Method and apparatus for obtaining images and measurements of density fluctuations in transparent media

U.S. Provisional Patent Application (No. 60/170,928)

S. B. Dalziel[1] G. O. Hughes[2] and B. R. Sutherland[3]

1 Introduction

This document describes in detail the setup and use of "synthetic schlieren", an inexpensive technique that can be used to visualise density changes, for example, the fluctuations due to heat rising above a hand. The technique can visualise changes in any solid, fluid or gaseous medium that is transparent to a portion of the electromagnetic spectrum. Thus, for example, the technique can be employed using x-rays to visualise density changes in the human body, or using visible light to visualise shock waves in air. For ease of discussion, here we illustrate the synthetic schlieren technique primarily using visible light to detect and measure changes of transparent liquid or gaseous flow.

A commonly observed phenomenon is the shimmer of light due to heat rising above an asphalt road. This occurs because the index of refraction of air decreases with increasing air temperature. Thus the path of sunlight is deflected when passing through turbulent, hot air. In an extreme circumstance, this can create the illusion of an mirage: a pool of water appears to lie in the road because light appears to reflect from the road surface. In fact downward propagating light rays approaching the road at glancing angles do not reflect but refract upwards as they pass through the extremely hot air above the pavement. This phenomenon is illustrated schematically in Figure 1.

In the above example, turbulent air is easily detected because one is looking beyond it to the horizon. As a result of the way light is deflected as it passes through hot air, the position of the horizon appears to rapidly fluctuate, creating a shimmering effect. The air well above the horizon can be quite hot, but the effect of heat shimmer is not so easily seen because there is no shimmering object in the distance, such as the horizon.

Synthetic schlieren makes use of this shimmering effect to visualise density differences, *e.g.* due to heating of air, not just along a one-dimensional object (like the horizon-line), but over a two-dimensional field of view. One way that this is done is to place in the distance an object (hereafter, the "object-image") composed of a stack of equally spaced horizontal lines. Each horizontal line plays the role of an artificial horizon. If heat or other density fluctuations occur between the object-image and the viewer, each line in the object-image appears to fluctuate. Thus, the vertical as well as the horizontal structure of the region of hot air can be visualised. (Though, note, no information is provided about the structure along

[1]Department of Applied Mathematics and Theoretical Physics, University of Cambridge, Cambridge, England

[2]Research School of Earth Sciences, The Australian National University, Canberra, Australia

[3]Department of Mathematical Sciences, University of Alberta, Edmonton, AB, Canada

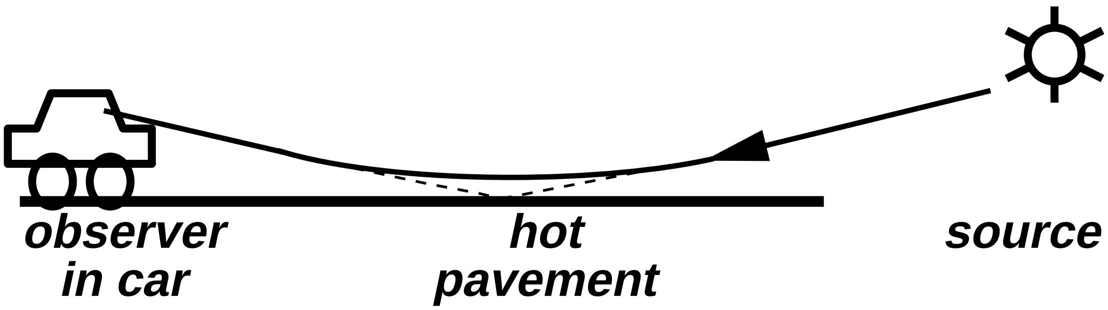

FIG. 1. Schematic showing the occurrence of a mirage. Light is bent due to the reduced index of refraction of hot air over pavement (solid line). To an observer it appears the light reflects from the road (dashed line) and results in the illusion of water in the road. Synthetic schlieren makes use of this effect to enhance images of heat shimmer and other phenomena.

the ray between the viewer and the image.) The selection of the object-image is somewhat arbitrary. To gain information about horizontal as well as vertical fluctuations, an array of randomly positioned dots, or other sharply resolved features, may be used instead of a set of lines.

In addition to visualising density fluctuations over a two dimensional area, by digitising and enhancing the object image on computer, synthetic schlieren is far more sensitive than the naked eye. Briefly, an object-image is compared with a snap shot of the object-image at an earlier time. This can be done effectively by subtracting or dividing the intensities of one image by the other, on a pixel-by-pixel basis. For ease of discussion, we assume below the digitised images are subtracted. If there is no change, subtracting the digitised images produces a uniform (*e.g.* black) schlieren image. If, for example, heat fluctuations cause the real-time object-image to differ from the initial object-image, then subtracting the digitised images will show where the differences occur, and so show the spatial extent of the fluctuations. The difference can be multiplied by an arbitrary constant to further enhance the fluctuations.

Synthetic schlieren has many advantages over other schlieren systems[1]. Traditional schlieren systems require two parabolic mirrors which limit the size of the region being examined to less than approximately 1 metre. The two parabolic mirrors, or a pair of masks in the Moiré method[2, 1], must be very accurately positioned. Mirrors and masks are prone to damage, thus prohibiting their use outside controlled laboratory conditions. In contrast, synthetic schlieren is much less expensive and easier to implement: the object-image (e.g. of lines or dots) can be as large as required to visualise the heat source and the placement of the object-image does not need to be precise. Furthermore, optical imperfections, which are problematic in classical schlieren methods, are digitally corrected in synthetic schlieren.

So far, the use of synthetic schlieren in "qualitative mode" has been outlined. This refers to the use of synthetic schlieren to observe the occurrence and extent of density fluctuations in real time. In many circumstances, synthetic schlieren can be also be used in "quantitative mode". In this mode, the strength of the fluctuations can themselves be measured.

We illustrate here the simplest use of the quantitative mode, which can be used if the density fluctuations in the region of interest have a known spanwise structure. That is, the variation in the fluctuations are assumed to be known along the line-of-sight between the observer/camera and the object-image. For example, a disturbance in a (non-turbulent) fluid caused by a moving horizontal cylinder is uniform along the direction of the cylinder axis. Thus one can assume the fluctuations are uniform over the line of sight in the direction along the cylinder axis.

More generally, standard tomographic techniques can be used to reconstruct a three-dimensional field from two or more simultaneous perspectives through region of interest.

The quantitative mode has been used to examine internal motions in a tank of water that is density stratified with varying concentrations of salt. As with many stratified fluids, the index of refraction of salt water is a function of salinity and, therefore, of density. The salinity of the water increases with depth in the tank. In a typical setup, an object-image is placed on one side of the tank and a camera on the opposite side of the tank is focussed on the object-image. Light from the object-image passes through the salt-stratified fluid and is deflected because more saline water has a larger index of refraction. In qualitative mode, synthetic schlieren can be used to visualise density perturbations within the fluid, for example due to waves that propagate within the fluid under the influence of buoyancy. The qualitative mode can be used to measure the wavelength and speed of propagation of the waves. In quantitative mode, if the waves are uniform along the span of tank (between the camera and the object-image) then the amplitude of the waves can also be determined[3].

This disclosure document describes, on a level understandable to a non-specialist, the setup and use of synthetic schlieren in both qualitative and quantitative modes. Section 2 describes how the camera and object-image should be set up to record movies that can be later played back or immediately processed by computer. The basics of image digitisation are also discussed. Section 3 describes the use of synthetic schlieren in qualitative mode. The information is supplemented with information in the appendix, which explicitly describes how this mode of operation is implemented using the image processing software package "DigImage". Section 4 describes the use of synthetic schlieren in quantitative mode, including the theory and computer algorithms used to compute density perturbations from observed fluctuations.

2 Setup of Synthetic Schlieren

The typical setup to visualise density changes using synthetic schlieren is shown in Figure 2. A camera (analogue or digital; video or still) is focussed on an object-image which, for example, may be a grid of horizontal black lines or a random pattern of black dots. (Even more generally, any image capture device that scans an area, line or even a single point will suffice.) For ease of discussion in what follows, the object-image is assumed to be composed of lines. The camera may be hooked up to a computer so that synthetic schlieren is used instantaneously. Otherwise, the object-image may be recorded on tape or stored digitally to be played back at a later time and processed using synthetic schlieren. If possible, digital

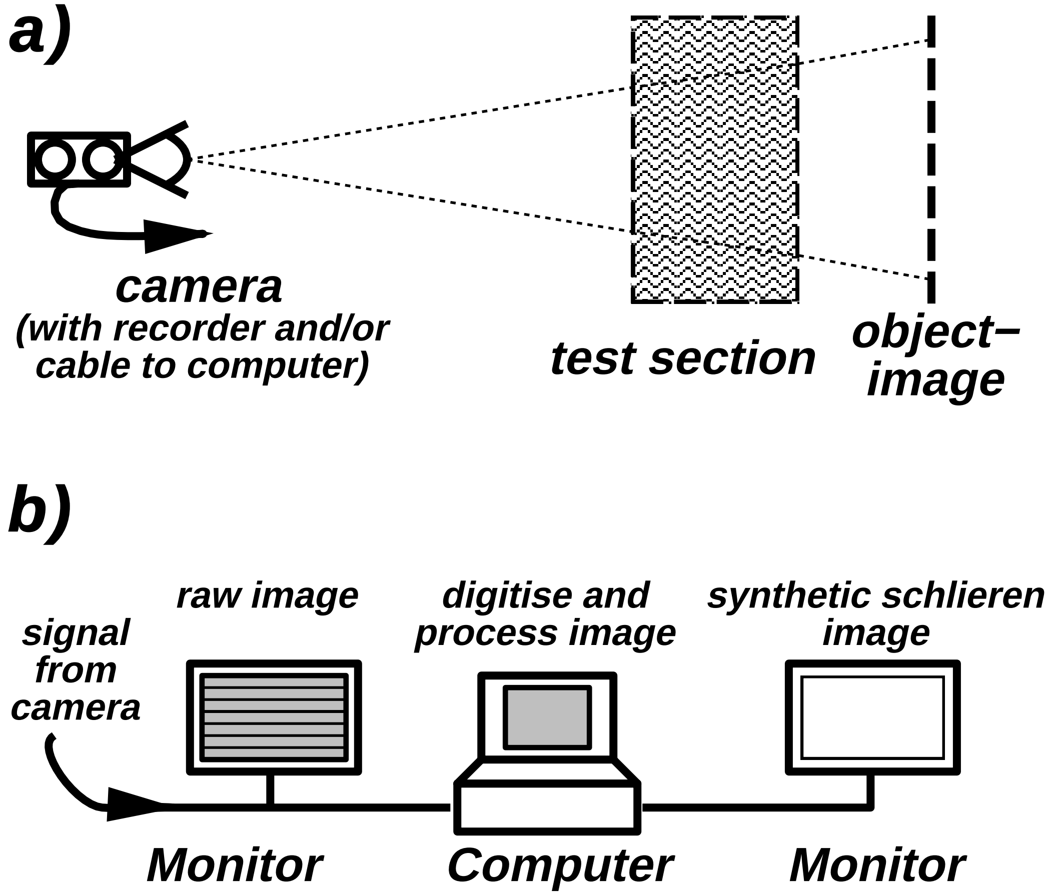

FIG. 2. Set-up of the Synthetic Schlieren system. a) A camera records an object-image (*e.g.* of a grid or horizontal black lines or an array of dots). Between the camera and the object-image is the object being studied, for example, heat rising from a surface or density changes resulting from wave-like motions in a density stratified fluid. b) The signal (either in real time or played back from recording) is processed on a computer which has a frame-grabber card installed (see text). The computer performs calculations to make a synthetic schlieren image on the computer screen or on a second monitor.

storage is preferred because this helps to reduce signal noise and prevents signal degradation over time.

The distance from the camera to the object-image is at the discretion of the user. In principle it could be very close for microscopic applications or hundreds of metres away for large industrial applications. However, the size of detail (*e.g.* the thickness of the lines) in the object-image depends on the resolution and field of view of the camera. When the object-image is digitised each line should be at least as wide as one pixel. The lines should be spaced sufficiently far apart so that the distance between any two lines spans at least five pixels. Synthetic schlieren is most sensitive and offers the highest spatial resolution if the lines are as small and as closely spaced as possible. (In an object-image of dots, each dot should occupy one or two pixels and they should be separated on average by five or six pixels in any direction).

For example, suppose the camera is focussed on a 1 square metre area of the object-image, and suppose the digitised image has a resolution of 512 by 512 pixels. Then, for optimal use of synthetic schlieren, the horizontal lines on the object-image should be spaced apart by a distance of approximately $5 \times \frac{1\,\mathrm{metre}}{512} \simeq 1\,\mathrm{cm}$. Each line should be approximately 0.2 cm thick.

The test section illustrated in Figure 2a indicates a region where the density variations of interest occur, for example, the region above a hot plate or the volume of salt stratified fluid in a glass tank. Synthetic schlieren is more sensitive to tiny density fluctuations if the test section is farther from the object-image and nearer the camera. The test section is thus best placed as close to the camera as possible so that the region of interest fills the field of view, but not so close that the effects of parallax are significant. The object-image should be placed as far away as physical constraints and cost allows. Preferably, the object-image and test section should both be within the depth of field of the camera lens.

Either in real-time during filming (if a computer setup is on-site) or when the film or digitised image is played back, a computer equipped with a frame grabber card and image processing software may be used to apply synthetic schlieren. A typical setup for analysis is shown in Figure 2b. Ideally, the signal from the camera is split so that the raw image is shown on a monitor while being processed on computer. The computer digitises the signal and performs calculations that enhance small changes in the object-image (*e.g.* due to heat shimmer) over time. The resulting enhanced image (hereafter, the "schlieren image") may be shown on a second monitor or on the computer screen itself.

The details of synthetic schlieren itself are described in the next section. So that this ensuing discussion is clear, the process of digitisation is described here. Digitisation may be done by the camera itself or the analogue signal may be converted to a digital signal by a "frame grabber card" on a computer, or by a high-resolution scanner. For example, this can be done using one of Data Translation's frame grabber cards DT2861 or DT2862 (see appendix C), which can be inserted in a ISA slot in a PC.

Figure 3 schematically illustrates the process of digitisation. In the example, it is supposed that the camera records a greyscale picture of an object-image of horizontal black lines. (The object-image can be colour, but the computer makes practical use only of the intensity information). The digitised image subdivides the object-image into a regular grid of tiny rectangular regions, called "pixels". The incident light is averaged over the area of the pixel so that, for example, a pixel centred on the edge of a black line, will be gray - the average of the black and white regions which each span half the pixel area. If a larger area of the black line spans the pixel then the pixel will be a darker shade of gray.

In the digitised image, each pixel is assigned a number, representing the average intensity of light over the pixel. In the following example, the pixels are represented by 8-bit numbers (0-255): if there is no light on the pixel (if it is black) then it is assigned the value 0 (zero); if the pixel is saturated with light (if it is white) then it is assigned the value 255; pixels that are gray hold an integer value between 0 and 255 depending on the shade of gray. (The total number of values a pixel can hold is $256 = 2^8$; each value is conveniently represented as

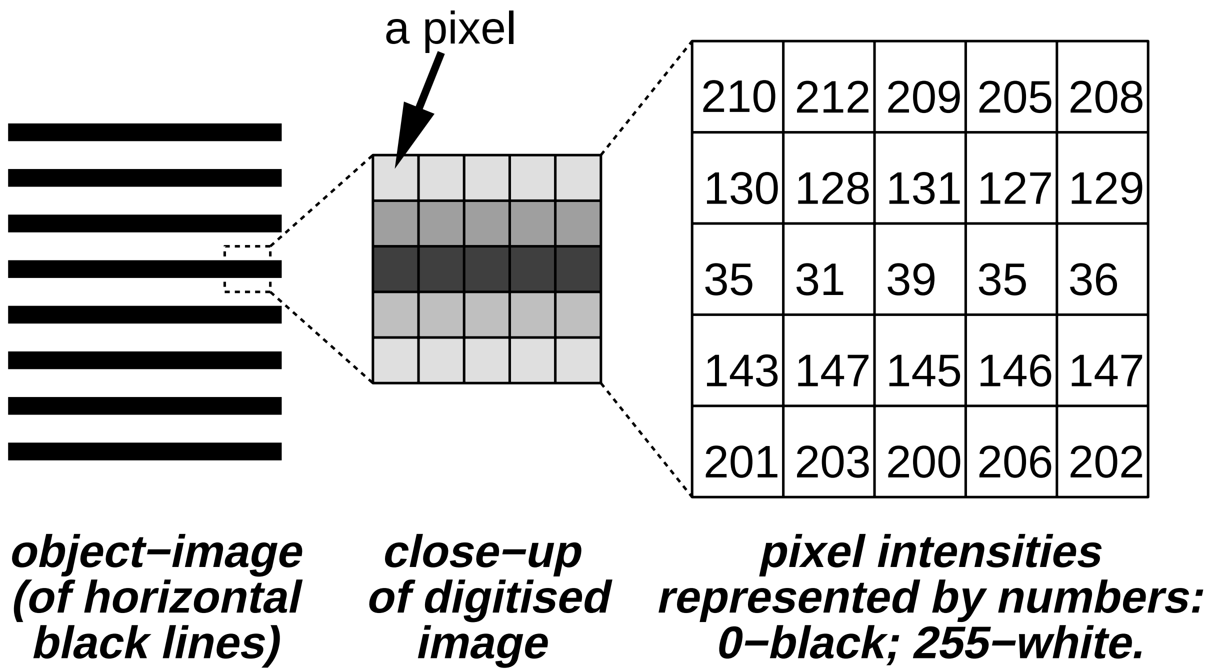

FIG. 3. The object-image (here a grid of horizontal black lines) is digitised giving an array (*e.g.* a 512 by 512 matrix) of pixels. The pixel intensity is the average intensity of the image over the area of the pixel. The intensity of each pixel is a number ranging, typically, from 0 (black) to 255 (white). Thus the digitised image may be represented by a matrix of integers.

a binary number on a computer using the memory equivalent to one byte.) More generally, the intensity of the pixel may be represented by a floating point number. But for ease of discussion, below it is assumed the intensities are represented by integers in the range 0 to 255.

Thus a digitised image may be stored on a computer as a matrix of integers. The size of the matrix depends upon the number of pixels that horizontally and vertically span the entire image. The Data Translation Frame Grabber Cards DT2861 and DT2862 subdivide the image into a 512 by 512 matrix of pixels. Thus, a digitised image requires $512 \times 512 \simeq 256$ kbytes of memory. Some frame grabber cards have memory chips on the board sufficient for storing at least one digitised image. (The cards DT2862 and DT2861 can store 4 and 16 digitized images, respectively.) Frame grabber cards can also access computer memory directly though the speed at which data can be stored in computer memory or disk may be restrictive.

3 Qualitative Mode

Synthetic schlieren can be run in either "qualitative" or "quantitative mode". The latter mode, which is described in detail in the next section, is used in special circumstances to determine how the variations in the object-image correspond to the actual values of the density fluctuations.

In qualitative mode, synthetic schlieren can be used to visualise instantaneous changes in an object-image due, for example, to heat shimmer. Briefly, the technique works by comparing one digitised image with another taken at an earlier time. The object-image at an earlier time is referred to as the "initial image". (Though, possibly, the initial image could be the time-average of many images taken at an earlier time. Time-averaging can be used in this way to reduce signal noise.) The other object-image, referred to as the "current image", may be a snapshot taken at a time after the initial image is taken. Alternately, the current image may be taken continuously in real time so that the the current and initial images are compared continuously. The images are compared digitally on a pixel-by-pixel basis and then enhanced so as to make small changes more apparent. This may be done, for example, by taking the difference between respective pixels on two images and then multiplying the result by an "enhancement factor", a number that multiplies the difference and so makes small changes more apparent.

For ease of discussion, the examples below assume the comparison is taking the absolute value of the difference between two images and multiplying the result by an enhancement factor.

A flowchart illustrating the steps to the qualitative mode of synthetic schlieren is shown in Figure 4. The steps are described in detail below.

The initial image (either a "snap-shot" or "time-average") is digitised. The image is thus represented on the computer by a matrix of integers. Each element of the matrix corresponds to a pixel of the image. The integer value of that element corresponds to the average intensity of light over the pixel. In this discussion, the matrix of initial data is represented symbolically by I_{init}, and the elements of the matrix are represented by $(I_{\text{init}})_{ij}$, where i is the i'th row of the matrix and j is the j'th column of the matrix. The intensity of the pixel in the top left corner of the digitised image is given by the value of $(I_{\text{init}})_{00}$. Each element of I_{init} can be an integer between 0 and $2^b - 1$, where b is the number of bits of computer memory used to resolve the actual intensity. Typically, one byte (8 bits) of memory is used. Thus the intensity of a pixel (a shade of gray) is represented by a number between 0 and 255. For example, if $(I_{\text{init}})_{00} = 0$, the top corner pixel is black; if $(I_{\text{init}})_{00} = 255$, the top corner pixel is white.

An enhancement factor is entered. In this discussion, the enhancement factor is represented by an integer, m. Typically the enhancement factor equals 5, though it may be larger if the fluctuations of the image over time are small and greater enhancement is required. Likewise, if the intensity variation of the image with time is large, then no enhancement is required and the enhancement factor may be entered as 1.

Next the current image is digitised. This may be done from a snapshot or done continuously while the image is being recorded or played back. In this discussion, the matrix of integers resulting from this digitised image is represented by I_{current}, which has elements $(I_{\text{current}})_{ij}$.

The computer then computes the value of $m|I_{\text{current}} - I_{\text{init}}|$, where the vertical lines denote the absolute value (*e.g.* $|-2| = 2$). The result is a matrix I_{synth} with elements $(I_{\text{synth}})_{ij} = m|(I_{\text{current}})_{ij} - (I_{\text{init}})_{ij}|$. If $(I_{\text{synth}})_{ij}$ lies above the range allowed by computer memory, *i.e.*

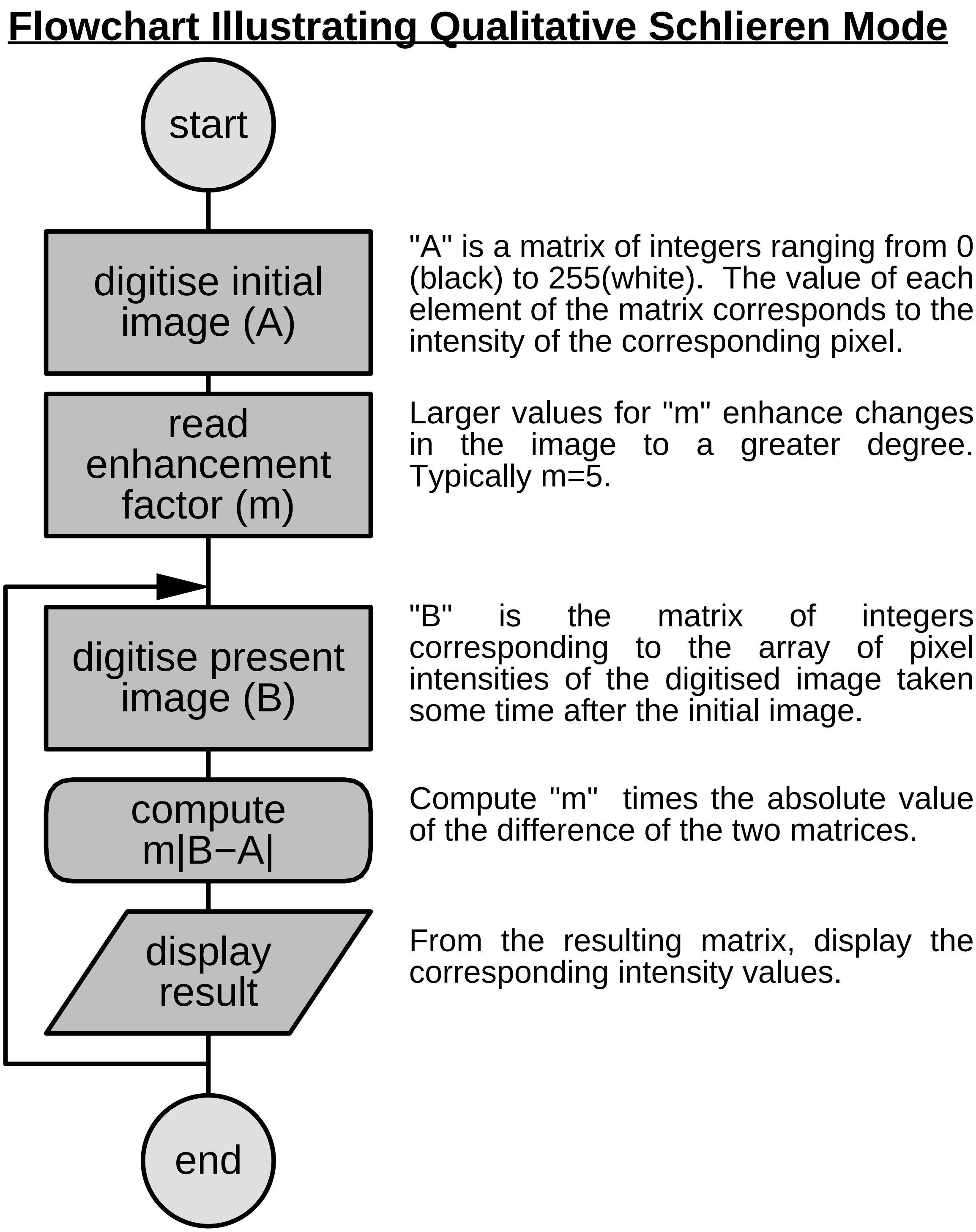

FIG. 4. Flow chart describing the "qualitative mode" of synthetic schlieren. In a typical set up, the loop is run continuously so that the constantly changing value of the schlieren image, given by values of the array $m|B-A|$, can be visualised.

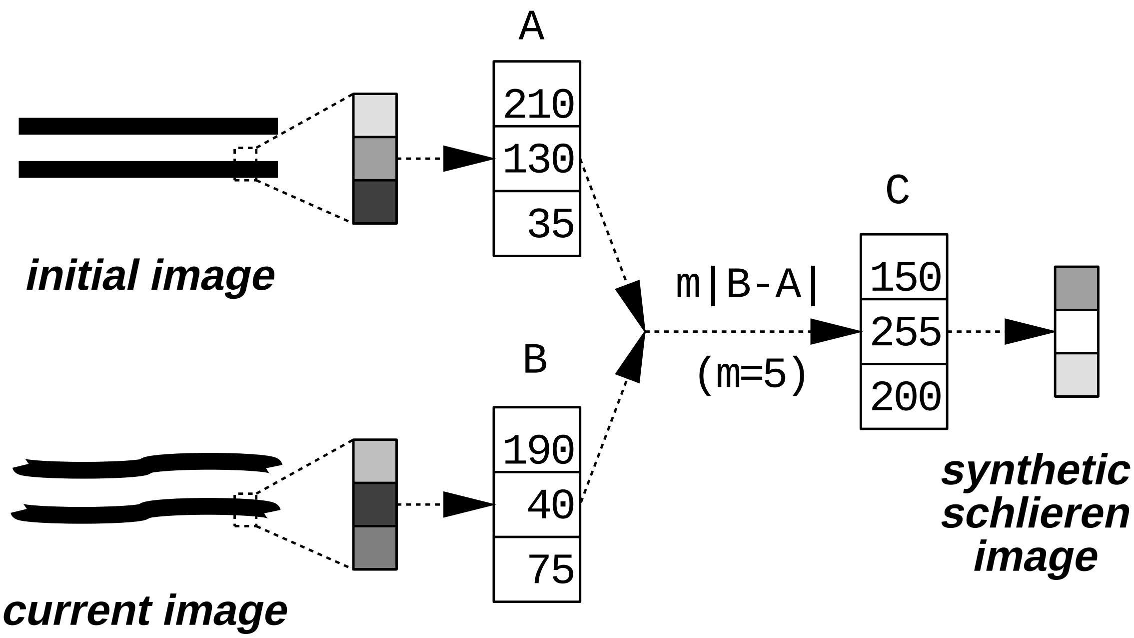

FIG. 5. Example showing the calculation used to enhance the difference between two images to generated a "synthetic schlieren" image in qualitative mode.

$(I_{\rm synth})_{ij} \geq 2^b$, then $(I_{\rm synth})_{ij}$ is instead assigned the maximum allowed value, $(I_{\rm synth})_{ij} \to 2^b - 1$. Likewise, if $(I_{\rm synth})_{ij}$ is negative, then $(I_{\rm synth})_{ij} \to 0$.

Finally, the elements of $I_{\rm synth}$ are interpreted as pixels of an image whose intensities are given by the values $(I_{\rm synth})_{ij}$. This "synthetic schlieren" image may be displayed on a monitor. Often, this is done in false colour to make small changes even more apparent.

An example of this calculation procedure is illustrated in Figure 5.

The calculation $m|I_{\rm current} - I_{\rm init}|$ is useful because the response to changes, both small and large, is linear. However, one can also compute, for example, $m(I_{\rm current} - I_{\rm init})^2$ to exaggerate the effect of large fluctuations. Or one can compute $m\sqrt{(|I_{\rm current} - I_{\rm init}|)}$ to exaggerate the effect of small fluctuations. Similarly, other functions of $I_{\rm current} - I_{\rm init}$ may be computed, depending upon the application and the quantities of interest to be visualised.

The qualitative mode of synthetic schlieren has been implemented using DigImage, an image processing software package (see appendix A).

An example of DigImage's implementation of the qualitative mode of synthetic schlieren is shown in Figure 6. These plates were produced by focussing the camera on a 12cm by 8cm region of a grid of horizontal black lines. The grid is 3.5 m from the camera. Between the camera and the image is a 20 cm wide tank filled with salt-stratified water. A periodic disturbance at the water surface creates waves in the fluid that propagate downward. Because the disturbed fluid induces density perturbations in the tank the path of light rays from the image to the camera is deflected and the current image (Fig. 6b) is distorted. (In this example, the signal is particularly strong and distortions of the current object-image are easily apparent. Even if the distortions are not apparent on the current image, synthetic schlieren can reveal them.)

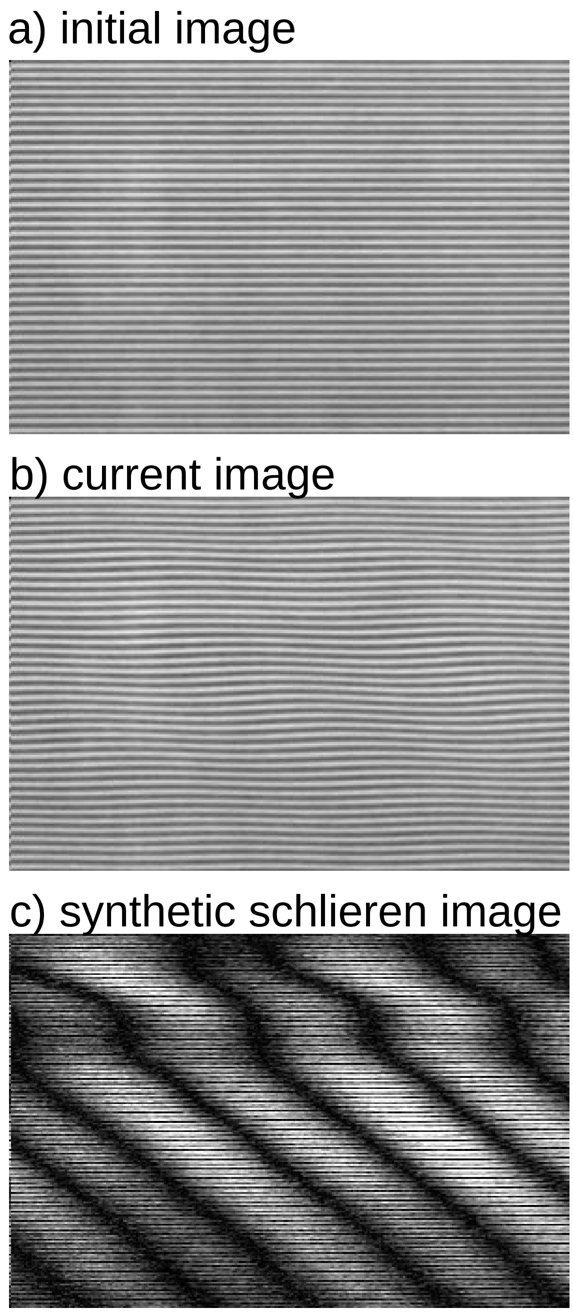

FIG. 6. Qualitative mode of synthetic schlieren used to visualise waves in a tank of salt-stratified fluid: a) initial image of horizontal black lines; b) current image distorted by waves; c) synthetic schlieren image found by taking difference of the digitised current and initial images and multiplying the absolute value of the result by an enhancement factor of 5.

Fig. 6a shows the initial image taken before the surface is disturbed. All the lines are parallel and horizontal. Fig. 6b shows the current image taken while the surface is disturbed and waves propagate downward. Indeed, a slight deflection of the lines is evident in this image. Fig. 6c shows the synthetic schlieren image produced by calculating $m|I_{\text{current}} - I_{\text{init}}|$ using an enhancement factor of $m = 5$. The slight deflections in the lines are immediately apparent as the bright regions in this image.

4 Quantitative Mode

Synthetic schlieren may be used in quantitative mode to determine how the fluctuations in an image correspond to the magnitude of the density fluctuations themselves. We illustrate its implementation here under laboratory conditions in which the density fluctuations are assumed to be spanwise uniform over the test section. (More generally, a pre-existing knowledge of the spatial structure (*e.g.*uniform or axisymmetric) allows the quantitative mode to be used with only one camera providing a single perspective of the test section. If, by using additional cameras or mirrors, two or more simultaneous perspectives of the test section are examined, then more complex three dimensional geometries may be reconstructed both qualitatively and quantitatively.)

In the discussion below, it is assumed that the test section is a tank filled with salt-stratified fluid. Assuming the fluid is initially static, the density of the fluid decreases with increasing distance above the bottom of the tank as the fluid becomes less saline. This "background" density distribution as a function of distance z above the bottom of the tank is represented by $\overline{\rho}(z)$. (In general, the background density may vary either along or across the camera's line of sight.)

A simple calculation shows that as a light ray passes through the fluid, entering the tank from the side at a small angle ϕ_0 to the horizontal, the ray follows a parabolic path, as illustrated schematically in Figure 7a. The vertical position z of the ray as it passes at distance y through the stratified fluid is

$$z(y) = z_0 + y \tan \phi_0 - \frac{1}{2} \left(\frac{1}{n_0} \frac{dn}{d\rho} \right) \left(-\frac{d\overline{\rho}}{dz} \right) y^2, \tag{1}$$

where z_0 is the vertical position at which the light ray enters the tank, $n_0 = 1.3330$ is the index of refraction of water, $\rho_0 = 0.9982\,\mathrm{g/cm^3}$, the density of fresh water at room temperature, and $dn/d\rho = 0.246$ is the rate of change of the index refraction with increasing density of saline water. The quantity $-\frac{d\overline{\rho}}{dz}$ is positive because the fluid density decreases as z increases.

Equation (1) may also be written in terms of the more standard quantity (called the "squared buoyancy frequency"),

$$N^2 = -(g/\rho_0) d\overline{\rho}/dz. \tag{2}$$

Hence $z(y) = z_0 + y \tan \phi_0 - \frac{1}{2}\gamma N^2 y^2$, where $\gamma = (\rho_0/g n_0) dn/d\rho = 1.878 \times 10^{-4}\,\mathrm{s^2/cm}$.

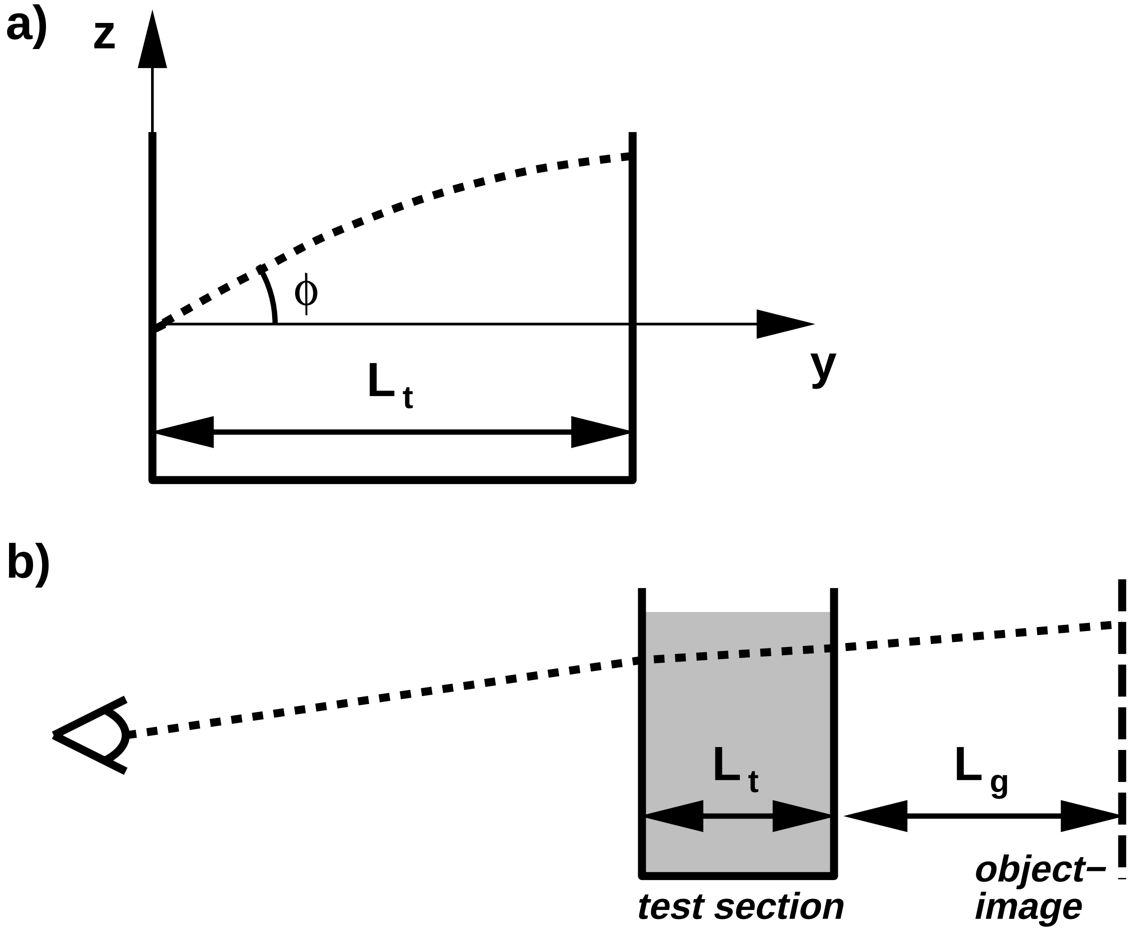

FIG. 7. a) The path of a light ray (dashed line) through a stratified fluid. The ray enters the tank at an angle ϕ to the horizontal and is bent along a parabolic arc. The degree to which it is bent depends upon the rate of change of density with depth $d\overline{\rho}/dz$. b) The path of the light ray from the image to camera passing through a tank, of width L_t, that is filled with stratified fluid. Note, in both diagrams the vertical scale is greatly exaggerated; typically the system is set up so that the angle $\phi \simeq 0$.

It follows immediately from equation (1) that light is bent to a greater degree if the density gradient is larger and it is bent to a lesser degree if the density gradient is smaller.

Now assume a wave propagates within the tank inducing a disturbance to the density field. This is a wave that moves under the influence of buoyancy forces within a density stratified fluid. If the wave is uniform across the span of the tank, the effect of the wave is to increase and decrease the local density gradients. By determining how much the light is deflected, it is possible to use (1) to measure how much the density gradient has changed, and so measure the amplitude of the wave.

Explicitly, by comparing initial and current images of a grid of horizontal lines, the vertical displacement, Δz, of pixels on the edge of the lines can be measured. Assuming the thickness of the tank walls is negligibly small, the resulting change in the density gradient in the tank is

$$\frac{\partial \rho}{\partial z} = \Delta z \frac{n_0}{dn/d\rho}\left[\frac{1}{2}{L_t}^2 + L_t n_0 \left(\frac{L_g}{n_a}\right)\right]^{-1}, \quad (3)$$

where $n_a = 1.000$ is the index of refraction of air, L_t is the width of the tank and L_g is the distance from the tank to the image, as illustrated in Figure 7b.

The quantity

$$\Delta N^2 = -g/\rho_0 \frac{\partial \rho}{\partial z} \quad (4)$$

may be computed from the value of the perturbation density gradient given by (3). In equation (4), $g \simeq 980\,\mathrm{cm/s^2}$ is the acceleration of gravity and $\rho_0 \simeq 1.0\,\mathrm{g/cm^3}$ is the density of fresh water.

Figure 8 shows three stages in the generation of the ΔN^2 field found from equations (3) and (4). The computed images are found from the initial and current images shown in figs 6a and b, respectively. In fig. 8a the values of ΔN^2 are computed for pixels at the edges of the black lines in the initial image. The black lines indicate regions where the computation was not performed in order to reduce signal noise. Fig. 8b is found from Fig. 8a by replacing black pixels with pixels whose intensities hold the average of surrounding non-black pixels. Effectively this uses the calculation that determined Fig. 8 and interpolates over regions where the calculation could not be performed. Finally, a low pass Fourier filter is applied to smooth the result and produce the quantitative schlieren image of the ΔN^2 field shown in Fig. 8c.

Synthetic schlieren is also usefully applied to a time series. This is a digitised image that displays how a cross-section through a spatial image evolves in time. Alternately, the time series image may be created directly from a line-scan device which directly records the time evolution of an image over a single line of pixels.

For example, Figure 9 shows the evolution in time (horizontal axis) of a vertical slice through the image shown in fig. 6a. The vertical slice is effectively a column in the matrix representing the digitised array of pixels. The periodic disturbance is initiated at a time corresponding to the left side of the picture. As time evolves, successive vertical slices (columns of the matrix) are successively placed proceeding from left to right in the picture. The slices are taken at a rate determined by the person processing the image. (See appendix AA.4.) In fig. 6a, slices are taken every 0.2 sec for 1 minute.

By comparing each column of the resulting digitised matrix with the first column of the matrix (a vertical slice through the initial image), the time series of the ΔN^2 field can be found. This is shown in fig. 9b. This quantitative schlieren image clearly shows the leading edge of the disturbances propagating downward as time progresses, from left to right. (The disturbances are waves whose front moves downward while the wave crests propagate upward.) Furthermore, by comparing a vertical slice with a vertical slice taken a short time earlier (typically a fraction of a second), the time rate of change of the ΔN^2 field can be found. This is shown in fig. 9c. The latter is useful, because it filters out slowly evolving disturbances.

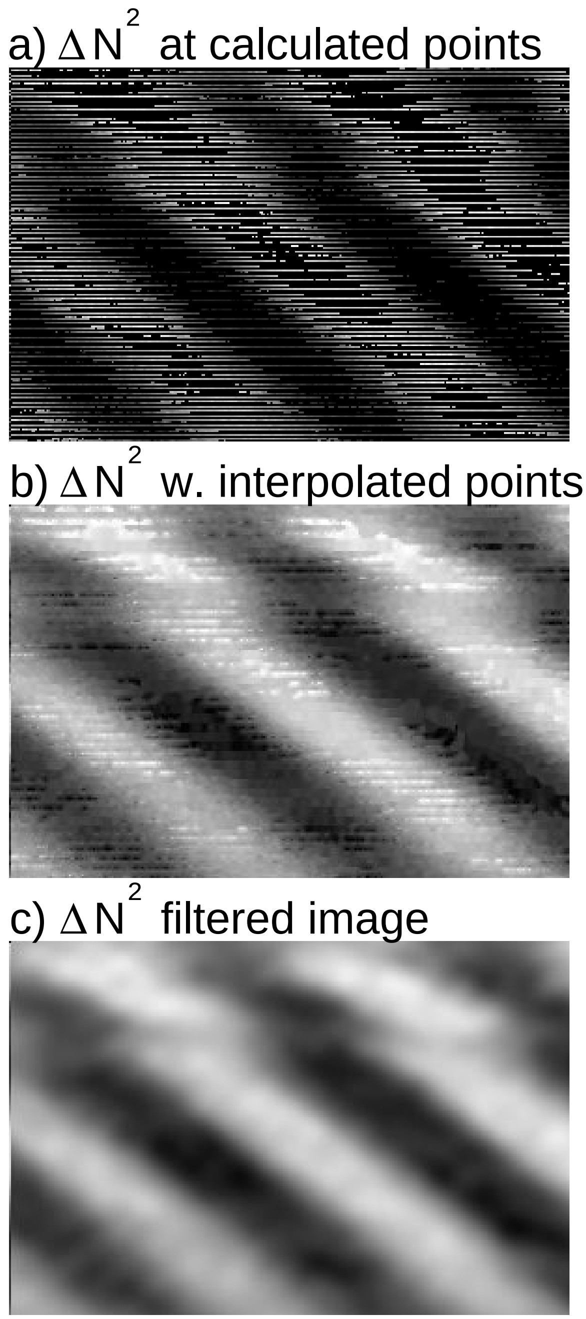

FIG. 8. Quantitative mode of synthetic schlieren used to visualise waves in a tank of salt-stratified fluid. The initial and current images are the same as those shown in Fig. 6a and b. a) the calculation determines the change in ΔN^2, which is proportional to the change in density gradient $\partial\rho/\partial z$ induced by the waves. The calculation is performed on the edges of the lines in the initial image in Fig. 6a. Pixels where the calculation are not performed are left black (intensity zero); b) pixels that are black are assigned intensities equal to the average of the surrounding pixels; c) the result is smoothed by a low pass filter.

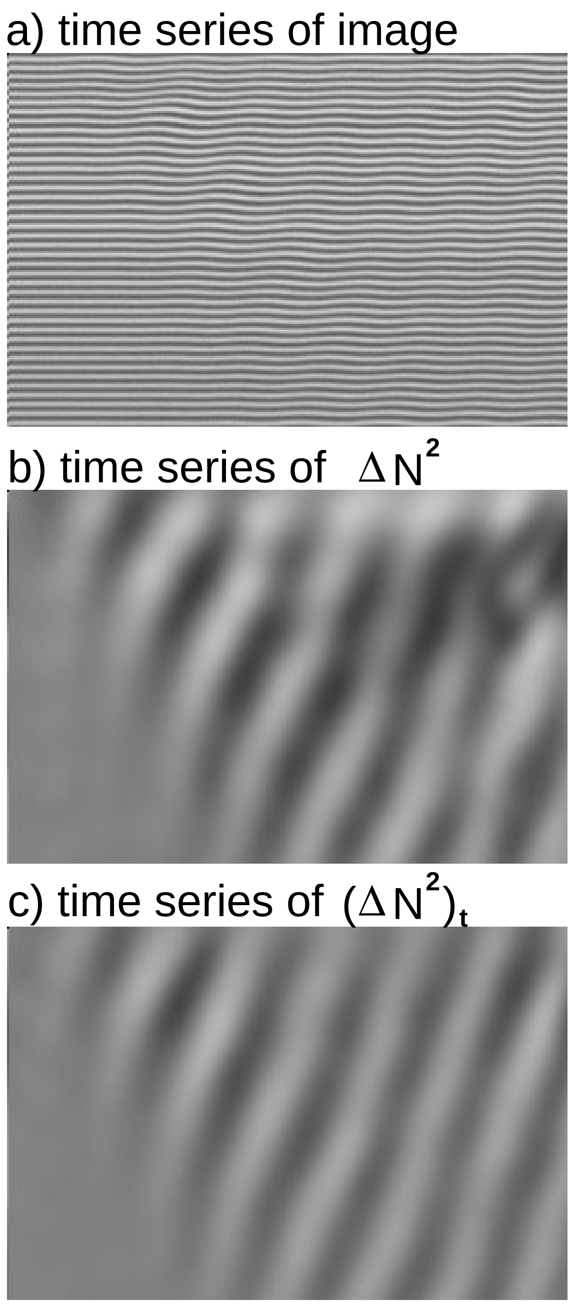

FIG. 9. Quantitative mode of synthetic schlieren used to visualise time series of waves in a tank of salt-stratified fluid. a) image of time series showing the evolution of the image along a vertical slice taken along the left edge of the image shown, for example, in Fig. 6. b) Time series of the ΔN^2 field (see text). The image shows the wave front propagating downward in time (though the crests of the waves move upward). c) Time series of time derivative of the ΔN^2 field (see text). This image filters motions that evolve more slowly.

5 Applications

Synthetic schlieren may cost-effectively replace present techniques that visualise and measure density changes which are used in industry, medicine and other sciences. Some examples illustrating the potential breadth of application are listed here.

- Visualisation and measurement of heat: Synthetic schlieren can be used to monitor heat rising from a human body, machinery, pavements or other objects. This could be used to measure heat loss, or detect leakage of heat from insulated objects - for example, around windows or from the roof of a house. It could be used to monitor wind gusts near runways.

- Visualisation and measurement of shock waves: Shock waves, for example in air, compress and expand gas and so locally heat and cool it, respectively. The waves can thus be visualised. Synthetic schlieren could be used to examine shock waves, for example, from supersonic aircraft (or models of them in laboratory conditions) or from a gun being fired.

- Visualisation of non-homogeneous turbulence: turbulence is easily apparent in a convecting fluid, whether due to heat (hot under cold fluid) or, for example, salinity (fresh under salty, dense fluid). In a fluid that is stably stratified (light fluid over dense), turbulence due to mixing can be visualised. This has applications in identifying the extent and longevity of turbulence in a non-homogeneous fluid. Examples are turbulence due to combustion in the combustion chamber of a car's engine; the turbulent wake behind a submarine in the ocean (whose salinity and temperature varies with depth); mechanical mixing by jets or a stirrer in a vat filled with liquids of varying concentration.

- Visualisation and measurement of disturbances in non-homogeneous objects: If the index of refraction of a solid, liquid or gas is non-uniform (whether due to density or compositional changes), then synthetic schlieren can detect and measure the magnitude of time-variations of density due to sound waves (in solids) and due to waves, such as internal waves (which move due to buoyancy effects) in liquids and gases. This has applications, for example, in solid state physics (*e.g.* detecting defects in silicon wafers), and detecting internal waves behind submarines.

- Visualization and measurement of changes in internal organs: If x-rays, rather than visual light, is used, synthetic schlieren can be applied to examine density variations in the human body, and so detect chemical changes and defects in organs. If more than one perspective is of the organ is examined, standard tomographic techniques can be employed to measure the three-dimensional spatial structure of the variations.

A DigImage Commands

DigImage is a versatile menu-driven software program that runs on PCs equipped with a Data Translation Frame Grabber Card (DT2861 or DT2862). Contact information is provided in Appendix C.

Digitised images are stored in memory. Many images can be stored at the same time in different memory locations. These locations are called buffers. For example, four images can be stored and manipulated in buffers 0, 1, 2 and 3.

A.1 Grab a Single Image

The following sequence of commands allows a single image to be digitised and stored in a buffer. In the example below the image is stored in buffer 0 after the space bar is pressed.

Command	Explanation
;	Go to main menu
G	Grab images menu
G	Grab an image
0	Buffer (memory location) where image is put
⟨enter⟩	grab image when enter key is pressed

A.2 Grab a Sequence of Images

The following sequence of commands allows a sequence images to be digitised and stored in buffers. In the example below the images are grabbed at times 0, 4, 8 and 12 seconds after the space bar is pressed. The images are stored in buffers 1, 2, 3 and 4.

Command	Explanation
;	Go to Main Menu
G	Grab Images Menu
S	Grab a Sequence of Images
0	time (in sec) to grab first image
4	grab second image after 4 seconds
8	grab third image after 8 seconds
12	grab fourth image after 12 seconds
-1	done specifying when to grab frames
⟨return⟩	Don't save results to file
⟨space⟩	grabbing images begins as soon as pressed

A.3 Continuously Acquire an Image

The following sequence of commands allows an image to be digitised continuously. DigImage can then perform real-time arithmetic operations on the evolving image.

Command	Explanation
;	Go to main menu
G	Grab images menu
G	Grab an image
0	Buffer (memory location) where image is put
C	Continuously acquire image

A.4 Make a Time Series

The following sequence of commands will make an image showing how a column or row of pixels evolve over time. In the particular example below, a vertical time series is created from a continuously evolving image. A vertical time series is formed by taking a vertical slice through the image at successive intervals. Each slice is stacked one after the other thus creating a new image which varies vertically in space and horizontally in time.

An initial image is assumed to be already stored in buffer 1. Cross-hairs are oriented on this image at the position where the vertical time series is to be taken. In order to span the screen (filling 512 pixels horizontally) the time series is taken for 17.033 seconds at a sampling period of 1/30th second. (This is the standard video rate for NTSC systems, such as that used in North America. For PAL systems, as used in most of Europe, the sampling period is 1/25th second, and the shortest time series that fills the screen is taken for 20.44 seconds.) Time series can be taken for arbitrarily longer times at the expense of reduced temporal resolution.

Command	Explanation
;	Go to Main Menu
T	Time Series Menu
L	Time Series along a horizontal or vertical line
1	Locate position of line in buffer 1 using cross-hairs
P	After locating position exit to continue
C	Take vertical time series (L for horizontal)
2	Put image of time series in buffer 2
N	Don't take time series anywhere else
17.033	Time series 17.033 sec. long: time increment is $\Delta t = 0.033$ sec between samples
M	Take image by pressing a key on the keyboard
⟨return⟩	Image taking begins when return is pressed

A.5 Save a bitmap (.bmp) Image to a File

Command	Explanation
;	Go to Main Menu
K	Save or restore image Menu
C	save a .BMP file to file
1	save image in buffer number 1
img1.bmp	save to file called "img1.bmp"
S	save the screen
300	number of horizontal pixels
300	number of vertical pixels
8	number of bits/pixel

A.6 Load a Bitmap (.bmp) Image into DigImage

Command	Explanation
;	Go to Main Menu
K	Save or restore image Menu
B	read a .BMP file into DigImage
1	put image in buffer number 1
img1	read in file called "img1.bmp"
S	make image fill the screen

A.7 Save Image to a .PIC file (DigImage's special image format)

Command	Explanation
;	Go to Main Menu
K	Save or restore image Menu
S	save a .PIC file
1	save image in buffer number 1
img1.pic	save to file called "img1.pic"
y	compress image
8	number of bits/pixel

A.8 Load an image from a .PIC file

Command	Explanation
;	Go to Main Menu
K	Save or restore image Menu
R	load image in a .PIC file to a buffer
1	load image into buffer number 1
img1	load image from file called "img1.pic"

A.9 Set Up Physical Co-ordinate System

Command	Explanation
;	Start from to main menu
P	Co-ordinate system menu
W	map pixel to physical co-ordinate menu
I	initialize co-ordinate system
cm	co-ordinate in centimetres
L	locate specific co-ordinate to make map
0	locate co-ordinates in buffer number 1
⟨cursor keys⟩	move cross-hairs to co-ordinate
P	exit
0 0	co-ordinates of point in cm: x=0 cm; z=0 cm
Y	locate another co-ordinate
⟨cursor keys⟩	move cross-hairs to another co-ordinate
P	exit
0 30	co-ordinates at: x=0 cm; z=30 cm
Y	locate another co-ordinate
⟨cursor keys⟩	move cross-hairs to another co-ordinate
P	exit
30 0	co-ordinates at: x=30 cm; z=0 cm
N	stop getting co-ordinates

A.10 Filter noise from an image

Command	Explanation
;	Start from main menu
F	filter image
F	perform Fourier filters
L	Low pass filter: get rid of small scale (high frequency) noise ... low frequencies remain
1	Filter image in buffer 1
B	filter both horizontal and vertical directions
32	cut off variations with frequencies faster than 32/512 pixels

A.11 Run Synthetic Schlieren in Qualitative Mode

Qualitative mode may be run by typing each of the commands below.

Command	Explanation
;	Go to main menu
G	Grab images menu
G	Grab an image
1	Buffer (memory location) where image is put
⟨enter⟩	grab image when enter key is pressed
;	Go to main menu
G	Grab images menu
M	Image Manipulation
A	Grab an image and perform arithmetic operations on it
38	Subtract one buffer from another
0	Leave present image as is before performing operations
V	read formula that defines what to do with result
abs(p-256)*5	difference is value from 0-511 (256-511 correspond to negative numbers); subtract 256 from difference and find absolute value. Multiply result by "enhancement factor" of 5.
S	Set result to 255 if actual result exceeds 255
2	Buffer containing current image
1	Buffer containing initial image
3	Buffer in which to put synthetic schlieren image
1	do not zoom in on result
0	no vertical offset
0	no horizontal offset

Alternately, these commands may be put in a slightly modified form in a file called, for example, "synth.cmd":

;	Start from main menu
G:	Grab/Display Menu
G	Grab single frame
!^	Grab reference image
1	put result in buffer 1
A	grab image and perform arithmetic operations on it
B	grab a single image
0	put result in buffer 0
1	compare result with image buffer 1
38	subtract image in buffer 1 from image in buffer 0; result is denote by "P"
0	do not adjust result ...
V	except to perform following calculation after enhancement factor is entered
!^ Enhance?	Get enhancement factor
!!0:=!!GK	Assign enhancement factor to variable called !!0
abs(P-256)*!!G0	take result "P", subtract 256, find absolute value and multiply by !!0
S	perform operation on whole screen
!L C	grab frame continuously

DigImage automatically executes the commands prompting the user for input as shown below:

Command	Explanation
!P synth	execute commands in file "synth.cmd"
⟨enter⟩	grab image when enter key is pressed
5	Enter enhancement factor

A.12 Run Synthetic Schlieren in Quantitative Mode

If DigImage is compiled with the Fortran subroutines that calculate how fluctuations in an image of horizontal black lines correspond to density fluctuations, then the following commands are typed to process the image.

The commands assume the user starts from the synthetic schlieren menu.

Command	Explanation
N	Find ΔN^2 field
S	field is spatial, not space-time image
1	buffer containing initial image
2	buffer containing current image
S	process entire image
34.2	L_g: distance from tank to grid (in cm)
350.0	L_c: distance from camera to tank (in cm)
0.0	horizontal position of center of image (in cm)
13.0	vertical position of center of image (in cm)
10	tolerance: process only if intensity difference between upper and lower pixels exceeds 10
-0.1	values $\Delta N^2 \leq -0.1$ are given pixel intensity 1
0.1	values $\Delta N^2 \geq +0.1$ are given pixel intensity 255
	(for $-0.1 \leq \Delta N^2 \leq 0.1$ pixel intensity is found by linear interpolation.)
3	put resulting "quantitative schlieren" image in buffer 3
Y	accept result (other try other values of min and max ΔN^2
Y	accept result (other try other values of tolerance)

Now interpolate over black regions (where calculation was not performed) in order to compute values everywhere on the image.

Command	Explanation
I	Average over regions of zero intensity
3	buffer containing "quantitative schlieren" image
S	process entire image
4	put result in buffer 4
A	calculate average of surrounding pixels
5	average over box 5 pixels wide ...
5	and 5 pixels tall
W	perform Gaussian weighted average (points within standard deviation σ given more weight)
2.0	σ: standard deviation
B	only perform average over pixels that are "black" (zero intensity)

At this point, the resulting image may be filtered using the commands, for example, in subsection A.10.

B DigImage Source Code to Run Synthetic Schlieren in Quantitative Mode

Special software subroutines have been written for use by DigImage to use synthetic schlieren in quantitative mode. The source codes are interpreted by Microsoft Fortran version 5.0 or 5.1 and are compiled along with the DigImage source code. When running DigImage, the subroutines allow the user to supply data as prompted through a series of menus. The data is then used to calculate the magnitude of density changes from the observed fluctuations of an object-image. The resulting calculation is shown as image which may be further enhanced to interpolate over regions where the calculation could not accurately be performed.

For simplicity, it is assumed in the discussion below that the image is a grid of horizontal lines.

Two main subroutines are listed in subsection B.1 below. The first, “Schlieren2DSpace”, computes the density fluctuation field given an initial and final image, and the second, “SchlierenSpaceTime”, computes the density fluctuation field from a vertical time series - an image showing the time-evolution of a column of pixels. Each subroutine performs a similar set of operations, reading in data and calling other subroutines, as illustrated in Figure 10.

Both subroutines begin by reading in the matrices of data representing the images to be processed. For example, these may be the initial and current images. The programs then ask the user to supply information about the setup of the camera, object-image and test section. Both “Schlieren2DSpace” and “SchlierenSpaceTime” call the specialised subroutines in the main part of the routine.

In order to reduce signal noise, the calculation is not performed for every pixel in the image. Part of the calculation requires interpolating between the intensities of a stack of three pixels. In order for the interpolation to be unambiguous, the intensities from top to bottom must increase or decrease monotonically. Thus, the calculation is performed effectively for pixels at the upper and lower edges of each horizontal line, but it is not performed at the center of the lines or the centre of the bright areas between the lines.

Furthermore, in order to reduce signal noise, the calculation is performed for a particular pixel at the edge of a line, the calculation being performed only if its intensity differs by more than some threshold, typicaly 10, from the intensity of the pixel immediately above and beneath it. An example is the close-up of the digitised image illustrated in Figure 3

Subsection B.2 lists some of the FORTRAN 90 subroutines used to calculate disturbances from fluctuations in arbitrary images, *e.g.* an array of dots.

B.1 Source Code for Horizontal Line Images

The following subroutines are used, optimally, to calculate how density changes by observing fluctuations in a grid of horizontal lines.

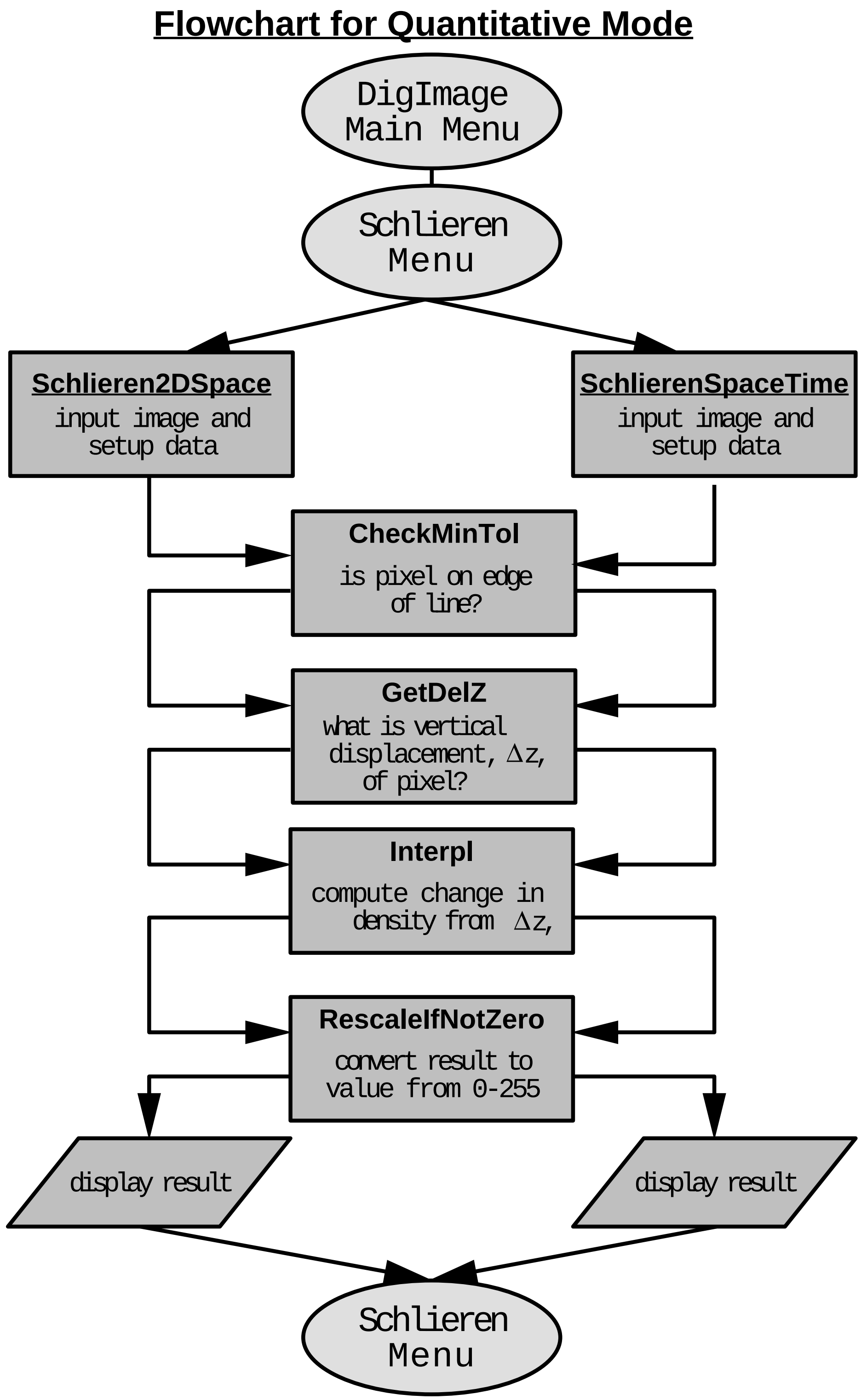

FIG. 10. Flow chart describing the "quantitative mode" of synthetic schlieren. One of two subroutines may be called, one that calculates the density fluctuation field given an initial and present image, and one that calculates a vertical time series of the density fluctuation field from a time series of the raw image.

Schlieren2DSpace

```
C%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C% Schliern.FOR                          OVERLAY                         %
C%    Subroutines for calculating vertical displacements                 %
C%    of grid lines and fluid parcels.                                   %
C%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
C$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
C$    Calls defined by SCHLIERN.FOR:                                     $
C$        Schlieren2DSpace                                               $
C$        SchlierenSpaceTime                                             $
C$        GetDelZ                                                        $
C$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
C^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
C^    Calls made by SCHLIERN.FOR:                                        ^
C^        CALL AcceptBuffer                                              ^
C^        CALL AcceptInteger                                             ^
C^        CALL GetLineFromBuffer                                         ^
C^        CALL MoveArrayToWindow                                         ^
C^        CALL MoveBufferToMemory                                        ^
C^        CALL SetCurrentHelpFile                                        ^
C^        CALL ShowBuffer                                                ^
C^        CALL WindowOrScreen                                            ^
C^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
$STORAGE:2
$DECLARE
C*************************************************************************
C* Schlieren2DSpace                                                      *
C*   This subroutine is designed to find the motion of lines             *
C*   on a schlieren grid due to density changes in a stratified fluid    *
C*   Options determine whether the relative motion of lines, physical    *
C*   displacement of lines, change in N^2 of water, vertical             *
C*   displacement of lines, fluctuation density, or vertical velocity    *
C*   fields should be found.                                             *
C*************************************************************************
      SUBROUTINE  Schlieren2DSpace(Mode)
$INCLUDE:'All.INC'
$INCLUDE:'WorkSpce.INC'
$INCLUDE:'Configur.INC'
C==Define and dimension variables
      LOGICAL NotDone,iError
      INTEGER*1 jCol,jCol2,iRow,iRow2
      INTEGER*2 iWant1,iWant2,iWant3,iHandle,iOnboard
      INTEGER*2 iw0,iw1,jw0,jw1,i,j
```

```
      INTEGER*2 Ir,Irn,Irp,In,I0,Ip,Id,iMinTol
      INTEGER*2 nmax,nn,nc,nz,iz
      REAL Ltank,Lperspex,Lgrid,Lcam,nwater,nperspex,nair,nw2nsqr
      REAL PyCam,PzCam,ScaleTG,Lpg,coefdelz,Rho0OverG
      REAL maxfld,minfld,maxfldi,minfldi
      REAL delz,dt,yw0,zw0,n2zw
      REAL zzmin,zzmax,n2min,n2max
      REAL fz,z,dfzdz,zz,n2
      CHARACTER Mode,AcptScaling,AcptMinTol,SetBndyConds
      CHARACTER N2file*(20)

      PARAMETER (nmax=256)
      DIMENSION jCol(0:511),jCol2(0:511),iRow(0:511),iRow2(0:511)
      DIMENSION n2(nmax),zz(nmax)
      DIMENSION dfzdz(nmax),z(nmax),fz(nmax)
      EQUIVALENCE (iWork1(0),jCol(0))
      EQUIVALENCE (iWork1(512),jCol2(0))
      EQUIVALENCE (iWork1(1024),iRow(0))
      EQUIVALENCE (iWork1(1536),iRow2(0))
      EQUIVALENCE (rWork1(0),fz(1))
      EQUIVALENCE (rWork1(256),z(1))
      EQUIVALENCE (rWork1(512),dfzdz(1))
      EQUIVALENCE (rWork1(768),zz(1))
      EQUIVALENCE (rWork1(1024),n2(1))

C==Set default options
      DATA iMinTol /10/
      DATA minfld,maxfld /-0.1,0.1/
C==default lengths in cm: span of tank, perspex, and distance to grid
      DATA Ltank /20.3/, Lperspex /1.1/, Lgrid /34.2/, Lcam /343./
C==position of camera in grid co-ordinates
      DATA PyCam /0.0/, PzCam /13.0/
C==indices of refraction for water, perspex and air
      DATA nwater /1.3330/, nperspex /1.49/, nair /1.0000/
C==scaling from index of refraction gradient to N^2  (in s^2/cm)
C==  (d(nw)/dz)/nw = -nw2nsqr*N^2
      DATA nw2nsqr /0.0001878/
C==Density of water divided by accn of gravity (in g s^2/cm^4)
      DATA Rho0OverG /0.001019/
C
C      CALL SetCurrentHelpFile('Norminty.F01',' ')
C
C= Make sure option passed to this subroutine is allowed
```

```
      IF (Mode .EQ. 'G') THEN
        WRITE(6,*) 'Calculate line displacement in grid world co-ords'
      ELSEIF (Mode .EQ. 'N') THEN
        WRITE(6,*) 'Calculate Delta N^2 field'
      ELSEIF (Mode .EQ. 'P') THEN
        WRITE(6,*) 'Calculate time derivative of Delta N^2'
      ELSEIF (Mode .EQ. 'R') THEN
        WRITE(6,*) 'Calculate flucutation density field'
      ELSEIF (Mode .EQ. 'S') THEN
        WRITE(6,*) 'Calculate relative shift in line displacement'
      ELSEIF (Mode .EQ. 'T') THEN
        WRITE(6,*) 'Calculate line displacement in tank world co-ords'
      ELSEIF (Mode .EQ. 'U') THEN
        WRITE(6,*) 'Calculate horizontal velocity field'
      ELSEIF (Mode .EQ. 'W') THEN
        WRITE(6,*) 'Calculate vertical velocity field'
      ELSEIF (Mode .EQ. 'Z') THEN
        WRITE(6,*) 'Calculate vertical displacement field'
      ELSE
        WRITE(6,*)'Option to SchlierenTank not valid'
        RETURN
      ENDIF

C==Get the buffer containing initial picture of a grid
      WRITE(6,*)'What is buffer containing initial image?'
      CALL AcceptBuffer(iWant1,1)
      IF (EscapePressed) THEN
        RETURN
      ENDIF
      CALL ShowBuffer(iWant1,iOnboard,.FALSE.)

C==Get the buffer containing instantaneous picture of a grid
      WRITE(6,*)'What is buffer containing current image?'
      CALL AcceptBuffer(iWant2,1)
      IF (EscapePressed) THEN
        RETURN
      ENDIF
      CALL ShowBuffer(iWant2,iOnboard,.FALSE.)
      CALL WindowOrScreen(iWant2,iw0,iw1,jw0,jw1,.TRUE.,' ',' ')

C==Get time difference between buffers
      IF (Mode .EQ. 'P' .OR. Mode .EQ. 'U' .OR. Mode .EQ. 'W') THEN
        dt = 0.1
```

```
        WRITE(6,*)'What is time difference, dt, between buffers?'
        CALL AcceptReal(dt,0.01,10.0,2)
      ENDIF

C==Get file containing N^2 profile and set boundary conds for integration
      IF (Mode .EQ. 'U' .OR. Mode .EQ. 'W' .OR. Mode .EQ. 'Z') THEN
        N2file='N2T*.XPT'
        WRITE(6,*)'XPlot filename of N^2 profile'
        CALL AcceptString(N2file,2)

        CALL OpenFile(iHandle,' ',N2file,
     &       'unknown','formatted','sequential',iError)
        IF (iError .OR. iHandle .EQ. 0) THEN
          CALL Warning('Cannot open specified N2 file')
        ELSE
          READ(iHandle,'(A)') N2file
          READ(iHandle,'(E11.3,/,E11.3)') n2min,n2max
          READ(iHandle,'(E11.3,/,E11.3)') zzmin,zzmax
          READ(iHandle,'(I3,/,I3)') nc,nn
          WRITE(6,*) 'File contains nn=',nn,' points of (n2,z) ...'
          IF (nn .GT. nmax) THEN
            WRITE(6,*) 'too many points for n2 and zz'
          ELSE
            DO i=1,nn
              READ(iHandle,'(2E11.3)') n2(i), zz(i)
            ENDDO
          ENDIF
          CALL CloseFile(iHandle)
        ENDIF
C==Prompt to determine setting of boundary conditions
        IF (Mode .EQ. 'U') THEN
          WRITE(6,*)'Bndy cond: fld=0 at Left, Right, or on Average'
          Call PressOneOf(SetBndyConds,'ALR',1)
        ELSE
          WRITE(6,*)'Bndy cond: fld=0 at Bottom, Top, or on Average'
          Call PressOneOf(SetBndyConds,'ABT',1)
       ENDIF
      ENDIF

C=========================================================================
C= Now define coefficients used to scale from observed vertical          =
C= motion of grid to change in N^2 in tank.                              =
C=    Delta N^2 = coefdelz * Delta Zgrid                                 =
```

```
C=    ScaleTG calculates world co-ordinates on Schlieren grid from world  =
C=        co-ordinates given by grid placed in centre of tank.            =
C=                                                                         =
C= Delta N^2 gives the vertical gradient of density fluctuation which,    =
C= in turn, gives the gradient of the vertical displacement, Delta Ztank. =
C=    Rho' = (d(Rho_bkgd)/dz) Delta Ztank                                 =
C=         = -(Rho_0/G) (N_bkgd)^2 Delta Ztank.                           =
C=    d(Delta Ztank)/dz = 1/(N_bkgd)^2   Delta N^2                        =
C==========================================================================
      IF (Mode .EQ. 'S' .OR. Mode .EQ. 'T') THEN
        ScaleTG = 0.0
      ELSE
C==Prompt to change lengths and indices of refraction
        WRITE(6,*)'What is width of tank (in cm)?'
        CALL AcceptReal(Ltank,0.0,100.0,2)
        WRITE(6,*)'What is distance (in cm) from tank to grid?'
        CALL AcceptReal(Lgrid,0.0,100.0,2)
        WRITE(6,*)'What is distance (in cm) from camera to tank?'
        CALL AcceptReal(Lcam,0.0,500.0,2)
        WRITE(6,*)'What is horiz. posn. (in tank co-ords) of camera?'
        CALL AcceptReal(PyCam,-50.0,50.0,2)
        WRITE(6,*)'What is vertical posn. (in tank co-ords) of camera?'
        CALL AcceptReal(PzCam,0.0,40.0,2)

        Lpg = Lperspex/Nperspex + Lgrid/Nair
        coefdelz = -1.0/(Nw2Nsqr*(0.5*Ltank*Ltank + Nwater*Lpg*Ltank))
        ScaleTG = 0.5*(Ltank/Nwater + Lperspex/Nperspex)*Nair
        ScaleTG = (ScaleTG + Lgrid)/(ScaleTG + Lcam)

C==Allow user to reset minimum tolerance to reduce noise
      WRITE(6,*) 'Min. intensity diff. between upper and lower pixels?'
      CALL AcceptInteger(iMinTol,1,255,2)

C==Prompt to set min and max field values between intensities of 1 and 255
        WRITE(6,*)'I=1 is what min. field value?'
        CALL AcceptReal(minfld,-100.0,100.0,2)
        WRITE(6,*)'I=255 is what max. field value?'
        CALL AcceptReal(maxfld,-100.0,100.0,2)
      ENDIF

C==Get the output buffer
      WRITE(6,*)'Put result in which buffer?'
      CALL AcceptBuffer(iWant3,1)
```

```
      IF (iWant3 .EQ. iWant1 .OR. iWant3 .EQ. iWant2) THEN
        CALL ERROR('This buffer is one of the grid buffers!!')
      ENDIF

C============================================================================
C=  Whew!  At last, the MAIN LOOP                                           =
C============================================================================

      NotDone = .TRUE.
      DO WHILE (NotDone)
        maxfldi = 0.0
        minfldi = 0.0
        CALL EraseBuffer(iWant3)
        CALL MoveBufferToMemory(iMemory,iWant1)

        IF (Mode .EQ. 'U') THEN
C==========================================================================
C= Find horizontal velocity field                                         =
C==========================================================================
          DO i=iw0,iw1
            CALL CheckIfEscape
            IF (EscapeNotPressed) THEN
              CALL GetLineFromBuffer(iWant2,i,iRow)

C==Find vertical displacement of grid and field as reqd by Mode
              iz=0
              DO j=jw0,jw1
                CALL GetColumnFromBuffer(iWant2,j,jCol)
                CALL GetIFromVector(i,jCol(0),iw0,iw1,Irn,Ir,Irp)
                CALL CheckMinTol(Irn,Ir,Irp,i,iw0,iw1,iMinTol,Id)
                IF (Id .GT. 0) THEN
                  CALL GetIFromMatrix(i,j,iw0,iw1,In,I0,Ip)
                  CALL CheckMinTol(In,I0,Ip,i,iw0,iw1,iMinTol,Id)
                  IF (Id .GT. 0) THEN
                    CALL GetDelZ(Ir,In,I0,Ip,i,j,iw0,iw1, yw0,zw0,
     &                           PyCam,PzCam,ScaleTG, delz,Id)
                  ENDIF
                ENDIF
                iRow2(j) = Id

                IF (Id .GT. 0) THEN
                  iz = iz+1
                  CALL INTERPL(zz(1),n2(1),nn,zw0,n2zw)
```

```
                  z(iz) = yw0
                  dfzdz(iz) = -coefdelz*delz/(n2zw*dt)
                ENDIF
              ENDDO
              nz = iz

C==Now integrate dfzdz to get fz.  fz(1)=0.0 by default
              CALL IntegrateWithBCs(z(1),dfzdz(1),nz,fz(1),SetBndyConds)

C==Finally, return scaled fz at all points where Id>0
              CALL RescaleIfNotZero(iRow2(0),jw0,jw1,fz(1),nz,
     &                 minfld,maxfld,minfldi,maxfldi)

              CALL PutLineInBuffer(iWant3,i,iRow2)
            ENDIF
          ENDDO
        ELSE
C=========================================================================
C= Find vertical displacement of grid and field as required by Mode     =
C=========================================================================
          DO j=jw0,jw1
            CALL CheckIfEscape
            IF (EscapeNotPressed) THEN
              CALL GetColumnFromBuffer(iWant2,j,jCol)
              iz=0
              DO i=iw1,iw0,-1
                CALL GetIFromVector(i,jCol(0),iw0,iw1,Irn,Ir,Irp)
                CALL CheckMinTol(Irn,Ir,Irp,i,iw0,iw1,iMinTol,Id)
                IF (Id .GT. 0) THEN
                  CALL GetIFromMatrix(i,j,iw0,iw1,In,I0,Ip)
                  CALL CheckMinTol(In,I0,Ip,i,iw0,iw1,iMinTol,Id)
                  IF (Id .GT. 0) THEN
                    CALL GetDelZ(Ir,In,I0,Ip,i,j,iw0,iw1, yw0,zw0,
     &                           PyCam,PzCam,ScaleTG, delz,Id)
                  ENDIF
                ENDIF
                jCol2(i) = Id

                IF (Mode .NE. 'S' .AND. Id .GT. 0) THEN
                  iz = iz+1
                  IF (Mode .EQ. 'G' .OR. Mode .EQ. 'T') THEN
                    fz(iz) = delz
                  ELSEIF (Mode .EQ. 'N') THEN
```

```
                fz(iz) = coefdelz*delz
              ELSEIF (Mode .EQ. 'P') THEN
                fz(iz) = coefdelz*delz/dt
              ELSEIF (Mode .EQ. 'R') THEN
                z(iz) = zw0
                dfzdz(iz) = -coefdelz*Rho0OverG*delz
              ELSE
                CALL INTERPL(zz(1),n2(1),nn,zw0,n2zw)
                z(iz) = zw0
                IF (Mode .EQ. 'Z') THEN
                  dfzdz(iz) = coefdelz*delz/n2zw
                ELSE
                  dfzdz(iz) = coefdelz*delz/(n2zw*dt)
                ENDIF
              ENDIF
            ENDIF
          ENDDO
          nz = iz

C==Now integrate dfzdz to get fz.  fz(1)=0.0 by default
          IF (Mode .EQ. 'R' .OR. Mode .EQ. 'W' .OR.
     &        Mode .EQ. 'Z') THEN
            CALL IntegrateWithBCs(z(1),dfzdz(1),nz,fz(1),
     &                SetBndyConds)
          ENDIF

C==Finally, return fz at all points where Id>0
          IF (Mode .NE. 'S') THEN
            CALL RescaleIfNotZero(jCol2(0),iw1,iw0,fz(1),nz,
     &                minfld,maxfld,minfldi,maxfldi)
          ENDIF

          CALL PutColumnInBuffer(iWant3,j,jCol2)
        ENDIF
      ENDDO
    ENDIF

    CALL ShowBuffer(iWant3,iOnboard,.FALSE.)
    WRITE(6,*) 'Actual Field range:',minfldi,'... ',maxfldi
    WRITE(6,*) 'Is scaling from Field -> Intensity acceptable?'
    CALL PressOneOf(AcptScaling,'NY',1)

    IF (AcptScaling .EQ. 'N') THEN
```

```
            WRITE(6,*)'I=1 is what min. field value?'
            CALL AcceptReal(minfld,-100.0,100.0,2)
            WRITE(6,*)'I=255 is what max. field value?'
            CALL AcceptReal(maxfld,-100.0,100.0,2)
          ENDIF

          WRITE(6,*) 'Accept Min. inten. diff. of surrounding pixels?'
          CALL PressOneOf(AcptMinTol,'NY',1)

          IF (AcptMinTol .EQ. 'N') THEN
            WRITE(6,*)'What is new minimum intensity?'
            CALL AcceptInteger(iMinTol,0,255,2)
          ENDIF

          IF (AcptMinTol .EQ. 'Y' .AND. AcptScaling .EQ. 'Y') THEN
            NotDone = .FALSE.
          ENDIF
        ENDDO

        RETURN
        END
```

SchlierenSpaceTime

```
C*********************************************************************
C* SchlierenSpaceTime                                                *
C*   This subroutine takes a space time image from a schlieren grid  *
C*   experiment and works out space-time fields of Delta N^2,        *
C*   Delta Ztank, or vertical velocity.                              *
C*********************************************************************
      SUBROUTINE  SchlierenSpaceTime(Mode)
$INCLUDE:'All.INC'
$INCLUDE:'WorkSpce.INC'
$INCLUDE:'Configur.INC'
C==Define and dimension variables
      LOGICAL NotDone,iError
      INTEGER iMinTol
      INTEGER*1 jCol,jCol2,iRow,iRow2,iRC1n,iRC1p,iRCn,iRCp,iRC0
      INTEGER*2 iWant1,iWant2,iWant2n,iWant2p,iWant3,iHandle,iOnboard
      INTEGER*2 iw0,iw1,jw0,jw1,i,j,ixy
      INTEGER*2 imin,imax,jmin,jmn,jmx,jj,ii,iz,nz
      INTEGER*2 Ir,Irn,Irp,Id,I0,In,Ip
      INTEGER*2 nn,nc,nmax,ndt
      REAL Ltank,Lperspex,Lgrid,Lcam,nwater,nperspex,nair,nw2nsqr
```

```
      REAL PyCam,PzCam,ScaleTG,Lpg,coefdelz,Rho0OverG
      REAL n2min,n2max,zzmin,zzmax,yw0,zw0,n2zw,delz
      REAL minfld,maxfld,minfldi,maxfldi,dt
      REAL fz,z,dfzdz,n2,zz
      CHARACTER Mode,SpaceAxis,AcptScaling,AcptMinTol,SetBndyConds
      CHARACTER N2File*(20)

      PARAMETER (nmax=256)
      DIMENSION jCol(0:511),jCol2(0:511),iRow(0:511),iRow2(0:511)
      DIMENSION iRC1n(0:511),iRC1p(0:511),iRCn(0:511),iRCp(0:511)
      dimension iRC0(0:511)
      DIMENSION fz(nmax),z(nmax),dfzdz(nmax),n2(nmax),zz(nmax)
      EQUIVALENCE (iWork1(0),jCol(0))
      EQUIVALENCE (iWork1(512),jCol2(0))
      EQUIVALENCE (iWork1(1024),iRow(0))
      EQUIVALENCE (iWork1(1536),iRow2(0))
      EQUIVALENCE (iWork1(2048),iRC1n(0))
      EQUIVALENCE (iWork1(2560),iRC1p(0))
      EQUIVALENCE (iWork1(3072),iRCn(0))
      EQUIVALENCE (iWork1(3584),iRCp(0))
      EQUIVALENCE (iWork1(4096),iRC0(0))
      EQUIVALENCE (rWork1(0),fz(1))
      EQUIVALENCE (rWork1(256),z(1))
      EQUIVALENCE (rWork1(512),dfzdz(1))
      EQUIVALENCE (rWork1(768),zz(1))
      EQUIVALENCE (rWork1(1024),n2(1))

C==Set default options
      DATA ixy /0/, iMinTol /10/
      DATA minfld,maxfld /-0.1,0.1/, dt,ndt /0.1, 1/
      DATA N2File /'n2******.xpt'/
C==default lengths in cm: span of tank, perspex, and distance to grid
      DATA Ltank /20.3/, Lperspex /1.1/, Lgrid /34.2/, Lcam /343./
C==position of camera in grid co-ordinates
      DATA PyCam /0.0/, PzCam /13.0/
C==indices of refraction for water, perspex and air
      DATA nwater /1.3330/, nperspex /1.49/, nair /1.0000/
C==scaling from index of refraction gradient to N^2  (in s^2/cm)
C==  (d(nw)/dz)/nw = -nw2nsqr*N^2
      DATA nw2nsqr /0.0001878/
C==Density of water divided by accn of gravity (in g s^2/cm^4)
      DATA Rho0OverG /0.001019/
C
```

```
C      CALL SetCurrentHelpFile('Norminty.F01',' ')
C
C==Check that option passed to this routine is valid
      if (Mode .eq. 'G') then
         write(6,*) 'S-T plot: Find vertical disp. of grid lines'
      elseif (Mode .eq. 'N') then
         write(6,*) 'S-T plot: Find change in N^2 in tank'
      ELSEIF (Mode .EQ. 'P') THEN
        WRITE(6,*) 'S-T plot: Calculate time derivative of Delta N^2'
      ELSEIF (Mode .EQ. 'R') THEN
        WRITE(6,*) 'S-T plot: Calculate flucutation density field'
      ELSEIF (Mode .EQ. 'S') THEN
        WRITE(6,*) 'S-T plot: Calculate relative shift in lines'
      ELSEIF (Mode .EQ. 'T') THEN
        WRITE(6,*) 'S-T plot: Find line displc. in tank world co-ords'
      elseif (Mode .eq. 'U') then
         write(6,*) 'S-T plot: Find horizontal velocity in tank'
      elseif (Mode .eq. 'W') then
         write(6,*) 'S-T plot: Find vertical velocity in tank'
      elseif (Mode .eq. 'Z') then
         write(6,*) 'S-T plot: Find vertical disp. in tank'
      else
         write(6,*) 'Option passed to SchlierenSpaceTime is invalid'
         write(6,*) 'You are a TeFal-head'
         return
      endif

C==Get the buffer containing initial picture of a grid
      IF (Mode .NE.'P' .AND. Mode .NE.'U' .AND. Mode .NE.'W') THEN
        WRITE(6,*)'What is buffer containing initial image?'
        CALL AcceptBuffer(iWant1,1)
        if (EscapePressed) then
          return
        endif
        CALL ShowBuffer(iWant1,iOnboard,.FALSE.)
      ENDIF

C==Get the buffer containing space-time plot
      WRITE(6,*)'What is buffer containing space-time image?'
      CALL AcceptBuffer(iWant2,1)
      if (EscapePressed) then
        return
      endif
```

```
      CALL ShowBuffer(iWant2,iOnboard,.FALSE.)
      CALL WindowOrScreen(iWant2,iw0,iw1,jw0,jw1,.TRUE.,' ',' ')

C==Determine dir'n of space axis on space-time plot and see if Mode is valid
      write(6,*)'Is space axis horizontal or vertical?'
      CALL PressOneOf(SpaceAxis,'HV',1)
      IF (SpaceAxis .EQ. 'V' .AND. Mode .EQ. 'U') then
        call Warning('Cannot find U from vertical space axis')
        RETURN
      else IF (SpaceAxis .EQ. 'H' .AND. (Mode .EQ. 'R'
     &    .OR. Mode .EQ. 'Z' .OR.  Mode .EQ. 'W')) THEN
        call Warning('Cannot find Rho/delZ/W from horiz. space axis')
        return
      ENDIF

C==Get vertical pixel position of horizontal cut
      if (SpaceAxis .eq. 'V') then
         WRITE(6,*)'What is horizontal pixel co-ord. of vertical slice?'
      else
         WRITE(6,*)'What is vertical pixel co-ord. of horizontal slice?'
      endif
      CALL AcceptInteger(ixy,0,511,0)

C==For U, need to have space-time plots of upper and lower pixels too
      IF (Mode .EQ. 'U') THEN
        WRITE(6,*)'Buffer of space-time image at vert. pixel ',ixy-1
        CALL AcceptBuffer(iWant2n,1)
        WRITE(6,*)'Buffer of space-time image at vert. pixel ',ixy+1
        CALL AcceptBuffer(iWant2p,1)
      endif

C==Get time difference between buffers
      IF (Mode .EQ. 'P' .OR. Mode .EQ. 'U' .OR. Mode .EQ. 'W') THEN
        WRITE(6,*)'What is time difference, dt, between pixels?'
        CALL AcceptReal(dt,0.01,10.0,2)
        WRITE(6,*)'Take time derivative across how many dt?'
        CALL AcceptInteger(ndt,1,10,2)
      ENDIF

C==Get background N^2 profile if determining delz or delw
      IF (Mode .EQ. 'U' .OR. Mode .EQ. 'Z' .OR. Mode .EQ. 'W') THEN
        WRITE(6,*)'For rho->Delta Z conversion: XPlot file of N^2(z)'
        CALL AcceptString(N2File,2)
```

```
        CALL OpenFile(iHandle,' ',N2file,
     &       'old','formatted','sequential',iError)
        IF (iError .OR. iHandle .EQ. 0) THEN
          CALL Warning('Cannot open specified N2 file')
          RETURN
        ELSE
          READ(iHandle,'(A)') N2File
          READ(iHandle,'(E11.3,/,E11.3)') n2min,n2max
          READ(iHandle,'(E11.3,/,E11.3)') zzmin,zzmax
          READ(iHandle,'(I3,/,I3)') nc,nn
          WRITE(6,*) 'File contains nn=',nn,' points of (n2,z) ...'
          IF (nn .GT. nmax) THEN
            WRITE(6,*) 'too many points for n2 and zz'
          ELSE
            DO i=1,nn
              READ(iHandle,'(2E11.3)') n2(i), zz(i)
            ENDDO
          ENDIF
          CALL CloseFile(iHandle)
        ENDIF
C==Prompt to determine setting of boundary conditions
        IF (Mode .EQ. 'U') THEN
          WRITE(6,*)'Zero bndy. cond. at Left, Right, Average'
          Call PressOneOf(SetBndyConds,'ALR',1)
        ELSE
          WRITE(6,*)'Zero bndy. cond. at Bottom, Top, Average'
          Call PressOneOf(SetBndyConds,'ABT',1)
        ENDIF

      ENDIF

C============================================================================
C= Now define coefficients used to scale from observed vertical            =
C= motion of grid to change in N^2 in tank.                                =
C=    Delta N^2 = coefdelz * Delta Zgrid                                   =
C=    ScaleTG calculates world co-ordinates on Schlieren grid from world  =
C=        co-ordinates given by grid placed in centre of tank.             =
C=                                                                         =
C= Delta N^2 gives the vertical gradient of density fluctuation which,    =
C= in turn, gives the gradient of the vertical displacement, Delta Ztank. =
C=    Rho' = (d(Rho_bkgd)/dz) Delta Ztank                                  =
C=         = -(Rho_0/G) (N_bkgd)^2 Delta Ztank.                            =
```

```
C=    d(Delta Ztank)/dz = 1/(N_bkgd)^2   Delta N^2                         =
C==========================================================================

      IF (Mode .EQ. 'S' .OR. Mode .EQ. 'T') THEN
        ScaleTG = 0.0
      ELSE
C==Prompt to change lengths and indices of refraction
        WRITE(6,*)'What is width of tank (in cm)?'
        CALL AcceptReal(Ltank,0.0,100.0,2)
        WRITE(6,*)'What is distance (in cm) from tank to  grid?'
        CALL AcceptReal(Lgrid,0.0,100.0,2)
        WRITE(6,*)'What is distance (in cm) from camera to tank?'
        CALL AcceptReal(Lcam,0.0,500.0,2)
        WRITE(6,*)'What is horiz. posn. (in tank co-ords) of camera?'
        CALL AcceptReal(PyCam,-50.0,50.0,2)
        WRITE(6,*)'What is vertical posn. (in tank co-ords) of camera?'
        CALL AcceptReal(PzCam,0.0,40.0,2)

        Lpg = Lperspex/Nperspex + Lgrid/Nair
        coefdelz = -1.0/(Nw2Nsqr*(0.5*Ltank*Ltank + Nwater*Lpg*Ltank))
        ScaleTG = 0.5*(Ltank/Nwater + Lperspex/Nperspex)*Nair
        ScaleTG = (ScaleTG + Lgrid)/(ScaleTG + Lcam)
      ENDIF

C==Allow user to reset minimum tolerance to reduce noise
      WRITE(6,*)'Min. intensity diff. between upper and lower pixels?'
      CALL AcceptInteger(iMinTol,1,255,2)

C==Prompt to scale from min/max field to intensities between 1 and 255
      WRITE(6,*)'I=1 is what min. field value?'
      CALL AcceptReal(minfld,-100.0,100.0,2)
      WRITE(6,*)'I=255 is what max. field value?'
      CALL AcceptReal(maxfld,-100.0,100.0,2)

C==Get the output buffer
      WRITE(6,*)'Put result in which buffer?'
      CALL AcceptBuffer(iWant3,1)
      if (EscapePressed) then
         RETURN
      ENDIF
      IF (iWant3 .EQ. iWant2 .OR. iWant3 .EQ. iWant1) THEN
        CALL Warning('Warning this buffer is one of the grid buffers!!')
        RETURN
```

```
      ENDIF
C===========================================================================
C=  Whew!  At last, the MAIN LOOP                                           =
C===========================================================================

      NotDone = .TRUE.
      DO WHILE (NotDone)
        minfldi = 0.0
        maxfldi = 0.0
        CALL EraseBuffer(iWant3)
        IF (Mode .NE.'P' .AND. Mode .NE.'U' .AND. Mode .NE.'W') THEN
          CALL MoveBufferToMemory(iMemory,iWant1)
        ENDIF

        IF (Mode .EQ. 'U') THEN
C========================================================================
C= Find horizontal velocity field                                       =
C= Note, "0,512" passed to CheckMinTol ensures i is not at upper or     =
C= lower boundary.                                                      =
C========================================================================
          DO j=jw0,jw1
            iRow2(j)=0
          enddo
          CALL PutLineInBuffer(iWant3,iw0,iRow2)

          DO i=iw0+1,iw1
            CALL CheckIfEscape
            IF (EscapeNotPressed) THEN
              CALL GetLineFromBuffer(iWant2,i,iRow)
              CALL GetLineFromBuffer(iWant2n,i,iRC1n)
              CALL GetLineFromBuffer(iWant2p,i,iRC1p)
              CALL GetLineFromBuffer(iWant2,i-1,iRC0)
              CALL GetLineFromBuffer(iWant2n,i-1,iRCn)
              CALL GetLineFromBuffer(iWant2p,i-1,iRCp)

              iz=0
              DO j=jw0,jw1
                Ir = I2fromI1(iRow(j))
                Irn = I2fromI1(iRC1n(j))
                Irp = I2fromI1(iRC1p(j))
                CALL CheckMinTol(Irn,Ir,Irp,i,0,512,iMinTol,Id)

                IF (Id .GT. 0) THEN
```

```
                  I0 = I2fromI1(iRC0(j))
                  In = I2fromI1(iRCn(j))
                  Ip = I2fromI1(iRCp(j))
                  CALL CheckMinTol(In,I0,Ip,i,0,512,iMinTol,Id)
                  IF (Id .GT. 0) THEN
                    CALL GetDelZ(Ir,In,I0,Ip,ixy,j,0,512, yw0,zw0,
     &                          PyCam,PzCam,ScaleTG, delz,Id)
                  ENDIF
                ENDIF
                iRow2(j) = Id

                IF (Id .GT. 0) THEN
                  iz = iz+1
                  CALL INTERPL(zz(1),n2(1),nn,zw0,n2zw)
                  z(iz) = yw0
                  dfzdz(iz) = -coefdelz*delz/(n2zw*dt)
                ENDIF
              ENDDO
              nz = iz

C==Now integrate dfzdz to get fz.  fz(1)=0.0 by default
              CALL IntegrateWithBCs(z(1),dfzdz(1),nz,fz(1),SetBndyConds)

C==Finally, return scaled fz at all points where Id>0
              CALL RescaleIfNotZero(iRow2(0),jw0,jw1,fz(1),nz,
     &                 minfld,maxfld,minfldi,maxfldi)

              CALL PutLineInBuffer(iWant3,i,iRow2)
            ENDIF
          ENDDO
        ELSE
C=======================================================================
C= Find vertical displacement of grid and field as required by Mode     =
C=======================================================================
          IF (Mode .EQ. 'P' .OR. Mode .EQ. 'W') THEN
            DO i=iw0,iw1
              jCol2(i)=0
            enddo
            CALL PutColumnInBuffer(iWant3,jw0,jCol2)
            jmin = jw0+1
          ELSE
            jmin = jw0
          ENDIF
```

```
          DO j=jmin,jw1
            CALL CheckIfEscape
            If (EscapeNotPressed) THEN
              IF (Mode .EQ. 'P' .OR. Mode .EQ. 'W') THEN
                jmn = MAX(jw0, INT(REAL(j)-0.5*ndt) )
                jmx = min(jw1, INT(REAL(j)+0.5*ndt) )
                CALL GetColumnFromBuffer(iWant2,jmn,iRC0)
                CALL GetColumnFromBuffer(iWant2,jmx,jCol)
c                WRITE(6,*)j,jmn,jmx
c                call PressAnyKeyToContinue
              else
                CALL GetColumnFromBuffer(iWant2,j,jCol)
              ENDIF

              iz=0
              DO i=iw1,iw0,-1
                IF (SpaceAxis .EQ. 'H') THEN
                  Ir = I2fromI1(jCol(i))
                  Id = Ir
                  ii = ixy
                  jj = j
                  imin=-1
                  imax=512
                ELSE
                  CALL GetIFromVector(i,jCol(0),iw0,iw1,Irn,Ir,Irp)
                  CALL CheckMinTol(Irn,Ir,Irp,i,iw0,iw1,iMinTol,Id)
                  ii = i
                  jj = ixy
                  imin = iw0
                  imax = iw1
                ENDIF

                IF (Id .GT. 0) THEN
                  IF (Mode .EQ. 'P' .OR. Mode .EQ. 'W') THEN
                    CALL GetIFromVector(i,iRC0(0),iw0,iw1,In,I0,Ip)
                  ELSE
                    CALL GetIFromMatrix(ii,jj,imin,imax,In,I0,Ip)
                  ENDIF
                  CALL CheckMinTol(In,I0,Ip,ii,imin,imax,iMinTol,Id)
                  IF (Id .GT. 0) THEN
                    CALL GetDelZ(Ir,In,I0,Ip,ii,jj,imin,imax, yw0,zw0,
     &                           PyCam,PzCam,ScaleTG, delz,Id)
```

```
                ENDIF
              ENDIF
              jCol2(i) = Id

              IF (Mode .NE. 'S' .AND. Id .GT. 0) THEN
                iz = iz+1
                IF (Mode .EQ. 'G') THEN
                  fz(iz) = delz
                ELSEIF (Mode .EQ. 'N') THEN
                  fz(iz) = coefdelz*delz
                ELSEIF (Mode .EQ. 'P') THEN
                  fz(iz) = coefdelz*delz/(ndt*dt)
                ELSEIF (Mode .EQ. 'R') THEN
                  z(iz) = zw0
                  dfzdz(iz) = -coefdelz*Rho0OverG*delz
                ELSE
                  CALL INTERPL(zz(1),n2(1),nn,zw0,n2zw)
                  z(iz) = zw0
                  IF (Mode .EQ. 'Z') THEN
                    dfzdz(iz) = coefdelz*delz/n2zw
                  ELSE
                    dfzdz(iz) = coefdelz*delz/(n2zw*ndt*dt)
                  ENDIF
                ENDIF
              ENDIF
            ENDDO
            nz = iz

C==Now integrate dfzdz to get fz.  fz(1)=0.0 by default
            IF (Mode .EQ. 'R' .OR. Mode .EQ. 'W' .OR.
     &          Mode .EQ. 'Z') THEN
              CALL IntegrateWithBCs(z(1),dfzdz(1),nz,fz(1),
     &                    SetBndyConds)
            ENDIF

C==Finally, return fz at all points where Id>0
            IF (Mode .NE. 'S') THEN
              CALL RescaleIfNotZero(jCol2(0),iw1,iw0,fz(1),nz,
     &                  minfld,maxfld,minfldi,maxfldi)
            ENDIF

            CALL PutColumnInBuffer(iWant3,j,jCol2)
          ENDIF
```

```
          ENDDO
        ENDIF

        CALL ShowBuffer(iWant3,iOnboard,.FALSE.)
        WRITE(6,*) 'Actual Field range:',minfldi,' ... ',maxfldi
        WRITE(6,*) 'Is scaling from Field -> Intensity acceptable?'
        CALL PressOneOf(AcptScaling,'NY',1)

        IF (AcptScaling .EQ. 'N') THEN
           WRITE(6,*)'I=1 is what min. field value?'
           CALL AcceptReal(minfld,-100.0,100.0,2)
           WRITE(6,*)'I=255 is what max. field value?'
           CALL AcceptReal(maxfld,-100.0,100.0,2)
        ENDIF

        WRITE(6,*) 'Accept Min. inten. diff. of surrounding pixels?'
        CALL PressOneOf(AcptMinTol,'NY',1)

        IF (AcptMinTol .EQ. 'N') THEN
           WRITE(6,*)'What is new minimum intensity?'
           CALL AcceptInteger(iMinTol,0,255,2)
        ENDIF

        IF (AcptMinTol .EQ. 'Y' .AND. AcptScaling .EQ. 'Y') THEN
           NotDone = .FALSE.
        ENDIF
      ENDDO

      CALL ShowBuffer(iWant3,iOnboard,.FALSE.)
      RETURN
      END
```

GetDelZ

```
C*************************************************************************
C* GetDelZ                                                               *
C*   This subroutine takes an intensity Ir, compares it with the         *
C*   upper, lower, and middle intensities of the background grid         *
C*   and returns the interpolated displacement in terms of an            *
C*   intensity Id (255 for one pixel up, 1 for one pixel down) and       *
C*   physical displacement, delz calculated from world co-ordinates.     *
C*   NB: Id=0 if interpolation is inaccurate as determined by iMinTol.   *
C*   Background grid is stored in iMemory(j,i).                          *
C*************************************************************************
```

```
      SUBROUTINE GetDelZ(Ir,In,I0,Ip,i,j,imin,imax,
     &                   yw0,zw0,y0,z0,scltg,delz,Id)
$INCLUDE:'All.INC'
$INCLUDE:'WorkSpce.INC'
$INCLUDE:'Configur.INC'
C==Declare variables
      INTEGER*2 i,j,imin,imax,Id,Ip,I0,In,Ir
      REAL delz,yw,yw0,zwp,zw0,zwn
      REAL y0,z0,scltg
      REAL t1,t2,t3

      CALL Map2DWorldCoordinates(REAL(i),REAL(j),yw0,zw0)
      zw0 = zw0 + scltg*(zw0-z0)

      IF (i .EQ. imin) THEN
C==Perform linear interpolation near top boundary using i=0,i=1
        CALL Map2DWorldCoordinates(REAL(i+1),REAL(j),yw,zwn)
        zwn = zwn + scltg*(zwn-z0)

        t1=REAL(Ir-I0)/REAL(Ip-I0)
        delz = (zwn-zw0)*t1
        Id=MIN(255,MAX(1,INT(-127*t1+128.0)))
      ELSEIF (i .EQ. imax) THEN
C==Perform linear interpolation near bottom boundary using i=510,i=511
        CALL map2DWorldCoordinates(REAL(i-1),REAL(j),yw,zwp)
        zwp = zwp + scltg*(zwp-z0)

        t1=REAL(Ir-I0)/REAL(In-I0)
        delz = (zwp-zw0)*t1
        Id=MIN(255,MAX(1,INT(127*t1+128.0)))
      ELSE
C==Perform quadratic interpolation between i-1 (255), i (128), i+1 (1)
        CALL map2DWorldCoordinates(REAL(i-1),REAL(j),yw,zwp)
        zwp = zwp + scltg*(zwp-z0)
        CALL map2DWorldCoordinates(REAL(i+1),REAL(j),yw,zwn)
        zwn = zwn + scltg*(zwn-z0)

        t1 = REAL((Ir-I0)*(Ir-Ip))/REAL((In-I0)*(In-Ip))
        t2 = REAL((Ir-In)*(Ir-Ip))/REAL((I0-In)*(I0-Ip))
        t3 = REAL((Ir-I0)*(Ir-In))/REAL((Ip-I0)*(Ip-In))
        delz = (zwp-zw0)*t1 + (zwn-zw0)*t3
        Id = MIN(255,MAX(1,INT(255*t1+128*t2+t3)))
      ENDIF
```

```
      RETURN
      END
```

CheckMinTol

```
C*******************************************************************
C* CheckMinTol                                                      *
C*   This subroutine checks whether the intensity difference        *
C*   between three consecutive pixels is sufficiently large and the *
C*   gradient uniform to perform interpolation routines later on.   *
C*******************************************************************
      SUBROUTINE CheckMinTol(In,I0,Ip,i,imin,imax,iMinTol,Id)
$INCLUDE:'All.INC'
$INCLUDE:'WorkSpce.INC'
$INCLUDE:'Configur.INC'
C==Declare variables
      INTEGER*2 imin,imax,i,Ip,I0,In,iMinTol,Id

      Id = I0
      IF (i .EQ. imin) THEN
        IF (ABS(Ip-I0) .LT. iMinTol) THEN
          Id=0
        ENDIF
      ELSEIF (i .EQ. imax) THEN
        IF (ABS(In-I0) .LT. iMinTol) THEN
          Id=0
        ENDIF
      ELSE
        IF ( (Ip-I0)*(I0-In) .LE. 0 .OR.
     &       ABS(In-I0) .LT. iMinTol .OR.
     &       ABS(Ip-I0) .LT. iMinTol     ) THEN
          Id=0
        ENDIF
      ENDIF

      RETURN
      END
```

RescaleIfNotZero

```
C*******************************************************************
C* RescaleIfNotZero                                                 *
C*   This subroutine replaced intensities in a vector with rescaled *
```

```
C*   values function scaled between 1 and 255.  No scaling occurs if   *
C*   the intensity in the vector is zero to begin with.                *
C*                                                                     *
C*   Note: f(iz) is ordered from 1 to nz corresponding to points       *
C*   from imn to imx.  If imn>imx then have to loop backwards.         *
C***********************************************************************
      SUBROUTINE RescaleIfNotZero(iVec,imn,imx,f,n,
     &              sclmn,sclmx,fmn,fmx)
$INCLUDE:'All.INC'
$INCLUDE:'WorkSpce.INC'
$INCLUDE:'Configur.INC'
C==Declare variables
      INTEGER*1 iVec(0:511)
      INTEGER*2 n,imn,imx,iz,i,Id
      REAL f(n),sclmn,sclmx,fmn,fmx

      iz=0
      if (imn .le. imx) then
        DO i=imn,imx
          IF (I2fromI1(iVec(i)) .GT. 0) THEN
            iz = iz+1
            Id = INT(254*(f(iz)-sclmn)/(sclmx-sclmn)+1.0)
            Id = MAX(1,MIN(255,Id))
            iVec(i) = Id
            fmx = MAX(f(iz),fmx)
            fmn = MIN(f(iz),fmn)
          ENDIF
        ENDDO
      else
        DO i=imn,imx,-1
          IF (I2fromI1(iVec(i)) .GT. 0) THEN
            iz = iz+1
            Id = INT(254*(f(iz)-sclmn)/(sclmx-sclmn)+1.0)
            Id = MAX(1,MIN(255,Id))
            iVec(i) = Id
            fmx = MAX(f(iz),fmx)
            fmn = MIN(f(iz),fmn)
          ENDIF
        ENDDO
      endif

      RETURN
      END
```

B.2 Source Code for Arbitrary Images

AnalyseSchlieren

```
C
C*******************************************************************
C* AnalyseSchlieren                  AnalyseSchlieren_Do           *
C*    Calculate the optimal shift for the entire array of windows.   *
C*        Par                        The control parameters.
C*******************************************************************
      subroutine AnalyseSchlieren_Do(Par)
C=====Parameters
      type (D_AnalyseSchlieren) Par
C=====Local variables
      type (F_Window) Window,TestWindow
      integer (4) i,j,ii,jj,ix,jy,nx,ny,k,i0,i1,j0,j1,n
      integer (4) iCoarse,jCoarse,iLastPass,jLastPass,iPass,iSubPix
      integer (4) iw0,iw1,jw0,jw1
      type (F_Location) Shift
      type (F_Location), Allocatable :: Location(:,:)
      type (F_WLocation), Allocatable :: Displace(:,:)
      real, Allocatable :: Density(:,:)
      real  xMin,xMax,yMin,yMax,Step,Mean,xMean,yMean
      real  a,b,d,x,y
      real u,v,u2,v2,uv,us,vs
      integer nPoints
      type (F_Image) Grad
      type (F_View)  DispView
      type (F_LUT) LUT
      type (F_LeastSquares) LSBase,LS
      type (F_ImageStatistics) Stats
      logical Again,OK,InterpolatedGuess
C=====Open source images
      call OpenImage(Par%Back)
      call OpenImage(Par%Fore)
      if (Par%InWindow%Right .eq. Par%InWindow%Left) then
        Par%InWindow = CurrentWindow(Par%Back)
      endif
C==================================================================
C=    Initialisation                                              =
C==================================================================
      ! Set constants
      nx = (Par%InWindow%Right - Par%InWindow%Left)/Par%iStep
      ny = (Par%InWindow%Top - Par%InWindow%Bottom)/Par%jStep
```

```
      ! Allocate memory for arrays
      allocate(Location(0:nx-1,0:ny-1))
      allocate(Displace(0:nx-1,0:ny-1))
      allocate(Density(0:nx,0:ny))

      ! Setup least squares problem: a + bx + cy + dxy + ex^2 + fy^2
      call Create(LSBase,9,6,1)
      call Create(LS,9,6,1)
      k = 0
      do i=-1,1
        do j=-1,1
          LSBase%A(k,0) = 1.0
          LSBase%A(k,1) = i
          LSBase%A(k,2) = j
          LSBase%A(k,3) = i*j
          LSBase%A(k,4) = i*i
          LSBase%A(k,5) = j*j
          k = k + 1           ! Equation number
        enddo
      enddo

      ! Initialise displacements
      do j=0,ny-1
        do i=0,nx-1
          Displace(i,j)%x = Huge(0.0)
          Displace(i,j)%y = Huge(0.0)
        enddo
      enddo

      ! Select output colour scheme
      call OpenLUT(LUT,'Test.LUT')

C========================================================================
C=    Display Difference images                                         =
C=    This is effectively the qualitative mode output.                  =
C========================================================================
      if (Use(Par%Diff)) then
        call CreateImage(Par%Diff,Width(Par%Back),Height(Par%Back))
        Par%Diff = Par%Back - Par%Fore
        Par%Diff = LUT    ! Set LUT
        call DisplayImage(Par%Diff,DisplayImage$Resize
     &       +DisplayImage$NewView)
```

```
      endif

C=========================================================================
C=    Set-up output image and location arrays                            =
C=========================================================================
      call CreateView(DispView,'Displacement',Width(Par%Back),
     &     Height(Par%Back),255)
      call SetLUT(DispView,LUT)

C=========================================================================
C=    Initialise for quantitative mode scans                             =
C=========================================================================
      ! Vector spacing for first pass
      iCoarse = 4
      jCoarse = 4

      ! Initialise
      iLastPass = nx
      jLastPass = ny
      iPass = 0

C=========================================================================
C=    Quantitative mode scans of image pairs                             =
C=========================================================================
      do while (iCoarse .gt. 0 .and. jCoarse .gt. 0 .and. Continue())
        iPass = iPass + 1
        write(6,*)'Pass:',iPass

C-------------------------------------------------------------------------
C-      Scan through image at required spatial resolution for this pass-
C-------------------------------------------------------------------------
        do j=0,ny-1,jCoarse
          ! Position within image
          jy = Par%InWindow%Bottom + (float(j) + 0.5)*Par%jStep
          j0 = jLastPass*(j/jLastPass)
          j1 = min(j0 + jLastPass,ny-1)
          if (j1 .ne. j0) then
            b = float(j1-j)/float(j1-j0)
          else
            b = 0.0
          endif
          do i=0,nx-1,iCoarse
            ix = Par%InWindow%Left + (float(i) + 0.5)*Par%iStep
```

```
            ! Set up window and test for texture level
            Window%Bottom = max(Par%InWindow%Bottom,jy - Par%ySize/2)
            Window%Top = min(Par%InWindow%Top,jy + Par%ySize/2)
            Window%Left = max(Par%InWindow%Left,ix - Par%xSize/2)
            Window%Right = min(Par%InWindow%Right,ix + Par%xSize/2)
            TestWindow%Bottom = max(Par%InWindow%Bottom,jy-Par%jStep/2)
            TestWindow%Top = min(Par%InWindow%Top,jy + Par%jStep/2)
            TestWindow%Left = max(Par%InWindow%Left,ix - Par%iStep/2)
            TestWindow%Right = min(Par%InWindow%Right,ix + Par%iStep/2)
            Stats = Statistics(Par%Back,Window)
            if (Abort()) then
              ! Do nothing more
            elseif (Stats%StdDev .lt. Par%MinTexture) then
              ! Too little texture - do not pattern match
              Displace(i,j) = F_WLocation(Huge(1.0),Huge(1.0))
            elseif (Contains(Par%Back,Window,0)) then
              ! There is a zero here - do not pattern match
              Displace(i,j) = F_WLocation(Huge(1.0),Huge(1.0))
            elseif (mod(i,iLastPass) .eq. 0 .and.
     &              mod(j,jLastPass) .eq. 0) then
              ! Already have details of this one
            else
C.......................................................................
C.            Pattern match for this location                          .
C.......................................................................
              Location(i,j)%i = ix
              Location(i,j)%j = jy

              ! Initial guess for displacement
              i0 = iLastPass*(i/iLastPass)
              i1 = min(i0 + iLastPass,nx-1)
              if (Displace(i0,j0)%x  .eq. Huge(0.0) .or.
     &            Displace (i1,j0)%x .eq. Huge(0.0) .or.
     &            Displace(i0,j1)%x  .eq. Huge(0.0) .or.
     &            Displace(i1,j1)%x  .eq. Huge(0.0)) then
                ! None of the surrounding points have displacement vectors.
                ! Interpolation not possible, so must start from scratch.
                ! Do integer search.
                Displace(i,j) = F_WLocation(0.0,0.0)
              else
                ! Have information from surrounding displacement vectors
                ! on which an estimate for this vector may be based.
```

```
                ! Use bi-linear interpolation.
                if (i1 .ne. i0) then
                  a = float(i1-i)/float(i1-i0)
                else
                  a = 0.0
                endif
                Displace(i,j) = a*b*Displace(i0,j0)
     &                       + (1.0-a)*b*Displace(i1,j0)
     &                       + a*(1.0-b)*Displace(i0,j1)
     &                       + (1.0-a)*(1.0-b)*Displace(i1,j1)
              endif
C......................................................................
C.            Determine optimal shift and interpolate to get it       .
C.            subpixel                                                .
C......................................................................
              call Copy(LS,LSBase)

              ! Get optimal shift to pixel resolution, then interpolate
              ! in neighbourhood of this position
              call PixelPasses(OK,Displace(i,j),Par,Window,LS)

              if (Displace(i,j)%x .ne. huge(0.0)) then
                ! First estimate for optimal solution found.
                ! Now refine this estimate using sub-pixel interpolated
                ! images.
                Step = 0.25
                do iSubPix=0,Par%iSubPixelPasses-1
                  call Copy(LS,LSBase)
                  call SubPixelPasses(OK,Displace(i,j),Par,Window,LS,
     &                 Step)
                  Step = Step/2.0
                enddo
              endif
C......................................................................
C.            Display apparent displacement arrows                    .
C......................................................................
              call SetColour(0)
              if (Displace(i,j)%x .ne. Huge(0.0)) then
                Shift = Par%VectorScale*Displace(i,j) ! Rescale
                call Arrow(Location(i,j),Shift)
              endif
            endif
          enddo
```

```
        enddo
C.......................................................................
C.      Refine mesh for next pass to pick up vectors missed            .
C.......................................................................
        iLastPass = iCoarse
        jLastPass = jCoarse
        iCoarse = iCoarse/2
        jCoarse = jCoarse/2
      enddo
C-----------------------------------------------------------------------
C-   Rescale displacements to world units                              -
C-----------------------------------------------------------------------
      do j=0,ny-1
        do i=0,nx-1
          if (Displace(i,j)%x .ne. Huge(0.0)) then
            Displace(i,j)%y = Displace(i,j)%y*Par%Fore%AspectRatio
          endif
        enddo
      enddo

C=======================================================================
C=    Filter and tidy up apparent displacements                        =
C=======================================================================
      IF (Continue()) THEN
C-----------------------------------------------------------------------
C-      Remove outliers, replacing by local means                      -
C-----------------------------------------------------------------------
        if (Par%RemoveOutliers) then
          do j=0,ny-1
            do i=0,nx-1
              if (Displace(i,j)%x .ne. Huge(0.0)) then
                n = 0
                xMean = 0.0
                yMean = 0.0
                do jy=max(0,j-1),min(j+1,ny-1)
                  do ix=max(0,i-1),min(i+1,nx-1)
                    if (Displace(ix,jy)%x .ne. Huge(0.0) .and.
     &                  (i .ne. ix .or. j .ne. jy)) then
                      xMean = xMean + Displace(ix,jy)%x
                      yMean = yMean + Displace(ix,jy)%y
                      n = n + 1
                    endif
                  enddo
```

```
              enddo
              if (n .ge. 4) then
                xMean = xMean/real(n)
                yMean = yMean/real(n)
                if (abs(Displace(i,j)%x-xMean) .gt. 1.0 .or.
     &              abs(Displace(i,j)%y-yMean) .gt. 1.0) then
                  Displace(i,j)%x = xMean
                  Displace(i,j)%y = yMean
                endif
              endif
            endif
          enddo
        enddo
      endif
C------------------------------------------------------------------------
C-      Constrain mean displacements                                    -
C------------------------------------------------------------------------
      if (Par%MeanZero) then
C........................................................................
C.        Global mean constrained                                       .
C........................................................................
        xMean = 0.0
        yMean = 0.0
        n = 0
        do j=0,ny-1
          do i=0,nx-1
            if (Displace(i,j)%x .ne. Huge(0.0)) then
              xMean = xMean + Displace(i,j)%x
              yMean = yMean + Displace(i,j)%y
              n = n + 1
            endif
          enddo
        enddo
        if (n .gt. 0) then
          xMean = xMean/real(n)
          yMean = yMean/real(n)
          do j=0,ny-1
            do i=0,nx-1
              if (Displace(i,j)%x .ne. Huge(0.0)) then
                Displace(i,j)%x = Displace(i,j)%x - xMean
                Displace(i,j)%y = Displace(i,j)%y - yMean
              endif
            enddo
```

```
            enddo
          endif
        endif
        if (Par%xMeanZero) then
C.......................................................................
C.        Horizontal mean of x displacement for each row               .
C.......................................................................
          do j=0,ny-1
            Mean = 0.0
            n = 0
            do i=0,nx-1
              if (Displace(i,j)%x .ne. Huge(0.0)) then
                Mean = Mean + Displace(i,j)%x
                n = n + 1
              endif
            enddo
            if (n .gt. 0) then
              Mean = Mean/real(n)
              do i=0,nx-1
                if (Displace(i,j)%x .ne. Huge(0.0)) then
                  Displace(i,j)%x = Displace(i,j)%x - Mean
                endif
              enddo
            endif
          enddo
        endif
        if (Par%yMeanZero) then
C.......................................................................
C.        Vertical mean of y displacement for each column              .
C.......................................................................
          do i=0,nx-1
            Mean = 0.0
            n = 0
            do j=0,ny-1
              if (Displace(i,j)%y .ne. Huge(0.0)) then
                Mean = Mean + Displace(i,j)%y
                n = n + 1
              endif
            enddo
            if (n .gt. 0) then
              Mean = Mean/real(n)
              do j=0,ny-1
                if (Displace(i,j)%y .ne. Huge(0.0)) then
```

```
              Displace(i,j)%y = Displace(i,j)%y - Mean
            endif
          enddo
        endif
      enddo
    endif

C=================================================================
C=      Produce and display output images                        =
C=================================================================
      if (Use(Par%DensLarge) .or. Use(Par%DensSmall)) then
C-----------------------------------------------------------------
C-       Calculate potential field: this is the density perturbation  -
C-----------------------------------------------------------------
        ! Can make this multi-grid to get faster convergence!
        call CalculatePotential(Density,Displace,nx,ny,1.0/float(nx),
     &         1.0/float(ny)*Par%Fore%AspectRatio)

        ! Construct image of the density
        call CreateImage(Par%DensSmall,nx+1,ny+1)
        call RenameImage(Par%DensSmall,'Density field')
        call SetImageLUT(Par%DensSmall,LUT)
        do j=0,ny
          do i=0,nx
            if (Density(i,j) .eq. Huge(0.0)) then
              call SetPixel(Par%DensSmall,i,j,0)
            else
              call SetPixel(Par%DensSmall,i,j,min(254.0,max(1.0,
     &               128.0+511.0*Par%DensityScale*Density(i,j))))
            endif
          enddo
        enddo

        ! Display density image
        ! One pixel per density point.
        call DisplayImage(Par%DensSmall,DisplayImage$Resize
     &         +DisplayImage$NewView)

        if (Use(Par%DensLarge)) then
          ! Display a copy of the density image, rescaled to the size
          ! of the source images. Use bi-linear interpolation.
          call CreateImage(Par%DensLarge,Width(Par%Back),
     &           Height(Par%Back))
```

```
          call RenameImage(Par%DensLarge,'Density field')
          call SetImageLUT(Par%DensLarge,LUT)
          call RescaleImage(Par%DensLarge,Par%DensSmall)
          call DisplayImage(Par%DensLarge,DisplayImage$Resize
     &           +DisplayImage$NewView)
        endif
      endif

      if (Use(Par%xGrad)) then
C-------------------------------------------------------------------------
C-        x component of gradient                                       -
C-------------------------------------------------------------------------
        ! Create small image
        call CreateImage(Grad,nx,ny)
        call RenameImage(Grad,'x Gradient')
        call SetImageLUT(Grad,LUT)

        ! Copy the gradient data to the image
        do j=0,ny-1
          do i=0,nx-1
            if (Displace(i,j)%x .eq. Huge(0.0)) then
              call SetPixel(Grad,i,j,0)
            else
              d = min(254.0,max(1.0,
     &              128.0+32.0*Par%GradientScale*Displace(i,j)%x))
              call SetPixel(Grad,i,j,d)
            endif
          enddo
        enddo

        ! Create final output image
        call CreateImage(Par%xGrad,Width(Par%Back),Height(Par%Back))
        call RenameImage(Par%xGrad,'Density gradient: x component')
        call SetImageLUT(Par%xGrad,LUT)

        ! Rescale small image to output image (bi-linear interpolation)
        call RescaleImage(Par%xGrad,Grad)

        ! Display output image and destroy small image
        call DisplayImage(Par%xGrad,DisplayImage$Resize
     &         +DisplayImage$NewView)
        call DestroyImage(Grad)
      endif
```

```
      if (Use(Par%yGrad)) then
C------------------------------------------------------------------------
C-        y component of gradient                                       -
C------------------------------------------------------------------------
        ! Create small image
        call CreateImage(Grad,nx,ny)
        call RenameImage(Grad,'z Gradient')
        call SetImageLUT(Grad,LUT)

        ! Copy the gradient data to the image
        do j=0,ny-1
          do i=0,nx-1
            if (Displace(i,j)%y .eq. Huge(0.0)) then
              call SetPixel(Grad,i,j,0)
            else
              d = min(254.0,max(1.0,
     &              128.0+32.0*Par%GradientScale*Displace(i,j)%y))
              call SetPixel(Grad,i,j,d)
            endif
          enddo
        enddo

        ! Create final output image
        call CreateImage(Par%yGrad,Width(Par%Back),Height(Par%Back))
        call RenameImage(Par%yGrad,'Density gradient: z component')
        call SetImageLUT(Par%yGrad,LUT)

        ! Rescale small image to output image (bi-linear interpolation)
        call RescaleImage(Par%yGrad,Grad)

        ! Display output image and destroy small image
        call DisplayImage(Par%yGrad,DisplayImage$Resize
     &        +DisplayImage$NewView)
        call DestroyImage(Grad)
      endif

      IF (Use(Par%Disp)) THEN
C........................................................................
C.        Show apparent displacement vectors on density image           .
C........................................................................
        ! Need a copy of the full-size density image
        if (Use(Par%DensLarge)) then
```

```
          call CopyImage(Par%Disp,Par%DensLarge)
        else
          call CreateImage(Par%Disp,Width(Par%Back),Height(Par%Back))
          Par%Disp = LUT
        endif

        ! Display the density image
        call DisplayImage(Par%Disp,DisplayImage$Resize
     &         +DisplayImage$NewView)
        call SetColour(255)

        ! Superimpose the displacement vectors
        do j=0,ny-1
          do i=0,nx-1
            if (Displace(i,j)%x .ne. Huge(0.0)) then
              Shift = Par%VectorScale*Displace(i,j)
              call Arrow(Location(i,j),Shift)
            endif
          enddo
        enddo
        call ReadDisplay(Par%Disp)
      endif
    endif

C================================================================
C=    Calculate statistics                                      =
C================================================================
      u = 0.0
      v = 0.0
      u2 = 0.0
      v2 = 0.0
      uv = 0.0
      nPoints = 0
      do j=0,ny-1
        do i=0,nx-1
          if (Displace(i,j)%x .ne. Huge(0.0)) then
            u = u + Displace(i,j)%x
            v = v + Displace(i,j)%y
            u2 = u2 + Displace(i,j)%x*Displace(i,j)%x
            v2 = v2 + Displace(i,j)%y*Displace(i,j)%y
            uv = uv + Displace(i,j)%x*Displace(i,j)%y
            nPoints = nPoints + 1
          endif
```

```
        enddo
      enddo
      if (nPoints .gt. 0) then
        u = u/real(nPoints)
        v = v/real(nPoints)
        u2 = sqrt(u2/real(nPoints))
        v2 = sqrt(v2/real(nPoints))
        us = sqrt(u2*u2-u*u)
        vs = sqrt(v2*v2-v*v)
        uv = (uv/real(nPoints) - u*v)/(us*vs)
        write(*,'(1x,"Mean       :",f10.6,1x,f10.6)')u,v
        write(*,'(1x,"RMS        :",f10.6,1x,f10.6)')u2,v2
        write(*,'(1x,"Std Dev    :",f10.6,1x,f10.6)')us,vs
        write(*,'(1x,"Correlation:",f10.6)')uv
      endif

C=========================================================================
C=    Save images and tidy up                                            =
C=========================================================================
      call TidyUp(Par%Disp)
      call TidyUp(Par%yGrad)
      call TidyUp(Par%xGrad)
      call TidyUp(Par%DensLarge)
      call TidyUp(Par%DensSmall)
      call DestroyView(DispView)
      call TidyUp(Par%Diff)
      call Destroy(LS)
      call Destroy(LSBase)
      deallocate(Density)
      deallocate(Displace)
      deallocate(Location)
      call TidyUp(Par%Fore)
      call TidyUp(Par%Back)
      return
      end subroutine
```

PixelPasses

```
C
C*************************************************************************
C* PixelPasses                       PixelPasses                         *
C*    Do the subpixel passes                                             *
C*************************************************************************
      subroutine PixelPasses(OK,Disp,Par,Window,LS)
```

```
C=====Parameters
      logical OK
      type (F_WLocation) Disp
      type (D_AnalyseSchlieren) Par
      type (F_Window) Window
      type (F_LeastSquares) LS
C=====Local variables
      type (F_Location)Shift,Check
      type (F_Window) BaseWindow
      integer (4) iw0,iw1,jw0,jw1,i,j,k,ijMax,ijWindow
      real (8) Diff(-32:32,-32:32),tmp,dx,dy
      logical Again,Reject
C=========================================================================
C=    Initialise                                                         =
C=========================================================================
      Again = .true.
      iw0 = Par%iSearch0
      iw1 = Par%iSearch1
      jw0 = Par%jSearch0
      jw1 = Par%jSearch1
      Shift%i = Disp%x
      Shift%j = Disp%y
      BaseWindow = Window
      ijWindow = 0                    ! Size increase in window
      do while (Again)
C-------------------------------------------------------------------------
C-      Determine optimal shift, repeating until it is found             -
C-------------------------------------------------------------------------
        Reject = .false.
        call OptimalShift_I(Shift,Diff,Par%Back,Par%Fore,Window,
     &                  iw0,iw1,jw0,jw1,Par%DifferenceType)
        if (Shift%i .eq. Huge(1)) then
C.......Can't do the optimisation here
          Again = .false.
          Reject = .true.
        elseif (Shift%i .le. Par%iMin .or.
     &          Shift%i .ge. Par%iMax .or.
     &          Shift%j .le. Par%jMin .or.
     &          Shift%j .ge. Par%jMax) then
C.........Shifted past its limits
          Reject = .true.
          Again  = .false.
        elseif (Shift%i .gt. iw0 .and.
```

```
     &          Shift%i .lt. iw1 .and.
     &          Shift%j .gt. jw0 .and.
     &          Shift%j .lt. jw1) then
C.........The internal point is optimum
          ijMax = max(abs(Shift%i),abs(Shift%j))
          ijMax = min(ijMax,Par%SizeIncreaseLimit)
          if (ijMax .gt. ijWindow) then
C...........Should do again with a bigger window
            Again = .true.
            ijWindow = ijMax + 1
            Window%Left   = BaseWindow%Left   - ijWindow
            Window%Right  = BaseWindow%Right  + ijWindow
            Window%Bottom = BaseWindow%Bottom - ijWindow
            Window%Top    = BaseWindow%Top    + ijWindow
          elseif (Diff(Shift%i,Shift%j) .gt. Par%MaxDifference)
     &            then
C...........A minimum is found, but it is not a very good one.
C...........Enlarge search region
            iw0 = max(Par%iMin,iw0 - 2)
            iw1 = min(Par%iMax,iw1 + 2)
            jw0 = max(Par%jMin,jw0 - 2)
            jw1 = min(Par%jMax,jw1 + 2)
            Again = (iw0 .gt. Par%iMin .or. iw1 .lt. Par%iMax .or.
     &               jw0 .gt. Par%jMin .or. jw1 .lt. Par%jMax)
            Reject = (Diff(Shift%i,Shift%j) .gt. Par%AcceptDifference)
          else
C...........This is a good minimum - use it!
            Again = .false.
          endif
        else
C.........Shift search window and try again
          if (Shift%i .eq. iw0) then
            iw0 = Shift%i - 2
            iw1 = Shift%i + 1
          elseif (Shift%i .eq. iw1) then
            iw0 = Shift%i - 1
            iw1 = Shift%i + 2
          else
            iw0 = Shift%i - 1
            iw1 = Shift%i + 1
          endif
          if (Shift%j .eq. jw0) then
            jw0 = Shift%j - 2
```

```
            jw1 = Shift%j + 1
          elseif (Shift%j .eq. iw1) then
            jw0 = Shift%j - 1
            jw1 = Shift%j + 2
          else
            jw0 = Shift%j - 1
            jw1 = Shift%j + 1
          endif
          iw0 = max(Par%iMin,iw0)
          iw1 = min(Par%iMax,iw1)
          jw0 = max(Par%jMin,jw0)
          jw1 = min(Par%jMax,jw1)
          ijWindow = min(ijWindow + 1,Par%SizeIncreaseLimit)
          Window%Left   = BaseWindow%Left   - ijWindow
          Window%Right  = BaseWindow%Right  + ijWindow
          Window%Bottom = BaseWindow%Bottom - ijWindow
          Window%Top    = BaseWindow%Top    + ijWindow
        endif
      enddo
      if (Reject) then
C========================================================================
C=      Flag as no good                                                 =
C========================================================================
        Shift%i = Huge(1)
        Shift%j = Huge(1)
        Disp%x = Huge(1.0)
        Disp%y = Huge(1.0)
        OK = .false.
      else
C========================================================================
C=      Interpolate to subpixel accuracy                                =
C========================================================================
        k = 0
        do i=-1,1
          do j=-1,1
            LS%A(k,6) = Diff(Shift%i+i,Shift%j+j)
            k = k + 1           ! Equation number
          enddo
        enddo
        call LeastSquares(LS)
        k = LS%n
        ! Check on the curvature as in DigImage
        tmp = LS%A(3,k)*LS%A(3,k)-4.0*LS%A(4,k)*LS%A(5,k)
```

```
        if (tmp .ne. 0.0 .and. LS%A(4,k) .gt. Par%MinCurvature .and.
     &      LS%A(5,k) .gt. Par%MinCurvature) then
          dx = (2.0*LS%A(1,k)*LS%A(5,k)-LS%A(2,k)*LS%A(3,k))/tmp
          dy = (2.0*LS%A(2,k)*LS%A(4,k)-LS%A(1,k)*LS%A(3,k))/tmp
          dx = max(-1.0,min(1.0,dx))
          dy = max(-1.0,min(1.0,dy))
          Disp%x = real(Shift%i) + dx
          Disp%y = real(Shift%j) + dy
          OK = .true.
        else
          Disp%x = Shift%i
          Disp%y = Shift%j
          OK = .false.
        endif
      endif
      return
      end subroutine
```

SubPixelPasses

```
C
C*********************************************************************
C* SubPixelPasses                     SubPixelPasses                 *
C*    Do the subpixel passes                                         *
C*********************************************************************
      subroutine SubPixelPasses(OK,Disp,Par,Window,LS,Step)
C=====Parameters
      logical OK
      type (F_WLocation) Disp
      type (D_AnalyseSchlieren) Par
      type (F_Window) Window
      type (F_LeastSquares) LS
      real (8) Step
C=====Local variables
      integer (4) i,j,k
      real (8) tmp,dx,dy
C=====Code
      k = 0
      do i=-1,1
        dx = Disp%x + real(i)*Step
        do j=-1,1
          dy = Disp%y + real(j)*Step
          LS%A(k,6) = WindowDifference_R(Par%Back,Par%Fore,Window,
     &                dx,dy,Par%DifferenceType)
```

```
          k = k + 1           ! Equation number
        enddo
      enddo
      call LeastSquares(LS)
      k = LS%n
      ! Check on curvature as in DigImage
      tmp = LS%A(3,k)*LS%A(3,k)-4.0*LS%A(4,k)*LS%A(5,k)
      if (tmp .ne. 0.0 .and. LS%A(4,k) .gt. Par%MinCurvature .and.
     &    LS%A(5,k) .gt. Par%MinCurvature) then
        dx = (2.0*LS%A(1,k)*LS%A(5,k)-LS%A(2,k)*LS%A(3,k))/tmp
        dy = (2.0*LS%A(2,k)*LS%A(4,k)-LS%A(1,k)*LS%A(3,k))/tmp
        dx = max(-2.0,min(2.0,dx))
        dy = max(-2.0,min(2.0,dy))
        Disp%x = Disp%x + dx*Step
        Disp%y = Disp%y + dy*Step
        OK = .true.
      else
C.......Leave Disp unchanged
        OK = .false.
      endif
      return
      end subroutine
```

OptimalShift

```
C
C************************************************************************
C* OptimalShift                       OptimalShift_I                    *
C*    Calculates the (pixel) optimal shift based on the absolute        *
C*    difference between two images.                                    *
C*        Shift                                                         *
C*        Diff                                                          *
C*        Image0                                                        *
C*        Image1                                                        *
C*        Window                                                        *
C*        iMin,iMax,jMin,jMax         Limits on shift                   *
C*        DifferenceType              The type of difference to be      *
C*                                    optimised.                        *
C************************************************************************
      subroutine OptimalShift_I(Shift,Diff,Image0,Image1,Window,
     &           iMin,iMax,jMin,jMax,DifferenceType)
C=====Parameters
      type (F_Location) Shift
      integer (4) iMin
```

```
      integer (4) iMax
      integer (4) jMin
      integer (4) jMax
      real (8) Diff(-32:32,-32:32)
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      integer (4) DifferenceType
C=====Local variables
      integer (4) iOff
      integer (4) jOff
      integer (4) iLo,jLo
      real Lower
C=====Code
      Lower = Huge(1.0)
      iLo   = Huge(1)
      jLo   = Huge(1)
      do jOff=jMin,jMax
        do iOff=iMin,iMax
          Diff(iOff,jOff) = WindowDifference_I(Image0,Image1,
     &         Window,iOff,jOff,DifferenceType)
          if (Diff(iOff,jOff) .lt. Lower) then
            if (Diff(iOff,jOff) .ge. 0.0) then
              Lower = Diff(iOff,jOff)
              iLo = iOff
              jLo = jOff
            endif
          endif
        enddo
      enddo
      Shift = F_Location(iLo,jLo)
      return
      end subroutine
```

WindowDifference

```
C
C***********************************************************************
C* WindowDifference                   WindowDifference_I              *
C*    Calculates mean absolute difference between images. Ignores     *
C*    difference when one of the pixels is zero.                      *
C*        Image0                                                      *
C*        Image1                                                      *
C*        Window                                                      *
```

```
C*       iOff,jOff                                                      *
C*       DifferenceType                                                 *
C***********************************************************************
      function WindowDifference_I(Image0,Image1,Window,iOff,jOff,
     &         DifferenceType)
C=====Parameters
      real WindowDifference_I
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      integer (4) iOff
      integer (4) jOff
      integer (4) DifferenceType
C=====Code
      select case (DifferenceType)
       case (AnalyseSchlieren$AbsoluteDiff)
        WindowDifference_I =
     &      WindowDifference_I_Absolute(Image0,Image1,Window,iOff,jOff)
       case (AnalyseSchlieren$SquareDiff)
        WindowDifference_I =
     &      WindowDifference_I_Square(Image0,Image1,Window,iOff,jOff)
       case (AnalyseSchlieren$CrossCorr)
        WindowDifference_I = 1.0 -
     &      WindowDifference_I_CrossCorr(Image0,Image1,Window,iOff,jOff)
      end select
      return
      end function
```

WindowDifferenceAbsolute

```
C
C***********************************************************************
C* WindowDifference                   WindowDifference_I_Absolute      *
C*    Calculates mean absolute difference between images. Ignores      *
C*    difference when one of the pixels is zero.                       *
C*       Image0                                                         *
C*       Image1                                                         *
C*       Window                                                         *
C*       iOff,jOff                                                      *
C***********************************************************************
      function WindowDifference_I_Absolute(Image0,Image1,Window,iOff,
     &         jOff)
C=====Parameters
      real WindowDifference_I_Absolute
```

```
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      integer (4) iOff
      integer (4) jOff
C=====Local variables
      integer (4) n,Sum
      integer (4) Pix0,Pix1
      call PresetPixel(Window)
      Sum = 0
      n = 0
      do while (NextPixel(Window))
        Pix0 = Pixel(Image0,Window)
        Pix1 = PixelRel(Image1,Window,iOff,jOff)
        if (Pix0 .gt. 0 .and. Pix1 .gt. 0) then
          Sum = Sum + abs(Pix0 - Pix1)
          n = n + 1
        endif
      enddo
      if (n .gt. 0) then
        WindowDifference_I_Absolute = float(Sum)/float(n)
      else
        WindowDifference_I_Absolute = -1.0 ! Flag as invalid
      endif
      return
      end function
```

WindowDifferencSquare

```
C
C***********************************************************************
C* WindowDifference                   WindowDifference_I_Square        *
C*    Calculates mean Square difference between images. Ignores        *
C*    difference when one of the pixels is zero.                       *
C*        Image0                                                       *
C*        Image1                                                       *
C*        Window                                                       *
C*        iOff,jOff                                                    *
C***********************************************************************
      function WindowDifference_I_Square(Image0,Image1,Window,iOff,
     &         jOff)
C=====Parameters
      real WindowDifference_I_Square
      type (F_Image) Image0
```

```
      type (F_Image) Image1
      type (F_Window) Window
      integer (4) iOff
      integer (4) jOff
C=====Local variables
      integer (4) n,Sum
      integer (4) Pix0,Pix1
      call PresetPixel(Window)
      Sum = 0
      n = 0
      do while (NextPixel(Window))
        Pix0 = Pixel(Image0,Window)
        Pix1 = PixelRel(Image1,Window,iOff,jOff)
        if (Pix0 .gt. 0 .and. Pix1 .gt. 0) then
          Sum = Sum + (Pix0 - Pix1)*(Pix0 - Pix1)
          n = n + 1
        endif
      enddo
      if (n .gt. 0) then
        WindowDifference_I_Square = float(Sum)/float(n)/256.0
      else
        WindowDifference_I_Square = -1.0 ! Flag as invalid
      endif
      return
      end function
```

WindowDifferenceCrossCorr

```
C
C*********************************************************************
C* WindowDifference                   WindowDifference_I_CrossCorr     *
C*    Calculates mean CrossCorr difference between images. Ignores     *
C*    difference when one of the pixels is zero.                       *
C*        Image0                                                       *
C*        Image1                                                       *
C*        Window                                                       *
C*        iOff,jOff                                                    *
C*********************************************************************
      function WindowDifference_I_CrossCorr(Image0,Image1,Window,iOff,
     &         jOff)
C=====Parameters
      real WindowDifference_I_CrossCorr
      type (F_Image) Image0
      type (F_Image) Image1
```

```
      type (F_Window) Window
      integer (4) iOff
      integer (4) jOff
C=====Local variables
      integer (4) n
      real Pix0,Pix1,Sum2,SumP0,Sum2P0,SumP1,Sum2P1
      call PresetPixel(Window)
      SumP0 = 0.0
      SumP1 = 0.0
      Sum2P0 = 0.0
      Sum2P1 = 0.0
      Sum2 = 0.0
      n = 0
      do while (NextPixel(Window))
        Pix0 = float(Pixel(Image0,Window))
        Pix1 = float(PixelRel(Image1,Window,iOff,jOff))
        if (Pix0 .gt. 0.0 .and. Pix1 .gt. 0.0) then
          SumP0 = SumP0 + Pix0
          SumP1 = SumP1 + Pix1
          Sum2P0 = Sum2P0 + Pix0*Pix0
          Sum2P1 = Sum2P1 + Pix1*Pix1
          Sum2 = Sum2 + Pix0*Pix1
          n = n + 1
        endif
      enddo
      if (n .gt. 0) then
        Sum2P0 = Sum2P0 - SumP0*SumP0/float(n)
        Sum2P1 = Sum2P1 - SumP1*SumP1/float(n)
        if (Sum2P0 .gt. 0.0 .and. Sum2P1 .gt. 0.0) then
          WindowDifference_I_CrossCorr = (Sum2 - SumP0*SumP1/float(n))
     &                                   /sqrt(Sum2P0*Sum2P1)
        else
           WindowDifference_I_CrossCorr =  2.0 ! Flag as invalid
        endif
      else
        WindowDifference_I_CrossCorr =  2.0 ! Flag as invalid
      endif
      return
      end function
```

OptimalShiftR

```
C
C************************************************************************
```

```
C* OptimalShift                          OptimalShift_R                   *
C*    Calculates the (subpixel) optimal shift based on the absolute    *
C*    difference between two images. A return value of exactly 0.0     *
C*    indicates no optimal found.                                      *
C*        Image0                                                       *
C*        Image1                                                       *
C*        Window                                                       *
C*        xMin,xMax,yMin,yMax         Limits on shift                  *
C*        xStep,yStep                 The shifts                       *
C*        DifferenceType              The type of difference to be     *
C*                                    optimised.                       *
C***********************************************************************
      function OptimalShift_R(Image0,Image1,Window,xMin,xMax,yMin,yMax,
     &         xStep,yStep,DifferenceType)
C=====Parameters
      type (F_WLocation) OptimalShift_R
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      real  xMin
      real  xMax
      real  yMin
      real  yMax
      real  xStep
      real  yStep
      integer (4) DifferenceType
C=====Local variables
      real  xLo,yLo,xOff,yOff
      real Lower,Diff
      real, parameter :: Limit = 1.0E10
C=====Code
      Lower = Limit
      yOff = yMin
      do while (yOff .le. yMax)
        xOff = xMin
        do while (xOff .le. xMax)
          Diff = WindowDifference_R(Image0,Image1,Window,xOff,
     &         yOff,DifferenceType)
          if (Diff .lt. Lower) then
            if (Diff .ge. 0.0) then
              Lower = Diff
              xLo = xOff
              yLo = yOff
```

```
          endif
        endif
        xOff = xOff + xStep
      enddo
      yOff = yOff + yStep
    enddo
    if (Lower .lt. Limit) then
      OptimalShift_R%x = xLo
      OptimalShift_R%y = yLo
    else
      OptimalShift_R%x = Huge(0.0)
      OptimalShift_R%y = Huge(0.0)
    endif
    return
    end function
```

WindowDifferencR

```
C
C*****************************************************************
C* WindowDifference                   WindowDifference_R          *
C*    Calculates mean difference between images. Ignores difference   *
C*    when one of the pixels is zero.                               *
C*        Image0                                                    *
C*        Image1                                                    *
C*        Window                                                    *
C*        xOff,yOff                                                 *
C*        DifferenceType                                            *
C*****************************************************************
      function WindowDifference_R(Image0,Image1,Window,xOff,yOff,
     &         DifferenceType)
C=====Parameters
      real WindowDifference_R
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      real  xOff
      real  yOff
      integer (4) DifferenceType
C=====Code
      select case (DifferenceType)
       case (AnalyseSchlieren$AbsoluteDiff)
        WindowDifference_R =
     &      WindowDifference_R_Absolute(Image0,Image1,Window,xOff,yOff)
```

```
       case (AnalyseSchlieren$SquareDiff)
        WindowDifference_R =
     &      WindowDifference_R_Square(Image0,Image1,Window,xOff,yOff)
       case (AnalyseSchlieren$CrossCorr)
        WindowDifference_R = 1.0 -
     &      WindowDifference_R_CrossCorr(Image0,Image1,Window,xOff,yOff)
      end select
      return
      end function
```

WindowDifferenceRAbsolute

```
C
C*********************************************************************
C* WindowDifference                   WindowDifference_R_Absolute      *
C*    Calculates mean absolute difference between images.             *
C*        Image0                                                      *
C*        Image1                                                      *
C*        Window                                                      *
C*        xOff,yOff                                                   *
C*********************************************************************
      function WindowDifference_R_Absolute(Image0,Image1,Window,xOff,
     &         yOff)
C=====Parameters
      real WindowDifference_R_Absolute
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      real  xOff
      real  yOff
C=====Local variables
      integer (4) n
      real Pix0,Pix1,Sum
      call PresetPixel(Window)
      Sum = 0.0
      n = 0
      do while (NextPixel(Window))
        Pix0 = float(Pixel(Image0,Window))
        Pix1 = PixelRel(Image1,Window,xOff,yOff)
        if (Pix0 .gt. 0 .and. Pix1 .gt. 0.0) then
          Sum = Sum + abs(Pix0 - Pix1)
          n = n + 1
        endif
      enddo
```

```
      if (n .gt. 0) then
        WindowDifference_R_Absolute = Sum/float(n)
      else
        WindowDifference_R_Absolute = -1.0 ! Flag as invalid
      endif
      return
      end function
```

WindowDifferenceRSquare

```
C
C*********************************************************************
C* WindowDifference                   WindowDifference_R_Square        *
C*    Calculates mean Square difference between images.                *
C*        Image0                                                       *
C*        Image1                                                       *
C*        Window                                                       *
C*        xOff,yOff                                                    *
C*********************************************************************
      function WindowDifference_R_Square(Image0,Image1,Window,xOff,yOff)
C=====Parameters
      real WindowDifference_R_Square
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      real  xOff
      real  yOff
C=====Local variables
      integer (4) n
      real Pix0,Pix1,Sum
      call PresetPixel(Window)
      Sum = 0.0
      n = 0
      do while (NextPixel(Window))
        Pix0 = float(Pixel(Image0,Window))
        Pix1 = PixelRel(Image1,Window,xOff,yOff)
        if (Pix0 .gt. 0 .and. Pix1 .gt. 0.0) then
          Sum = Sum + (Pix0 - Pix1)*(Pix0-Pix1)
          n = n + 1
        endif
      enddo
      if (n .gt. 0) then
        WindowDifference_R_Square = Sum/float(n)/256.0
      else
```

```
        WindowDifference_R_Square = -1.0 ! Flag as invalid
      endif
      return
      end function
```

WindowDifferenceRCrossCorr

```
C
C*********************************************************************
C* WindowDifference                   WindowDifference_R_CrossCorr     *
C*    Calculates mean CrossCorr difference between images.             *
C*        Image0                                                      *
C*        Image1                                                      *
C*        Window                                                      *
C*        xOff,yOff                                                   *
C*********************************************************************
      function WindowDifference_R_CrossCorr(Image0,Image1,Window,xOff,
     &         yOff)
C=====Parameters
      real WindowDifference_R_CrossCorr
      type (F_Image) Image0
      type (F_Image) Image1
      type (F_Window) Window
      real  xOff
      real  yOff
C=====Local variables
      integer (4) n
      real Pix0,Pix1,Sum2,SumP0,Sum2P0,SumP1,Sum2P1
      call PresetPixel(Window)
      SumP0 = 0.0
      SumP1 = 0.0
      Sum2P0 = 0.0
      Sum2P1 = 0.0
      Sum2 = 0.0
      n = 0
      do while (NextPixel(Window))
        Pix0 = float(Pixel(Image0,Window))
        Pix1 = PixelRel(Image1,Window,xOff,yOff)
        if (Pix0 .gt. 0.0 .and. Pix1 .gt. 0.0) then
          SumP0 = SumP0 + Pix0
          SumP1 = SumP1 + Pix1
          Sum2P0 = Sum2P0 + Pix0*Pix0
          Sum2P1 = Sum2P1 + Pix1*Pix1
          Sum2 = Sum2 + Pix0*Pix1
```

```
          n = n + 1
        endif
      enddo
      if (n .gt. 0) then
        Sum2P0 = Sum2P0 - SumP0*SumP0/float(n)
        Sum2P1 = Sum2P1 - SumP1*SumP1/float(n)
        if (Sum2P0 .gt. 0.0 .and. Sum2P1 .gt. 0.0) then
          WindowDifference_R_CrossCorr = (Sum2 - SumP0*SumP1/float(n))
     &                                   /sqrt(Sum2P0*Sum2P1)
        else
           WindowDifference_R_CrossCorr = 2.0 ! Flag as invalid
        endif
      else
        WindowDifference_R_CrossCorr = 2.0 ! Flag as invalid
      endif
      return
      end function
```

CalculatePotential

```
C
C*****************************************************************
C* CalculatePotential                CalcualtePotential          *
C*    Integrates vector field to obtain potential. The vector and    *
C*    potential fields are arranged as follows:                  *
C*    P0n     P1n     P2n     P3n   ...         Pmn              *
C*        v0n     v1n     v2n     v3n   v(m-1)(n-1)              *
C*     .   .   .   .   .   .   .   .       .     .               *
C*     .   .   .   .   .   .   .   .       .     .               *
C*     .   .   .   .   .   .   .   .       .     .               *
C*        v01     v11     v21     v31   v(m-1)1                  *
C*    P01     P11     P21     P31     ...       Pm1              *
C*        v00     v10     v20     v30   v(m-1)0                  *
C*    P00     P10     P20     P30     ...       Pm0              *
C*        Potential(0:nx,0:ny)        Returns the potential.     *
C*        Vector(0:nx-1,0:ny-1)       The vector field to be     *
C*                                    integrated.                *
C*        nx,ny                       Size of grid               *
C*        dx,dy                       The mesh spacing.          *
C*****************************************************************
      subroutine CalculatePotential(Potential,Vector,nx,ny,dx,dy)
C=====Parameters
      integer (4) nx
      integer (4) ny
```

```
      real Potential(0:nx,0:ny)
      type (F_WLocation) Vector(0:nx-1,0:ny-1)
      real dx
      real dy
C=====local variables
      integer (4) i,j,nPotential,iStart,jStart
      integer (4) iCount,nPasses,iTotal
      real PotentialMin,PotentialMax,PotentialMean
C------------------------------------------------------------------------
C-    Initialise whole grid as unaccessed                                  -
C------------------------------------------------------------------------
      do j=0,ny
        do i=0,nx
          Potential(i,j) = Huge(0.0)
        enddo
      enddo
C------------------------------------------------------------------------
C-    Set the potential for some arbitrary point at the centre of the  -
C-    grid.                                                                -
C------------------------------------------------------------------------
      iStart = nx/2
      jStart = ny/2
      Potential(iStart,jStart) = 0.0
C------------------------------------------------------------------------
C-    Repeatedly scan through the grid until all the points             -
C-    connected to the starting point have been set.                    -
C------------------------------------------------------------------------
      nPasses = 0
      iCount = 1
      iTotal = 0
      do while ((nPasses .eq. 0 .or. iCount .gt. 0 .or.
     &   iTotal .lt. nx*ny/8) .and. iStart .lt. nx)
        nPasses = nPasses + 1
        call PassForPotential(iCount,Potential,Vector,nx,ny,dx,dy)
        if (nPasses .eq. 1 .and. iCount .eq. 0 .and.
     &      iStart .lt. nx) then
C........................................................................
C.        On the first pass, we were unable to update any stream       .
C.        function values. This suggests the starting point was        .
C.        not a valid flow point. We shall try again with a            .
C.        different point, slowly moving out along the x axis          .
C.        until we find a suitable one                                 .
C........................................................................
```

```
          do while (Potential(iStart,jStart) .ne. Huge(0.0) .and.
     &       iStart .lt. nx)
            iStart = iStart + 1
          enddo
          Potential(iStart,jStart) = 1.0
          nPasses = 0
        endif
C........................................................................
C.      Keep track of the total number of grid points set              .
C........................................................................
        iTotal = iTotal + iCount
      enddo
      do i=0,40*nx !5*nx !40*nx
C.......Do a couple of extra passes to smooth the error
        call PassForPotential(iCount,Potential,Vector,nx,ny,dx,dy)
      enddo
C------------------------------------------------------------------------
C-    Determine mean value of potential                                 -
C------------------------------------------------------------------------
      PotentialMean = 0.0
      PotentialMin = 1.0
      PotentialMax = 1.0
      nPotential = 0
      do i=0,nx
        do j=0,ny
          if (Potential(i,j) .ne. Huge(0.0)) then
            PotentialMean = PotentialMean + Potential(i,j)
            PotentialMin = min(PotentialMin,Potential(i,j))
            PotentialMax = max(PotentialMax,Potential(i,j))
            nPotential = nPotential + 1
          endif
        enddo
      enddo
      PotentialMean = PotentialMean/float(max(1,nPotential))
      do i=0,nx-1
        do j=0,ny-1
          if (Vector(i,j)%x .eq. Huge(0.0)) then
C...........The velocity field is not valid, so the Potential field is
C...........also not valid. Indicate by setting to Huge
            Potential(i,j) = Huge(0.0)
            Potential(i+1,j) = Huge(0.0)
            Potential(i,j+1) = Huge(0.0)
            Potential(i+1,j+1) = Huge(0.0)
```

```
          elseif (Potential(i,j) .ne. Huge(0.0)) then
C...........Give zero mean
            Potential(i,j) = Potential(i,j) - PotentialMean
          endif
        enddo
      enddo
      return
      end subroutine
```

PassForPotential

```
C
C*****************************************************************
C* PassForPotential                  Internal: PassForPotential      *
C*    Calculates updated potential for the grid.                   *
C*****************************************************************
      subroutine PassForPotential(iCount,Potential,Vector,nx,ny,dx,dy)
C=====Parameters
      integer (4) iCount
      integer (4) nx
      integer (4) ny
      real Potential(0:nx,0:ny)
      type (F_WLocation) Vector(0:nx-1,0:ny-1)
      real dx
      real dy
C=====local variables
      integer (4) i,j,is,js,i0,i1,j0,j1
      real Phi
C=====Central point
C=================================================================
C=    Divide domain into four quadrants and integrate over all four    =
C=    simultaneously. This speeds up convergence as it allows the      =
C=    seed points to more rapidly fill the domain.                     =
C=================================================================
      is = nx/2
      js = ny/2
      iCount = 0
      do i=0,is
        i0 = max(0,is-i)
        i1 = min(nx,is+i)
        do j=0,js
          j0 = max(0,js-j)
          j1 = min(ny,js+j)
C.................................................................
```

```
C.        i0j0 quadrant                                                .
C......................................................................
          Phi = PotentialForPoint(i0,j0,Vector,Potential,nx,ny,dx,dy)
          if (Phi .ne. Huge(0.0)) then
            if (Potential(i0,j0) .ne. Huge(0.0)) then
              Potential(i0,j0) = 0.3*Potential(i0,j0) + 0.7*Phi
            else
              Potential(i0,j0) = Phi
              iCount = iCount + 1
            endif
          endif
C......................................................................
C.        i0j1 quadrant                                                .
C......................................................................
          Phi = PotentialForPoint(i0,j1,Vector,Potential,nx,ny,dx,dy)
          if (Phi .ne. Huge(0.0)) then
            if (Potential(i0,j1) .ne. Huge(0.0)) then
              Potential(i0,j1) = 0.3*Potential(i0,j1) + 0.7*Phi
            else
              Potential(i0,j1) = Phi
              iCount = iCount + 1
            endif
          endif
C......................................................................
C.        i1j0 quadrant                                                .
C......................................................................
          Phi = PotentialForPoint(i1,j0,Vector,Potential,nx,ny,dx,dy)
          if (Phi .ne. Huge(0.0)) then
            if (Potential(i1,j0) .ne. Huge(0.0)) then
              Potential(i1,j0) = 0.3*Potential(i1,j0) + 0.7*Phi
            else
              Potential(i1,j0) = Phi
              iCount = iCount + 1
            endif
          endif
C......................................................................
C.        i1j1 quadrant                                                .
C......................................................................
          Phi = PotentialForPoint(i1,j1,Vector,Potential,nx,ny,dx,dy)
          if (Phi .ne. Huge(0.0)) then
            if (Potential(i1,j1) .ne. Huge(0.0)) then
              Potential(i1,j1) = 0.3*Potential(i1,j1) + 0.7*Phi
            else
```

```
              Potential(i1,j1) = Phi
              iCount = iCount + 1
            endif
          endif
        enddo
      enddo
      return
      end subroutine
```

PotentialForPoint

```
C
C*****************************************************************
C* PotentialForPoint                  Internal: PotentialForPoint      *
C*    Evaluates an updated estimate of the potential for this point.   *
C*    The points are arranged as follows:                              *
C*                      Pi1                                            *
C*                 v01      v11                                        *
C*             P0j      Pij      P1j                                   *
C*                 v00      v10                                        *
C*                      Pi0                                            *
C*    The potential indices are i0,i1,j0,j1. The Vector indices are    *
C*    iv0,iv1,jv0,jv1.                                                 *
C*****************************************************************
      function PotentialForPoint(i,j,Vector,Potential,nx,ny,dx,dy)
C=====Parameters
      real PotentialForPoint
      integer (4) i
      integer (4) j
      integer (4) nx
      integer (4) ny
      real Potential(0:nx,0:ny)
      type (F_WLocation) Vector(0:nx-1,0:ny-1)
      real dx
      real dy
C=====local variables
      integer (4) i0,i1,j0,j1,iv0,iv1,jv0,jv1,nCount
      real u0,u1,v0,v1,u00,u01,u10,u11,v00,v01,v10,v11
      real Phi
C=====Neighbouring potential grid points
      i0 = max(0,i-1)
      i1 = min(nx,i+1)
      j0 = max(0,j-1)
      j1 = min(ny,j+1)
```

```
C=====Neighbouring velocity grid points
      iv0 = max(i-1,0)
      iv1 = min(i,nx-1)
      jv0 = max(j-1,0)
      jv1 = min(j,ny-1)
C=====Contributing velocities
      if (Vector(iv0,jv0)%x .eq. Huge(0.0)) then
        u00 = Vector(iv1,jv0)%x
        v00 = Vector(iv0,jv1)%y
      else
        u00 = Vector(iv0,jv0)%x
        v00 = Vector(iv0,jv0)%y
      endif
      if (Vector(iv0,jv1)%x .eq. Huge(0.0)) then
        u01 = Vector(iv1,jv1)%x
        v01 = Vector(iv0,jv0)%y
      else
        u01 = Vector(iv0,jv1)%x
        v01 = Vector(iv0,jv1)%y
      endif
      if (Vector(iv1,jv0)%x .eq. Huge(0.0)) then
        u10 = Vector(iv0,jv0)%x
        v10 = Vector(iv1,jv1)%y
      else
        u10 = Vector(iv1,jv0)%x
        v10 = Vector(iv1,jv0)%y
      endif
      if (Vector(iv1,jv1)%x .eq. Huge(0.0)) then
        u11 = Vector(iv0,jv1)%x
        v11 = Vector(iv1,jv0)%y
      else
        u11 = Vector(iv1,jv1)%x
        v11 = Vector(iv1,jv1)%y
      endif
        if (u00 .eq. Huge(0.0)) then
          u00 = 0.0
      endif
        if (u01 .eq. Huge(0.0)) then
          u01 = 0.0
      endif
        if (u10 .eq. Huge(0.0)) then
          u10 = 0.0
      endif
```

```
        if (u11 .eq. Huge(0.0)) then
          u11 = 0.0
      endif
        if (v00 .eq. Huge(0.0)) then
          v00 = 0.0
      endif
        if (v01 .eq. Huge(0.0)) then
          v01 = 0.0
      endif
        if (v10 .eq. Huge(0.0)) then
          v10 = 0.0
      endif
        if (v11 .eq. Huge(0.0)) then
          v11 = 0.0
      endif
      u0 = (u00 + u01)/2.0
      u1 = (u10 + u11)/2.0
      v0 = (v00 + v10)/2.0
      v1 = (v01 + v11)/2.0
C=====Now accumulate the contributions for this point
      nCount = 0
      Phi = 0.0
C-----Treat as velocity potential
      if (Potential(i0,j) .ne. Huge(0.0) .and. i .ne. i0)
     &    then
        Phi = Phi + Potential(i0,j) + u0*dx
        nCount = nCount + 1
      endif
      if (Potential(i1,j) .ne. Huge(0.0) .and. i .ne. i1)
     &    then
        Phi = Phi + Potential(i1,j) - u1*dx
        nCount = nCount + 1
      endif
      if (Potential(i,j0) .ne. Huge(0.0) .and. j .ne. j0)
     &    then
        Phi = Phi + Potential(i,j0) + v0*dy
        nCount = nCount + 1
      endif
      if (Potential(i,j1) .ne. Huge(0.0) .and. j .ne. j1)
     &    then
        Phi = Phi + Potential(i,j1) - v1*dy
        nCount = nCount + 1
      endif
```

```
C=====Finally, determine the mean, provided nCount nonzero
      if (nCount .eq. 0) then
C.......Indicate invalid value
        Phi = Huge(Phi)
      else
C.......Calcualte mean
        Phi = Phi/float(nCount)
      endif
      PotentialForPoint = Phi
      return
      end function
```

C Contact Information

C.1 DigImage Software

DL Research Partners
c/o Stuart Dalziel
Dept. of Applied Mathematics and Theoretical Physics
University of Cambridge
Silver Street
Cambridge, UK CB3 9EW

C.2 Frame Grabber Cards

A large number of companies offer "A2D" cards that can be used to translate an analogue image into a digital image. The software package DigImage requires the PC to be equipped with one of two frame grabber cards (DT2861 or DT2862) sold by Data Translation.

The contact information for the American headquarters is given below. (See the web for more international contacts.)

Data Translation Inc.
100 Locke Drive
Marlboro, MA 01752-1192
U.S.A.
Phone: 1 (508) 481-3700
Fax: 1 (508) 481-8620
Web: http://www.datx.com/
Email: info@datx.com

References

[1] S. B. Dalziel, G. O. Hughes, and B. R. Sutherland. Whole field density measurements. *Experiments in Fluids*, 28:322–335, 2000.

[2] S. Sakai. Visualisation of internal gravity waves by Moiré method. *Kashika-Joho*, 10:65–68, 1990.

[3] B. R. Sutherland, S. B. Dalziel, G. O. Hughes, and P. F. Linden. Visualisation and measurement of internal waves by "synthetic schlieren". Part 1: Vertically oscillating cylinder. *J. Fluid Mech.*, 390:93–126, 1999.