the image size increases. For the higher subsampling in the rotation transformation the SimpleFlow algorithm has a lower error per pixel, however this is at the cost of a longer processing time per frame of video. As the sample sequences grow larger, the Farnebäck algorithm starts to show an advantage in the error per pixel. Consistently throughout the different frame sizes, the Lucas-Kanade method is the fastest.

Looking at the results for the scaling transformation, plotted in figure 4.2 and listed in table C.2 of Appendix C, they are similar to those of the rotation transformation. That is to say, that the SimpleFlow algorithm has the longest processing time, while the Lucas-Kanade algorithm has the shortest processing time, with the Farnebäck algorithm lying somewhere

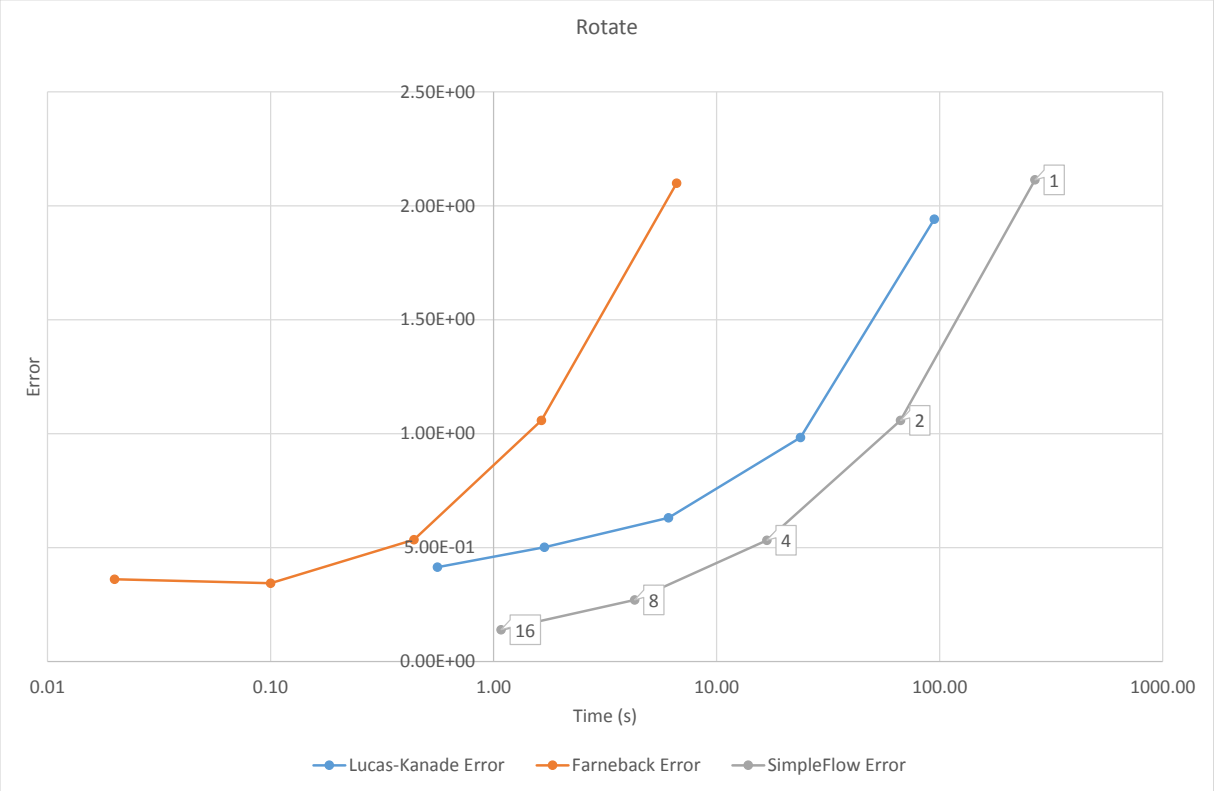


Fig. 4.1 Rotate: Accuracy vs Speed

in between. The main difference between scaling and rotation is that the Lucas-Kanade algorithm consistently wins out against the other two on accuracy.

Looking at the results for the shear transformation, plotted in figure 4.3 and listed in table C.3 of Appendix C, they present a different view to either of the two transformations previously mentioned. All three algorithms have approximately uniform error across all subsampling amounts, however the processing time increases as the subsampling decreases, or conversely, as the frame size increases. SimpleFlow presents the lowest error per pixel of all three optical flow algorithms, however on larger images it has the longest processing time per frame. The other two algorithms present higher errors per pixel, however the Farnebäck algorithm appears to be very efficient at processing shear transformations compared to the other two algorithms.

Looking at the results for the translate transformation, plotted in figure 4.4 and listed in

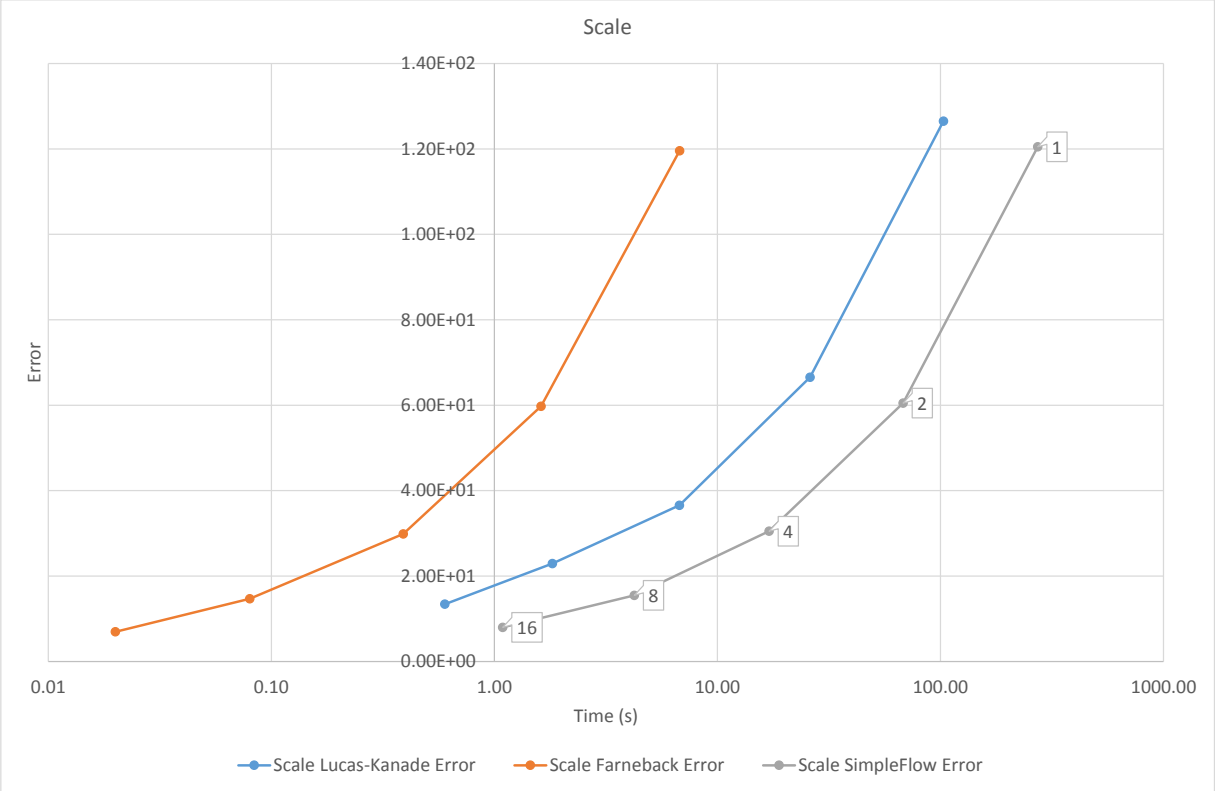


Fig. 4.2 Scale: Accuracy vs Speed

table C.4 and of Appendix C, all three optical flow algorithms appear to produce approximately the same error per pixel across all amounts of subsampling. The slight exception is the Farnebäck algorithm, which slightly outperforms the other two. The SimpleFlow algorithm is once again shown to be the slowest algorithm, while the Lucas-Kanade algorithm is shown to be the quickest.

The author was unable to construct the space dependent flow field required to calculate the accuracy of the fracture transformation, therefore only the timing information is included in table 4.1 for posterity.

Looking at the results as a whole, it is clear when choosing an optical flow algorithm to be used on a dataset there are trade-offs that need to be made. When looking at an online, realtime application, the Farnebäck optical flow algorithm should be the forerunner due to the consistent low processing time across all different levels of subsampling, as well as all

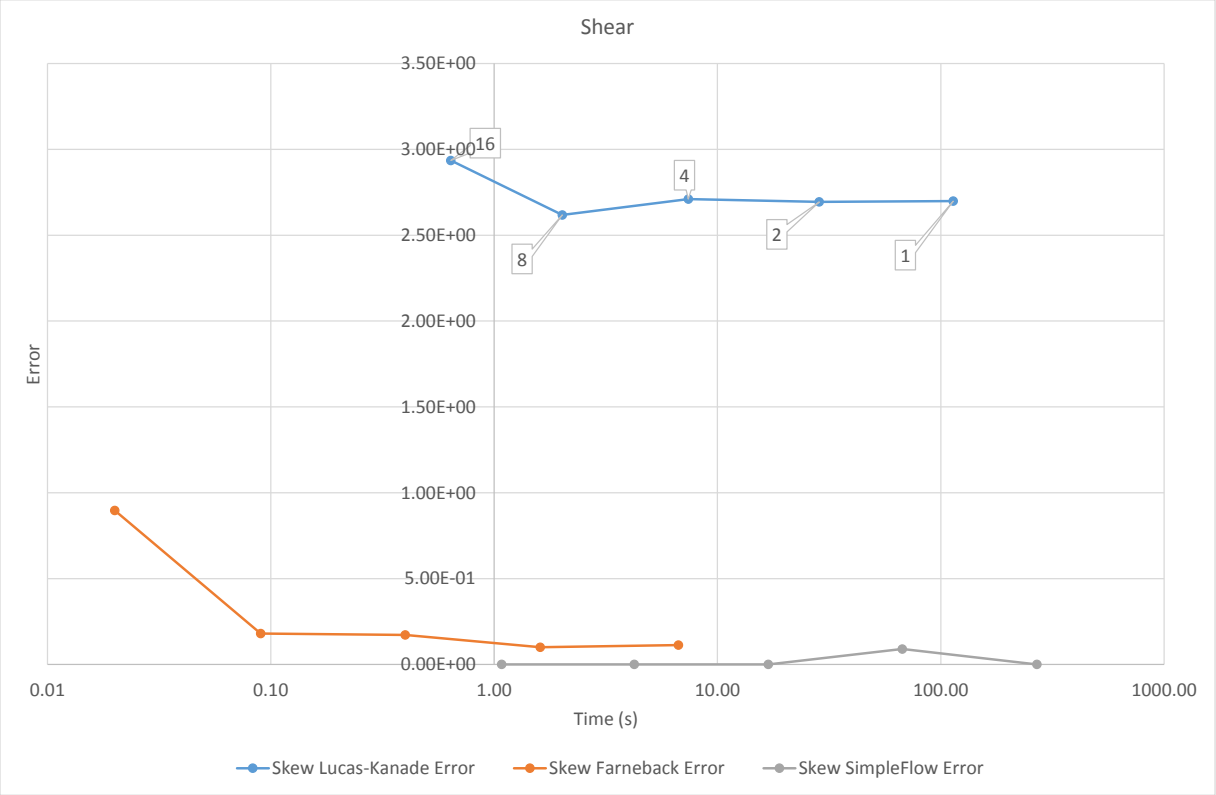


Fig. 4.3 Shear: Accuracy vs Speed

different affine transformations. The disadvantage to this choice is the lower accuracy obtained. If the application requires high accuracy, and is an offline processing application, then the SimpleFlow optical flow algorithm provides high accuracy, but this comes at the cost of a long processing time. As noted before, the SimpleFlow algorithm is highly parallelisable, and is designed for GPU acceleration. On future versions of the Raspberry Pi, or on workstation PCs, which may have these features, the SimpleFlow algorithm should perform better than on a single core, low clock CPU. If the application requires online, but only near-realtime processing, then the Lucas-Kanade optical flow algorithm is a strong candidate. Its processing time is consistently between that of the Farnebäck and SimpleFlow algorithms, and while is does provide good accuracy, its accuracy is not consistent across all affine transformations. As stated in section 2, Theory and design of experiment, optical flow methods introduce an additional constrain to enable estimation of the optical flow, and

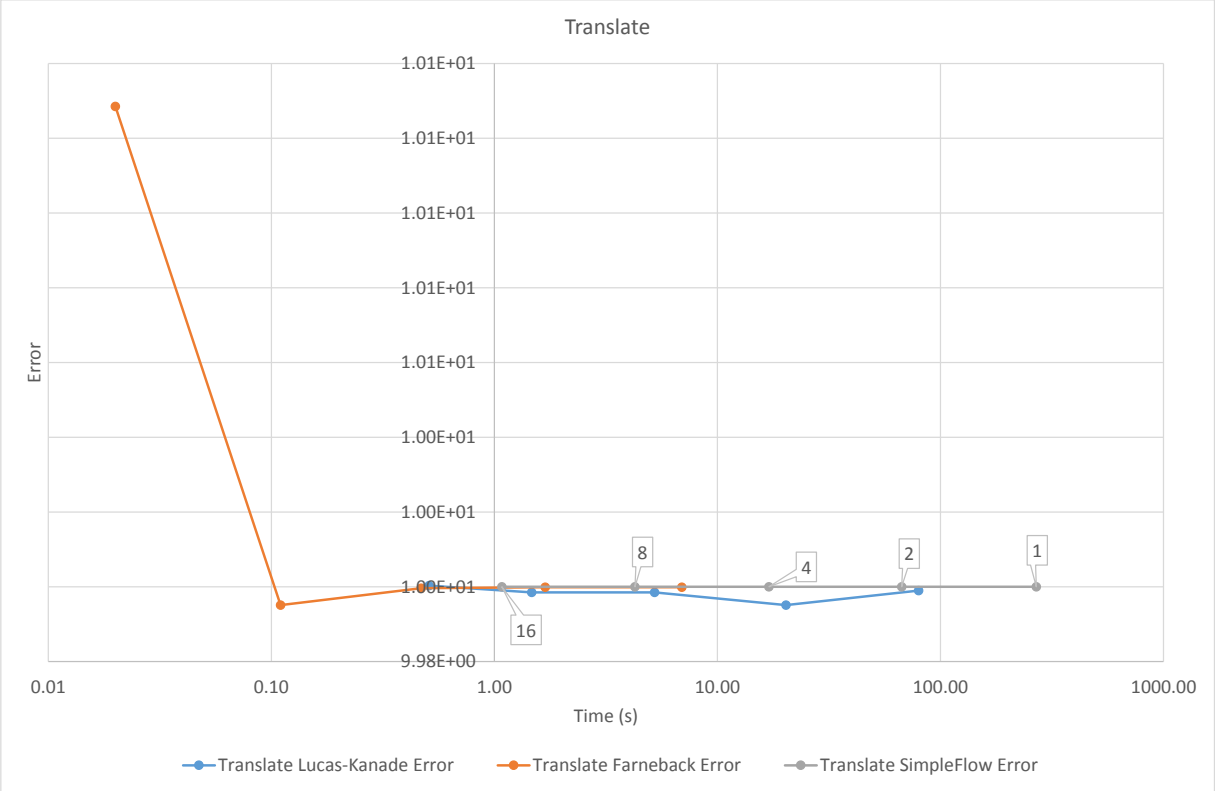


Fig. 4.4 Translate: Accuracy vs Speed

the assumptions or approximations used in each optical flow algorithm may have an effect on how accurate it is with different affine transformations.

4.1.2 Real World Data

As there is no accurate, prior knowledge of the vector flow field, these datasets can only be used to compare the three optical flow algorithms for speed quantitatively, and for accuracy qualitatively. Table 4.2 shows the processing time for the two data sets provided by Dr Alexandre Kabla [9], a lecturer and researcher in CUED, and Mustafa Kamal, A PhD student in the Hopkinson Lab.

It is apparent that the Lucas-Kanade optical flow algorithm is the fastest, this is most likely due to it being a sparse method. Following closely is the Farnebäck algorithm, and

	Lucas-Kanade		Farnebäck		SimpleFlow	
Subsampling	Time (s)	Error	Time (s)	Error	Time (s)	Error
1	74.60		6.37		267.70	
2	18.73		1.61		67.18	
4	4.90		0.44		16.92	
8	1.39		0.10		4.31	
16	0.49		0.02		1.09	

Table 4.1 Fracture

	Processing Time (s)		
Frame Size (MP)	Lucas-Kanade	Farnebäck	SimpleFlow
0.3	1.39	3.45	418
0.7	2.40	6.65	718

Table 4.2 Real Word Data

lastly the SimpleFlow algorithm. The SimpleFlow algorithm's poor speed performance is most likely due to it being designed to be a highly parallelisable, GPU accelerated algorithm. As the Raspberry Pi has a single core ARM CPU, and its GPU is not able to utilised for CUDA or OpenCL accelerated processing, it should not be surprising that the SimpleFlow algorithm performs poorly on the Raspberry Pi. This shows that when the Lucas-Kanade algorithm is used as a true sparse optical flow algorithm, it has the ability to outperform the Farnebäck and SimpleFlow algorithms, as opposed to being forced to be a dense optical flow algorithm, in the previous section, when it operates more slowly than the Farnebäck algorithm.

In order to aid the visualisation of the vector field of the real world data, and therefore assess the qualitative accuracy of the optical flow algorithms, the vectors calculated from the optical flow methods are drawn on top of each frame of the video. The method used is similar to the MATLAB function `quiver` in that it automatically scales the arrows depending on the magnitude of the vector. An example of the output can be seen in figure 4.5.

The vector field can also be represented using a colour map. This is achieved by converting the Cartesian vector coordinates into polar coordinates which are then mapped onto the HSV colour space as: Hue = Angle, Saturation = 255, Value = Magnitude. This can be seen in figure 4.6 which depicts the colour mapping of the scaling operation from the artificial dataset. This visualisation method is very useful to gain a quick insight into the apparent motion of the object being observed, without having to further process the vector field from

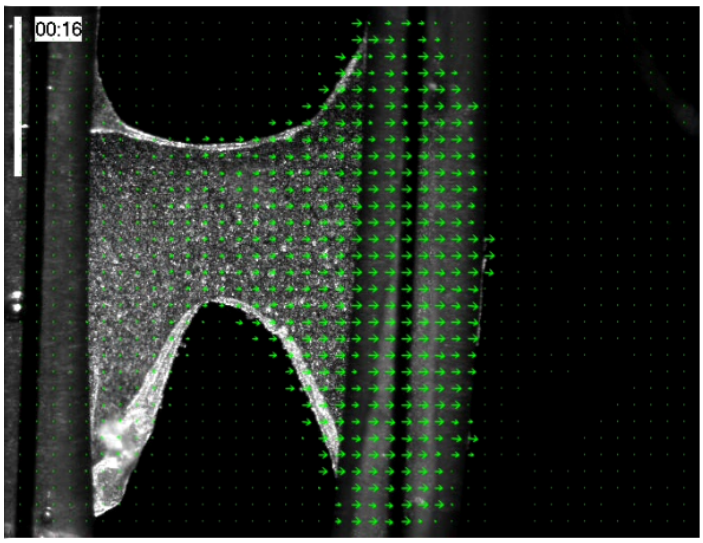
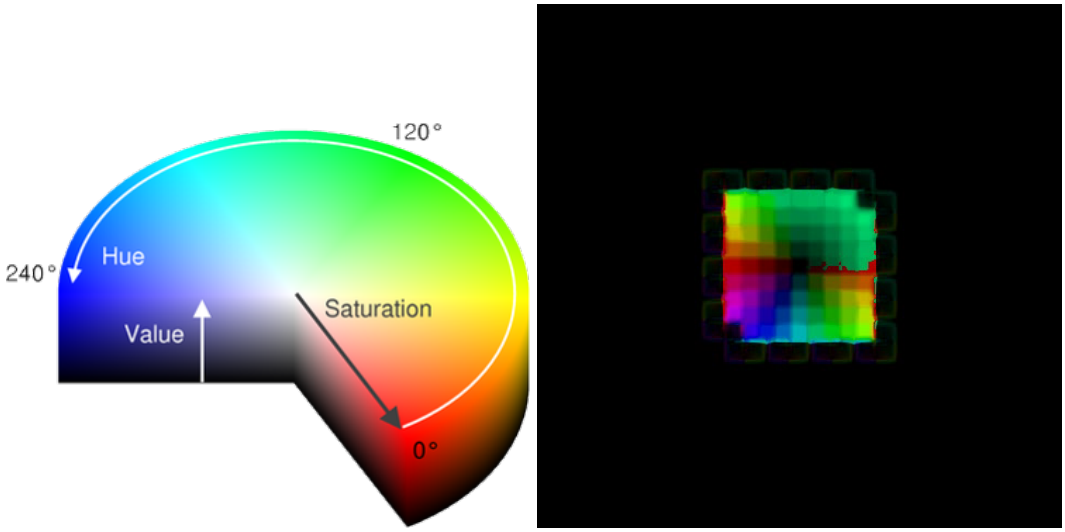


Fig. 4.5 Vector flow depicted on top of the video frame



(a) Mapping of vectors to HSV colour space (b) Colour mapping of a scaling

Fig. 4.6 Mapping the vector field to the HSV colour space

the optical flow algorithm. It should be noted that while this colour mapping is possible for all optical flow algorithms, it is most effective for dense optical flow algorithms. Therefore the Lucas-Kanade algorithm will be required to track all points to achieve a similar output to that of the Farnebäck and SimpleFlow algorithms.

4.2 Mechanical Testing

One of the other goals of this project was the integrate with other projects in the OpenLabTools initiative. One such project was a mechanical testing rig designed and built by Josie Hughes. It allows for tensile testing of materials, and is based around the Arduino, an open-source electronics prototyping platform. Therefore two programs were authored in order to demonstrate online and offline processing. The online processing program makes use of the Lucas-Kanade algorithm due to the ability to select points to be tracked, however it also features an automatic point selector which calculates the minimal eigenvalue of gradient matrices for corner detection. The help text for the online processing program is as follows

```
Hot keys:
ESC - quit the program
r - auto-initialize tracking
c - delete all the points
n - switch the "night" mode on/off
To add/remove a feature point click it
```

Night mode only shows the vector arrows of the flow field over a black background, in an effort to make the optical flow of the points being tracked easier to view. The source code for the online processing program can be seen in full online [12]

The offline processing program uses the Farnebäck algorithm, and while it is a dense optical flow algorithm, processing time is not as important as we are processing offline. Images for the offline method can be obtained by using the `raspistill` command line program [3]. The help text for the offline processing program is as follows:

```
Usage: ./offline frame1 frame2 output...
```

The source code for the online processing program can be seen in full online [12]

In order to test the functionality of the programs, a tensile test of a rubber band was setup. A frame from the testing sequence with the flow field overlaid can be seen in figure 4.7. This image shows that there is some optical flow occurring in the sequence, however the algorithm, in this example Farnebäck optical flow as this was processed offline, has mostly focused on the motion of the rig holding the rubber band in place. This is most likely due to the high contrast between the rig, and the background. Better framing of the rubber band in the sequence, and a high contrast background behind the mechanical testing rig should eliminate this problem.

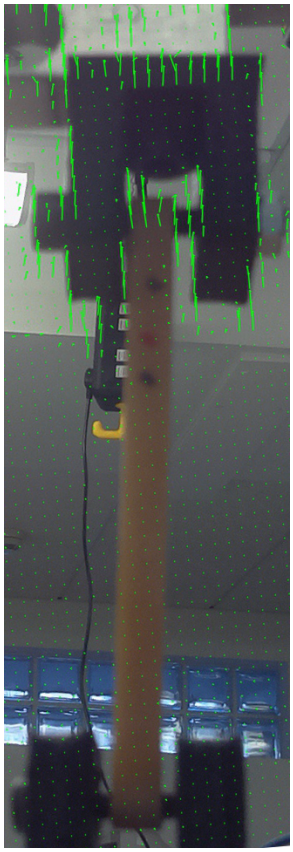


Fig. 4.7 Tensile test of rubber band

CONCLUSIONS

It has been demonstrated that executing computer vision methods, specifically optical flow algorithms, on the low power and low cost Raspberry Pi system, it is indeed possible to obtain not only high accuracy, but also high speed results within certain requirements. Performing a comparison between Lucas-Kanade, Farnebäck, and SimpleFlow optical flow algorithms for an artificial dataset comprised of individual affine transformations, and forcing Lucas-Kanade optical flow to be a dense optical flow algorithm, there are obvious choices depending on the application. For applications requiring realtime results Farnebäck optical flow appears to be the best choice. For applications which place accuracy above all else, SimpleFlow optical flow shows the best results. Finally, for applications requiring a compromise between accuracy and processing speed, Lucas-Kanade optical flow is a possible choice.

For real world data the results show a slightly different view, mainly due to Lucas-Kanade optical flow being used as a sparse optical flow method instead of being forced to be a dense method. This is another decision that needs to be made when choosing an optical flow method for an application: whether or not a dense flow field is required, or merely flow vectors for key points in the field. Nevertheless, as a true sparse optical flow method, Lucas-Kanade optical flow performs best in terms of processing time per frame, followed by Farnebäck optical flow then SimpleFlow optical flow.

Another aim of this project was to provide a clear set of instructions aimed at undergraduate level. This was achieved early on in the project as many of the tools, such as OpenCV, were used heavily in this project. The instructions can be seen in code listing A.1 in Appendix A and will soon be found on the OpenLabTools website. In addition, to show the practical applications of optical flow methods in other areas of engineering, a further aim of the project was to integrate with other projects in the OpenLabTools. One such project was a mechanical testing rig designed and built by Josie Hughes, which allows for tensile testing of materials. Using optical flow methods the strain can be measured, and there is recent

research which allows the Poisson ratio to be calculated from flow field data from optical flow methods [7]. While this project does not explore these avenues, there is certainly scope for doing so using the robust tool set used in this project.

There is scope for extension in several areas of this project. Within the comparison only individual affine transformations were explored, whereas combinations of affine transformations are not. Single affine transformations are less likely to occur in real world data, than arbitrary affine transformations, therefore it may be worth investigating. The tools used to analyse the artificial dataset was written to support, not only single affine transformations, or even arbitrary affine transformations, but projective transformations too. Therefore it should be relatively simple to pursue this area of research. Better visualisation of the vector field is also another possible extension. Currently the vector field can be visualised using vector arrows, or as a colour mapping to HSV colour space, however showing the curl or the divergence of the vector field may also be a useful tool. If the vector field represents the flow velocity of a moving fluid, then the curl is the circulation density of the fluid and the divergence measures the magnitude of a vector field's source or sink at a given point.

REFERENCES

[1] Bcm2835 media processor; broadcom. URL http://www.broadcom.com/products/BCM2835.

[2] OpenCV | OpenCV. URL http://opencv.org/.

[3] Raspistill. URL http://www.raspberrypi.org/documentation/usage/camera/raspicam/raspistill.md.

[4] Simplecv. URL http://simple.org/.

[5] Linux projects - documentation. URL http://www.linux-projects.org/modules/sections/index.php?op=viewarticle&artid=14.

[6] M Bertrand, J Meunier, M Doucet, and G Ferland. Ultrasonic biomechanical strain gauge based on speckle tracking. In *Ultrasonics Symposium, 1989. Proceedings., IEEE 1989*, pages 859–863. IEEE, 1989.

[7] Feifei Chen, Dong-Joong Kang, and Jun-Hyub Park. New measurement method of poisson's ratio of pva hydrogels using an optical flow analysis for a digital imaging system. *Measurement Science and Technology*, 24(5):055602, 2013.

[8] Gunnar Farnebäck. Two-frame motion estimation based on polynomial expansion. In *Proceedings of the 13th Scandinavian Conference on Image Analysis*, LNCS 2749, pages 363–370, Gothenburg, Sweden, June-July 2003.

[9] Andrew R Harris, Loic Peter, Julien Bellis, Buzz Baum, Alexandre J Kabla, and Guillaume T Charras. Characterizing the mechanics of cultured cell monolayers. *Proceedings of the National Academy of Sciences*, 109(41):16449–16454, 2012.

[10] Stephen J Huston and Holger G Krapp. Visuomotor transformation in the fly gaze stabilization system. *PLoS biology*, 6(7):e173, 2008.

[11] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *Proceedings of the 7th International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'81, pages 674–679, San Francisco, CA, USA, 1981. Morgan Kaufmann Publishers Inc.

[12] Jamie Magee. Github - jamiemagee/iib-project, 2013. URL https://github.com/JamieMagee/IIB-Project.

[13] Georges M Quénot, Jaroslaw Pakleza, and Tomasz A Kowalewski. Particle image velocimetry with optical flow. *Experiments in fluids*, 25(3):177–189, 1998.

[14] Y Rosenberg and M Werman. Representing local motion as a probability distribution matrix applied to object tracking. In *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, pages 654–659, 1997.

[15] Michael W. Tao, Jiamin Bai, Pushmeet Kohli, and Sylvain Paris. Simpleflow: A non-iterative, sublinear optical flow algorithm. *Computer Graphics Forum (Eurographics 2012)*, 31(2), May 2012. URL http://graphics.berkeley.edu/papers/Tao-SAN-2012-05/.

[16] Liz Upton. Two million!, 2013. URL http://www.raspberrypi.org/two-million/.

CODE LISTINGS

```
# OpenCV

This will guide you through installing OpenCV/SimpleCV on the Raspberry Pi

## 1. Setup Camera Hardware

http://www.raspberrypi.org/camera

To ensure that you have the latest firmware update the raspberry pi by running

```
sudo rpi-update
```

and reboot afterwards.

2. Install UV4L Driver

First download and import the PGP file to verify the integrity of the packages we're going to install

```
wget http://www.linux-projects.org/listing/uv4l_repo/lrkey.asc && sudo apt-key add ./lrkey.asc
```

We then need to add the repository to `/etc/apt/sources/list`

```
sudo sh -c "echo 'deb http://www.linux-projects.org/listing/uv4l_repo/raspbian/ wheezy main' >> /etc/apt/sources.list"
```

Then we can install the Userland Video 4 Linux Driver for the Raspberry Pi

```

```
```
sudo apt-get update
sudo apt-get install uv4l uv4l-raspicam
```

To test that the driver was installed correctly we can run an instance of it

```
uv4l --driver raspicam --auto-video_nr --width 640 --height 480 --encoding jpeg --nopreview
```

and save a image to the current directory using

```
dd if=/dev/video0 of=snapshot.jpeg bs=11M count=1
```

To terminate a running driver, close the applications and kill the corresponding uv4l process:

```
pkill uv4l
```

3. Install Libraries
3a. Install OpenCV

Building OpenCV from source takes 8-10 hours on the Raspberry Pi (Depending on overclock). It is strongly encouraged that you run this in terminal only mode as "Out of Memory Exceptions" can occur otherwise.

```
sudo apt-get remove ffmpeg x264 libx264-dev
sudo apt-get install libopencv-dev build-essential checkinstall cmake pkg-config yasm libtiff4-dev libjpeg-dev libjasper-dev libavcodec-dev libavformat-dev libswscale-dev libdc1394-22-dev libxine-dev libgstreamer0.10-dev libgstreamer-plugins-base0.10-dev libv4l-dev python-dev python-numpy libqt4-dev libgtk2.0-dev libmp3lame-dev libopencore-amrnb-dev libopencore-amrwb-dev libtheora-dev libvorbis-dev libxvidcore-dev x264 v4l-utils ffmpeg
mkdir OpenCV
cd OpenCV
wget https://github.com/Itseez/opencv/archive/2.4.6.2r3.zip
unzip 2.4.6.2r3.zip
rm 2.4.6.2r3.zip
cd opencv-2.4.6.2r3
mkdir build
cd build
```
```

```
cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local -D WITH_TBB=ON -D BUILD_NEW_PYTHON_SUPPORT=ON -D WITH_V4L=ON -D INSTALL_C_EXAMPLES=ON -D INSTALL_PYTHON_EXAMPLES=ON -D BUILD_EXAMPLES=ON -D WITH_QT=ON -D WITH_OPENGL=ON ..
make
sudo make install
sudo sh -c 'echo "/usr/local/lib" > /etc/ld.so.conf.d/opencv.conf'
sudo ldconfig
```

Once OpenCV has installed we can build the sample applications that come with OpenCV

```
cd ~/OpenCV/opencv-2.4.6.2r3/samples/c
./build_all.sh
```

The UV4L driver must be running, and we can run it at a higher priority to ensure better performance

```
uv4l --driver raspicam --auto-video_nr --encoding yuv420 --width 320 --height 240 --nopreview
sudo chrt -a -r -p 99 `pgrep uv4l`
```

Try out some of the sample applications and see what happens!

3b. Install SimpleCV

First install the necessary dependency packages required by SimpleCV

```
sudo apt-get install ipython python-opencv python-scipy python-numpy python-pygame python-setuptools python-pip
```

Next download and install SimpleCV itself

```
sudo pip install https://github.com/sightmachine/SimpleCV/archive/1.3.zip
```

To test that SimpleCV has been installed correctly open a text editor and paste the following

```
from SimpleCV import Camera
# Initialize the camera
cam = Camera()
```

```
# Loop to continuously get images
while True:
    # Get Image from camera
    img = cam.getImage()
    # Make image black and white
    img = img.binarize()
    # Draw the text "Hello World" on image
    img.drawText("Hello World!")
    # Show the image
    img.show()
```

The UV4L driver must be running, and we can run it at a higher priority to ensure better performance

```
uv4l --driver raspicam --auto-video_nr --encoding yuv420 --width 320 --height 240 --no-preview
sudo chrt -a -r -p 99 `pgrep uv4l`
```

Save the files as helloworld.py and run it from the terminal

```
python helloworld.py
```

If SimpleCV has installed correctly and an instance of the UV4L driver is running a binary image from the camera should be displayed with the text "Hello World" superimposed over it

Sources

1. http://linux-projects.org
2. http://raspberrypi.org
3. http://simplecv.org
4. http://opencv.org

Listing A.1 How to setup OpenCV on the Raspberry Pi

```
import glob
import re

for fname in glob.glob('*.txt'):
    with open(fname, "r") as myfile:
        f = open(fname[:-4]+'average.txt', 'w')
        average, frames, count = 0, 0, 0
        for line in myfile.readlines():
            if re.findall('\d+\.\d{2}', line):
                if count != 0:
                    average += float(''.join(re.findall('\d+\.\d{2}', line)))
                    frames += 1
```

```
                count += 1
            else:
                if line is not '\n' and not re.findall('0\.\d{2}', line)
    :
                    if re.findall('Output', line):
                        f.write(line)
                    if average is not 0:
                        f.write("{0:.2f}".format(average/frames)+'s
    average per frame\n\n')
                    average, frames, count = 0, 0, 0
        f.close()
```

Listing A.2 Calculate average processing time per frame

```
import re
import os
import fnmatch
from operator import sub
from datetime import datetime
from math import isnan

start = datetime.now()

matches = []
for root, dirnames, filenames in os.walk('map'):
    for filename in fnmatch.filter(filenames, '*.yml'):
        matches.append(os.path.join(root, filename))

for fname in matches:
    print('Processing '+fname)
    with open(fname, "r") as myfile:
        data, frames, nextpts, prevpts = [], [], [], []
        data = ''.join([line.replace('\n', '').replace(" ", "") for line
    in myfile.readlines()])
        for val in re.findall(r'\[([^]]*)\]', data):
            frames.append([round(float(x), 2) if x != '.Nan' else 0 for
   x in val.split(',')])
        if "lucas-kanade" in fname:
            for i in range(0, len(frames), 2):
                nextpts.append(frames[i])
                prevpts.append(frames[i+1])
            frames = []
            for i in range(0, len(nextpts)):
                frames.append(list(map(sub, nextpts[i], prevpts[i])))
    f = open(fname[:-4]+'.txt', 'w')
    print("There are "+str(len(frames))+" frames")
    frame_num = 1 #int(input('Which frame would you like to grab? '))
    for i in range(0, len(frames[frame_num]), 2):
        f.write(str("{0:0.2f}".format((frames[frame_num])[i]))+', '+str(
    "{0:0.2f}".format((frames[frame_num])[i+1]))+'\n')
    f.write('\n')
    f.close()
    print('Finished processing '+fname)
```

```
print("Processing completed\nTime taken: "+str(datetime.now()-start)+"s"
    )
```

Listing A.3 Process vector field output from optical-flow

```
close all

orig=checkerboard>.5;
subsampling = 16;
%orig=imresize(imnoise(ones(20),'gaussian'),4);
%orig=imresize(imnoise(ones(20),'salt & pepper'),4);

imshow(padarray(orig,[200 200],255));

tform=affine2d([1 0 0; .5 1 0; 0 0 1]);
writerObj = VideoWriter('skew16.avi');
open(writerObj);
frame=uint8(padarray(orig,[200 200],255).*255);
frame=frame(1:subsampling:end,1:subsampling:end);
writeVideo(writerObj,frame);
s=imwarp(orig,tform,'FillValue',255);
for i=1:10
    dr=floor(size(s,1)-size(orig,1));
    dc=floor(size(s,2)-size(orig,2));
    frame=uint8(padarray(s,[200-dr/2 200-dc/2],255).*255);
    frame=frame(1:subsampling:end,1:subsampling:end);
    writeVideo(writerObj,frame);
    s=imwarp(s,tform,'FillValue',255);
end
close(writerObj);

theta=1;
tform=affine2d([cosd(theta) -sind(theta) 0; sind(theta) cosd(theta) 0; 0
    0 1]);
writerObj = VideoWriter('rotate16.avi');
open(writerObj);
frame=uint8(padarray(orig,[200 200],255).*255);
frame=frame(1:subsampling:end,1:subsampling:end);
writeVideo(writerObj,frame);
r=imwarp(orig,tform,'FillValue',255);
for i=1:10
    dr=size(r,1)-size(orig,1);
    dc=size(r,2)-size(orig,2);
    frame=uint8(padarray(r,[200-dr/2 200-dc/2],255).*255);
    frame=frame(1:subsampling:end,1:subsampling:end);
    writeVideo(writerObj,frame);
    r=imwarp(r,tform,'FillValue',255);
end
close(writerObj);

writerObj = VideoWriter('translate16.avi');
open(writerObj);
frame=uint8(padarray(orig,[200 200],255).*255);
```

```
frame=frame(1:subsampling:end,1:subsampling:end);
writeVideo(writerObj,frame);
for i=1:10
    t=padarray(orig,[200-5*i 200-5*i],255,'pre');
    t=padarray(t,[200+5*i 200+5*i],255,'post');
    frame=uint8(t.*255);
    frame=frame(1:subsampling:end,1:subsampling:end);
    writeVideo(writerObj,frame);
end
close(writerObj);

writerObj = VideoWriter('scale16.avi');
open(writerObj);
for i=1:10
    scale=imresize(orig,(i+1)/2);
    dr=size(scale,1)-size(orig,1);
    dc=size(scale,2)-size(orig,2);
    scale=padarray(scale,[200-dr/2 200-dc/2],255);
    frame=uint8(scale.*255);
    frame=frame(1:subsampling:end,1:subsampling:end);
    writeVideo(writerObj,frame);
end
close(writerObj);

writerObj = VideoWriter('fracture16.avi');
open(writerObj);
frame=uint8(padarray(orig,[200 200],255).*255);
frame=frame(1:subsampling:end,1:subsampling:end);
writeVideo(writerObj,frame);
for i=1:11
    frac=cat(2,padarray(orig(:,1:size(orig,2)/2),[2*i 0],255,'pre'),
    padarray(orig(:,((size(orig,2)/2)+1):end),[2*i 0],255,'post'));
    dr=size(frac,1)-size(orig,1);
    dc=size(frac,2)-size(orig,2);
    frac=padarray(frac,[200-dr/2 200-dc/2],255);
    frame=uint8(frac.*255);
    frame=frame(1:subsampling:end,1:subsampling:end);
    writeVideo(writerObj,frame);
end
close(writerObj);

close all
```

Listing A.4 MATLAB code used to generate artifical sample sequences

```
function [V, r0, r1] = velfield(X, tform)
%VELFIELD Function that takes an image X and a transform object tform
    and
% returns a velocity field V, where each entry is the direction that
    each
% pixel of X is moved in when X undergoes the transform tform.
%
%   X is an n x m image array
```

```
%   tform is a transform object (affine2d or projective2d)
%
%   V is an n x m x 2 array, where V(n,m,1) is the x component that the
%       [n,m] pixel in X is moved by tform and V(n,m,2) is the y component
%       of that movement

% Define size of X.
[n,m] = size(X);

% Set up matrix of points r0.
r0 = ones(n*m, 3);
n_ = -n/2;
m_ = -m/2;
for i = 1:n
    r0(i:m:(n*m),2) = n_;
    n_ = n_ + 1;
end
for j = 1:m
    r0((n*(j-1) + 1):n*j,1) = m_;
    m_ = m_ + 1;
end

% Calculate new positions for all points r0 after transformation under
%   tform, which are r1.
r1 = r0*tform.T;
r1(:,1) = r1(:,1)./r1(:,3);
r1(:,2) = r1(:,2)./r1(:,3);

% Eliminate third column (now unneeded)
r0_ = r0(:,1:2);
r1_ = r1(:,1:2);

% Calculate stacked matrix of velocities, v.
v = r1_ - r0_;

% Reshape v into V
V = reshape(v, [n m 2]);
r0 = reshape(r0_, [n m 2]);
r1 = reshape(r1_, [n m 2]);

% Plotting code
% str = [];
% for i = 16:16:n
%     for j = 16:16:m
%         str = ([str, '[',int2str(j),',',int2str(j),' + V(',int2str(i),',',int2str(j),',1)],'...
%             '[',int2str(i),',',int2str(i),' + V(',int2str(i),',',int2str(j),',2)], ']);
%     end
% end
% figure; eval(['plot(',str,' ''k'');']);
```

```
56 % quiver(r0(:,:,1),flipud(r0(:,:,2)),V(:,:,1),V(:,:,2))
57 % axis('equal');
58 end
```

Listing A.5 Calculate velocity field of a transfromation applied to an image

RISK ASSESSMENT RETROSPECTIVE

The risk assessment submitted at the outset of this project identified only hazard - relating to extended use of a computer. As this project is mostly a computational project, this was the most frequently occurring hazard through the duration of the project. Most of the experiments used artificial, computer generated datasets, and the real world datasets that were used were collected by other individuals, therefore there was no hazard in collecting the datasets. The only other possible risk is a very small electrical and mechanical risk involved in the mechanical testing section.

RESULTS

	Lucas-Kanade		Farnebäck		SimpleFlow	
Subsampling	Time (s)	Error	Time (s)	Error	Time (s)	Error
1	94.67	1.94E+00	6.62	2.10E+00	267.59	2.11E+00
2	23.80	9.83E-01	1.64	1.06E+00	66.83	1.06E+00
4	6.09	6.31E-01	0.44	5.35E-01	16.80	5.32E-01
8	1.69	5.01E-01	0.10	3.44E-01	4.29	2.70E-01
16	0.56	4.14E-01	0.02	3.61E-01	1.08	1.40E-01

Table C.1 Rotation

	Lucas-Kanade		Farnebäck		SimpleFlow	
Subsampling	Time (s)	Error	Time (s)	Error	Time (s)	Error
1	103.12	1.27E+02	6.77	1.20E+02	273.10	1.21E+02
2	26.02	6.66E+01	1.62	5.97E+01	67.99	6.05E+01
4	6.75	3.66E+01	0.39	2.99E+01	17.05	3.05E+01
8	1.82	2.29E+01	0.08	1.47E+01	4.24	1.55E+01
16	0.60	1.34E+01	0.02	6.96E+00	1.09	8.00E+00

Table C.2 Scale

	Lucas-Kanade		Farnebäck		SimpleFlow	
Subsampling	Time (s)	Error	Time (s)	Error	Time (s)	Error
1	113.71	2.70E+00	6.69	1.12E-01	269.52	0.00E+00
2	28.57	2.69E+00	1.61	1.00E-01	67.25	9.00E-02
4	7.40	2.71E+00	0.40	1.72E-01	16.88	0.00E+00
8	2.02	2.62E+00	0.09	1.80E-01	4.24	0.00E+00
16	0.64	2.94E+00	0.02	8.97E-01	1.08	0.00E+00

Table C.3 Shear

	Lucas-Kanade		Farnebäck		SimpleFlow	
Subsampling	Time (s)	Error	Time (s)	Error	Time (s)	Error
1	79.79	1.00E+01	6.92	1.00E+01	268.98	1.00E+01
2	20.29	1.00E+01	1.69	1.00E+01	67.01	1.00E+01
4	5.23	1.00E+01	0.47	1.00E+01	17.02	1.00E+01
8	1.47	1.00E+01	0.11	1.00E+01	4.27	1.00E+01
16	0.51	1.00E+01	0.02	1.01E+01	1.08	1.00E+01

Table C.4 Translate