Technique for plume velocity determination using image correlation

LRAP-173

Karin Amnehagen
Göran Sandberg

Abstract

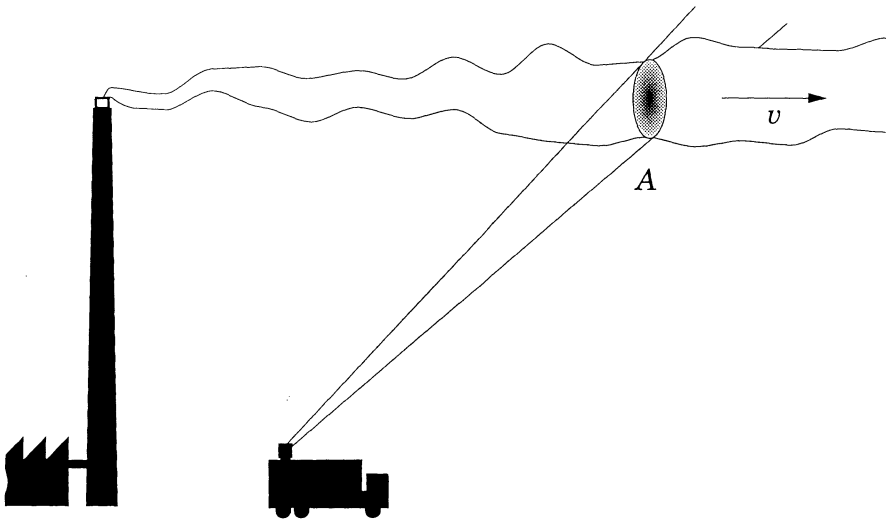
The possibility to use the smoke plume from a factory smoke stack as an anemometer was investigated. A method using a CCD video camera to monitor the smoke plume and a PC equipped with a video frame grabber to calculate the wind velocity by correlating subsequent images was developed.

Table of Contents

Introduction

The need for remote wind velocity determination

The LIDAR system developed at the department of Atomic Physics at Lund University has made it possible to make accurate remote measurements of the concentration of a given substance in the atmosphere[1]. This system can, for example, be employed to monitor the concentration of pollutants emitted from an industrial process.



LIDAR measurements

In the LIDAR system a powerful pulsed and frequency tuned dye laser beam is directed at the area of the atmosphere where the measurement is to be made. By using a telescope to observe the back scattered light from the laser beam when it excites its target substance, the concentration of the substance can be determined along beam. The laser beam can be made to sweep a given area of the atmosphere, giving a cross-section of the substance distribution.

This method can be used to determine the amount of pollutants in the smoke from a factory smoke stack. By letting the laser dissect the smoke plume the concentration, c, can be found. If the velocity, v, at which the smoke transverses the cross-section is also known, the actual emission of pollutants can be found using:

$$emmission\left[kg/s\right] = v\left[m/s\right] \cdot \int_A c\left[kg/m^3\right] dxdy \,,$$

where A is the area of the cross section of the plume.

This calls for a reliable method to measure the wind velocity at the top of the smoke stack. The obvious solution would be to use an anemometer. This is, however, not a good solution if the measurement is going to be made without the knowledge of the operator of the factory. It is also not a very practical solution, if the smoke stack is tall. Thus a remote measurement method has to be devised.

Selecting a remote wind velocity measurement method.

The basis for this work is the suggestion to use the smoke plume itself as the wind velocity detector. By somehow observing the motion of the smoke plume, which is assumed to move with the wind, the wind velocity can be measured.

The methods suggested to us were to use the LIDAR laser to track the edges of the smoke plume, an indirect method using the laser to track a balloon, and to use a video camera and image processing equipment to detect the motion of the smoke plume.

We also launched a large scale information search for articles concerning wind velocity measurements. Unfortunately very little material was to be found on this subject. The closest reference was a Japanese article concerning the use of a video camera to measure the movement of clouds. A French paper actually described the smoke plume velocity measurement, but the method suggested was to superimpose a stopwatch on a video image of the smoke plume. The user then manually had to watch the timer and the smoke plume to determine the speed. A fascinating method was to shine a Helium-Neon laser through the air and observing the speckle pattern using a CCD camera. By connecting the camera to a neural network, the speckle pattern could be used to determine the wind speed. With the proper equipment, it might also be possible to construct a laser-doppler anemometer.

After having looked through the material we decided first to try using a video camera and image processing software. The smoke plume edge tracking method had already been tried by Eva Wallinder and we felt a bit discouraged by the complexity of the LIDAR system. Besides, the LIDAR system will be busy making the atmospheric measurements, and it might be desired to measure the wind velocity during those measurements. Before making the decision we had the opportunity to test a brand new frame grabber that had been installed in a computer to which we had easy access. Experiments with false colour video display and a dark and cloudy sky clearly showed that the camera and frame grabber could separate levels of grey that the human eye could not distinguish. This reassured us that a method using a video camera should be able to work even on a cloudy day.

Implementing the wind velocity measurement program

Having decided to use a video camera, we had to find a way to calculate the wind velocity from the image data. Luckily we soon found a promising method in the book Digital Image Processing by Gonzalez and Wintz[2]. To find the velocity, a series of images were to be Fourier transformed, added together and transformed again. The method was supposed to be good at measuring the velocities of small objects against noisy backgrounds. This, we thought, would be handy to single out the moving smoke plume against a clouded sky.

Unfortunately the implementation of the above method was crippled by the limitations imposed by the speed of the computer system and frame grabbing hardware. The rate at which images could be captured was to low for the method to function properly, and unpredictable results were produced.

This called for a new method. During the experiments with the method above, we had found that the image data from the frame grabber was very good, and that it would not be any problem to see the smoke plume, not even against a rainy sky. This encouraged us to use straightforward correlation between two consecutive images. This method was implemented using a fast Fourier transform correlation algorithm. This time the results were highly accurate.

During the implementation we also faced practical problems such as how to interface the frame grabber to our program and how to handle smoke plumes that are not parallel to the camera image plane. These time consuming tasks are described in detail in the appropriate sections.

About this report

This report is divided in three sections describing different aspects of the project. Related information can be found in the appendices.

- Theory. Theoretical description of the methods employed to calculate the wind velocity
- Implementation. Technical description of the actual system implementation
- Evaluation of the methods

Appendices

- Users manual for the wind velocity measurement program
- The Frame grabber
- Program listings

Theory

When using a video camera and a frame grabber to measure the wind velocity, a method is needed to find the displacement of the smoke plume between two consecutive images.

In the search of such a method we had to consider several factors:

- The method should be able to distinguish the smoke plume from a clouded and noisy background.

  The initial experimentation with the frame grabber had shown that it would be possible to detect very small variations in light intensity across the image. With a suitable method we imagined that it would be possible to measure wind velocity even under the most difficult conditions of a cloudy and rainy sky.

- The method should work under widely varying wind velocities.

  When a video camera is used to capture the image of a smoke plume many factors influence the speed at which the plume traverses the image. Different distances, camera focal lengths and wind velocities require a wide range.

- The frame grabber could only store two images at a time.

  This suggested that the method had to use only two images to calculate the velocity. A way around this would be to copy each image, or part of it, to the computer RAM.

- Memory limits imposed by Turbo Pascal made it impractical to store more than one complete image in the computer RAM.

  This required much of the processing to be done directly on the image stored in the frame grabber.

With this in mind, we searched for a suitable method. After an extensive literature search a promising method was found in Digital Image Processing. The method is based on Fourier transforms of the images. It was implemented, but unfortunately it proved to be unsuitable, so another method had to be found. We then chose to make a cross correlation between two consecutive images. The methods are described in detail below.

Fourier transform method

The method suggested by [2] appeared to fill the above criteria well. According to experimental data from the book, the ability to measure

measure the velocity of an obscure object against a complex background was very good.

In the method, the pixel values are summed for each column of the image. This projection of the image data on the horizontal axis is then stored in the computer, requiring much less space than a complete image. Thus a number of images can be captured and stored in the computer RAM.

To measure the wind velocity, the following steps are carried out:

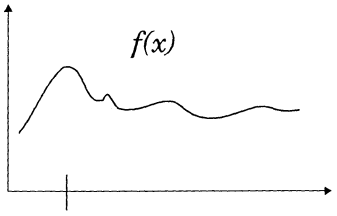
- $N=2^n$ images are captured
- The column sums are calculated to project the image on the horizontal axis.
- The projection of each image is Fourier transformed to form a periodic function whose period reflects the displacement between two captured images.
- This periodic function is then Fourier transformed to obtain the displacement in pixels per second.

Theory

The smoke plume is assumed to be perfectly rigid and moving at a constant velocity, v. At the sample intervals, $t=t_0,t_1,t_2...t_{N-1}$, the image projection is Fourier transformed giving:

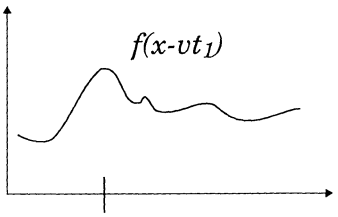
The smoke plume projection for different times, t

$t = t_0$



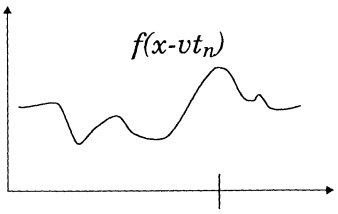
$$\mathcal{F}(f(x)) = \hat{f}(k)$$

$t = t_1$



$$\mathcal{F}(f(x - vt_1)) = e^{i\frac{2\pi}{N}kvt_1}\hat{f}(k)$$

$t = t_n$



$$\mathcal{F}(f(x - vt_n)) = e^{i\frac{2\pi}{N}kvt_n}\hat{f}(k)$$

where

$$t_n = n \cdot \Delta t$$

$N \geq v, \quad \text{for } k = 1$

The speed at which the plume transverses the image can be varied using the zoom lens to be below 32 pixels per sample. The user must adjust the lens manually until he estimates that a proper velocity is obtained. This allows us to select k using the condition from the Nyquist sampling theorem

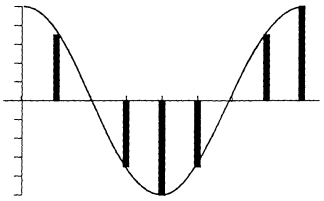
$$k \leq \frac{N}{v}$$

In order to get maximum velocity resolution it is desired to select k as large as possible. According to the statement above, k=1 must be chosen if the maximum expected velocity is 32 pixels/second. On the other hand, if the velocity is not expected to exceed 16 pixels/second, k=2 can be selected, and so on. See also the examples below.

The velocity, v, is 2 pixels per second. $\Delta t = 0,5s$, N=8 samples.

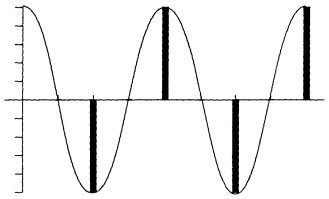
k=1

$v_{\max} = 8$



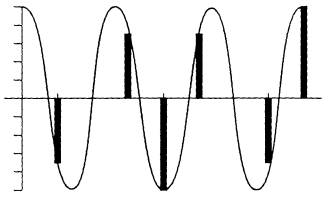
k=2

$v_{\max} = 4$



k=3

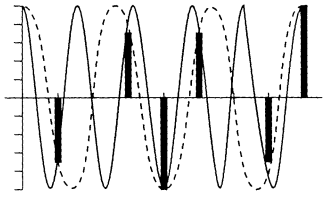
$v_{\max} = \frac{8}{3}$



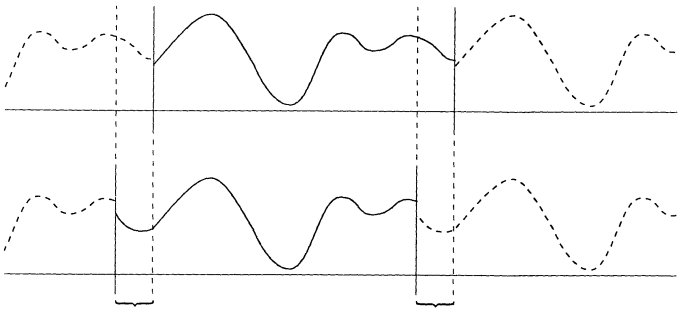
It is not possible to find the correct frequency with less than two samples per period.

k=5

$v_{\max} = \frac{8}{5} < 2$

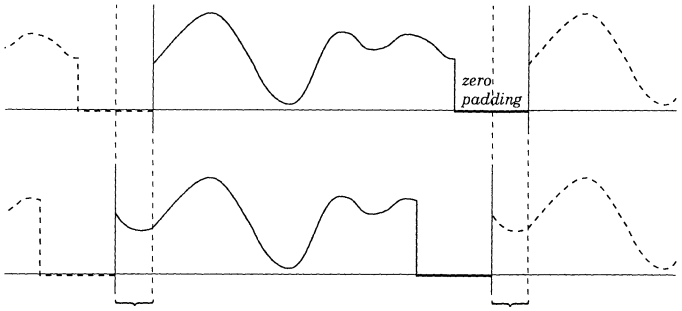


Without zero padding the correlation is polluted by wrap-around effects



Correlated with irrelevant data

With zero padding the wrap-around effects are eliminated



No contribution to correlation

Inside the CCD camera a special integrated circuit, the CCD chip, is placed in the image plane. The image is projected on the surface of the chip, which is sensitive to light and the image pattern is encoded to a video signal. CCD chips come in different sizes, and it is important to know the size in order to be able to calculate the actual size of an object in the image from the image data. The standard sizes are 1" and 1/2", measured across the diagonal. The ratio between the short and long edge is usually 9:13. The CCD size is used by the Wind16 program to calculate the wind velocity.

During the development it was found that it is not obvious how to use the CCD size to calculate the actual object size.

- It is difficult to find the exact CCD size and ratio for a given CCD camera.
- It is not safe to assume that the entire chip surface is used to produce the image.
- The frame grabber may not use all of the image data from the camera.

To allow for these factors, a camera calibration procedure was added to the Wind program. Any camera can be calibrated using an object of known size at a known distance. The CCD size is then only used to give a rough estimate of the calibration factor. This also makes it possible to use any video camera as the image source (although it would probably be hard to find a camera that doesn't use a CCD chip today).

We used system a JVC model TK-S300EG CCD camera having an 8-48mm zoom lens during the system development. We also used a portable camera with a built in video recorder to make field measurements.

The computer system

An ordinary IBM-PC compatible computer was used for the image processing. The images from the CCD camera are captured by a digital frame grabber card inside the computer (see appendix). For the Fourier transformation method described in the theory section above, it is important for it to be as fast as possible, since the method requires a relatively small displacement of the smoke plume between two images. The system used, a 25MHz 80386 with numeric coprocessor, proved to be too slow for the method to give predictable results. For the correlation method, on the other hand, it was adequate.

The software was developed in Turbo-Pascal to make it easier to integrate with the existing software for the LIDAR system. Special software routines had to be developed to make it possible to access the

frame grabber card from Turbo-Pascal. See appendix for complete program listings.

The software allows the evaluation of the two different methods accounted for in the theory section. It also allows the user to make a single shot measurement of the wind velocity and to manually evaluate and adjust the calculated velocity if necessary.

The video monitor

The output of the frame grabber is connected to a separate colour video monitor. This allows the image from the CCD camera to be viewed at all times. The monitor is also used to select the smoke plume area of the image and to calibrate the video camera.

The distance measurement system

In order to calculate the actual wind velocity it is necessary to know the exact distance to the smoke plume. To do this the existing LIDAR system is used. A laser pulse is fired at the plume and the return time for the reflection of the pulse from the plume is measured. This gives an accurate distance to the plume. By measuring the distance at two different angles, the angle between the plume and the camera image plane can be determined.

Measuring the wind velocity

Setting up the camera

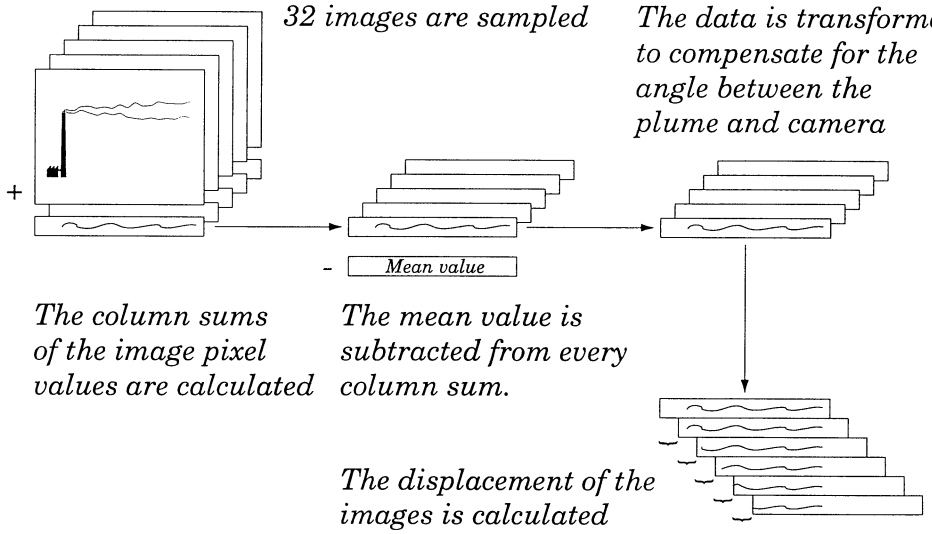
After the camera has been calibrated as described in the Wind Users manual, it must be properly positioned in relation to the smoke plume. The wind velocity measurement program makes corrections for different angles between the plume and the camera image plane, so there is no need for the image plane and plume to be parallel as long as the angle is known. The angle should preferably not be too great, since the plume is heavily distorted when projected on the image plane.

Because the program only measures the velocity along the horizontal axis of the image, the camera should be adjusted so that the plume transverses the image plane horizontally.

The focal length of the camera is then adjusted to make the plume pass the image field at an appropriate speed for the selected method. For the Fourier method the plume should transverse the image field slow enough for the measurement to be completed while some part of the

original plume is still visible. For the correlation method the plume should move significantly between the exposures.

After the camera has been set up, the Wind program allows the user to select a subpart of the image to be used for the measurement. This makes it possible to edit out other moving objects in the image field. For Fast Fourier Transform methods the number of pixels across the active area of the image is limited to powers of two. During the setup, the frame grabber operates in pass-thru mode, allowing the adjustments to be viewed on the video monitor connected to the frame grabber.



Software overview

Capturing images

During the measurement phase the frame grabber captures a predefined number of images with a known time interval. Since the frame grabber can store only two images, it is not possible to capture all the required images at once and each image must be processed before the next is acquired.

Both the measurement methods require the sums of the image pixel values to be calculated for every pixel column in the image. This calculation is made by an assembly language routine as soon as the frame grabber has finished capturing the image. Only the area previously selected is used for the calculation. This process takes about 0.5 seconds for a full image. The column sum array is stored in a linked list and the captured image data is overwritten by the new image. This is repeated for all the images to be captured.

The time it takes to calculate the column sum is dependent of the size of the area to be processed. Unfortunately the speed of the system does

not allow more than two images to be captured every second. This pu
a severe limitation on the usefulness of the Fourier method, which
needs short intervals.

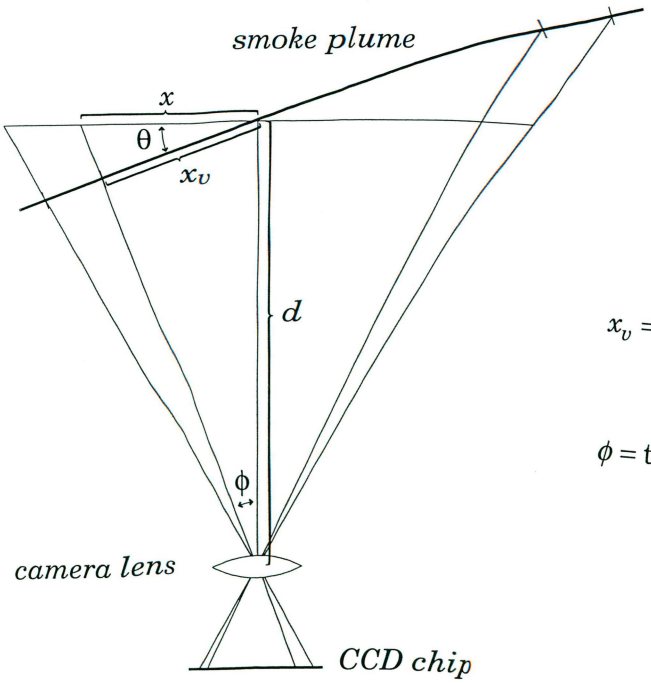
Another limiting factor is the amount of available memory. Since our
Turbo-Pascal implementation can only make use of the standard one
megabyte of IBM PC memory, storage is scarce. This limits the numbe
of image projections stored in arrays that can be held simultaneously i
memory to about 64. This also decreases the accuracy of the Fourier
method.

Subtracting the background

When all the images have been captured, the linked list of image pro-
jections is passed to a procedure which calculates the mean column sum
for every column. This value is subtracted from each of the images in
order to allow the dynamic data coming from the moving smoke plume
to be seen over the strong static background.

Making corrections for plume angle

If the camera image plane and the smoke plume are not parallel, the
image projections have to be transformed to compensate for the angular
deviation. Otherwise the plume velocity would vary along the horizon-
tal axis, see the figure below.



$$x_v = \frac{\cos\phi}{\cos(\theta+\phi)}x$$

$$\phi = \tan^{-1}(\frac{x}{d})$$

The transformation is done by projecting the image data on a plane parallel to the smoke plume.

Finding the displacement of two consecutive images

When all image data has had the background subtracted and the angles corrected it is passed to either of the two methods described in the theory section. Fourier method uses all the captured images, while the cross correlation method gives a result for every pair of images. To get a final result for the later method, the median result of all the images is used. The output from the methods is the displacement between two images given in pixels. The real wind velocity is then calculated using the following factors:

- Camera focal length
- Distance
- Angle
- CCD-size
- Camera calibration factor

Manual evaluation of the result

After the automatic velocity measurement, the image data can be viewed on the computer monitor to validate the result. The image projections for all the images are plotted below each other displaced using the calculated velocity. The users brain can then be used to perform image correlation, a task which it is very well suited for. The diagram can be adjusted until the user is satisfied that the correct correlation, and thus wind velocity, is found.

Evaluation

The Fourier method

After having implemented the Fourier transform method, the time had come to verify that it could be used in practice. We borrowed an ordinary video camera and went plume hunting in an industrial area on a rather windy day. Several plumes where filmed with different shapes and backgrounds, and from different distance and angle. We made a special effort to film the same plume from different angles and distances, and of course different focal lengths, to make sure that this did not effect the result.

The evaluation of our film made us rather disappointed. Some results seemed to be very close to reality, but others were out in the blue. What made the results so unpredictable?

As described in the theory chapter, the maximum wind speed that can be measured with the Fourier transform method is 32 pixels/second, that is if the number of pictures taken is 32 and the sample interval is 0,5s. From a distance of 500 meters and with a focal length of 50mm, this translates into roughly 4.5 meters/second, a not very impressive value. Even so, this speed would cause the plume to transverse the screen so fast that the last picture would not include any part of the plume from the first picture. Our approximations would thus become seriously questionable. With a focal length of 20mm, it would be possible to measure wind velocities up to around 11 meters/second, but the same reasoning applies for the validity of our approximations.

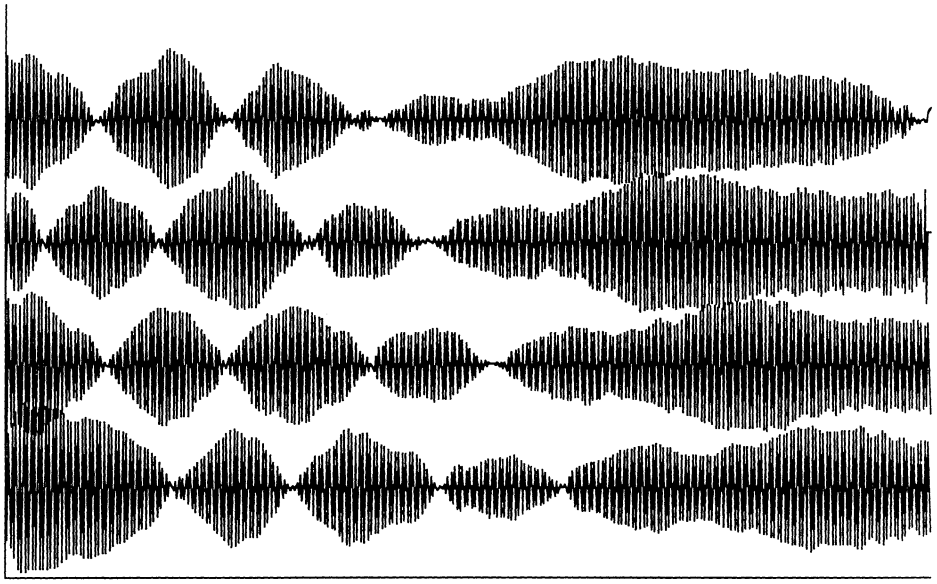
To make our approximations more correct, it is necessary to make sure that a great part of the plume present in the first picture is also present in the last. Since the computer does not allow us to take the pictures with a shorter time interval, the only way to accomplish this is to take fewer pictures. According to the requirements of the FFT algorithm the number of pictures must be of a power of two, so we half the number to 16. This value assures that 3/4 of the original plume are still present in the last picture, but the maximum wind speed that can be measured is reduced further to around 2.25 meters/second with a focal length of 50mm and 5.5 meters/second with a focal length of 20mm. When we made our film the wind speed was around 15 meters/second which explains our lousy results.

The Fourier method could probably be very useful with a faster computer that would allow pictures to be taken at a much higher frequency, but why bother - we found the one and only correlation method.

The correlation method

The correlation method does not impose any restrictions on what wind velocities are possible to measure or the number of pictures to take. It is, of course necessary that some part of the plume is present in both the pictures to correlate, but this is easily achieved since only two pictures are needed and for that the time limit set by the computer is not important.

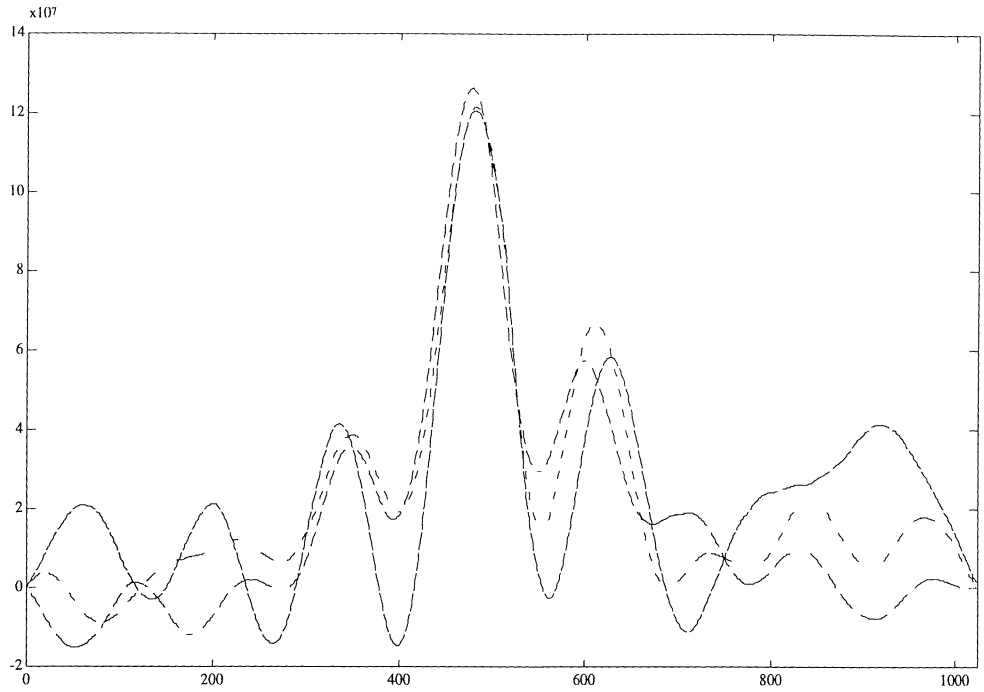
We were immediately encouraged by the results from using the correlation method on our video tape. As is shown below, the structure of the smoke plume is very evident in the consecutive samples from the video tape. The diagram below shows four samples where the displacement is clearly visible. The diagram was created by calculating the sum of the pixel values for each image pixel column. The data for each sample was then mirrored across the x-axis to create an illusion of a smoke plume.



Four consecutive smoke plume samples.

The results from the correlation of the images are shown on the top of next page. The correlations show very clear peaks, indicating that the plume structure is rigid enough in the time interval between two samples.

To evaluate the variation of the wind velocity given by the correlation method the correlation of 16 samples was calculated. The standard deviation of the samples was also calculated. See the table below:



The result of the correlation of the four samples in the diagram on the previous page.

Focal length=20 mm, Distance=560 m, Sample interval=0,5s

The wind velocity obtained from the above data

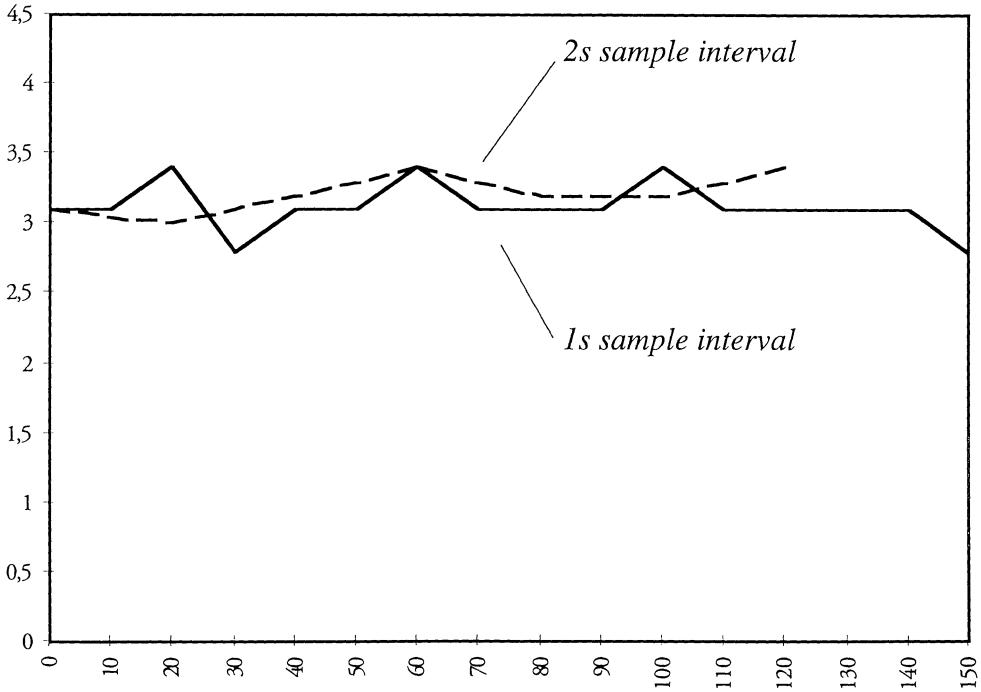
Time (s)	Displacement (pixels)	Wind velocity (m/s)
0	19	6,4
0,5	37	12,5
1	37	12,5
1,5	36	12,1
2	38	12,8
2,5	41	13,8
3	39	13,1
3,5	35	11,8
4	36	12,1
4,5	33	11,1
5	32	10,8
5,5	36	12,1
6	37	12,5
6,5	32	10,8
7	22	7,4
Median	36	12,1
St. dev.	6,0	2,0

If the first and last values are disregarded the following result is obtained:

St. dev.	2,6	0,89

Focal length=24 mm, Distance=560 m, Sample interval=1s

Time (s)	Displacement (pixels)	Wind velocity (m/s)
0	11	3,1
10	11	3,1
20	12	3,4
30	10	2,8
40	11	3,1
50	11	3,1
60	12	3,4
70	11	3,1
80	11	3,1
90	11	3,1
100	12	3,4
110	11	3,1
120	11	3,1
130	11	3,1
140	11	3,1
150	10	2,8
Median	11	3,1
St. dev.	0,57	0,17



A plot of the wind velocity measured over a period of 150s.

What is evident from the above measurements is that the method gives consistent values for the wind velocity under similar conditions. Thus it is only a matter of correct calibration of the measurement system to find a credible wind velocity using the correlation method.

Improvements and suggestions

From the discussion above is clear that the method needs to be thoroughly tested and calibrated. It would also be necessary to incorporate the wind velocity measurement system in the new LIDAR system which is presently being developed.

In its present state, our method is most suited for stand-alone use to make single shot wind velocity measurements. It would, however, be fairly easy to adopt it to continuous measurements. The scheme of making 32 samples and take the median value as the result presented could be abandoned. It would then be possible to make, lets say, two samples every minute and using them to find the wind velocity. The drawback of this is that the measurement could be disturbed by, for instance, a bird that flies across the image field.

Using a specially adopted CCD camera

The most computationally intense part of our velocity measurement method is the summing of the columns of the captured image data. Considering the nature of the CCD chip inside the camera, it would probably be easier to make this calculation inside the camera and only send the concentrated data to the computer. This could perhaps be done by rotating the CCD-chip 90 degrees and having special hardware to sum the pixel values as each line is read from the chip.

An alternative method

From the experimentation with the above methods it was obvious that a manual correlation of the image data would be a very powerful way to find the wind velocity. This manual method might actually be the fastest and most convenient wind velocity measurement method. Given the support of computerised scaling and velocity calculation, an experienced user would be able to easily find the correct velocity.

Users Manual

The wind velocity measurement program is started either by typing "wind16" at the prompt, or by loading it into Turbo Pascal and running it by selecting the "run" menu option. The program is entirely menu driven and when started the main menu appears:

```
*****************************************
*****    WINDSPEED MEASUREMENT MENU  *****
*****************************************

[F1]:  Set region of interest
[F2]:  Set parameters
[F3]:  Measure wind velocity
[F4]:  Plot smoke profile

[F5]:  Input source:         camera
[F6]:  Method:               correlation

[F7]:  Set debug options
[F8]:  Save smoke profile as text
[F9]:  Calibrate camera

[F10]:  End program.

Enter function key, please...
```

A detailed description of the menu options is found in the chapter Menu options below.

Measuring wind velocity

Before any measurement can be carried out, the camera to be used must be calibrated, if this has not already been done. The Wind16 program can store the calibration for up to five different cameras. For instructions on how to calibrate the camera, see main menu option F9.

After the camera has been properly set up and adjusted the wind velocity can be measured as follows:

1. Use [F2] to set the following measurement parameters:
   - The camera focal length
   - The distance to the smoke plume
   - The angle between the camera image plane and the smoke plume.

- 'F9' Angle. The angle, given in degrees, beween the image plane of the camera and the smoke plume. For smoke parallell to the image plane, the angle is 0.
- 'F10' returns to the main menu.

If the program has been set up to test the fourier measurement method one more menu option is available:

- 'F2' Number of samples. This selects the number of images to be captured for the measurement. The number of images must be a power of two. The more images, the better velocity resolution. The amount of free memory of the computer limits the number of samples to 64. When using the correlation method, the number of samples is always 32.

'Measure wind velocity' menu selection

Pressing 'F3' starts the wind velocity measurement. If no debug options hav been selected using menu option 'F7' the screen blanks until the measurement is completed. This process may take several minutes. When the measurement is finished the wind velocity is displayed on the screen. Press 'Enter' to continue.

If the debug option 'Save images' has been selected you will be prompted for a filename under which the data will be saved before the measurement starts. Also, if the debug option 'Screen output' is selected you will be asked to enter a comment. The screen will display debuging information showing the progress of the measurement.

Once 'Enter' has been pressed the main menu is redisplayed. In order to view the result again, 'F4' has to be pressed to plot smoke profile. The wind velocity can then be read in the top right corner of the display.

'Plot smoke profile' menu selection

By pressing 'F4' a graphical representation of the image data is presented. The image data for each image is ploted with a displacement depending on the maesured velocity. If all is well, the characteristics of the data for the consecutive images should be well aligned. The calcualted wind velocity is also displayed in the top right corner.

This option also gives the user the oportunity to manually determine the wind velocity from the sampled image data. By using the right and left cursor control keys on the computer keyboard, the display can be adjusted so that the features of the inage data aligns properly. The displayed wind velocity is changed accordingly.

To leave the graphics display, press 'F10'. Please note that any manual adjustments to the wind velocity are lost.

'Input source' menu selection

This allows the user to use a file previously stored to disk using the debug option 'Save images' to replace the data from the camera. When 'F5' is pressed, the menu changes to show that the input source is 'disk'. Press 'F5' again to select the camera as the input source.

When the input source is 'disk', the program will prompt for a file name when wind velocity measurement is started by pressing 'F3'. Please note that smoke profiles stored as text using 'F8' cannot be used as disk input.

'Method' menu selection

Pressing 'F6' toggles between the 'fourier' and the 'correlation' method. The fourier method is provided for evaluation purposes only, and should not be used during normal wind velocity measurement.

'Set debug options' menu selection

Pressing 'F7' displays the debug options submenu. This menu allows the user to display data on the screen showing the progress of the program and to save various information to the disk:

```
*****************************************
*****          Debug options          *****
*****************************************

                        Current value

[F1]:  Save images              Off

[F3]:  Save fourier data        Off
[F4]:  Screen output            Off
[F5]:  Save file                Off


[F10]:  Leave this menu.

Enter function key, please...
```

- 'F1' Save images. This option allows the user to save the image data to disk, so that it can be used to replace the camera as input source. When the measurement is started, the user is prompted for a file name.

- 'F3' Save fourier data. When using the experimental fourier method, this option allows the results from the fourier transforms to be written to text files.
- 'F4' Screen output. This option displays various internal values during wind velocity measurement.
- 'F5' Save file. When selected this option prompts the user for a file name under which all the screen debuging output is to be written. This useful feature allows the operation of the program to be carfully studied.
- 'F10' returns to the main menu.

'Save smoke profile as text' menu option

Pressing 'F8' after a measurement has been completed saves the image data as a text file. This is useful if the data is needed for a charting application, such as Microsoft Excel.

'Calibrate camera' menu option

In an ideal world it would be sufficient to know the focal length of the camera and the sizo of the CCD chip inside of it in order to calculate the correct wind velocity. Unfortunately the active size of the CCD may vary. This calls for an option to calibrate the camera.

The camera is calibrated by filming an object of known size at a known distance using a lens with a known focal length. It is therfore important to set the distance and focal length parameter in the 'Set parameters' menu option by pressing 'F2' prior to selecting 'F9'. The camera number to be calibrated must also be selected in the 'Set parameters' menu option. Once this is done follow these steps to calibrate the camera:

1. Press 'F9'. This starts video pass-thru mode. The camera image is shown on the video monitor.
2. Use the cursor control keys on the computer keyboard to position the cross-hair cursor at the left side of the object of known size.
3. Press 'Enter'. Red lines mark the left edge of the object.
4. Use the cursor control keys on the computer keyboard to position the cross-hair cursor at the right side of the object.
5. Press 'Enter'. A red line is drawn to the right of the object. On the computer screen you are prompted for the size of object.
6. Enter the size of the object and press 'Enter'. If you want to leave the calibration unchanged enter '0' for the size. The main menu is displayed.

If you want to change the calibration, simply press 'F9' again. To see the calibration factor, select 'F2' in the main menu and look at the 'Select camera no' line.

'End program' menu option

Selecting 'F10' ends the program. The settings of focal length, distance, angle, camera number, and camera calibrations are saved and reloaded the next time the program is started.

File locations

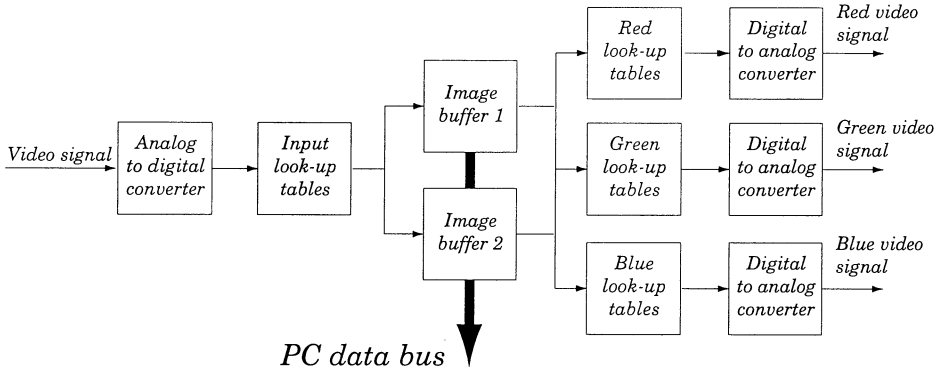
The Wind16 program uses two subdirectorys to store its data. The 'PREF' directory stores the WINDPREF file that stores the parameters when the program is quit. The 'DATA' directory holds the output of the various debug files.

These directorys have to be subdirectorys of the main Wind16 directory, or the program will crash.

The frame grabber

The DT2851 frame grabber

To capture the video images the computer was equipped with a Data Translation DT2851 frame-grabber card. This card has an input for video signals and an output for a video monitor. The operation of the card is entirely controlled by the host computer. A block diagram of the card is shown below:



The input video signal is passed through an analog to digital converter, where the signal brightness level is converted to an eight bit digital value (0-255). This value is used to index the input lookup table. This table maps every incoming digital value to a new, arbitrary eight bit digital value. The input lookup table makes it possible to adjust the brightness and contrast of the image, or to make special effects, such as inverting the image before it is stored. In order to store the image as is, the table is loaded with a linear series form 0 to 255, so that every input value is mapped to a similar output value. There are eight different input look-up tables that can be individually set up. It is then easy to direct the input through any of the tables.

The output from the input look-up table is then stored in one of the two frame buffers. The buffers each store 512 by 512 pixels of 8 bit (256 levels) greyscale image data. There are also facilities to protect selected bit levels so that only 7, 6 or 4 bit data is stored. This makes it possible to superimpose a previously stored image onto the image being sampled.

The contents of the frame buffers can be displayed on a colour video monitor. The image data is put through three different output look-up tables, one for each of the red, green and blue colours. It is then put through three digital to analog converters and fed to the RGB monitor outputs. This makes it possible to false-colour the greyscale image e.g. to enhance the visibility or to colour code the information. By making the three output look-up tables similar, the original greyscale image is displayed. The use of the output look-up tables has the advantage of making it possible to manipulate the image without changing the original data.

The DT2851 also contains advanced functions to allow the sampled images to be added, subtracted and otherwise transformed, but none of these functions were needed for our project.

The Turbo-Pascal interface for the frame grabber.

The frame grabber was delivered with a hands-on demonstration program, and a library of precompiled Fortran interface subroutines. Unfortunately none of the source code was available. At that time, the project was intended to be included in the present LIDAR software and for compatibility reasons it had to be carried through using Turbo-Pascal. Since Pascal uses a different subroutine calling convention from Fortran, the existing DT2851 interface subroutines could not be used.

The solution, unwillingly undertaken, was to write a new library of interface subroutines using Turbo-Pascal. Fortunately, the DT2851 documentation was of great help. We decided to mimic the supplied Fortran subroutines, but to limit ourselves to the routines we thought we could use in our project. We thought we could use functions to take care of the following tasks:

- Controlling the operations of the frame grabber
- Manipulating look-up tables
- Controlling the cursor
- General housekeeping functions

We also realized that we needed subroutines to access the data in the image buffers. These were not available among the Fortran routines, for reasons that would be apparent to us later on.

Controlling the operation of the frame grabber

The following procedures were needed to control the operation of the frame grabber:

- dt_wait

  Waits until the frame grabber has completed the current sample. This is used to synchronize calls to the different frame grabber functions.

- dt_display

  Turns the video display on and off.

- dt_passthru

  Starts video passthru mode. The video image being sampled is simultaneously output to the video monitor.

- dt_freeze_frame

  Stops the passthru mode and freezes the image in the frame buffer.

- dt_acquire

  Samples one video frame and stores it in the frame buffer.

- dt_select_input_frame

  Selects which of the two frame buffers should be used to store the sampled image.

- dt_select_output_frame

  Selects which frame buffer to display on the video monitor

- dt_set_sync_source

  Controls if video synchronization should be taken from the input video signal or generated on board.

- dt_load_mask

  Loads a mask to protect certain bits in the frame buffer, making it possible to superimpose images.

Manipulating look-up tables

We decided to implement most of the functions concerning the look-up tables. It was thought that there was a need to make us of contrast enhancement to be able to single out the smoke from a clouded, grey background. Practical experience has, however, shown that there is no need for such elaborate measures. The use of the look-up tables is therefore limited to the display of a red rectangle, showing the image area selected for processing. The implemented look-up table functions are:

- dt_fill_ilut

  Fills the specified part of a input look-up table with a given value.

- dt_load_ilut

  Fills a input look-up table with the values from a 256 element vector.

- dt_fill_olut

  Fills the specified part of the output look-up tables with given values for red, green and blue.

- dt_load_olut

  Fills the output look-up tables with the values from three 256 element vectors, one for each of the colours red, green and blue.

- dt_select_ilut

  Selects the input look-up table to be used.

- dt_select_olut

  Selects the output look-up tables to be used.

Controlling the cursor

Two simple functions control the built-in cursor of the DT2851 frame grabber. The cursor is used to designate the image area to be processed, and to calibrate the video camera against objects of known size.

- dt_cursor

  Turns the cursor display on and off.

- dt_set_cursor_position

  Places the cursor at a given position.

Accessing the frame grabber image data

In order to be able to process the captured images we had to be able to read the data from the frame buffer. The Fortran subroutine package did only provide a very rudimentary facility to save image data to the disk. This would be much too slow for our needs, and would quickly fill the disk, considering that every image occupies 256 kilobytes. Besides, we had no way of using the Fortran subroutines.

Due to the constraints imposed on memory access by the design of the IBM PC the direct access to the frame grabber image data proved to be a daunting task. An original IBM PC can only access 1 Mbyte of RAM-memory. By using various "standardized" constructions such as the Lotus-Intel-Microsoft Extended Memory System, LIM-EMS, modern computers can access more memory. The 512 kilobytes of frame grabber memory was addressed well beyond the limits of the standard PC. This made it impossible for us to read the contents of this memory from Turbo-Pascal.

Our first plan was to find out more about the workings of the LIM-EMS to see if it could help us to read the memory cells. To our great despair, the pages describing the subject was missing from the only book we could find on the subject. We did, however, eventually find out that it would take a monumental programming effort to solve the problem this way. A work-around had to be devised.

The solution was to do some dirty 80386 assembly language programming. The task was reluctantly undertaken, since neither one of us were familiar with 80386 assembly programming, or, for that matter, assembly programming at all. By using 32 bit addressing available in the 80386 processor, the computer could be tricked into reading the frame grabber memory. This is not the proper way to do it, though, and the computer can crash if it is not appropriately set up. Practical experience has shown that the program HIMEM.SYS and the disc cache SMARTDRV has to be loaded in order for this work-around to function properly.

The assembly language code can be found in Appendix X. The code was assembled once and for all using the Turbo Assembler, and the object file is linked to the Turbo Pascal wind velocity measurement program when it is complied. The following routines are added as external calls:

- get386mem

  Returns the value of the byte stored at a given memory location.

- put386mem

  Stores a byte at a given memory location.

- copy386roi

  Copys part of the image stored in the frame buffer to an other location.

- sumx386roi

  Calculates the column sums of image pixel values and stores them in a vector.

These procedures are called from pascal procedures:

- dt_copy_roi

  Calls copy386roi to copy image data.

- dt_sum_columns

  Calls sumx386roi to calculate column sums.

- dt_draw_line

  Uses put386mem to draw lines in the frame buffer. This is used to put a red rectangle on the video monitor, showing the selected area.

General housekeeping functions

To properly initialize the frame grabber and to load the various look-up tables with default values giving a greyscale image, we added the procedure:

- dt_initialize

To set up the region of interest, which designates the image area to be processed, a final procedure was added:

- dt_define_roi

The Wind program

All image processing and velocity calculation is performed by the Wind program. This section describes the data structures and functions used.

Data structures

The image data from the frame grabber is stored in a linked list of longint arrays:

```
picture_pointer = ^picture_frame;
picture_frame   = record
                     picture:array[0..512] of longint.
                     next   :picture_pointer;
                  end;
```

When the images are captured a region of interest can be defined. This limits the area of the processed part of the image.

The region of interest parameters are stored in the structure:

```
RoI = record
         buffer    :longint;
         xpos,xsize:word;
         ypos,ysize:word;
      end;
```

When the column sums are calculated for a given region of interest, the data is shifted so that the first column sum becomes the first element of the picture array.

Some variables needed by many parts of the program are globally defined. Examples of important globals are given below:

```
region       :RoI;       {The region presently defined}

picture_list :picture_pointer;{Pointer to the first
                               element of list of
                               stored image data}

focal_length :real; {Camera and measurement parameters}
CCD_size     :real;
CCD_xsize    :real;
```

```
distance      :real;
angle         :real;

magnification:real; {The image magnification calculated
                     from the above parameters}

scale         :real; {The angle scale factor}
```

The Wind program units

As new functions have been added the Wind program has grown to a rather inconvenient size. In order to make an overview easier, it has been divided into functional blocks using Turbo Pascal Units. The units are described below, with the exception of the frame grabber interface unit, dt_turbo, which is described in section ?. The units are:

- **Wind**. This is the main program providing initialization procedures, general calculations, frame grabber access, and the main menu selections loop. It also provides the wind velocity measurement procedure which calls the other modules.
- **Global unit**. This unit contains all global variables and a procedure to set them up to their default values.
- **User unit**. This unit provides the user interface with menus and cursor control. It is used to set up the image area to be processed and to make the camera calibration.
- **Debug unit**. Contains procedures to put debug information on screen or disk.
- **Corr unit**. All the procedures to measure the wind velocity using the correlation method.
- **Four unit**. All the procedures to measure the wind velocity using the Fourier method.
- **Dt_2**. Turbo-Pascal interface for the Data Translation DT2851 frame grabber.
- **Getput**. This is the assembly language interface for the DT2851 frame grabber.

In addition to the above units some general input units were reused from the present LIDAR software. These units are: Initial, Input_u and Readdir_u. They are documented in the LIDAR software documentation.

Wind

This unit contains the following procedures and functions:

- **initialize_frame_grabber**. Initializes the frame grabber card and selects appropriate input and output look-up tables. It then starts pass-thru mode, displaying the image on the video monitor.
- **CreatePictureList**. Reserves memory for the image data.
- **DisposePictureList**. Frees the memory allocated for image data.
- **Acquire_Pictures**. Acquires a series of images with a given interval. For each image the column sums of pixel values are calculated and stored in the memory set aside by CreatePictureList.
- **Velocity**. Calculates the real wind velocity from a given image displacement using information about the distance, angle, focal length, CCD size and camera calibration.
- **Angle_correction**. Transforms the image data from the camera to a plane parallel to the smoke plume.
- **Transform**. Used by Angle_correction to transform data.
- **Subtract_Mean**. Subtracts the mean value of the sample images from every image to reduce the static image background.
- **measure_wind_velocity**. This procedure calls the appropriate procedures to measure wind velocity using the selected method. It also presents the result of the measurement.

The main program initializes the frame grabber and then displays the main menu. Depending on the user selections calls are made to the different units.

Global unit

The global unit defines all the global variables shared between the units. It also defines one procedure:

- **Set_Globals**. This procedure sets the default values of some global variables. If the preferences file 'Windpref' can be found, the distance, angle, focal length, camera number and camera calibrations are read from this file. Otherwise they are set to default values.

User unit

This unit contains all user interface procedures:

- **Windspeed_Menu**. Displays the main menu.
- **Redefine_globals**. Displays the parameters menu. When the parameters have been changed the distance, angle, focal length, camera number and camera calibrations are written to the 'Windpref' file.

- **Set_debug_mode**. Displays the debug options menu.
- **Set_region_of_interest**. Allows the user to define the image area to be used for the measurement.
- **Camera_calibration**. Allows the user to calibrate the camera using the cursor controls.

Debug unit

Debugging procedures:

- **Save_smoke_profile**. Saves the sampled image data in a format that can be used as input to the program.
- **Save_smoke_text**. Saves the sampled image data in text format that can be used to export data to other programs.
- **Save_fourier** and **Save_fourier2**. Saves Fourier transform data as text.
- **Load_smoke_profile**. Reads the data saved by Save_smoke_profile.
- **Debug and Debugln**. Writes debug data to the screen or file if the options have been set in the debug options menu.
- **Plot_smoke_profile**. Plots the smoke profile and lets the user adjust the displacement.
- **Plot_fourier_data**. Plots Fourier transform input and output data.

Corr unit

This unit contains the procedures for the correlation velocity measurement method:

- **Corr_measure_wind_velocity**. The function returning the displacement between the images measured in pixels.
- **InitiateData** and **InitiateData2**. Clears the workspace needed by the calculation procedures.
- **Fourier. TwoFFT**, Performs fast Fourier transform of the input data. These procedures are adapted from Fortran procedures from 'Numerical Recipes'.
- **PeakSearch**. Locates the maximum value in a vector. This procedure is used to find the displacement for which the correlation has a maximum.
- **Correlate**. Uses the above procedures to perform cross correlation of the image data.
- **Median**. Calculates the median value of the image displacements. This is the result returned from Corr_measure_wind_velocity.

- **Directcorrelation**. This procedure calculates the cross correlation without using Fourier transforms. It is not used.
- **Fourcorrelation**. This procedure calculates the cross correlation using Fourier transforms. This is faster than the directcorrelation procedure and is the method used in the wind program.

Four unit

This unit contains the code for the experimental Fourier correlation method.

- **Fft_measure_wind_velocity**. The function returning the displacement between the images measured in pixels.
- **InitiateData**. Clears the workspace needed by the calculation procedures.
- **DisposeImageList**. Frees memory allocated by the calculation procedures.
- **Fourier**. Performs fast Fourier transform of the input data. This procedure is adapted from a Fortran procedure from 'Numerical Recipes'.
- **PeakSearch**. Locates the maximum value in a vector. This procedure is used to find the displacement for which the correlation has a maximum.
- **PrepareData1**, PrepareData2 and PrepareData3. Performs a special Fourier transform of the input image data. These are alternate experimental methods.
- **MakeVector**. Prepares the input data for the calculation procedures.

Program listings

On the following pages the program listings of the Wind program can be found.

Wind

```
{
This is the main program, providing initialization procedures, general calcualtions,
frame grabber access, and the main menu selections loop. It also provides the wind
velocity measurement procedure which calls the other modules.
}

{$R+}    {Range checking on             }
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on             }
{$I+}    {I/O checking on               }
{$N+}    {Numeric coprocessor           }

PROGRAM Wind;

USES
     Global_u,   { All program globals                                    }
     Dt_2,       { Interface routines for framegrabber                    }
     User_u,     { User interfaces, menues, set-ups                       }
     Debug_u,    { Writes debug information to screen or disk             }
     Four_u,     { The fourier method procedures                          }
     Corr_u,     { The correlation method procedures                      }
     Dos,        { TURBO-pascal Dos unit                                  }
     Input_u,    { General input procedures from existing LIDAR system }
     Crt;


{
------ The constants and variables defined in unit Global_u and dt_2 ----------

   CONST

     Buf0       frame buffer 0  from dt_2
     Buf1       frame buffer 1  from dt_2
     pixels     number of pixels across   from dt_2

     max_no_of_pictures      = 128;  number of pictures must be an integer
     power of two
     default_sample_interval = 50;   sec/100 Interval between samples
     default_no_of_pictures  = 32;
     default_CCD_size        = 7.40;
     default_focal_length    = 54;
     save_file_path          = 'C:\tp\plym\data\';
     preferences_file        = 'C:\tp\plym\prefs\windpref.prf';
     pixels2                 = 2*pixels;

TYPE

     Sum               =
        array[0 to pixels] of longint     from dt_2
     RoI               =
        RECORD  Region of interest selected in picture
           buffer    :longint;
           xpos,xsize:word;
           ypos,ysize:word;
        END;    from dt_2
```

```
     image_source_type = (disk , camera);
     methods           = (fourier, correlation);
     picture_pointer   = ^picture_frame;  The pixels of each column are added
     picture_frame     =  together, representing the picture
        RECORD  in an array with one element for each
           picture:Sum;    column
           next   :picture_pointer;
        END;

     transformfunction =
        array [1..pixels2] of real;
     fft_pointer       = ^fft_frame;
     fft_frame         =  The fourier transformed pictures
         RECORD
            data:transformfunction;
            next:fft_pointer;
         END;

     transformfunction2= array [1..2*pixels2] of real;


VAR
     region               :RoI; Region of interest in picture
     picture_list         :picture_pointer;
     select               :char;
     sample_interval      :word;     interval between pictures (sec/100)
     no_of_pictures       :word;
     focal_length         :real;
     CCD_size,CCD_xsize   :real;
     magnification        :real;
     distance             :real;
     angle                :real;
     image_source         :image_source_type;
     method               :methods;  correlation or fourier
     save_file_name       :string[40];
     ut                   :text;
     debug_mode           :set of (im, four, screen, dsk, result);
     debug_image          :integer;
     pixelvelocity        :integer;
     scale                :real;
     camera_cal           :array [1..5] of real;
     cal_no               :real;

-------------------------------------------------------------------------

Initialization and setup procedures
}

PROCEDURE initialize_frame_grabber;
{
Initializes the frame grabber card and selects appropriate input
and output lookup tables. It then starts passthru mode, displaying
the image on the video monitor.
The procedures can be found in Dt_2.
}

BEGIN
   dt_initialize;                {Set up frame grabber card.     }
   dt_select_ilut(0);            {Select specified input table.  }
   dt_select_olut(0);            {Select specified output table. }
```

```
   dt_select_input_frame(0);       {Read frames into and out from  }
   dt_select_output_frame(0);      {selected frame buffers.        }
   dt_set_sync_source(1);          {External sync is choosen.      }
   dt_display(1);                  {Display is turned on.          }
END;  {initialize_frame_grabber}
{
------------------------------------------------------------------------
}
PROCEDURE CreatePictureList(VAR  picture_list:picture_pointer;
                                 no_of_pictures:word);
{
Reserves memory for the image data in the form of a linked list with
"no_of_pictures" elements each containing an array of type "sum".
At least two images must be stored.
}

VAR
     current_picture:picture_pointer;
     x              :word;

BEGIN
   current_picture :=new(picture_pointer);
   picture_list    :=current_picture;

   FOR x:=2 TO no_of_pictures DO BEGIN
      current_picture^.next :=new(picture_pointer);
      current_picture       :=current_picture^.next;
   END;

   current_picture^.next:=nil;
END;  {create_picture_list}

{
------------------------------------------------------------------------
}

PROCEDURE DisposePictureList(VAR picture_list:picture_pointer);
{
Frees the memory allocated for image data by the linked list.
}

VAR
     next_picture:picture_pointer;

BEGIN
   WHILE picture_list<>nil DO BEGIN
      next_picture :=picture_list^.next;
      dispose(picture_list);
      picture_list :=next_picture;
   END;
END;  {Dispose_picture_list}

{
------------------------------------------------------------------------

Framegrabber interface procedures
}

PROCEDURE Acquire_Two_Pictures(region      :RoI;
                               interval    :word;
                               picture_list:picture_pointer);
```

```
{
Samples two images to the two frame grabber image buffers. The
procedures can be found in Dt_2.
}
BEGIN
   dt_select_olut(0);
   dt_select_ilut(0);
   dt_freeze_frame;
   dt_acquire_two(interval div 4);
   dt_sum_columns(region,picture_list^.picture);
   dt_sum_columns(region,picture_list^.next^.picture);
   dt_passthru;
END; {Acquire_Two_Pictures}

{
-----------------------------------------------------------------
}

PROCEDURE Acquire_Pictures(region      :RoI;
                           interval    :word;
                           picture_list:picture_pointer);
{
For every element in "picture_list" an image is acquired from the camera
with a time interval defined by "interval". The pixel values for each
column within the image region defined by "Region" are summed and stored in
the array "picture_list^.picture".
}
VAR
     clock               :longint;
     hour,min,sec,hundreds:word;

BEGIN
   dt_select_olut(0);              {Select appropriate lookup tables          }
   dt_select_ilut(0);

   GetTime(hour,min,sec,hundreds);

   REPEAT {for all (no_of_pictures) images}

      dt_acquire;                  {Sample a videoframe and store it in buffer}

      clock:=(min*6000+sec*100+hundreds+interval) mod 360000; {calculate end of
                                                               time interval   }
      dt_wait;

      dt_sum_columns(region,picture_list^.picture); {Add the pixels of each column}

      picture_list:=picture_list^.next;

      REPEAT  {until time interval is over}
         GetTime(hour,min,sec,hundreds);
      UNTIL (min*6000+sec*100+hundreds)>=clock;

   UNTIL picture_list=nil;

   dt_passthru;
END;  {acquire_pictures}
```

```
{
-------------------------------------------------------------------------------

General calculation procedures
}

FUNCTION Velocity(v,scale :real):real;
{
The velocity is translated to m/s from pixels/frame

v:               measured pixel velocity (pixels per image)
scale:           scale factor for angular correction (1)
magnification:   optical magnification of the camera (1)
pixel_size:      size of a pixel (mm)
sample_interval: time interval between pictures (sec/100)
}
VAR
   pixel_size:real;     { The physical size in the camera of a
                          framegrabber pixel (mm)                          }

BEGIN
   pixel_size:= CCD_xsize/pixels;
   Velocity  := v * (pixel_size/1000) * magnification/(sample_interval/100) *
                scale * camera_cal[Trunc(cal_no)];
END; { velocity }

{
-------------------------------------------------------------------------
}

PROCEDURE Angle_correction(VAR page  :picture_pointer;
                               region:RoI;
                           VAR scale :real);
{
Project smoke plume on a plane parallel to the camera image plane.

Move the coordinates so that zero is in the center of the image,
calculate the length of the projected plume and a scale factor so
that it fits in the old vector. Use geometry to find the corresponding
displacements and linear extrapolation to calculate the values for
discrete steps.
}
VAR
     result,last,step :picture_pointer;
     phi,lastphi      :real;    {angles at the end points of the region of interest}
     offset           :real;    {displacement from origo to end point of the projection}
     width            :real;    {region.xsize of the projection                  }
     meter_per_pixel  :real;

   PROCEDURE Transform(step, result :picture_pointer;
                       shift        :real);
   {
   Calculates the projection for a given displacement and uses linear
   extrapolation to calculate the values for discrete steps.
   }
   VAR
         ii,a,index :integer;
         xposition  :real;    {holds the displacement in pixels from origo to  }
                              {the image point,ii                              }
```

```
        x1,x2       :real;    {x1 is the displacement in pixels from origo to  }
                              {the image point,ii - 1, projected on a plane    }
                              {parallel to the camera.                         }
        phi         :real;    {the angle at the camera, between the point      }
                              {representing origo of the picture and the point }
                              {to be transformed                               }
  BEGIN
     x1    := 0.0;
     result^.picture[0] := step^.picture[0];

     FOR ii := 1 TO (region.xsize-1) DO BEGIN              {For all columns}
        xposition:=region.xpos-256.0+ii;
        phi := arctan(xposition*meter_per_pixel/distance);
        x2  := (xposition*cos(phi)/cos(phi+angle)-shift)/scale;

        IF (trunc(x2) - trunc(x1)) > 0 THEN BEGIN
        {the elements of the array must contain data from every column of}
        {pixels. If the displacement x1 to x2 is more than one pixel,     }
        {linear interpolation is used to approximate the data to store in}
        {the array.                                                       }

           FOR a := (trunc(x1) + 1) TO trunc(x2) DO BEGIN {discrete steps between
                                                          calculated displacements}
              result^.picture[a] :=
                 step^.picture[ii-1] +
                 round( (step^.picture[ii] - step^.picture[ii-1])/(x2-x1) *
                 (a-x1)  );
              {new value :                                }
              {   known value +                           }
              {   slope *                                 }
              {   displacement from known value, x1 <-> ii-1}
           END;{FOR}
        END;{IF}

        x1 := x2;

     END;{FOR}
  END;{Transform}



BEGIN
      meter_per_pixel := (magnification * CCD_xsize*1E-3)/pixels;
      phi     := arctan((region.xpos - 256.0) * meter_per_pixel/distance);
      offset  := (region.xpos-256.0) * cos(phi)/cos(phi+angle);
      lastphi := arctan((region.xpos + (region.xsize -1) - 256.0)
                 * meter_per_pixel/distance);
      width   := (region.xpos + (region.xsize - 1) - 256.0)
                 * cos(lastphi)/cos(lastphi + angle) - offset + 1;
      scale   := width/region.xsize;

      debug('Angle_Correction  phi=',phi);
      debug('width=',width);
      debugln('scale=',scale);

      last := nil;
      step := page;
      WHILE step <> nil DO      {for all pictures}
         BEGIN
            new(result);
            transform(step,result,offset);
```

```
            result^.next := step^.next;
            dispose(step);
            step := result^.next;
            IF last <> nil THEN
               BEGIN
                  last^.next := result;
                  last := result;
               END
            ELSE
               BEGIN
                  last := result;
                  page := result;      { Save a pointer to first element }
               END;
         END;
      step := nil;
      last := nil;
      result := nil;

   END;{ Angle_correction}

{
------------------------------------------------------------------
}

PROCEDURE Subtract_Mean(region      :RoI;
                        picture_list:picture_pointer);

{ Subtracts the mean value of the sample images from every image, }
{ to reduce the static image background.                          }


VAR
     pict   :picture_pointer;
     column :word;
     temp   :Sum;


BEGIN
   pict := picture_list;

   FOR column:= 0 to region.xsize-1 DO BEGIN
      temp[column] := 0;
   END; {FOR}

   WHILE pict <> nil DO
   {add all pictures and divide by number of pictures for each column}
      BEGIN
         FOR column:=0 TO region.xsize-1 DO BEGIN
            temp[column] := temp[column] + (pict^.picture[column] div
                            no_of_pictures);
         END; {FOR}
         pict := pict^.next;
      END;

   pict := picture_list;

   WHILE pict <> nil DO
   {subtract above calculated mean value from every picture and column}
      BEGIN
         FOR column:=0 TO region.xsize-1 DO BEGIN
            pict^.picture[column] := pict^.picture[column] - temp[column];
```

```
         END; {FOR}
         pict := pict^.next;
      END;
END;  {subtract_mean}


{
-------------------------------------------------------------------------------

Windspeed measurement procedures
}

PROCEDURE measure_wind_velocity(VAR region      :RoI;
                                VAR picture_list:picture_pointer);
{
This procedure calls the appropriate procedures to measure wind velocity
using the selected method. It also presents the result of the measurement.
}

VAR
     windspeed :real;
     comment   :string[255];      {used in debug mode}


BEGIN

   IF (screen in debug_mode) or (dsk in debug_mode) THEN BEGIN
      ClrScr;
      comment:=Input_string(1,1,'comment: ',9);

      debugln('comment',0);
      debugln(comment,0);
      debug('Distance: ',distance);
      debug('Angle: ',angle);
      debug('Focal: ',focal_length);
   END;

   IF image_source=camera THEN BEGIN

   {--set up the frame grabber - found in Dt_2-------------------------}
      dt_freeze_frame;
      dt_wait;
      dt_load_mask(0);         {erase potential red lines on the screen}

   {--acquire a series of images and store ----------------------------}
   {--in a practical format - found in this unit----------------------}
      Acquire_Pictures(region,sample_interval,picture_list);

   {--Reduce the static image background - found in this unit----------}
      Subtract_Mean(region,picture_list);

      IF im in debug_mode THEN BEGIN
         save_smoke_profile(region,picture_list);
      END; {IF}

   {--Transform the image data to a plane parallel--------------------}
   {--to the smoke plume - found in this unit-------------------------}
       Angle_correction(picture_list,region,scale);
   END

   ELSE BEGIN {image_source = disk}
```

```
   {--load image file - found in debug_u------------------------------}
      load_smoke_profile(region,picture_list);

   {--Transform the image data to a plane----------------------------}
   {--parallel to the smoke plume - found in this unit---------------}
         Angle_correction(picture_list,region,scale);
   END;{IF}

   debugln('Scale: ',scale);
   debug_image:=0;

   {--calculate the velocity - found in four_u and corr_u------------}
   CASE method OF
      fourier:     windspeed    :=fft_measure_wind_velocity(region,picture_list);
      correlation: pixelvelocity:=Round(corr_measure_wind_velocity(region,picture_list));
   END;{CASE}

   ClrScr;
   GotoXY(14,10);writeln('The windspeed is: ',velocity(pixelvelocity,scale):2:2,' m/s');
   GotoXY(14,12);Write('Press ''Enter'' to continue');
   readln;
END;{measure_wind_velocity}


{
--------------------------------------------------------------------------------

Main program
}

BEGIN
{--set up equipment and initialize data---------------------------------------}
   initialize_frame_grabber;
   dt_define_RoI(region,Buf0,0,0,pixels,pixels);  {Start with full screen region}
   set_globals;                                   {found in global_u            }
   CreatePictureList(picture_list,no_of_pictures);{found in this unit          }

   REPEAT
      Windspeed_Menu(select);                                    {found in user_u}
      CASE select OF
       F1  :set_region_of_interest(region,Buf0);                 {found in user_u}
       F2  :redefine_globals;                                    {found in user_u}
       F3  :measure_wind_velocity(region,picture_list);          {found in this u}
       F4  :plot_smoke_profile(region,picture_list,pixelvelocity);
       F5  :IF image_source=camera THEN
               image_source:=disk
            ELSE
               image_source:=camera;
       F6  :IF method=fourier THEN
               method:=correlation
            ELSE
               method:=fourier;
       F7  :set_debug_mode;
       F8  :save_smoke_text(region,picture_list);
       F9  :camera_calibration(region,Buf0);                     {found in user_u}

      END;{CASE}
   UNTIL select = F10;
   DisposePictureList(picture_list);
END.{Wind}
```

Global unit

```
{
This unit contains all global variables and a procedure to set them up to their
default values.
}

{$R+}    {Range checking off            }
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on             }
{$I+}    {I/O checking on               }
{$N+}    {Numeric coprocessor           }

UNIT Global_u;

INTERFACE

   USES Dt_2;

   CONST
   {
     Buf0       frame buffer 0  from dt_2
     Buf1       frame buffer 1  from dt_2
     pixels     number of pixels across   from dt_2
   }
     max_no_of_pictures      = 128;  {number of pictures must be an integer
      power of two                          }
     default_sample_interval = 50;   {sec/100 Interval between samples     }
     default_no_of_pictures  = 32;
     default_CCD_size        = 7.40;
     default_focal_length    = 54;
     save_file_path          = 'C:\tp\plym\data\';
     preferences_file        = 'C:\tp\plym\prefs\windpref.prf';
     pixels2                 = 2*pixels;

TYPE
     {
     Sum               =
        array[0 to pixels] of longint     from dt_2
     RoI               =
        RECORD  {Region of interest selected in picture}
           buffer    :longint;
           xpos,xsize:word;
           ypos,ysize:word;
        END;    from dt_2
     }

     image_source_type = (disk , camera);
     methods           = (fourier, correlation);
     picture_pointer   = ^picture_frame;  {The pixels of each column are added   }
     picture_frame     =  {together, representing the picture    }
        RECORD  {in an array with one element for each }
           picture:Sum;    {column                                }
           next   :picture_pointer;
        END;

     transformfunction =
        array [1..pixels2] of real;
     fft_pointer       = ^fft_frame;
```

```
     fft_frame          =  {The fourier transformed pictures      }
         RECORD
            data:transformfunction;
            next:fft_pointer;
         END;

     transformfunction2= array [1..2*pixels2] of real;


VAR
     region               :RoI; {Region of interest in picture         }
     picture_list         :picture_pointer;
     select               :char;
     sample_interval      :word;     {interval between pictures (sec/100)    }
     no_of_pictures       :word;
     focal_length         :real;
     CCD_size,CCD_xsize   :real;
     magnification        :real;
     distance             :real;
     angle                :real;
     image_source         :image_source_type;
     method               :methods;  {correlation or fourier               }
     save_file_name       :string[40];
     ut                   :text;
     debug_mode           :set of (im, four, screen, dsk, result);
     debug_image          :integer;
     pixelvelocity        :integer;
     scale                :real;
     camera_cal           :array [1..5] of real;
     cal_no               :real;
{
-------------------------------------------------------------------------------------
}

PROCEDURE set_globals;
{
This procedure sets the default values of some global variables. If the preferences
file 'Windpref' can be found, the distance, angle, focal length, camera number and
camera clibrations are read from this file, otherwise they are set to default values.
}

IMPLEMENTATION{
-------------------------------------------------------------------------------------
}

PROCEDURE set_globals;
{
Set initial values of global variables
}

VAR indata:file of real;
    i     :integer;

BEGIN
   sample_interval := default_sample_interval; { sec/100  Interval between pictures}
   no_of_pictures  := default_no_of_pictures;
   CCD_size        := default_CCD_size;
   CCD_xsize       := sqrt(9/13) * CCD_size;
   pixelvelocity   := 0;
   method          := correlation;
```

```
   Assign(indata,preferences_file);
   {$I-}  {No fault should occur if no file is
      available, so IO-check is turned off  }
   Reset(indata);
   {$I+}
   IF IOresult=0 THEN     {File available                          }
      BEGIN
         Read(indata,focal_length,distance,angle,cal_no);
         FOR i:=1 TO 5 DO
            Read(indata,camera_cal[i]);
         close(indata);
      END
   ELSE   {File not available                     }
      BEGIN
         focal_length := default_focal_length;
         distance     := 0;
         angle        := 0;
         cal_no       := 1.0;

         FOR i:=1 TO 5 DO
            camera_cal[i]:=1;
      END;
   magnification :=distance/(focal_length*1E-3);
   image_source  := camera;
   debug_mode    := [];
END;  {set_globals}

END.
```

User unit

```
{
This unit contains all user interface procedures.
}

{$R+}    {Range checking off}
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on}
{$I+}    {I/O checking on}
{$N+}    {Numeric coprocessor}

UNIT User_u;

INTERFACE

    USES
     global_u,  {All program globals                }
     dt_2,
     Crt,
     INPUT_U;

PROCEDURE Windspeed_Menu( VAR func_key: CHAR);
{
Displays the main menu.
}
```

```
PROCEDURE redefine_globals;
{
Displays the parameters menu. When the parameters have been changed,
the distance, angle, focal length, camera number and camera calibrations
are written into the 'Windpref' file.
}

PROCEDURE set_debug_mode;
{
Displays the debug options menu.
}

PROCEDURE place_cursor(xmin,ymin:word;power_of_two :boolean;VAR x,y:word);
{
Allows the user to place the cursor in order to define the image area to
be used. If a fourier method is used that requires the width of the area
to be an integer power of two, this is accounted for.
}

PROCEDURE set_region_of_interest(VAR region:RoI;buffer:longint);
{
Allows the user to define the image area to be used for measurement.
}

PROCEDURE camera_calibration(VAR region:RoI;buffer:longint);
{
Allows the user to calibrate the camera using the cursor controls.
}


IMPLEMENTATION

PROCEDURE Windspeed_Menu( VAR func_key: CHAR);
{
Displays the main menu.
}
CONST Xpos = 14;
      Ypos = 3;

BEGIN
     ClrScr;
     GotoXY(Xpos,Ypos);   Write('*****************************************');
     GotoXY(Xpos,Ypos+1); Write('*****    WINDSPEED MEASUREMENT MENU  *****');
     GotoXY(Xpos,Ypos+2); Write('*****************************************');
     LowVideo;
     GotoXY(Xpos,Ypos+4);    Write('[F1]:  Set region of interest');
     GotoXY(Xpos,Ypos+5);    Write('[F2]:  Set parameters');
     GotoXY(Xpos,Ypos+6);    Write('[F3]:  Measure wind velocity');
     GotoXY(Xpos,Ypos+7);    Write('[F4]:  Plot smoke profile');
     GotoXY(Xpos,Ypos+9);    Write('[F5]:  Input source:        ');
     IF image_source=camera THEN
        write('camera')
     ELSE
        write('disk  ');
     GotoXY(Xpos,Ypos+10);   Write('[F6]:  Method:              ');
     IF method=fourier THEN
        write('fourier    ')
     ELSE
        write('correlation');
```

```
     GotoXY(Xpos,Ypos+12);   Write('[F7]:  Set debug options');
     GotoXY(Xpos,Ypos+13);   Write('[F8]:  Save smoke profile as text');
     GotoXY(Xpos,Ypos+14);   Write('[F9]:  Calibrate camera');

     GotoXY(Xpos-1,Ypos+16); Write('[F10]:  End program.');
     NormVideo;
     func_key:= Input_Function_Key(Xpos,Ypos+18,
                                   'Enter function key, please...');
END; {Windspeed menu}

{
--------------------------------------------------------------------------------
}

PROCEDURE redefine_globals;
{
Displays the parameters menu. When the parameters have been changed,
the distance, angle, focal length, camera number and camera calibrations
are written into the 'Windpref' file.
}
CONST
     Xpos = 14;
     Ypos = 3;

VAR
     func_key:char;
     utdata:file of real;

BEGIN
  REPEAT
    ClrScr;
    GotoXY(Xpos,Ypos);   Write('*****************************************');
    GotoXY(Xpos,Ypos+1); Write('*****    Wind Velocity Parameters    *****');
    GotoXY(Xpos,Ypos+2); Write('*****************************************');
    LowVideo;
    GotoXY(Xpos,Ypos+4);
    IF method=fourier THEN
       write('Fourier method')
    ELSE
      write('Correlation method');
    GotoXY(Xpos+40,Ypos+4);  Write('Current value:');
    GotoXY(Xpos,Ypos+6);   Write('[F1]: Reset to default');
    GotoXY(Xpos,Ypos+7);   Write('[F2]: Number of samples ('
                                  ,default_no_of_pictures,')');
    GotoXY(Xpos+40,Ypos+7);  Write(no_of_pictures:4,' samples');
    GotoXY(Xpos,Ypos+8);   Write('[F3]: Interval between samples ('
                                   ,default_sample_interval,')');
    GotoXY(Xpos+40,Ypos+8);  Write(sample_interval:4,' sec/100');
    GotoXY(Xpos,Ypos+11);  Write('[F5]: Select camera no.');
    GotoXY(Xpos+40,Ypos+11); Write(cal_no:4:0,'      '
                                  ,camera_cal[Trunc(cal_no)]:3:2);
    GotoXY(Xpos,Ypos+12);  Write('[F6]: Camera focal length ('
                                  ,default_focal_length,')');
    GotoXY(Xpos+40,Ypos+12); Write(focal_length:4:0,' mm');
    GotoXY(Xpos,Ypos+13);  Write('[F7]: CCD size (',default_CCD_size:4:1,')');
    GotoXY(Xpos+40,Ypos+13); Write(CCD_size:4:1,' mm');
    GotoXY(Xpos,Ypos+14);  Write('[F8]: distance (0) m');
    GotoXY(Xpos+40,Ypos+14); Write(distance:4:1,' m');
    GotoXY(Xpos,Ypos+15);  Write('[F9]: angle (0)');
```

```
    GotoXY(Xpos+40,Ypos+15); Write(angle*180/pi:4:1);
    GotoXY(Xpos-1,Ypos+17);Write('[F10]: Leave this menu.');

    func_key:=Input_Function_Key(Xpos,Ypos+19,'Enter function key, please...');
    CASE func_key OF
      F1  :set_globals;
      F2  :no_of_pictures:=
              Input_Integer(Xpos,Ypos+20,'Enter new number of samples',4,128);
      F3  :sample_interval:=
              Input_Integer(Xpos,Ypos+20,'Enter new interval in sec/100',1,6000);
      F5  :cal_no:=
              Input_Real(cal_no,Xpos,Ypos+20,
                        'Enter new calibration no to be used',1,5);
      F6  :BEGIN
              focal_length:=
                 Input_real(focal_length,Xpos,Ypos+20,
                           'Enter new focal length',1,100);
              magnification:=distance/(focal_length*1E-3);
           END;
      F7  :BEGIN
              CCD_size:=
                 Input_real(CCD_size,Xpos,Ypos+20,'Enter new CCD-size',0,25.4);
              CCD_xsize := sqrt(9/13) * CCD_size;
           END;
      F8  :BEGIN
              distance:=
                 Input_real(distance,Xpos,Ypos+20,'Enter distance',0,3000);
              magnification:=distance/(focal_length*1E-3);
           END;
      F9  :BEGIN
              angle:=
                 Input_real(angle,Xpos,Ypos+20,'Enter angle',-90,90);
              angle:=angle*pi/180;
           END;
    END;
  UNTIL func_key=F10;

  Assign(utdata,preferences_file);
  {$I-}  {No fault should occur if no file is
     available, so IO-check is turned off }
  Rewrite (utdata);
  {$I+}

  IF IOresult=0 THEN BEGIN     {File was available                    }
    write(utdata,focal_length,distance,angle,cal_no);
    write(utdata,camera_cal[1],camera_cal[2],camera_cal[3],camera_cal[4],camera_cal[5]);
        close(utdata);
  END;
END;{redefine_globals}

{
------------------------------------------------------------------------------
}


PROCEDURE set_debug_mode;
{
Displays the debug options menu.
}
```

```
CONST
     Xpos = 14;
     Ypos = 3;

VAR
     func_key:char;
     utdata:file of real;

BEGIN
  REPEAT
    ClrScr;
    GotoXY(Xpos,Ypos);   Write('*****************************************');
    GotoXY(Xpos,Ypos+1); Write('*****          Debug options         *****');
    GotoXY(Xpos,Ypos+2); Write('*****************************************');
    LowVideo;
    GotoXY(Xpos,Ypos+4);
    GotoXY(Xpos+40,Ypos+4); Write('Current value:');
    GotoXY(Xpos,Ypos+6); Write('[F1]:  Save images');
    IF im in debug_mode THEN
    BEGIN
         GotoXY(Xpos+40,Ypos+6);Write('On ');
      END
    ELSE
      BEGIN
         GotoXY(Xpos+40,Ypos+6);Write('Off');
    END;

    GotoXY(Xpos,Ypos+8); Write('[F3]:  Save fourier data');
    IF four in debug_mode THEN
    BEGIN
         GotoXY(Xpos+40,Ypos+8);Write('On ');
      END
    ELSE
      BEGIN
         GotoXY(Xpos+40,Ypos+8);Write('Off');
    END;

    GotoXY(Xpos,Ypos+9); Write('[F4]:  Screen output');
    IF screen in debug_mode THEN
      BEGIN
         GotoXY(Xpos+40,Ypos+9);Write('On ');
      END
    ELSE
      BEGIN
         GotoXY(Xpos+40,Ypos+9);Write('Off');
      END;

    GotoXY(Xpos,Ypos+10); Write('[F5]:  Save file');
    IF dsk in debug_mode THEN
      BEGIN
         GotoXY(Xpos+40,Ypos+10);Write(save_file_name);
      END
    ELSE
      BEGIN
         GotoXY(Xpos+40,Ypos+10);Write('Off');
      END;

    GotoXY(Xpos-1,Ypos+17); Write('[F10]:  Leave this menu.');
    func_key:=Input_Function_Key(Xpos,Ypos+19,'Enter function key, please...');
    CASE func_key OF
```

```
      F1  :IF im in debug_mode THEN
              debug_mode:=debug_mode-[im]
          ELSE
              debug_mode:=debug_mode+[im];
      F3  :IF four in debug_mode THEN
              debug_mode:=debug_mode-[four]
          ELSE
              debug_mode:=debug_mode+[four];
      F4  :IF screen in debug_mode THEN
              debug_mode:=debug_mode-[screen]
          ELSE
              debug_mode:=debug_mode+[screen];
      F5  :IF dsk in debug_mode THEN
              BEGIN
                 debug_mode:=debug_mode-[dsk];
                 close(ut);
              END
          ELSE
              BEGIN
                 debug_mode:=debug_mode+[dsk];
                 save_file_name:=
                    Input_string(Xpos,Ypos+20,'Enter name for save file: ', 9);

                 Assign(ut,save_file_path+save_file_name+'.txt');
                 Rewrite(ut);
              END;
    END;
  UNTIL func_key=F10;
  Assign(utdata,preferences_file);
  {$I-}    {No fault should occur if no file is
     available, so IO-check is turned off  }
  Rewrite(utdata);
  {$I+}
  IF IOresult=0 THEN
     BEGIN
        write(utdata,focal_length,distance,angle);
        close(utdata);
     END;
END;{Set_debug_mode}

{
------------------------------------------------------------------------------
}

PROCEDURE place_cursor(xmin,ymin:word;power_of_two :boolean;VAR x,y:word);
{
Allows the user to place the cursor in order to define the image area to
be used. If a fourier method is used that requires the width of the area
to be an integer power of two, this is accounted for.
}
VAR
     step,count,t,i,imax:word;
     ch,prev_ch    :byte;

BEGIN
   IF power_of_two THEN
      BEGIN
         i:=4;
         WHILE i+xmin <= 512 DO
```

```
            i:=i*2;
         i:=i div 2;
         x:=i+xmin;
         imax:=i;
      END;
   dt_set_cursor_position(x,y);
   dt_cursor(1);
   ch:=0;
   prev_ch:=1;
   while ch<>13 DO
      BEGIN
         t:=0;

         WHILE (not KeyPressed) and (t<5000) DO
            t:=t+1;
         IF t=5000 THEN
            BEGIN
               t:=0;
               count:=0;
               step:=2;
            END;
         ch:=ord(ReadKey);
         IF ch=0 THEN
            BEGIN
               ch:=ord(ReadKey);
               IF ch<>prev_ch THEN
                  BEGIN
                     step:=2;
                     count:=0;
                  END;
               prev_ch:=ch;
               case ch of
                71 :BEGIN
                       x:=0;
                       y:=0;
                    END;
                72 :IF y>ymin+step THEN
                       y:=y-step
                    else
                       y:=ymin;
                75 :IF power_of_two THEN
                       BEGIN
                          IF i>=4 THEN
                             begin
                                i:=i div 2;
                                x:=xmin+i;
                             end;
                       END
                   ELSE
                       IF x>xmin+step THEN
                          x:=x-step
                       else
                          x:=xmin;
                77 :IF power_of_two THEN
                       BEGIN
                          IF i<imax THEN
                             BEGIN
                                i:=i*2;
                                x:=xmin+i;
```

```
                        END;
                    END
                ELSE
                    x:=x+step;
             79 :BEGIN
                    IF power_of_two THEN
                       x:=imax+xmin
                    ELSE
                       x:=512;
                    y:=512;
                END;
             80 :y:=y+step;
            END;
            IF x>512 THEN
               x:=512;
            IF y>512 THEN
               y:=512;
            dt_set_cursor_position(x,y);
            count:=count+1;
            CASE count OF
                10 :step:=4;
                15 :step:=6;
                20 :step:=8;
                25 :step:=10;
                30 :step:=12;
                35 :step:=14;
            END;
        END;
    END;
  dt_cursor(0);
END;  {place_cursor}

{
-------------------------------------------------------------------------------
}

PROCEDURE set_region_of_interest(VAR region:RoI;buffer:longint);
{
Allows the user to define the image area to be used for measurement.
}
VAR
    x1,x2,y1,y2:word;

BEGIN
  ClrScr;
  dt_freeze_frame;  {Stops the passthru mode              }
  dt_load_mask(0);  {Removes the write protection         }
  dt_passthru; {Overwrite red remainders from
     previous measurements                 }
  dt_freeze_frame;
  dt_select_ilut(5);     {Tables that will display the colour
     red.                                  }
  dt_select_olut(7);     {Red is used to show the selected
     region.                               }
  dt_passthru;
     x1:=0;
     y1:=0;
     write('Use the cursor-keys, (',chr(24),chr(25),chr(26),chr(27),'), ');
     writeln('to place cursor at the upper left');
```

```
      writeln('corner of the interesting area and press ''Enter''');
      place_cursor(0,0,false,x1,y1);
      dt_freeze_frame;
      dt_load_mask(0);
      dt_draw_line(buffer,x1,y1,511,y1,1);     {Draw a horizontal red line across the
      screen                                  }

      dt_draw_line(buffer,x1,y1,x1,511,1);     {Draw a vertical red line across the
      screen                                  }
      dt_load_mask(1);    {Write protect the colour red         }
      dt_passthru;
      writeln;
      x2:=511;
      y2:=511;
      write('Use the cursor-keys, (',chr(24),chr(25),chr(26),chr(27),'), ');
      writeln('to place cursor at the lower right');
      writeln('corner of the interesting area and press ''Enter''');
      IF method=fourier THEN
         place_cursor(x1+4,y1+4,true,x2,y2)    {width of selected are must be an
      integer power of two                    }
      ELSE
         place_cursor(x1+4,y1+4,false,x2,y2);
      dt_freeze_frame;
      dt_load_mask(0);    {Remove write protection of colour red}
      dt_draw_line(buffer,x1,y2,x2,y2,1); {Draw a vertical and a horizontal line
      to complete the chosen square           }
      dt_draw_line(buffer,x2,y1,x2,y2,1);
      dt_draw_line(buffer,x1,y2+1,x1,511,0);   {Remove the red lines that reaches out
      from the square to the edge of the
      screen                                  }
      dt_draw_line(buffer,x2+1,y1,511,y1,0);
      dt_load_mask(1);    {Write protect the colour red         }
      dt_passthru;
      dt_define_roi(region,buffer,x1,y1,x2-x1+1,y2-y1+1);
END;  {set_region_of_interest}

{
--------------------------------------------------------------------------------
}

PROCEDURE camera_calibration(VAR region:RoI;buffer:longint);
{
Allows the user to calibrate the camera using the cursor controls.
}
VAR
     x1,x2,y1,y2:word;
     realdist:real;
     utdata:file of real;

BEGIN
   ClrScr;
   dt_freeze_frame;  {Stops the passthru mode              }
   dt_load_mask(0);  {Removes the write protection         }
   dt_passthru; {Overwrite red remainders from
      previous measurements                }
   dt_freeze_frame;
   dt_select_ilut(5);     {Tables that will dispaly the colour
      red.                                 }
   dt_select_olut(7);     {Red is used to show the selected
      region.                              }
```

```
   dt_passthru;
      x1:=0;
      y1:=0;
      writeln('Calibration of camera no: ',cal_no:1:0,
              '    Current factor: ',camera_cal[Trunc(cal_no)]:3:2);
      writeln;
      writeln('Distance = ',distance:4:1,' m, Focal lenght = '
                                          ,focal_length:3:0,' mm');
      writeln;
      write(  'Use the cursor-keys, (',chr(24),chr(25),chr(26),chr(27),'), ');
      writeln('to place cursor at the left');
      writeln('side of an object of known size and press ''Enter''');

      place_cursor(0,0,false,x1,y1);
      dt_freeze_frame;
      dt_load_mask(0);     {Remove write protection of colour red}
      dt_draw_line(buffer,x1,y1,x1,511,1);     {Draw vertical line through x1        }
      dt_load_mask(1);     {Write protect the colour red         }
      dt_passthru;

      writeln;
      x2:=511;
      y2:=511;
      write(  'Use the cursor-keys, (',chr(24),chr(25),chr(26),chr(27),'), ');
      writeln('to place cursor at the right');
      writeln('side of the object and press ''Enter''');

      place_cursor(x1+4,y1+4,false,x2,y2);
      dt_freeze_frame;
      dt_load_mask(0);     {Remove write protection of colour red}
      dt_draw_line(buffer,x2,y1,x2,y2,1); {Draw vertical line through x2        }
      dt_load_mask(1);     {Write protect the colour red         }
      dt_passthru;

      writeln;
      write('Under the given circumstances the object should be ');
      writeln((magnification*(x2-x1)*CCD_xsize/pixels)/1000:4:1, ' meters');
      write('Enter the real size (0 to leave unchanged) ');
      readln(realdist);

      IF realdist>0.5 THEN
         camera_cal[Trunc(cal_no)]:=
             realdist/((magnification*(x2-x1)*CCD_xsize/pixels)/1000);

      Assign(utdata,preferences_file);
      {$I-}     {No fault should occur if no file is
      available, so IO-check is turned off }
      Rewrite(utdata);
      {$I+}
      IF IOresult=0 THEN
      BEGIN
         write(utdata,focal_length,distance,angle,cal_no);
         write(utdata,camera_cal[1],camera_cal[2],camera_cal[3],
                      camera_cal[4],camera_cal[5]);
         close(utdata);
      END;
   END;  {camera_calibration}

END.
```

Debug unit

```
{$R+}    {Range checking off}
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on}
{$I+}    {I/O checking on}
{$N+}    {Numeric coprocessor}

UNIT Debug_u;

INTERFACE

    USES
     global_u,        { All program globals }
     dt_2,
     graph,
     crt,
     INPUT_U;

PROCEDURE save_smoke_profile(region:RoI;picture_list:picture_pointer);

PROCEDURE save_smoke_text(region:RoI;picture_list:picture_pointer);

PROCEDURE save_fourier(data:transformfunction;nn:integer;question:string);

PROCEDURE save_fourier2(data:transformfunction2;nn:integer;question:string);

PROCEDURE load_smoke_profile(var region:RoI; picture_list:picture_pointer);

PROCEDURE debug(text:string;data:real);

PROCEDURE debugln(text:string;data:real);

PROCEDURE plot_smoke_profile(region:RoI;picture_list:picture_pointer;var
pixelvelocity:integer);

PROCEDURE Plot_fourier_data(data:transformfunction;mode:integer);


IMPLEMENTATION

PROCEDURE save_smoke_profile(region:RoI;picture_list:picture_pointer);

VAR
     utdata               :file of longint;
     image,x              :integer;
     xpos,xsize,number    :longint;
     save_image_name      :string[12];

BEGIN
   ClrScr;
   xpos:=region.xpos;
   xsize:=region.xsize;
   number:=no_of_pictures;
   save_image_name:=Input_string(1,1,'Ange filnamn för utdata',9);
   Assign(utdata,save_file_path+save_image_name+'.im');
   Rewrite(utdata);
   Write(utdata,xpos,xsize,number);
   FOR image:=1 to no_of_pictures do
      BEGIN
```

```
         FOR x:=0 TO region.xsize-1 DO
            Write(utdata,picture_list^.picture[x]);
         picture_list:=picture_list^.next;
      END;
   close(utdata);
END;

PROCEDURE save_smoke_text(region:RoI;picture_list:picture_pointer);

VAR
     utext                :text;
     image                :picture_pointer;
     x              :integer;
     xpos,xsize,number    :longint;
     save_image_name      :string[12];

BEGIN
   ClrScr;
   xpos:=region.xpos;
   xsize:=region.xsize;
   number:=no_of_pictures;
   save_image_name:=Input_string(1,1,'Ange filnamn för utdata: ',9);
   Assign(utext,save_file_path+save_image_name+'.txt');
   Rewrite(utext);
   Writeln(utext,xpos,' ',xsize,' ',number);
   FOR x:=0 TO region.xsize-1 DO
      BEGIN
         image:=picture_list;
         WHILE image<>nil DO
            BEGIN
               write(utext,image^.picture[x],', ');
               image:=image^.next;
            END;
         writeln(utext);
      END;
   close(utext);
END;

PROCEDURE save_fourier(data:transformfunction;nn:integer;question:string);

   VAR
       utext               :text;
       x                   :integer;
      save_file_name       :strng;

   BEGIN
      ClrScr;
      writeln(question);
      save_file_name:=Input_string(1,2,'Ange filnamn för fourierdata: ',9);
      IF save_file_name <>' ' THEN
         BEGIN
            Assign(utext,save_file_path+save_file_name+'.fou');
            Rewrite(utext);
            FOR x:=1 TO nn div 2 DO
               writeln(utext,data[x*2-1],', ',data[x*2]);
           close(utext);
         END;
   END;
```

```
PROCEDURE save_fourier2(data:transformfunction2;nn:integer;question:string);

   VAR
       utext                :text;
       x                    :integer;
      save_file_name        :strng;

   BEGIN
      ClrScr;
      writeln(question);
      save_file_name:=Input_string(1,2,'Ange filnamn för fourierdata: ',9);
      IF save_file_name <>' ' THEN
         BEGIN
            Assign(utext,save_file_path+save_file_name+'.fo2');
            Rewrite(utext);
            FOR x:=1 TO nn div 2 DO
               writeln(utext,sqrt(sqr(data[x*2-1])+sqr(data[x*2])));
           close(utext);
         END;
   END;


PROCEDURE load_smoke_profile(var region:RoI; picture_list:picture_pointer);

VAR
     indata               :file of longint;
     save_image_name      :string[255];
     image,x              :integer;
     xpos,xsize,number    :longint;

BEGIN
   ClrScr;
   Writeln('Ange filnamn för indata');
   save_image_name:=Input_string(1,1,'Ange filnamn: ',9);
   Assign(indata,save_file_path+save_image_name+'.im');
   Reset(indata);
   Read(indata,xpos,xsize,number);
   dt_define_RoI(region,Buf0,xpos,0,xsize,512);
   IF number>no_of_pictures THEN
      number:=no_of_pictures;
   FOR image := 1 to number do
      BEGIN
      FOR x:=0 TO region.xsize-1 DO
            Read(indata,picture_list^.picture[x]);
      picture_list := picture_list^.next;
      END;
   close(indata);
END;

PROCEDURE debug(text:string;data:real);

BEGIN
   IF screen in debug_mode THEN
      write(text,'  ',data,'  ');
   IF dsk in debug_mode THEN
      write(ut,text,'  ',data:6:2,'  ');
END;

PROCEDURE debugln(text:string;data:real);
```

```
BEGIN
   IF screen in debug_mode THEN
      writeln(text,'  ',data,'  ');
   IF dsk in debug_mode THEN
      writeln(ut,text,'  ',data:6:2,'  ');
END;

FUNCTION Velocity(v,scale :real):real;
{ The velocity is translated to m/s from pixels/frame }

VAR
   pixel_size:real;     { The physical size in the camera of a
                          framegrabber pixel (mm)}

BEGIN
   pixel_size:=CCD_xsize/pixels;
   Velocity := v*scale*magnification*(pixel_size/1000)/(sample_interval/100);
   { v:               measured pixel velocity (pixels per image)
     scale:           scale factor for angular correction (1)
     magnification:   optical magnification of the camera (1)
     pixel_size:      size of a pixel (mm)
     sample_interval: time interval between pictures (sec/100) }
END; { velocity }


PROCEDURE Draw_Screen_Boxes;

VAR Out_text:string;

BEGIN
   SetBKColor(blue);
   SetColor(LightBlue);
   Rectangle(0,0,639,46);
   Rectangle(480,2,637,44);
   Rectangle(0,48,639,452);
   SetColor(White);
   OutTextXY(485,19,'Windspeed:');
   Str(Velocity(pixelvelocity,scale):6:1,Out_text);
   OutTextXY(555,19,Out_text);
   OutTextXY(610,19,'m/s');
END;

PROCEDURE plot_smoke_profile(    region       :RoI;
                                 picture_list :picture_pointer;
                           var pixelvelocity:integer);

VAR
     gd,gm          :integer;
     y,x,start      :integer;
     max,y_scale    :integer;
     Stop_pict      :integer;
     Start_pict     :integer;
     current_picture:picture_pointer;
     x_scale        :real;
     ch             :char;
     Out_text       :string;
     exit           :boolean;
```

```
BEGIN
   gd:=Detect;
   InitGraph(gd,gm,'c:\bp\bgi');
   stop_pict:=no_of_pictures;
   start_pict:=1;
   if stop_pict > 12 then
      stop_pict:=12;
   max:=0;
   current_picture:=picture_list;
   WHILE current_picture <> nil DO
   BEGIN
      FOR x := 0 to region.xsize-1 DO
         IF abs(picture_list^.picture[x])>max THEN
            max := abs(picture_list^.picture[x]);
      current_picture := current_picture^.next;
   END;
   y_scale:=max div 20;
   setBkcolor(blue);
   ClearDevice;
   Draw_Screen_Boxes;
   SetFillStyle(SolidFill,blue);
   REPEAT
      SetColor(White);
      Str(Velocity(pixelvelocity,scale):6:1,Out_text);
      Bar(555,19,609,26);
      OutTextXY(555,19,Out_text);
      Current_picture := picture_list;
      Bar(4,50,634,450);
      setcolor(4);
      FOR x:=0 to 63 DO
      BEGIN
         MoveTo(x*10+4,50);
         LineTo(x*10+4,450);
      END;
      MoveTo(4,50);
      LineTo(634,50);
      MoveTo(4,450);
      LineTo(634,450);
      x_scale := 630.0/((stop_pict-1)*pixelvelocity+region.xsize);
      IF x_scale>1 THEN
         x_scale := 1;
      IF pixelvelocity < 0 THEN
         start:=0
      ELSE
         start:=631-Round(region.xsize*x_scale);
      y:=1;
      WHILE Start_pict>y DO
      BEGIN
         y:=y+1;
         Current_picture:=current_picture^.next;
      END;
      FOR y:=1 TO stop_pict DO
      BEGIN
         IF y mod 2 = 0 THEN
            setcolor(15)
         ELSE
            setcolor(7);
         moveto(start+4,((current_picture^.picture[0] div y_scale))+y*30+55);
         FOR x:=0 TO region.xsize-1 DO
```

```
            BEGIN
               LineTo(Round(x*x_scale)+start+4,((current_picture^.picture[x] div
y_scale))+y*30+55);
               LineTo(Round(x*x_scale)+start+4,(y*30-(current_picture^.picture[x] div
y_scale)+55));
            END;
         start:=start-Round(pixelvelocity*x_scale);
         current_picture:=current_picture^.next;
      END;

      exit:=false;
      REPEAT
         ch := ReadKey;
         IF ch = #0 THEN
         BEGIN
            ch := ReadKey;
            exit:=true;
            CASE ch OF
            #77 : pixelvelocity:=pixelvelocity-1;
            #75 : pixelvelocity:=pixelvelocity+1;
            #72 : IF start_pict>1 THEN
                     Start_pict:=Start_pict-1
                  ELSE
                     exit:=false;
            #80 : IF start_pict+stop_pict<no_of_pictures THEN
                     start_pict:=start_pict+1
                  ELSE
                     exit:=false;
            #71 : start_pict:=1;
            #79 : start_pict:=no_of_pictures-stop_pict;
            #68 :;
            ELSE
            exit:=false;
            END;
         END;
      UNTIL exit;
   UNTIL ch = #68;
   Closegraph;
END;

PROCEDURE Plot_fourier_data(data:transformfunction;mode:integer);

   VAR
      number :real;  { magnitude of fourietransform at frequency u }
      u,i,x,y:integer;
      maximum:real; { greatest number so far }
      temp   :real;
      gm,gd,pl:integer;

   BEGIN
    DetectGraph(gd,gm);
    InitGraph(gd,gm,'c:\bp\bgi');
      x:=4;
      y:=250;
      maximum := 0;
      for u:=0 to (no_of_pictures-1) do
      BEGIN
         i:=2*u+1;
         IF mode=1 THEN
```

```
            temp := Round(sqrt(sqr(data[i])+sqr(data[i+1])))
         ELSE
            temp := abs(Round(data[i]));
         if temp > maximum THEN
            maximum := temp;
      END;
      FOR u:=0 to (no_of_pictures-1) do
         BEGIN
            Moveto(x,y);
            i:=2*u+1;
            IF mode=1 THEN
               temp := Round(sqrt(sqr(data[i])+sqr(data[i+1])))
            ELSE
               temp := Round(data[i]);
            pl:=Round(temp*(200/maximum));
            x:=i*4;
            y:=250-pl;
            LineTo(x,y);
            Moveto(x,250);
            IF (u mod 5) = 0 THEN
               LineTo(x,270)
            ELSE
               Lineto(x,260);
         END;
      readln;
      closegraph;
   END;

   PROCEDURE TestArray(testlist:picture_pointer);

   VAR
      t,a:integer;

   BEGIN
      writeln('Testarray');
      write('   Försjutning mellan bilder (- för vind åt vänster): ');
      readln(a);
      FOR t:= 1 to no_of_pictures DO
         BEGIN
            IF a>0 THEN
               testlist^.picture[abs(t*a mod pixels+1)]:=10
            ELSE
               testlist^.picture[512-abs(t*a mod pixels+1)]:=10;
            testlist:=testlist^.next;
         END;
   END; {Testarray }

END.
```

Four unit

```
{
This unit contains the code for the experimental Fourier correlation method.
}
{$R+}    {Range checking off            }
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on             }
{$I+}    {I/O checking on               }
{$N+}    {Numeric coprocessor           }

UNIT Four_u;

INTERFACE

USES
   Global_u,
   Dt_2,
   Debug_u;

CONST
     isign =1;  {+1:Fourietransform,
      -1:Inverse fourietransform     }
     k     =1;  {k-value described in the theory chapter
      With a shorter sampling interval, this
      value should be increased or maybe
      changed into a variable   }

FUNCTION fft_measure_wind_velocity(VAR region       :RoI;
                                   VAR picture_list :picture_pointer
                                   ):real;
{
This function measures the displacement between the images in pixels.
}

IMPLEMENTATION{
-------------------------------------------------------------------------------
}

FUNCTION fft_measure_wind_velocity(VAR region       :RoI;
                                   VAR picture_list :picture_pointer
                                   ):real;
{
This function measures the displacement between the images in pixels.
}


VAR
     image         :transformfunction;    {storagearray for the     }
     {prepared pictures   }
     image_list    :fft_pointer;
     peak          :integer;    {frequency of greatest peak     }
     {of fourietransform  }
     pixelvelocity :real; {displacement in pixels per frame    }
     umax          :real; {maximum frequency allowed}
     frequency     :real;
     scale         :real;
     comment       :string[255];     {used in debug mode  }
```

```
{
-------------------------------------------------------------------------------
}
   PROCEDURE DisposeImageList(VAR image_list:fft_pointer);
   {
   Frees the memory occupied by the linked list image_list
   }

   VAR
     next_picture:fft_pointer;

   BEGIN
      REPEAT
         next_picture:=image_list^.next;
         dispose(image_list);
         image_list:=next_picture;
      UNTIL image_list=nil;
   END;  {Dispose_image_list}

{
-------------------------------------------------------------------------------
}
   PROCEDURE fourier(VAR data    : transformfunction;
                         nn,isign: integer);
   {
   Performs Fast Fourier Transform of the input data. This procedure is adapted
   from a Fortran procedure from 'Numerical Recipies'.
   }

   VAR
        ii,jj,n,mmax,m,j,istep,i : integer;
        wtemp,wr,wpr,wpi,wi,theta: real;
        tempr,tempi,pi2          : real;

   BEGIN
      pi2:=2*pi;
      n := 2*nn;
      j := 1;
      FOR ii := 1 TO nn DO BEGIN
         i := 2*ii-1;
         IF (j > i) THEN BEGIN
            tempr := data[j];
            tempi := data[j+1];
            data[j] := data[i];
            data[j+1] := data[i+1];
            data[i] := tempr;
            data[i+1] := tempi
         END;
         m := n DIV 2;
         WHILE ((m >= 2) AND (j > m))  DO BEGIN
            j := j-m;
            m := m DIV 2
         END;
         j := j+m
      END;
      mmax := 2;
      WHILE (n > mmax) DO BEGIN
         istep := 2*mmax;
         theta := pi2/(isign*mmax);
```

```
         wpr := -2.0*sqr(sin(0.5*theta));
         wpi := sin(theta);
         wr := 1.0;
         wi := 0.0;
         FOR ii := 1 TO (mmax DIV 2) DO BEGIN
            m := 2*ii-1;
            FOR jj := 0 TO ((n-m) DIV istep) DO BEGIN
               i := m + jj*istep;
               j := i+mmax;
               tempr := wr*data[j]-wi*data[j+1];
               tempi := wr*data[j+1]+wi*data[j];
               data[j] := data[i]-tempr;
               data[j+1] := data[i+1]-tempi;
               data[i] := data[i]+tempr;
               data[i+1] := data[i+1]+tempi
            END;
            wtemp := wr;
            wr := wr*wpr-wi*wpi+wr;
            wi := wi*wpr+wtemp*wpi+wi
         END;
         mmax := istep
      END
   END;  {fourier}

{
-------------------------------------------------------------------------------
}

   FUNCTION PeakSearch (VAR data  :transformfunction;
                            range :integer):integer;
   {
   Finds the greatest peak of the fourietransformed function DATA and
   places the result in FREQUENCY.
   }

   VAR
        number   :real;    {magnitude of fouriertransform at
      frequency u    }
        u,i,peak :integer;
        maximum  :real;    {greatest number so far    }

   BEGIN
      maximum:=0;
      peak   :=0;
      FOR u:= 1 to (range-1) DO
         BEGIN
            i:=2*u+1;      {i=0 corresponds to velocity = 0     }
            number:=sqr(data[i])+sqr(data[i+1]);
            IF number>maximum THEN
               BEGIN
                  maximum:=number;
                  peak   := u;
               END;
         END;
     Debugln('peak=',peak);

    IF peak > (range div 2) THEN     {same frequencies, but opposite
      wind direction }
         PeakSearch := range - peak
```

```
      ELSE
         PeakSearch := peak;
   END;{ Peaksearch }

{
-------------------------------------------------------------------------------
}

  PROCEDURE PrepareData(VAR picture_list:picture_pointer;
                        VAR image_list:fft_pointer);
  {
  Moves the data in picture_list to a complex array in image_list with the
  imaginary elements set to zero. FFT is performed on each picture in the list.
  }

   VAR
        ii,a                    :integer;
        choosepicture, done     :picture_pointer;
        temp,latest             :fft_pointer;

   BEGIN
      choosepicture:=picture_list;
      latest := nil;
      debugln('PrepareData  no_of_pics',no_of_pictures);
      debug('region.xpos: ',region.xpos);
      debug('region.xsize: ',region.xsize);
      debug('region.ypos: ',region.ypos);
      debugln('region.ysize: ',region.ysize);

      FOR a := 1 to no_of_pictures DO
         BEGIN
            new(temp);
            FOR ii:= 1 TO region.xsize DO {For every column do...    }
               BEGIN
                  temp^.data[ii*2-1] := choosepicture^.picture[ii-1];
                  temp^.data[ii*2] := 0.0;
               END;{FOR}
            done := choosepicture;
            choosepicture := choosepicture^.next;
            dispose(done);

            Fourier(temp^.data,region.xsize,isign);

            IF latest <> nil THEN
               BEGIN
                  latest^.next := temp;
                  latest := temp;
               END
            ELSE
               BEGIN
                  latest := temp;
                  image_list := temp;
                  picture_list := nil;
               END;
         END;
      temp^.next := nil;
   END;{ PrepareData }
```

```
{
------------------------------------------------------------------------------
}
   PROCEDURE MakeVector(image_list :fft_pointer;
                        VAR image  :transformfunction);
   {
   Places the data representing a specific k-value in each picture into an array.
   This array represents all pictures for this specific k-value.
   }
   VAR
      ii            :integer;
      choosepicture :fft_pointer;

   BEGIN
      choosepicture := image_list;

      FOR ii := 1 TO (no_of_pictures) DO
         BEGIN
            image[ii*2-1] := choosepicture^.data[k*2+1];
            image[ii*2] := choosepicture^.data[k*2+2];
            choosepicture := choosepicture^.next;
         END;{FOR}
   END;{MakeVektor}
{
------------------------------------------------------------------------------
}

BEGIN{fft_measure_wind_velocity}

   umax:=1/(2*(sample_interval/100));     {Max frequency according to Nyquist  }

   PrepareData(picture_list,image_list);  {FFT each picture    }
   MakeVector(image_list,image);     {Make one array from choosen k-value }
   Fourier(image,no_of_pictures,isign);
   peak         := Peaksearch(image,no_of_pictures);
   frequency    := peak * (umax / (no_of_pictures div 2));
   pixelvelocity:= (no_of_pictures*frequency/k)*sample_interval/100; {pixels/frame}

   debug('k=',k);
   debug('peak=',peak);
   debug('freq=',frequency);
   debug('pixel v=',pixelvelocity);

   fft_measure_wind_velocity := pixelvelocity;
   DisposeImageList(image_list);

   debugln('*********************************************',0);
END;  {fft_measure_wind_velocity}

END.
```

Corr unit

```
{
This unit contains the procedures for the correlation velocity measurement method.
}

{$R+}    {Range checking off}
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on}
{$I+}    {I/O checking on}
{$N+}    {Numeric coprocessor}

UNIT Corr_u;

INTERFACE

USES
   Global_u,
   Dt_2,
   Debug_u;

FUNCTION corr_measure_wind_velocity(VAR region:RoI;
                                    VAR picture_list:picture_pointer):real;
{
This function returns the displacement between images measured in pixels.
}

IMPLEMENTATION
{
Windspeed measurement procedures

-------------------------------------------------------------------------------
}

FUNCTION corr_measure_wind_velocity(VAR region       :RoI;
                                    VAR picture_list :picture_pointer) :real;
{
This function returns the displacement between images measured in pixels.
}
VAR

   pixelvelocity: real;
   peak      :integer; { frequency of greatest peak of fourietransform }
   umax      :real;
   frequency :real;
   speed     :real;

   {
   ---------------------------------------------------------------------------
   }

   PROCEDURE fourier(VAR data     :transformfunction2;
                         nn,isign :integer);
   {
   Performs Fast Fourier Transform of the input data. Stolen with pride.
   }
   VAR
        ii,jj,n,mmax,m,j,istep,i : integer;
        wtemp,wr,wpr,wpi,wi,theta: real;
```

```
      tempr,tempi,pi2          : real;

BEGIN
   pi2:=2*pi;
   n := 2*nn;
   j := 1;
   FOR ii := 1 TO nn DO BEGIN
      i := 2*ii-1;
      IF (j > i) THEN BEGIN
         tempr := data[j];
         tempi := data[j+1];
         data[j] := data[i];
         data[j+1] := data[i+1];
         data[i] := tempr;
         data[i+1] := tempi
      END;
      m := n DIV 2;
      WHILE ((m >= 2) AND (j > m))  DO BEGIN
         j := j-m;
         m := m DIV 2
      END;
      j := j+m
   END;
   mmax := 2;
   WHILE (n > mmax) DO BEGIN
      istep := 2*mmax;
      theta := pi2/(isign*mmax);
      wpr := -2.0*sqr(sin(0.5*theta));
      wpi := sin(theta);
      wr := 1.0;
      wi := 0.0;
      FOR ii := 1 TO (mmax DIV 2) DO BEGIN
         m := 2*ii-1;
         FOR jj := 0 TO ((n-m) DIV istep) DO BEGIN
            i := m + jj*istep;
            j := i+mmax;
            tempr := wr*data[j]-wi*data[j+1];
            tempi := wr*data[j+1]+wi*data[j];
            data[j] := data[i]-tempr;
            data[j+1] := data[i+1]-tempi;
            data[i] := data[i]+tempr;
            data[i+1] := data[i+1]+tempi
         END;
         wtemp := wr;
         wr := wr*wpr-wi*wpi+wr;
         wi := wi*wpr+wtemp*wpi+wi
      END;
      mmax := istep
   END
END;  {fourier}

{
--------------------------------------------------------------------------------
}

FUNCTION PeakSearch (VAR data  :transformfunction2;
                         range :integer):integer;
{
Locates the maximum value in a vector. This procedure is used to
find the displacement for which the correlation has a maximum. The
```

```
displacement zero is never considered since that would indicate
no wind at all.
}
VAR
     number   :real;
     u,i,peak :integer;
     maximum  :real;

BEGIN
   maximum:=0;
   peak   :=0;

   FOR u:= 1 to (range-1) DO BEGIN   {for all possible correlations}
      i:=2*u+1;                      {data is a complex array      }
      number:=sqr(data[i])+sqr(data[i+1]);

      IF number>maximum THEN BEGIN
         maximum :=number;
         peak    := u;
      END;{IF}

   END;{FOR}

   Debugln('Peak: ',peak);
   IF peak > (range div 2) THEN     {the correlation was found by shifting the
                                     first picture to the left               }
      PeakSearch :=  range - peak
   ELSE                             {the correlation was found by shifting the
                                     first picture to the right              }
      PeakSearch := 0 - peak;
END;{ Peaksearch }

{
--------------------------------------------------------------------------
}

PROCEDURE Twofft(var data1,data2:transformfunction;
                 var fft1,fft2:transformfunction2; n:integer);
{
This procedure calculates the Fast Fourier Transform of the
real input functions data1 and data2. The two functions,
data1 and data2, are packed into the array fft1 in such a way
that their individual transforms can be separated from the
result. This can be done considering the symmetry of the
transform of a purely real function FN-n = (Fn)*, and the
symmetry of the transform of a purely complex function
GN-n = -(Gn)*. data1 and data2 are packed as the real and
imaginary parts respectively of the complex input array fft1
of procedure Fourier. The resulting transform array can be
unpacked with the aid of the two symmetries.
}
VAR
   i,j,u            :integer;
   H1r,H1i,H2r,H2i  :real;

BEGIN
   FOR j := 1 TO n DO               {The functions are packed             }
      BEGIN
         i := 2 * j - 1;
```

```
            fft1[i] := data1[j];
            fft1[i+1] := data2[j];
         END;
      Fourier(fft1,n,1);
      fft2[1] := fft1[2];
      fft2[2] := 0.0;
      fft1[2] := 0.0;
      FOR j := 2 TO n div 2 + 1 DO      {The transformed functions are unpacked}
         BEGIN
            i := 2 * j - 1;
            u := (n + 2 - j) * 2;
            H1r := 0.5 * (fft1[i] + fft1[u-1]);
            H1i := 0.5 * (fft1[i+1] - fft1[u]);
            H2r := 0.5 * (fft1[i+1] + fft1[u]);
            H2i := -0.5 * (fft1[i] - fft1[u-1]);
            fft1[i] := H1r;
            fft1[i+1] := H1i;
            fft1[u-1] := H1r;
            fft1[u] := -H1i;
            fft2[i] := H2r;
            fft2[i+1] := H2i;
            fft2[u-1] := H2r;
            fft2[u] := -H2i;
         END;
   END;(* Twofft *)

   {
   ------------------------------------------------------------------------
   }

PROCEDURE Correlate(    picture_list  :picture_pointer;
                    VAR pixelvelocity :real);

TYPE
   corr_pointer=^transformfunction;
   shiftarray  = array [1..63] of integer;

VAR
    picture1,picture2,temp   :corr_pointer;
    period,ii                :integer;
    shift,image              :integer;
    choosepicture            :picture_pointer;
    ans                      :transformfunction2;
    shifts                   :shiftarray;

   {
   ------------------------------------------------
   }
   FUNCTION Median(var shifts:shiftarray;
                       nn     :integer):integer;
   VAR  x,y,temp:integer;

   BEGIN
      FOR x:= 2 to nn DO
         FOR y:=x to nn DO
            IF shifts[y]<shifts[y-1] THEN
               BEGIN
                  temp := Shifts[y-1];
                  Shifts[y-1] := Shifts[y];
```

```
                Shifts[y] := temp;
              END;
    Median:=Shifts[nn div 2];
END; {Median}

{
------------------------------------------------------------
}
PROCEDURE Fourcorrelation(VAR data1,data2 :transformfunction;
                              n           :integer;
                          VAR ans         :transformfunction2);
{
Computes the correlation of of two real data sets, data1 and
data2, each of length n. n must be an integer number of two.
The answer is returned as the first n points in ans stored in
wraparound order, i.e. correlations at increasingly negative lags
are in ans(n) on down to ans(n/2+1), while correlations at
increasingly positive lags are in ans(1) (zero lag) on up to
ans(n/2). ans must be at lenght at least 2*n, since it is also
used as working space. Sign convention of this routine: if data1
lags data2, i.e. is shifted to the right of it, then ans will show
a peak at positive lags.
The theory behind this algorithm is that the correlation can be
calculated by FFT the two data sets, multiply one resulting transform
by the complex conjugate of the other, and inverse transform the
product. The result will be a complex vector of length n.
}

VAR
   i,u,p    :integer;
   temp     :real;
   fft      :transformfunction2;

BEGIN
   Twofft(data1,data2,fft,ans,n);          {FFT the two data sets}

   {
   Multiply one resulting transform by the complex conjugate of the other.
   }
   temp   := (fft[1] * ans[1] + fft[2] * ans[2])/(n div 2);
   ans[2] := (fft[2] * ans[1] - fft[1] * ans[2])/(n div 2);
   ans[1] := temp;
   FOR i := 2 TO n div 2 + 1 DO
      BEGIN
         u := 2*i - 1;
         p := (n + 2 - i) * 2;
         temp := (fft[u] * ans[u] + fft[u+1] * ans[u+1])/(n div 2);
         ans[u+1] := (fft[u+1] * ans[u] - fft[u] * ans[u+1])/(n div 2);
         ans[u] := temp;
         ans[p-1] := ans[u];
         ans[p] := -ans[u+1];
      END;

   Fourier(ans,n,-1);                          {inverse transform the product}
END;(*Fourcorrelation *)


BEGIN {Correlate}
```

```
      choosepicture := picture_list;
      image := 1;
      new(picture1);

      {To get around the problem of not having a periodic function, zero
      padding is used. For this purpose we move our data to an array with
      more elements, and add zeros at the end.}

      FOR ii := 0 TO region.xsize -1 DO BEGIN
         picture1^[ii+1] := choosepicture^.picture[ii];
      END;{FOR}

      new(picture2);
      period := pixels * 2;                          {must be integer power of two}
      FOR ii := region.xsize+1 TO period DO
         BEGIN
            picture1^[ii] := 0.0;
            picture2^[ii] := 0.0;
         END;
      WHILE (choosepicture^.next <> nil) DO BEGIN
          choosepicture := choosepicture^.next;

          FOR ii := 0 TO region.xsize - 1 DO BEGIN
             picture2^[ii+1] := choosepicture^.picture[ii];
          END;{FOR}

          Fourcorrelation(picture1^,picture2^,period,ans);

          shift := Peaksearch(ans,period);
          shifts[image] := shift;
          image:= image+1;
          temp := picture1;
          picture1 := picture2;
          picture2 := temp;
      END;{WHILE}

      IF image > 2 THEN
         pixelvelocity := Median(shifts,image-1)
      ELSE
         pixelvelocity := shifts[1];

      dispose(picture1);
      dispose(picture2);
   END;{Correlate}



BEGIN
   Correlate(picture_list,pixelvelocity);
   debug('pixel v=',pixelvelocity);
   corr_measure_wind_velocity := pixelvelocity;
   debugln('***********************************************',0);
END;  {corr_measure_wind_velocity}




END.
```

DT_2 unit

```
{
This unit is the Turbo Pascal interface for the Data Translation DT2851 frame
grabber.
}

{$R+}    {Range checking off}
{$B+}    {Boolean complete evaluation on}
{$S+}    {Stack checking on}
{$I+}    {I/O checking on}
{$N+}    {Numeric coprocessor}

UNIT Dt_2;

INTERFACE

CONST
     Base     =$390;
     Buf0     =$A00000;
     Buf1     =$A40000;
     pixels   =512;

TYPE
     Lut      =array[0..255] of word;
     Sum      =array[0..pixels] of longint;
     RoI      =RECORD
                  buffer     :longint;
                  xpos,ypos  :word;
                  xsize,ysize:word;
               END;


PROCEDURE dt_wait;

{  Returns true if the frame-grabber is busy                          }


FUNCTION get386mem(adress:longint):byte;

{  Returns the value of the byte at adress                            }


PROCEDURE put386mem(adress:longint;value:byte);

{  Puts the value at memory location given by adress                  }


PROCEDURE copy386roi(source,dest:longint;columns,rows:word);

{  Copys data from adress pointed to by source to adress pointed to
    by dest. Copies 4*columns * rows of data in the frame-grabber
    memory. The source and destination regions must not overlap.      }
```

```
PROCEDURE sumx386roi(VAR result:Sum;source:longint;columns,rows:word);

{  Returns the sum of pixel-values from the columns of the region of
    interest in the array result.                                    }


PROCEDURE dt_load_mask(mask_value:word);

{  Loads the write-protection mask.
    Bit 0, value 1, protects data bit  0
    Bit 1, value 2, protects data bit  1
    Bit 2, value 4, protects data bits 2 & 3
    Bit 3, value 8, protects data bits 4,5,6 & 7                     }


PROCEDURE dt_draw_line(buffer:longint;x1,y1,x2,y2,draw:word);

{  Draws a line in buffer from x1,y1 to x2,y2                        }


PROCEDURE dt_define_roi(VAR region:RoI;buf:longint;xpos,ypos,xsize,ysize:word);

{  Returns a region defined by xpos,ypos,xsize,ysize in the record
    region. This region is used as data to the PROCEDURE dt_copy_roi }


PROCEDURE dt_fill_ilut(table,start,stop,val:word);

{  Fills the input lookup table specified by table from start
    to stop with val.                                                }


PROCEDURE dt_load_ilut(table:word;VAR ilut:lut);

{  Loads the input lookup table specified by table with the
    contents of ilut.                                                }


PROCEDURE dt_load_olut(table:word;VAR red,green,blue:lut);

{  Loads the output lookup table specified by table with the contents
    of the red, green and blue arrays.                               }


PROCEDURE dt_fill_olut(table,start,stop,red,green,blue:word);

{  Fills the output lookup table specified by table from start to
    stop with the values of red, green and blue. If a value greater
    than 255 is given, the lookuptable for that colour will remain
    unchanged.                                                       }
```

```
PROCEDURE dt_select_ilut(table:word);

{  Selects the specified input lookup table                          }


PROCEDURE dt_select_olut(table:word);

{  Selects the specified output lookup table                         }


PROCEDURE dt_set_sync_source(source:word);

{  Selects the internal sync if source=0, otherwise external sync
    is selected.                                                      }


PROCEDURE dt_display(display:word);

{  Turns on the display if display=1                                 }


PROCEDURE dt_select_input_frame(frame:word);

{  Selects the frame buffer into which the frame is read             }


PROCEDURE dt_select_output_frame(frame:word);

{  Selects the frame buffer from which the frame is displayed        }


PROCEDURE dt_passthru;

{  Places the frame-grabber in passthru mode                         }


PROCEDURE dt_acquire;

{  Acquires one frame to the selected input frame                    }


PROCEDURE dt_acquire_two(interval:word);

{  Acquires two frames to the input buffers. Interval sets the
    time interval between samples (1/25)s                           }


PROCEDURE dt_freeze_frame;

{  Stops the passthru mode                                           }
```

```
PROCEDURE dt_copy_roi(source,dest:RoI);

{  Copys data in the frame-grabber from source to destination
    Note that the regions must not overlap. Source and dest are
    convieniently difined by the PROCEDURE dt_define_roi.          }


PROCEDURE dt_sum_columns(source:RoI;VAR result:Sum);

{  Calls sum386roi to calculate column sums                        }


PROCEDURE dt_cursor(cursor_state:word);

{  Turns the cursor display on and off                             }


PROCEDURE dt_set_cursor_position(x,y:word);

{  Places the cursor at a given position                           }


PROCEDURE dt_initialize;

{  Initialize the frame grabber and load the various look up
    tables with default values, giving a greyscale image           }


{=============================================================================}

IMPLEMENTATION


{$L getput}


PROCEDURE dt_wait;

{  Returns true if the frame-grabber is busy                       }

BEGIN
   WHILE ((portW[base] and 128)>0) DO;
END;  {dt_wait}

{
-----------------------------------
}

{$F+}
FUNCTION get386mem(adress:longint):byte;
{  Returns the value of the byte at adress                         }
external;

PROCEDURE put386mem(adress:longint;value:byte);
{  Puts the value at memory location given by adress               }
external;
```

```
PROCEDURE copy386roi(source,dest:longint;columns,rows:word);
{  Copys data from adress pointed to by source to adress pointed to
    by dest. Copies 4*columns * rows of data in the frame-grabber
    memory. The source and destination regions must not overlap.     }
external;

PROCEDURE sumx386roi(VAR result:Sum;source:longint;columns,rows:word);
{  Returns the sum of pixel-values from the columns of the region of
    interest in the array result.                                    }
external;
{$F-}

{
-------------------------------------
}

PROCEDURE dt_load_mask(mask_value:word);

{  Loads the write-protection mask.
    Bit 0, value 1, protects data bit  0
    Bit 1, value 2, protects data bit  1
    Bit 2, value 4, protects data bits 2 & 3
    Bit 3, value 8, protects data bits 4,5,6 & 7                     }

BEGIN
   dt_wait;
   portW[base+2]:=(portW[base+2] and 240) or (mask_value and 15);
END;  {dt_load_mask}

{
-------------------------------------
}

PROCEDURE dt_draw_line(buffer:longint;x1,y1,x2,y2,draw:word);

{  Draws a line in buffer from x1,y1 to x2,y2                        }

VAR
     x,y   :word;
     adress:longint;

BEGIN
   x:=x1;
   y:=y1;
   if (x2-x1)>(y2-y1) then
      FOR x:=x1 TO x2 DO
         BEGIN
            if x2<>x1 then
               y:=y1+(y2-y1)*(x-x1) div (x2-x1);
            adress:=y;
            adress:=adress*pixels+buffer+x;
            IF draw=1 THEN
               put386mem(adress,get386mem(adress) or 1)
            ELSE
               put386mem(adress,get386mem(adress) and 254);
         END
   else
      FOR y:=y1 TO y2 DO
         BEGIN
```

```
            if y2<>y1 then
               x:=x1+(x2-x1)*(y-y1) div (y2-y1);
            adress:=y;
            adress:=adress*pixels+buffer+x;
            IF draw=1 THEN
               put386mem(adress,get386mem(adress) or 1)
            ELSE
               put386mem(adress,get386mem(adress) and 254);
         END;
END;  {dt_draw_line}

{
-------------------------------------
}

PROCEDURE dt_define_roi(VAR region:RoI;buf:longint;xpos,ypos,xsize,ysize:word);
{  Returns a region defined by xpos,ypos,xsize,ysize in the record
    region. This region is used as data to the PROCEDURE dt_copy_roi }


BEGIN
   region.buffer:=buf;
   region.xpos:=xpos;
   region.ypos:=ypos;
   region.xsize:=xsize;
   region.ysize:=ysize;
END;  {dt_define_roi}

{
-------------------------------------
}

PROCEDURE dt_fill_ilut(table,start,stop,val:word);

{  Fills the input lookup table specified by table from start
    to stop with val.                                              }



VAR
     lut_index    :word;
     inscr1,inscr2:word;

BEGIN
   dt_wait;  { Wait for vertical sync }
   inscr1:=portW[base];
   inscr2:=portW[base+2];
   portW[base+2]:=64;               { Set load lut mode      }
   portW[base]:=(table and 7) or 8; { Select lut             }
   FOR lut_index:=start TO stop DO
      BEGIN
         portW[base+8]:=lut_index;
         portW[base+10]:=val;
      END;
   portW[base]:=inscr1;
   portW[base+2]:=inscr2;
END;  { dt_fill_ilut }
```

```
{
-------------------------------------
}

PROCEDURE dt_load_ilut(table:word;VAR ilut:lut);

{  Loads the input lookup table specified by table with the
    contents of ilut.                                          }

VAR
     lut_index    :word;
     inscr1,inscr2:word;

BEGIN
   dt_wait;  { Wait for vertical sync }
   inscr1:=portW[base];
   inscr2:=portW[base+2];
   portW[base+2]:=64;              { Set load lut mode      }
   portW[base]:=(table and 7) or 8; { Select lut            }
   FOR lut_index:=0 TO 255 DO
      BEGIN
         portW[base+8]:=lut_index;
         portW[base+10]:=ilut[lut_index];
      END;
   portW[base]:=inscr1;
   portW[base+2]:=inscr2;
END;  { dt_load_ilut }

{
-------------------------------------
}

PROCEDURE dt_load_olut(table:word;VAR red,green,blue:lut);

{  Loads the output lookup table specified by table with the contents
    of the red, green and blue arrays.                         }

VAR
     lut_index    :word;
     inscr1,inscr2:word;

BEGIN
   dt_wait;  { Wait for vertical sync }
   inscr1:=portW[base+2];
   inscr2:=portW[base+4];
   portW[base+2]:=64;              { Set load lut mode      }
   portW[base+4]:=(table and 7);   { Select lut             }
   FOR lut_index:=0 TO 255 DO
      BEGIN
         portW[base+8]:=lut_index;
         portW[base+12]:=red[lut_index]+green[lut_index]*256;
         portW[base+14]:=blue[lut_index];
      END;
   portW[base+2]:=inscr1;
   portW[base+4]:=inscr2;
END;  { dt_load_olut }

{
-------------------------------------
}
```

```
PROCEDURE dt_fill_olut(table,start,stop,red,green,blue:word);

{  Fills the output lookup table specified by table from start to
    stop with the values of red, green and blue. If a value greater
    than 255 is given, the lookuptable for that colour will remain
    unchanged.                                                    }

VAR
     lut_index,rg :word;
     inscr1,inscr2:word;

BEGIN
   dt_wait;                { Wait for vertical sync }
   inscr1:=portW[base+2];
   inscr2:=portW[base+4];
   portW[base+2]:=64;              { Set load lut mode      }
   portW[base+4]:=(table and 7);   { Select lut             }
   FOR lut_index:=start TO stop DO
      BEGIN
         portW[base+8]:=lut_index;
         if green<256 then
            rg:=green*256
         else
            rg:=0;
         if red<256 then
            rg:=rg+red;
         if (red<256) or (green<256) then
            portW[base+12]:=rg;
         if blue<256 then
            portW[base+14]:=blue;
      END;
   portW[base+2]:=inscr1;
   portW[base+4]:=inscr2;
END;  { dt_fill_olut }

{
_____________________________________
}

PROCEDURE dt_select_ilut(table:word);

{  Selects the specified input lookup table                       }

BEGIN
   portW[base]:=((portW[base] and 248) or (table and 7));
END; {dt_select_ilut}

{
_____________________________________
}

PROCEDURE dt_select_olut(table:word);

{  Selects the specified output lookup table                       }

BEGIN
   portW[base+4]:=((portW[base+4] and 240) or (table and 7));
END; {dt_select_olut}
```

```
{
-------------------------------------
}

PROCEDURE dt_set_sync_source(source:word);

{  Selects the internal sync if source=0, otherwise external sync
    is selected.                                                    }

BEGIN
   if source=0 then
      portW[base+4]:=(portW[base+4] and (not 32))
   else
      portW[base+4]:=(portw[base+4] or 32);
END; {dt_set_sync_source}

{
-------------------------------------
}

PROCEDURE dt_display(display:word);

{  Turns on the display if display=1                              }

BEGIN
   if display=0 then
      portW[base+4]:=(portW[base+4] and (not 128))
   else
      portW[base+4]:=(portw[base+4] or 128);
END; {dt_display}

{
-------------------------------------
}

PROCEDURE dt_select_input_frame(frame:word);

{  Selects the frame buffer into which the frame is read          }

BEGIN
   if frame=0 then
      portW[base+2]:=(portW[base+2] and (not 128))
   else
      portW[base+2]:=(portw[base+2] or 128);
END; {dt_select_input_frame}

{
-------------------------------------
}

PROCEDURE dt_select_output_frame(frame:word);

{  Selects the frame buffer from which the frame is displayed     }

BEGIN
   if frame=0 then
      portW[base+4]:=(portW[base+4] and (not 16))
   else
      portW[base+4]:=(portw[base+4] or 16);
END; {dt_select_output_frame}
```

```
{
-------------------------------------
}

PROCEDURE dt_passthru;

{  Places the frame-grabber in passthru mode                          }

BEGIN
   dt_wait;
   portW[base+2]:=((portW[base+2] and 143) or 16);
   portW[base]:=(portW[base] and 135);
   portW[base]:=(portW[base] or 128);
END; {dt_passthru}

{
-------------------------------------
}
{
-------------------------------------
}

PROCEDURE dt_acquire;

{  Acquires one frame to the selected input buffer                     }

BEGIN
   dt_wait;
   portW[base+2]:=((portW[base+2] and 143) or 16);
   portW[base]:=(portW[base] and 135);
   portW[base]:=(portW[base] or 136)
END; {dt_acquire}

{
-------------------------------------
}

PROCEDURE dt_acquire_two(interval:word);

{  Acquires two frames to the input buffers. Interval sets the
    time interval between samples (1/25)s                             }

BEGIN
   dt_wait;
   portW[base+4]:=(portW[base+4] and (not 128));
   portW[base+2]:=((portW[base+2] and 143) or 16);
   portW[base]:=(portW[base] and 135);
   portW[base]:=(portW[base] or 136);
   REPEAT
      interval:=interval-1;
      dt_wait;
   UNTIL interval<1;
   portW[base+2]:=(portw[base+2] or 128);
   portW[base+2]:=((portW[base+2] and 143) or 16);
   portW[base]:=(portW[base] and 135);
   portW[base]:=(portW[base] or 136);
   portW[base+4]:=(portW[base+4] and (not 128))
END; {dt_acquire_two}
```

```
{
_____________________________________
}

PROCEDURE dt_freeze_frame;

{  Stops the passthru mode                                          }

BEGIN
   portW[base]:=portW[base] or 8;
END; {dt_freeze_frame}



PROCEDURE dt_copy_roi(source,dest:RoI);
{  Copys data in the frame-grabber from source to destination
    Note that the regions must not overlap. Source and dest are
    convieniently difined by the PROCEDURE dt_define_roi.           }

VAR
     sadr,dadr:longint;

BEGIN
   sadr:=pixels*source.ypos+source.buffer+source.xpos;
   dadr:=dest.ypos;
   dadr:=pixels*dadr+dest.buffer+dest.xpos;
   copy386roi(sadr,dadr,dest.xsize,(dest.ysize div 4));
   dt_freeze_frame;
END; {dt_copy_roi}

{
_____________________________________
}

PROCEDURE dt_sum_columns(source:RoI;VAR result:Sum);
{  Calls sum386roi to calculate column sums                         }

VAR
     sadr     :longint;

BEGIN
   sadr:=pixels*source.ypos+source.buffer+source.xpos;
   dt_wait;
   Sumx386roi(result,sadr,source.xsize,source.ysize);
   dt_freeze_frame;
END; {dt_sum_columns}

{
_____________________________________
}

PROCEDURE dt_cursor(cursor_state:word);
{  Turns the cursor display on and off                              }

BEGIN
   if cursor_state=1 then
      portW[base+4]:=portW[base+4] or 64
   else
      portW[base+4]:=portW[base+4] and 191;
END;  {dt_cursor}
```

```
{
-------------------------------------
}

PROCEDURE dt_set_cursor_position(x,y:word);
{  Places the cursor at a given position                              }

BEGIN
   portW[base+6]:=y*128+(x div 2);
END;  {dt_set_cursor_position}

{
-------------------------------------
}

PROCEDURE dt_initialize;
{  Initialize the frame grabber and load the various look up
    tables with default values, giving a greyscale image              }

VAR
     ilut     :lut;
     lut_index:word;

BEGIN

   { Stop all card operations and turn display off }

   portW[base+4]:=0;
   portW[Base]:=8;
   portW[Base]:=8;

   { Clear frame buffers }

   dt_fill_ilut(7,0,255,0);
   dt_wait; {wait until ready }
   portW[base+4]:=0;
   portW[base+2]:=0;
   portW[base]:=143;
   dt_wait;
   portW[base+2]:=128;
   portW[base]:=143;

   { Create and load input lut #0 }

   FOR lut_index:=0 TO 255 DO
      ilut[lut_index]:=lut_index;
   dt_load_ilut(0,ilut);

   { Load output lut #0 for red,green and blue }

   dt_load_olut(0,ilut,ilut,ilut);

   dt_wait;

   { Create and load input lut #5 }

   FOR lut_index:=0 TO 127 DO
      BEGIN
         ilut[2*lut_index]:=2*lut_index;
         ilut[2*lut_index+1]:=2*lut_index;
```

```
      END;
   dt_load_ilut(5,ilut);

   { Load output lut #7 };

   dt_load_olut(7,ilut,ilut,ilut);
   FOR lut_index:=0 TO 127 DO
      dt_fill_olut(7,lut_index*2+1,lut_index*2+1,255,0,0);

END; {dt_initialize}


End.
```

GetPut

```
        .MODEL    TPASCAL
        .CODE
        PUBLIC    Get386mem,Put386mem,Copy386roi,Sumx386roi

Get386mem       PROC FAR hi:WORD,lo:WORD

        .386
        movzx     ebx,[hi]
        shl       ebx,16
        movzx     eax,[lo]
        add       eax,ebx
        xchg      eax,ebx
        push      ds
        mov       ax,0
        mov       ds,ax
        mov       al,[ebx]
        pop       ds
        .8086

        ret
        ENDP

Put386mem       PROC FAR hi:WORD,lo:WORD,val:BYTE

        .386
        movzx     ebx,[hi]
        shl       ebx,16
        movzx     eax,[lo]
        add       eax,ebx
        xchg      eax,ebx
        mov       cl,[val]
        push      ds
        mov       ax,0
        mov       ds,ax
        mov       [ebx],cl
        pop       ds
```

```
        .8086

        ret
        ENDP

Copy386roi      PROC FAR hi1:WORD,lo1:WORD,hi2:WORD,lo2:WORD,col:WORD,row:WORD

        .386
        movzx   esi,[hi1]
        shl     esi,16
        movzx   eax,[lo1]
        add     esi,eax
        movzx   edi,[hi2]
        shl     edi,16
        movzx   eax,[lo2]
        add     edi,eax
        movzx   ecx,[row]
        movzx   edx,[col]
        push    ds
        mov     ax,0
        mov     ds,ax
        mov     es,ax
OuterLoop:
        xchg    bx,cx
        mov     cx,dx
InnerLoop:
        mov     eax,[esi+ecx*4]
        mov     [edi+ecx*4],eax
        loop    InnerLoop
        add     esi,512
        add     edi,512
        xchg    ebx,ecx
        loop    OuterLoop
        pop     ds
        .8086

        ret
        ENDP

Sumx386roi      PROC FAR o:WORD,s:WORD,hi3:WORD,lo3:WORD,col2:WORD,row2:WORD

        .386
        movzx   esi,[hi3]
        shl     esi,16
        movzx   eax,[lo3]
        add     esi,eax
        mov     ax,[o]
        mov     es,ax
        movzx   edi,[s]
        movzx   ecx,[col2]
        mov     ax,[row2]
        mov     fs,ax
        mov     ebp,esi
        push    ds
        mov     ax,0
        mov     ds,ax
OutLoop:
        mov     ebx,ecx
        mov     cx,fs
        mov     esi,ebp
```

```
        mov     eax,0
InLoop:
        movzx   edx,BYTE PTR [esi+ebx]
        add     eax,edx
        add     esi,512
        loop    InLoop
        mov     es:[edi+ebx*4],eax
        mov     cx,bx
        loop    OutLoop
        mov eax,edi;
        mov dx,es
        pop     ds
        .8086

        ret
        ENDP

        END
```

References

1 H. Edner, K. Fredriksson, A. Sunesson, S. Svanberg, L. Unéus and W. Wendt, "Mobile remote sensing system for atmospheric monitoring", Appl. Opt. 26, 4330-4338 (1987).

2 Gonzales, Wintz, "Digital Image Processing", Addison Wesly, (1987)

3 W. Press, B. Flannery, S. Teukolsky, W. Vetterling, "Numerical Recipes: the art of scientific computing", Cambridge university press, (1986)

4 R. Ramirez, "The FFT: fundamentals and concepts", Prentice Hall (1985)