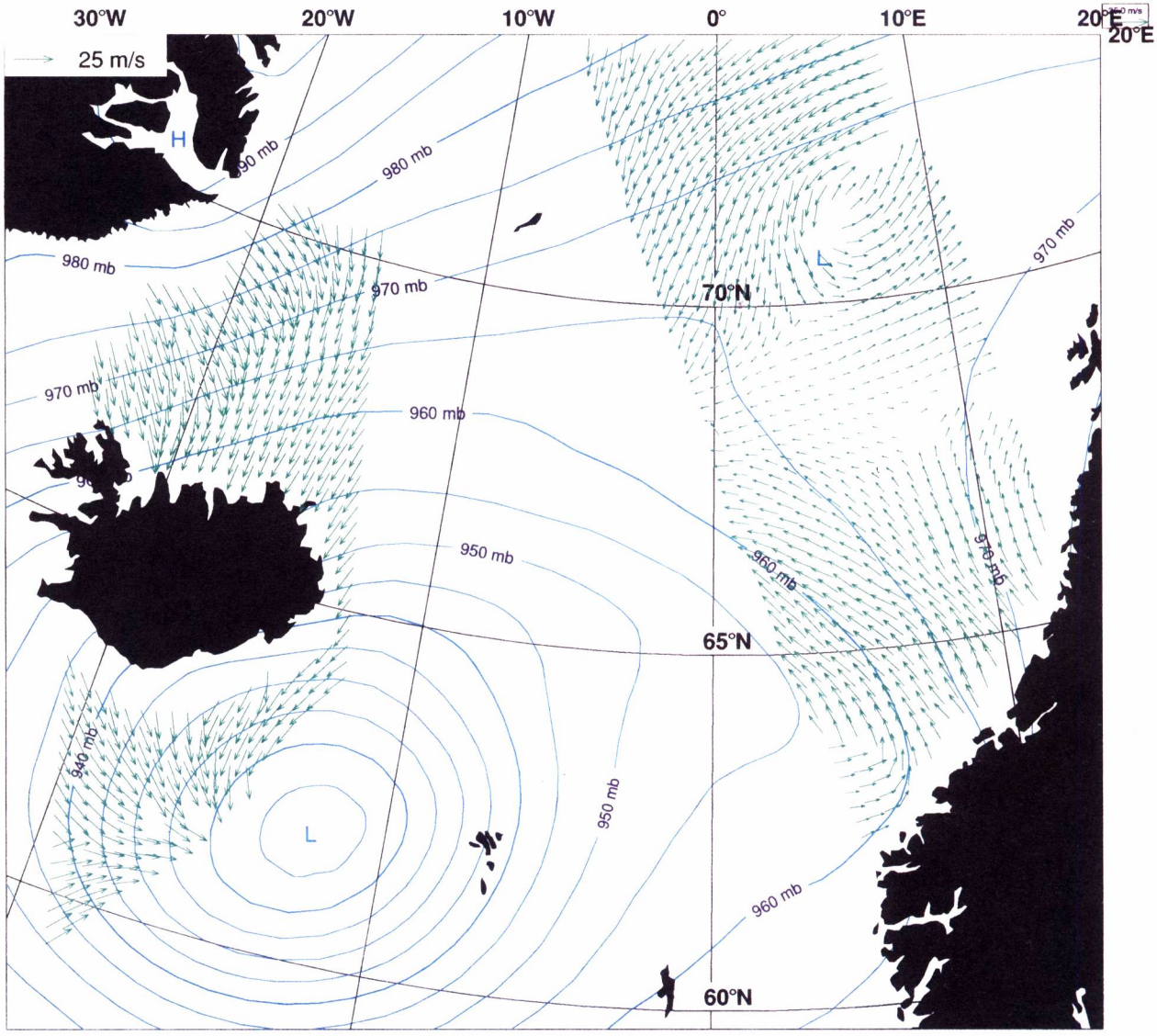
ECMWF Newsletter

Number 66 - Summer 1994

Chosen solutions for CMOD4
3 hour forecast for 93011021



Shinfield Park, Reading, Berkshire RG2 9AX, England. Telephone: U.K. (0734) 499000,
International (+44 734) 499000, Telex: 847908 ECMWF G, Telefax (0734) 869450

European Centre for Medium-Range Weather Forecasts
Europäisches Zentrum für mittelfristige Wettervorhersage
Centre européen pour les prévisions météorologiques à moyen terme

IN THIS ISSUE

COVER: Scatterometer winds (processed at ECMWF) in the North Atlantic on 11 January 1993, when the deepest-ever low pressure system was reported. The first track of scatterometer winds (right) was measured at 21:30 UTC and the second at 23:10. The surface pressure contours are from the operational 3-hour forecast valid at 21:00 UTC.

This Newsletter is edited and produced by User Support.

The next issue will appear in Autumn 1994.

In this edition of the Newsletter we conclude our series of reprints of articles by Jeanne Adams on Fortran 90. In the computing field, there is a description of the features of the POSIX shell under UNICOS 8, and users will also find a helpful article on the Mark 16 news on the NAG Library.

The main meteorological article this time is a detailed discussion on the use and treatment of surface wind observations received from the ERS scatterometers.

As regular items at this time of year, the Newsletter contains the announcements of the ECMWF Annual Seminar (on the parametrization of physical processes in models), and of its sixth workshop on the use of parallel processors in meteorology.

* * * * * * * * * * * * *

CHANGES TO THE OPERATIONAL FORECASTING SYSTEM

Recent changes

A change was made to the fields of soil humidity used in the operational system, starting from the analysis for 4 July 1994 at 00 UTC. In order to compensate for a drying out of the model boundary layer which has been noticed over the last few weeks, particularly over Europe and East Asia, the humidity of the soil was reset to field capacity.

It resulted in a noticeable impact on the temperature in the daytime boundary layer over continental areas. There was a reduction in the warm bias by several degrees in places, and the positive impact was also be seen at 850 and 700 hPa. There was also significant reduction in the bias of the dew point temperature at 2 metres.

Planned changes

A change to the pre-selection of cloud motion wind (SATOB) data for the analysis will be implemented shortly.

The 3D-variational analysis is planned for implementation in the second half of 1994.

- Bernard Strauss

* * * * * * * * * * * *

SURFACE WIND OBSERVATIONS FROM THE ERS SCATTEROMETERS

The first wind observations over the sea were made indirectly. A sailor would observe the ocean surface and estimate the strength of the wind velocity. In fact these Beaufort estimates are still received on GTS and used in daily operations at ECMWF. This indirect or remote approach is mimicked to observe the surface wind velocity from space by use of a scatterometer. The European Space Agency (ESA) launched the first ESA Remote Sensing satellite (ERS-1) on 17 July 1991 carrying a scatterometer on board. The interpretation of its measurements has proved less straightforward than anticipated before the launch. Three years after the launch, the scatterometer wind product is of excellent quality and the challenge of using it in Numerical Weather Prediction (NWP) clearly lies in the field of data assimilation of surface winds. In this newsletter article I shall describe the ERS scatterometer and the problems associated with the interpretation of its measurements. Subsequently, the use of scatterometer data in "optimal interpolation" (OI, as in cu' , ıt ECMWF operations) and variational data assimilation schemes will be discussed.

The ERS satellites

The pre-operational ERS-1 satellite is dedicated to measuring ocean parameters, sea state and ice conditions. The operation of the microwave instruments is not hindered by cloud and measurements are available in all weather conditions. The Active Microwave Instrumentation (AMI) includes a Synthetic Aperture Radar (SAR) to measure wave spectra, an altimeter to measure wave height and wind speed at nadir, and a wind scatterometer to obtain the wind field over the oceans (see Fig. 1). Other instrumentation such as an infra-red radiometer (ATSR) for measuring sea surface temperature, a passive microwave sounder (MWS) to obtain total precipitable water in the atmosphere and a satellite tracking instrument (PRARE) are experimental.

The ERS-2 satellite will be launched in about half a year from now. It will carry instruments similar to those in ERS-1, with the addition of a total ozone measuring instrument (GOME).

The scatterometer

The wind- and wave- measuring instruments on board the ERS satellites use a radar wavelength of roughly 5 cm. The radar beam emitted from the scatterometer will hit the ocean surface at an angle of between 18° and 57°. When the ocean surface is smooth the radar beam will be reflected and no radar power will be returned in the direction of the satellite. However, when capillary ocean waves with a wavelength of at around 5 cm exist on the ocean surface they will interfere

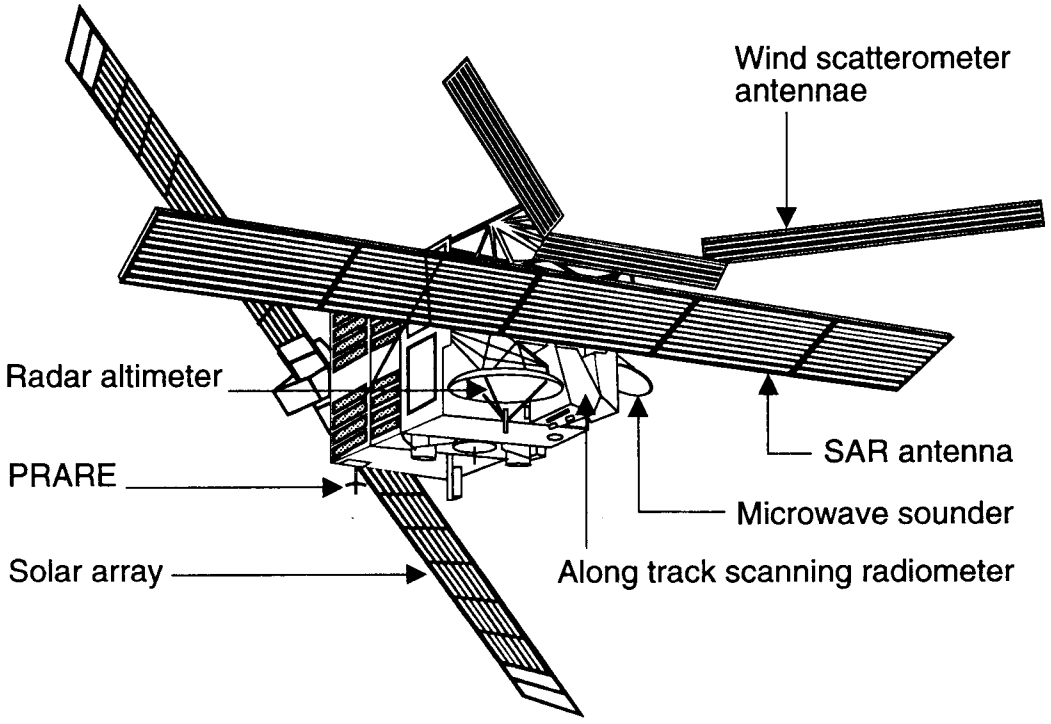


Fig. 1: Overview of the ERS-1 satellite.

with the radar beam, which will be scattered in all directions. The higher the amplitude of the ocean capillary waves (i.e. the rougher the surface), the stronger the scattering will be, and the more radar power will be scattered back to the satellite. Since the roughness is related to the surface wind speed, the measured back-scattered radar power is related to the surface wind speed as well.

The generation of capillary waves from wind can easily be observed in a puddle. This confirms qualitatively the roughness-wind relationship. One can also see that the direction of the capillary waves is the same as that of the wind, i.e. the generation of capillary waves is anisotropic. The consequence of this is that the back-scattered power received by a scatterometer depends on the wind direction. By looking only at the shape of the ripples in the puddle it is difficult to decide whether the wind is blowing from the left or the right. In practice this results in a so-called upwind/downwind ambiguity, i.e. from the scatterometer measurements we will find two possible wind vectors of similar amplitude blowing in roughly opposite directions.

The qualitative description of the scatterometer does not translate into a proper physical model of the effects involved. For instance, the interaction of the radar beam with the complicated topography of the ocean surface is not well understood. To describe the behaviour of the scatterometer quantitatively, an empirical relationship between back-scattered radar power and surface wind speed and direction (at 10m height) has been derived. Experiments were carried out to establish such an empirical relationship in preparation for the launch of ERS-1. Measurements of scatterometers mounted on aircraft were collocated with direct measurements of the surface wind conditions, and the radar back-scattered power was related to wind speed and direction. This relationship, called the "transfer function", has to be inverted to obtain information on wind speed and direction from measurements of back-scattered power. Since two quantities (speed and direction) are wanted, at least two measurements from a different horizontal view angle (azimuth) are required.

The ERS scatterometer

In fact two measurements of back-scattered power are not sufficient to resolve the directional dependency, and as a result four ambiguous wind vector solutions are generally found. Therefore, the ERS-1 and ERS-2 scatterometers (which are identical) have three antennae, viewing the ocean surface in different directions. As the satellite progresses, each node on the Earth's surface will be viewed first by the fore beam, then by the mid beam and last by the aft beam, within a period of 2 to 7 minutes. The back-scattered power is an average over an area of roughly 50 km diameter (called footprint). The swath is over-sampled on a 25 km rectangular grid (see Fig. 2).

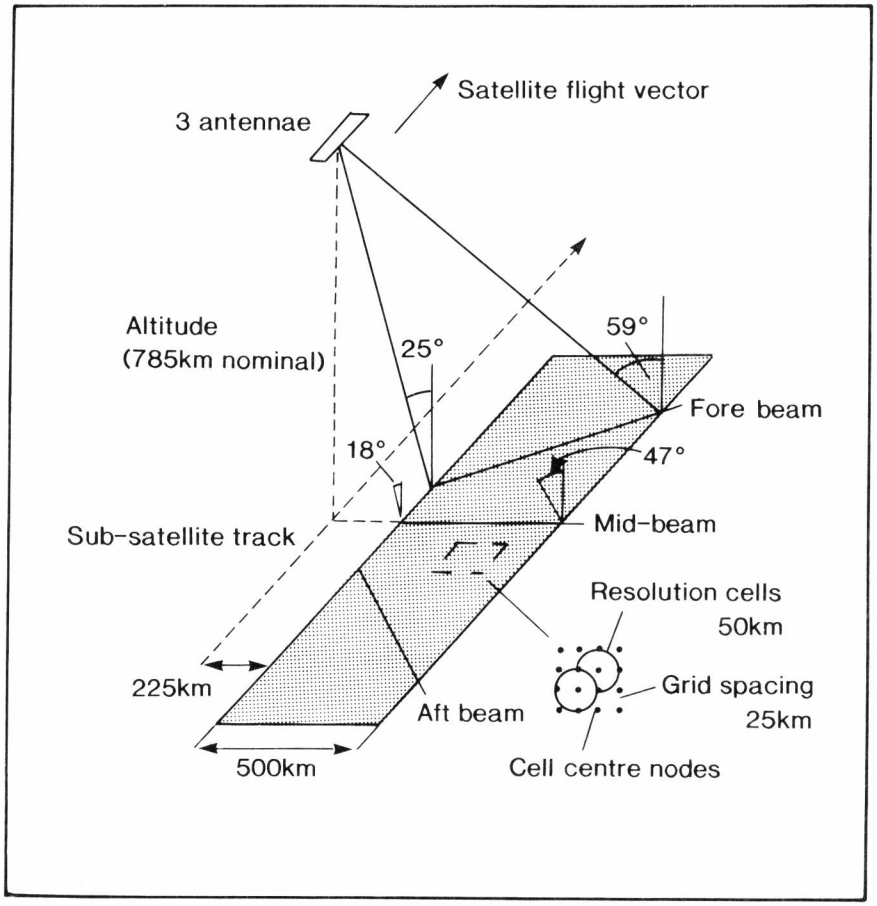


Fig. 2: Schematic representation of the ERS scatterometer.

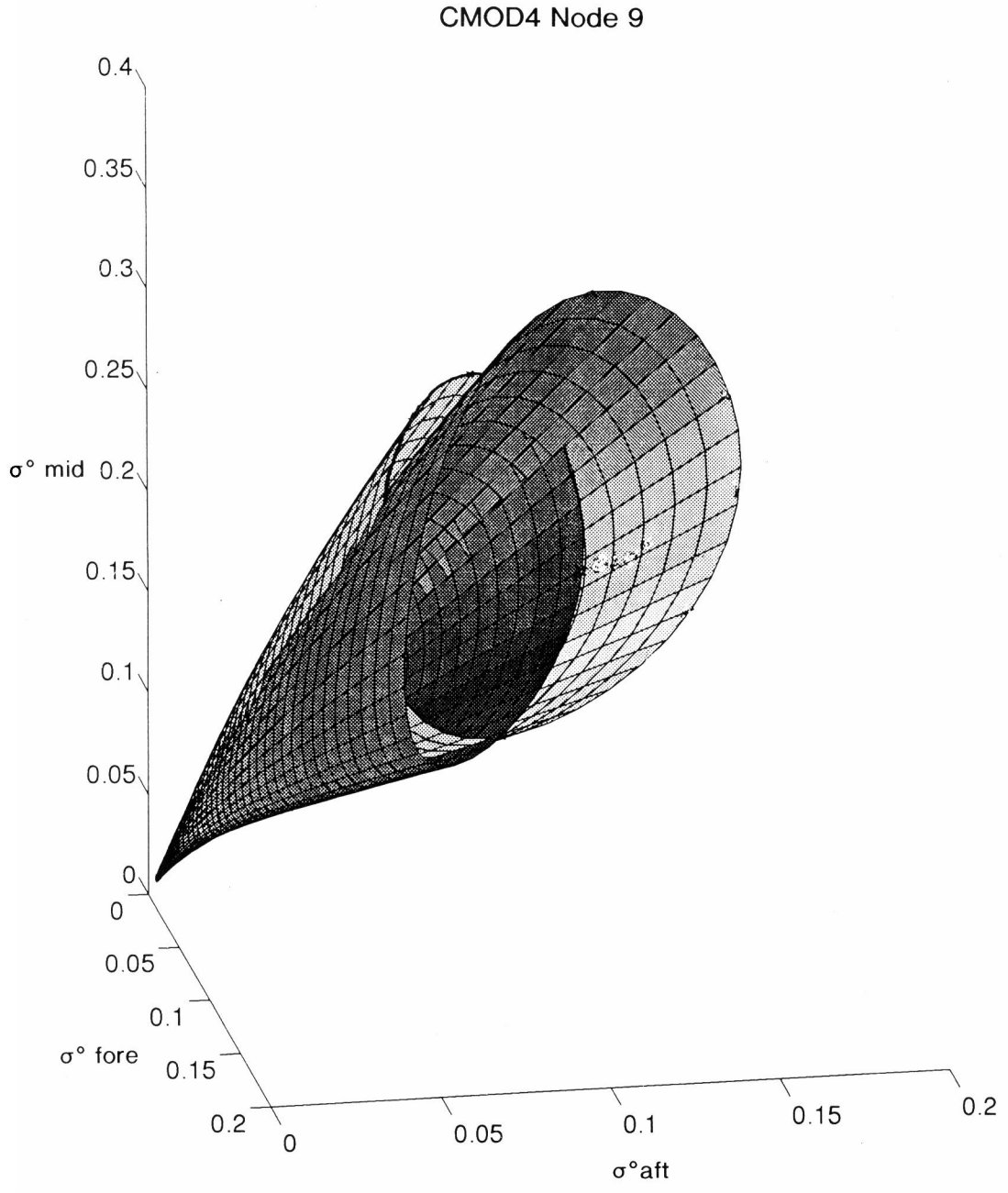


Fig. 3: 3D Measurement space. Triplets of measured back-scattered power are expected to lie close to a cone-shaped surface.

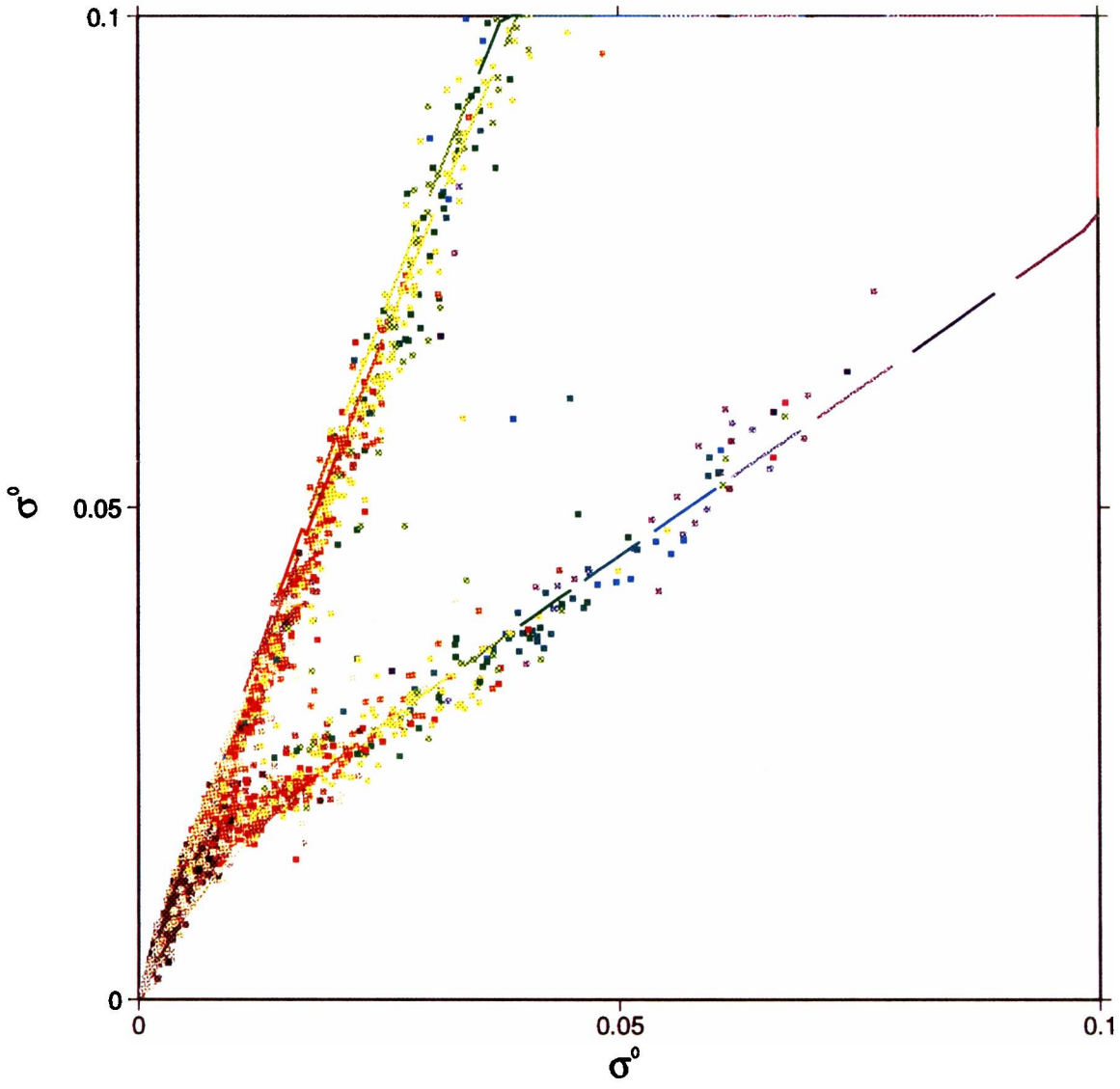


Fig. 4: Example of the distribution of measured triplets in a cross-section along the cone. The expected cone surface is clearly visible. From the lower left to the upper right the wind speed is increasing. Triplets are coloured according to the collocated ECMWF analysis wind speed, where a different colour is used for each 1 m/s wind speed interval. The curve represents the transfer function derived at ECMWF.

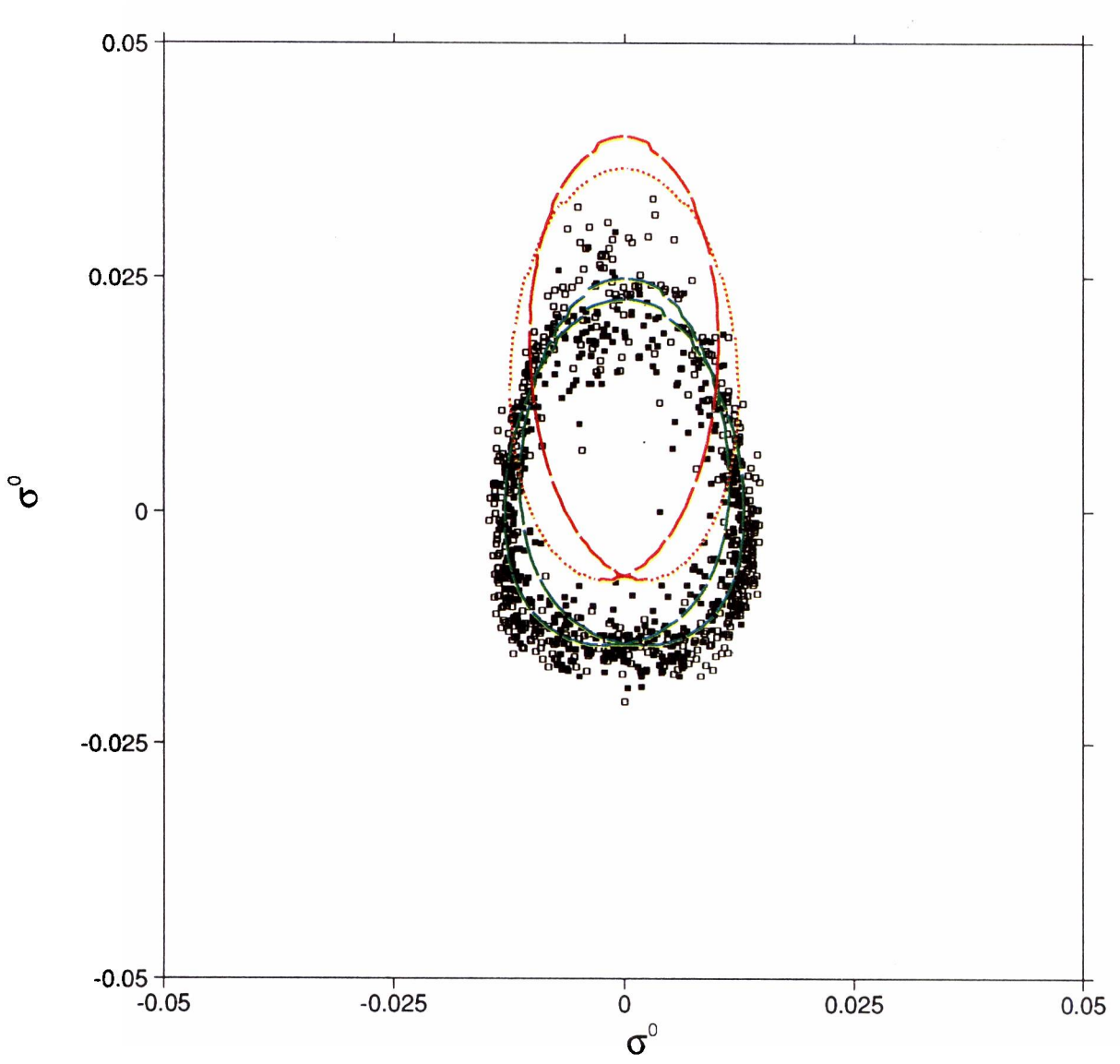


Fig. 5: Example of the distribution of measured triplets in a cross-section across the cone. The expected cone surface can easily be identified. The collocated analysis wind speed for the triplets plotted is roughly 8 m/s.

■:	**"upwind" triplets, according to ECMWF analysis wind direction**
□:	**"downwind" triplets, according to ECMWF analysis wind direction**
red curve:	**the pre-launch transfer function**
green curve:	**transfer function derived at ECMWF (CMOD4)**

Calibration and validation

A calibration and validation campaign was carried out after the launch of ERS-1. Calibration of the ERS-1 scatterometer was planned using both transponders and the Amazon rain forest. When a transponder is hit by a scatterometer radar beam, an amplified and controlled radar signal is sent back to the satellite, allowing an absolute calibration of the antennae. The rain forest is a homogeneous and isotropic scatterer for radar radiation, allowing a relative calibration (i.e. of one antenna with respect to another). Unfortunately, directly after launch it appeared that the two calibrations were mutually inconsistent. At ECMWF we developed a method to calibrate the antennae over the oceans by using ECMWF analysed winds. This confirmed the correctness of the rain forest calibration. Further, we were able to notify ESA of an incorrect implementation of a calibration table within one day. The wealth of data available at ECMWF has proved invaluable for an efficient calibration of the ERS-1 instruments.

Measurement space

The three measurements at each node can be interpreted as a point in a three-dimensional (3D) space. The measurement by each antenna will then represent a coordinate along one axis. A distribution of these points can be plotted in the 3D space. Subsequently, the space can be investigated by making cross-sections through it.

We found that the distribution of measured triplets follows closely a surface in the 3D space and that this surface is cone-shaped (as in Fig. 3). Figs. 4 and 5 give examples of 2D cross-sections through the cone and show how the measured triplets lie close to this cone. The extension of the cone into the 3D space was shown to be related to the wind speed, such that different locations along the cone can be related to different wind speeds. The opening of the cone is related to the anisotropy of the back-scattering, and so different locations around the cone surface can be identified with different wind directions. In fact, two closely overlapping cones exist; one for when the system measures "upwind" and one for "downwind". Conversely, using the transfer function for a range of speeds and directions to simulate the radar back-scatter for the three antennae, we find that these triplets in the 3D space span a cone-shaped surface. This surface should fit the distribution of measured triplets. We have verified that this is not the case for the transfer function proposed using pre-launch information. Further, the formulation of this function does not allow for some of the characteristics of the observed cone.

Occasionally, measured triplets of radar back-scattered power are not close to the general distribution of triplets. We have discovered that these anomalous measurements are found in areas where the wind is changing rapidly, e.g. close to tropical cyclones, lows and fronts. A quality control procedure has been set up to identify and eliminate these points.

New transfer function

To derive an empirical transfer function a set of collocated radar back-scatter measurements and surface winds is necessary. An estimation algorithm is then used to find the most probable set of coefficients defining the transfer function.

Naturally the higher the accuracy of the collocated wind data used, the better the quality of the resulting transfer function will be. ESA set up a calibration and validation campaign in the Norwegian Sea (close to Halbenbanken) to obtain an accurate set of wind data. Unfortunately, during the campaign only a limited range of wind speeds and directions were observed. The US (NOAA) buoys were also used for collocation, but then a long time span is needed to cover the full range of speeds and directions. At ECMWF analysis winds were used to collocate with the scatterometer measurements.

We filtered the winds to avoid mutual correlation of the winds and to reject cases where the winds were unsteady and therefore probably of lower quality. In a comparison exercise the transfer function derived at ECMWF was selected for implementation in ESA operations. At ECMWF we verified that the cone surface of this transfer function fitted the distribution of measured radar back-scatter triplets very well (see Figs. 4 and 5).

Inversion

Given a triplet of measured back-scattered power we may locate the position on the cone surface that is closest. We described earlier how each position on the cone surface is related to a wind speed and a wind direction. This mapping is defined by the transfer function. Thus, the identification of the closest position on the cone reveals a wind speed and direction solution. Because a closely overlapping "upwind" and "downwind" cone exists, two almost equally probable wind solutions will emerge.

Ambiguity removal

Only after resolving the dual ambiguity at each node we do have a resulting wind field from the scatterometer. Ambiguity removal schemes based on the scatterometer radar power triplets only (autonomons) were attempted, but these are not successful since both solutions are almost equally probable. Using a forecast wind field, 95% of the ambiguities can be removed successfully. Here, the quality of the forecast strongly determines the ambiguity removal skill. We developed a scheme that corrects wrongly selected solutions (5%), due to a wrong forecast. The scheme attempts to select the field with the highest wind vector consistency. Areas where the forecast wind vector is close to one of the two scatterometer wind vector solutions, and where the scatterometer wind

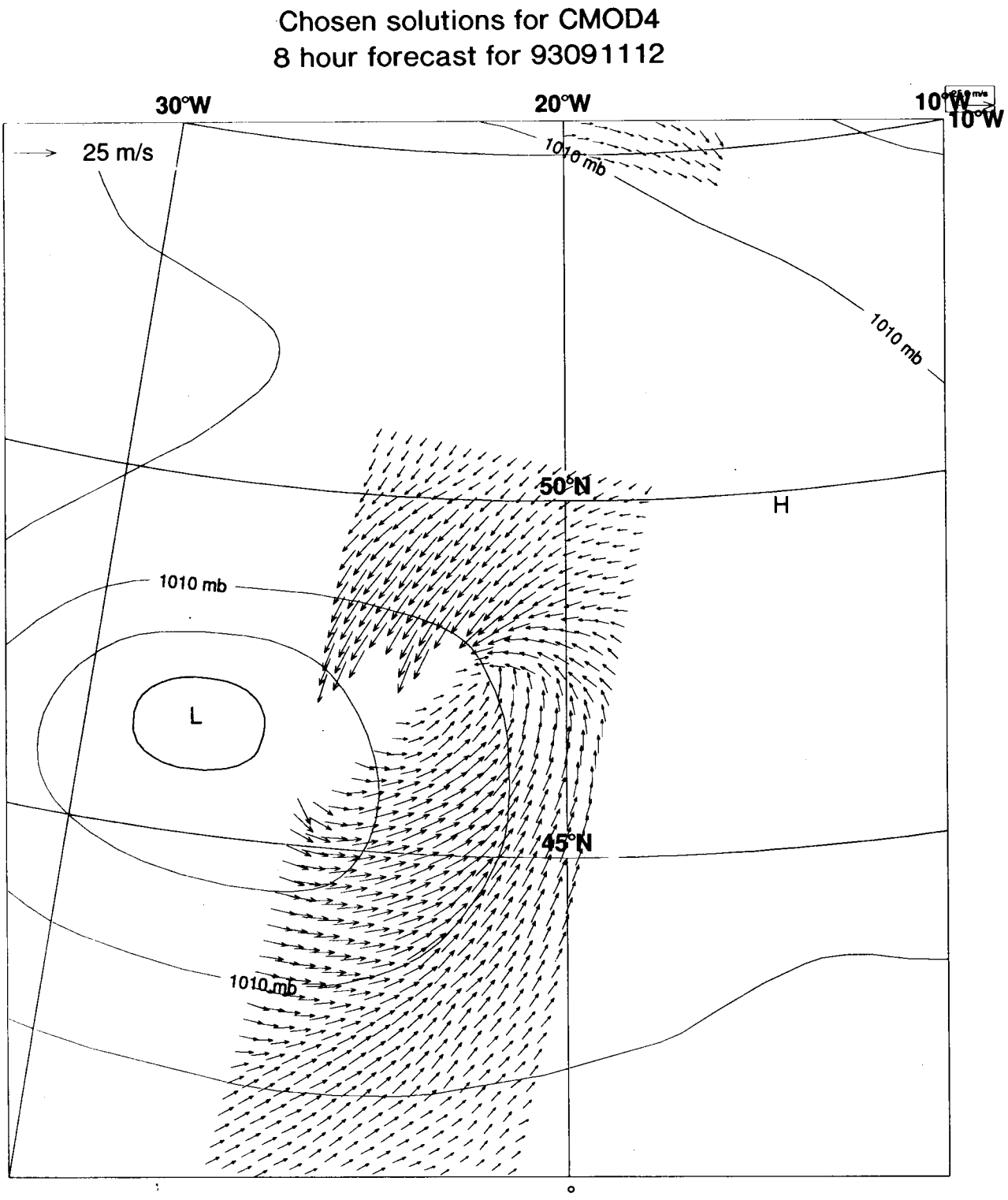


Fig. 6: ERS-1 scatterometer winds (PRESCAT) around the remainder of tropical storm Floyd, 36 hours before it hit the European continent with an unexpected amount of precipitation. The ECMWF 6 hour forecast of mean sea level pressure is shown for comparison.

directions are relatively accurate, are initially given high confidence. The filter propagates information from these high confidence areas to areas where the confidence in the selected solution is lower.

The scheme, called PRESCAT, has considerable skill, but in areas where the forecast is of very low quality and forecast directions are wrong by nearly 180° problems may still occur. An example of the scatterometer winds retrieved with PRESCAT is shown in Fig. 6 for a case where the remains of tropical storm Floyd were "seen" by the scatterometer.

Quality of retrieved winds

PRESCAT winds were verified against the ECMWF 6 hour forecasts. Similar verifications are done routinely for SHIP winds. We found to our surprise that the scatterometer winds generally compare better to the ECMWF forecast than SHIP winds (see Fig. 7). One of the main reasons for this concerns the so-called representativeness error. To use an observation for assimilation into the analysis, we would like it to represent spatial and temporal scales that can be resolved by the analysis. The current ECMWF model represents spatial scales down to roughly 200 km and temporal scales of roughly 30 minutes. For the ERS scatterometers, the corresponding scales are typically 50 km and 5 minutes, but for conventional wind data these are one metre and 10 minutes. The turbulence on scales between one metre and 200 km (with generally a vector variance in the wind of ~ 2 m/s) will make a substantial contribution to the difference between SHIP and short range forecast. It also follows from this that SHIPs will report extreme winds more often than the scatterometer or the ECMWF model will do.

One problem affecting remotely sensed data is the potential for horizontally correlated error. This will render the data less useful. For scatterometer winds we verify that no substantial horizontal error correlation is present. As a result, spatial wind spectra made from scatterometer retrieved wind fields compare favourably with spectra obtained from conventional anemometers.

Assimilation

Given the quality of the PRESCAT scatterometer winds it seems worthwhile to assimilate them into the ECMWF model. We have explored both the assimilation into the current "optimal" interpolation (OI) analysis scheme, and into the 3D- and 4D-variational schemes (3D-VAR and 4D-VAR).

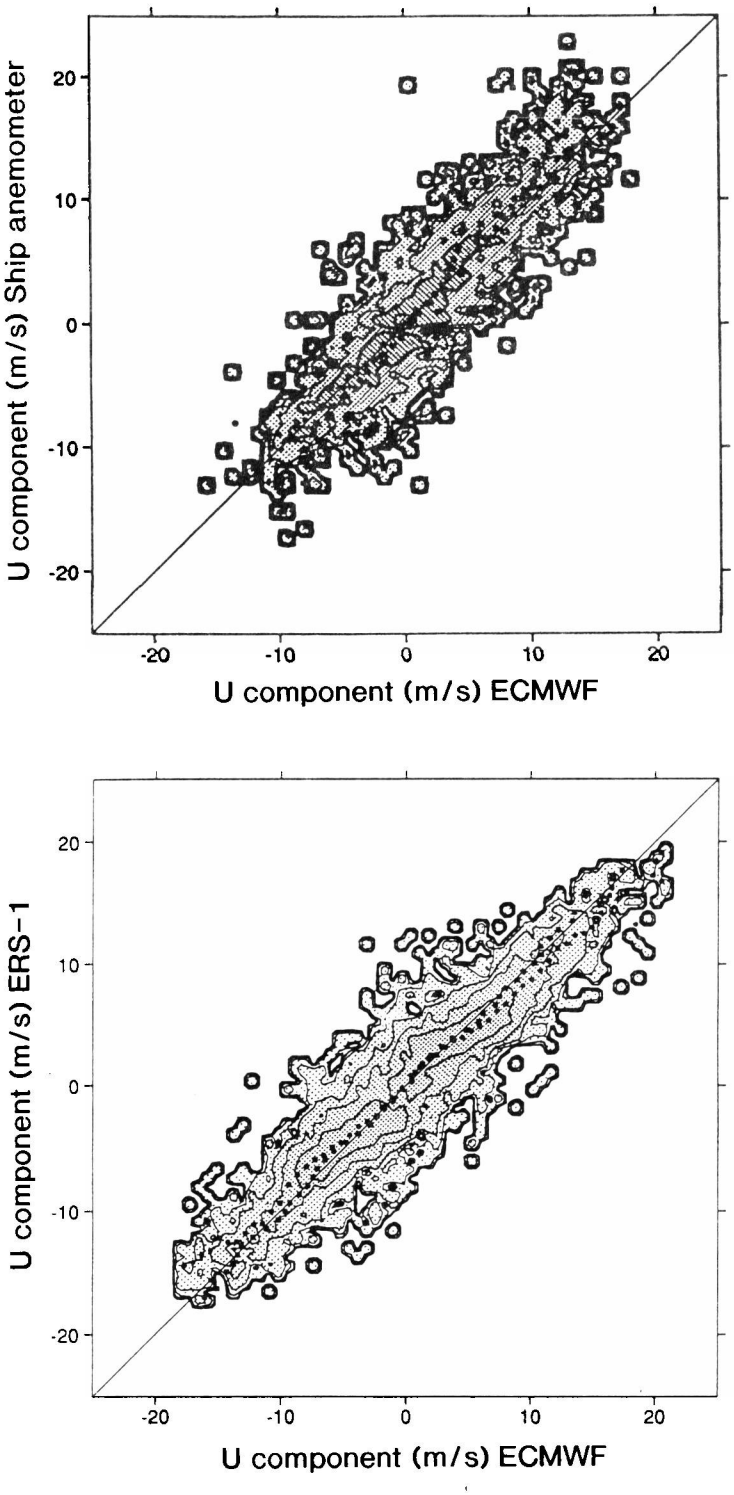


Fig. 7: a) **Verification of ship anemometer winds against the ECMWF forecast (3 to 9 hours forecast lead).**

b) **As a), but for scatterometer winds in the middle of the swath (at node 11).**

OI

The OI problem of the assimilation of scatterometer winds is similar to the assimilation of conventional surface wind data. However, a greater impact is expected since the scatterometer data are more numerous and spatially consistent.

We did indeed find that the scatterometer has an impact on the analyses in the Southern Hemisphere. In the Northern Hemisphere however, the effect of the scatterometer on the weather analyses is less. The changes in the Southern Hemisphere were verified to be beneficial. Also the short range wind forecasts (up to 12 hours) were found to be improved.

Scatterometer winds did not improve the medium range forecasts at ECMWF. In fact, they were found to be redundant with satellite temperature soundings. In data assimilation schemes assumptions are made on the structure of error of the forecast. This means for example that if the forecast wind at a certain location is found to be wrong, then it is assumed that the forecast wind, pressure and temperature are also wrong in an area above, below and around this location. So, when we have only temperature data, then we will improve on the temperature fields of the forecast, and have an accurate temperature analysis. However, because of our assumptions on forecast error structure, we will also adapt the forecast wind field. In verification of the operational surface wind analyses and forecasts using scatterometer winds, it was found that the changes to the forecast surface wind field through the use of upper air data are not always beneficial. This implies that our assumptions on the structure of the forecast error are not adequate. The most limiting assumption is probably that the forecast error structure is assumed to be constant and independent of the meteorological conditions. In 4D-VAR experiments the weakness of this approximation has already been shown. It offers an explanation for the neutral effect of the scatterometer winds on the medium range forecast skill and the redundancy between scatterometer and satellite sounding data.

3D- and 4D-VAR

In a variational data assimilation scheme a penalty function is minimised. At the minimum the most likely "true" meteorological state of the atmosphere is found. This state will be a compromise between a short range forecast (6 hour forecast in case of 3D-VAR) and the observations. In order to compute this compromise properly we need to know what the probability of an observed wind vector is, given the "true" wind vector, in the case of a wind measurement. In the case of conventional wind data this probability simply depends on experience of how well the observational system fits the 6 hour forecast wind in general. Similarly, for scatterometer data it has also been found that the observation error can easily be described using the components of the wind. In terms of back-scattered power however the observation error structure is very complicated. This is due to the complex (non-linear) wind-to-radar relationship, embodied in the transfer function, and the relatively small detection error of the measured radar power. Therefore, also in 3D- and in 4D-VAR, scatterometer data are most easily treated like other surface wind data.

By defining an observation cost function with two minima the ambiguity of scatterometer winds can be taken into account. Meteorological balance constraints on the resulting analysis will then be taken into account for the selection of the proper wind direction solution. This should result in a better ambiguity removal than the constraint of wind vector consistency currently used in PRESCAT. In this respect the first results from 3D-VAR look promising.

It is expected that the best use of surface wind data can be made within 4D-VAR. Scatterometer information will then effect the atmosphere in a dynamically consistent manner that depends on meteorological conditions.

- Ad Stoffelen, ECMWF
(current affiliation - KNMI)

* * * * * * * * * * * *

THE POSIX SHELL UNDER UNICOS 8

Introduction

With UNICOS 8 the standard command and programming language shell is the POSIX shell, i.e. **/bin/sh** is the POSIX shell, not the Bourne shell. The POSIX shell is functionally equivalent to the Korn shell, so much so that at UNICOS 8 they are one and the same binary i.e. file ***/bin/ksh*** is a link to file ***/bin/sh***, and both contain the binary for the POSIX shell. The Bourne shell is the default shell under UNICOS 7, where file ***/bin/ksh*** contains the Korn shell binary and file ***/bin/sh*** contains the Bourne shell binary. All Bourne shell scripts should work unchanged under the POSIX shell, since the features of this shell are a superset of those of the Bourne shell.

In the past users have generally used the Bourne shell for Batch work and the C shell for Interactive work. This is because the Bourne shell, on UNICOS at least, is more efficient than the C shell and allows redirection of the standard error file. The C shell with its command history and "alias" capabilities is more suited to interactive use. The Korn shell attempts to incorporate, and in many ways improve upon, some of the useful C shell features, while still allowing Bourne shell scripts to execute unchanged.

This article will mainly concentrate on the new features of the POSIX shell over and above those provided by the Bourne shell. These include: command history, command editing, tilde substitution, aliases, exporting of aliases and functions, the "ENV" file, the ability to do integer arithmetic and to use array variables, regular expressions and pattern-matching operators, the "select" control structure, and extra shell variables and special built-in commands.

Command History

The POSIX shell, like the C shell, keeps a record of the most recently typed commands. The number of interactive commands recorded is determined by the shell variable **HISTSIZE**, which, if not set, defaults to 128. A record of the commands is kept in a file determined by the shell variable **HISTFILE**, which if not set, defaults to **$HOME/.sh_history**, though on the Cray computers at ECMWF this variable has been set to **$TMPDIR/.sh_history**. To list the last 16 commands input to the shell all you need to type is:

history (or **h**, both are aliases for **fc -l**)

To see more history, just supply the amount as a negative number, e.g.

h -30

Command Editing

On the Cray computers at ECMWF the shell variable **VISUAL** has been set to **/usr/bin/vi**. This means that while typing an interactive command it is possible to correct errors by using the POSIX shell's built-in vi-like editor (there is also a built-in emacs-like editor, though this will not be described here). To enter the editor command mode you need to press the ***ESCAPE*** key. You can then use many of the normal "vi" commands to correct the line of text, and the press the ***RETURN*** key to execute the modified command. The "vi" commands include:

[*count*]**h**	to position 1 (or *count*) character(s) to the left
[*count*]**l**	to position 1 (or *count*) character(s) to the right
i	to insert text; press the ***ESCAPE*** key to terminate insert mode
[*count*]**x**	to delete 1 (or *count*) character(s)
[*count*]**r**	to replace 1 (or *count*) character(s)
cw	to replace a word
u	to undo the last modification
U	to undo all modifications
*	to do filename generation
\	to do filename completion

While in editor command mode it is possible to abandon the command being typed, and edit any of the commands in the history file. This is done again by using standard "vi" commands, which include:

[*count*]**k**	to fetch the previous (*count*) command
[*count*]-	same as [*count*]**k**
[*count*]**j**	to fetch the next (*count*) command
[*count*]+	same as [*count*]**j**
[*count*]**G**	to fetch command number *count*. If *count* is not specified then it defaults to the least recent history command, unlike in "vi" where G means the last line in the file.
/string	to search backward in the history file for a previous command containing *string* (less recent). *string* is terminated by the ***RETURN*** key. If *string* is preceded by a caret '^' then the matched command must begin with *string*.
?string	to search forward in the history file (more recent).

Another way of editing and re-executing commands from the history file is to use the built-in command "**fc**". An alias is available, called "**r**" for "repeat", which uses "fc". Some examples should make it easy to see how to use it:

r	this will repeat the previous command
r 122	this will repeat command number 122
r cf	this will repeat the last command beginning with the string "cf"
r abc=xyz	this will repeat the last command, replacing the string "abc" within the command by the string "xyz"
r lib="-l lib" cc	this will repeat the last command beginning with the string "cc", replacing the string "lib" within the command by the string "-l lib"

Tilde "~" Substitution

Like the C shell, the POSIX shell allows you to specify a user's $HOME directory by using the string "***~userid***". If *userid* is not a valid user identifier then no substitution is performed. A ~ by itself or in front of a / is interpreted as the current user's $HOME directory. A ~ followed by a + or - is interpreted as $PWD and $OLDPWD, respectively.

Aliases

POSIX shell aliases are similar to, but different from, C shell aliases. The syntax for defining them is different and they do not allow arguments. Aliases can be exported to child processes and you can also set up "tracked" aliases, this being the POSIX shell equivalent of the C shell hashing mechanism. The format for defining an alias is:

alias [-t] [-x] ***name=command***

The '-x' makes this an exported alias, and the "-t" makes it a tracked alias. Without the "*name=command*" portion the command lists the aliases currently in effect.

Aliases are frequently used as short forms for full path names. A tracked alias has as part of its initial value the full path name of the command as defined by the PATH variable and becomes undefined each time the PATH variable is reset. The alias is tracked so that the next subsequent reference will redefine the value. Aliases can be removed by using the special command:

unalias ***name***

The following example shows how to set up and list some aliases:

```
alias -x ls="ls -l "
alias
      ls=ls -l
alias -t dir="ls -CF "
alias
      dir=ls -CF
      ls=ls -l
alias -x
      ls=ls -l
alias -t
      ls=/bin/ls
PATH=$PATH:$HOME/bin
alias -t
      none
```

Exporting Aliases and Functions and the "ENV" File

As mentioned above, aliases may be exported to POSIX shell child processes by defining them with the "-x" option specified. Functions may be exported by defining them and then using the "-f -x" options of the special command "typeset" e.g.:

```
cdls( )
{
      cd     $1
      ls -la $1
}
typeset  -fx cdls
```

The "typeset" special command has many uses and will be described later in this article. The first line of the above function definition "**cdls( )**" is similar to the way the C language defines a function; it could be replaced by the more Fortran-like syntax "**function cdls**", which is equivalent.

Unless specified, aliases and functions, as with normal shell variables, are not exported to child processes. To explore this further we will explain the steps which are followed when the POSIX shell is invoked and shell scripts are executed.

If your login shell is set to the POSIX shell, or if you run a batch job under the default (POSIX) shell, then the following sequence of events takes place.

The file **/etc/profile** is read by the shell and the commands within this file are executed. Next the file **$HOME/.profile** is read and the commands executed. Usually $HOME/.profile has in it a reference to the file **$HOME/.user_profile**. This file is also read and its commands executed. After this the POSIX shell then checks the **ENV** shell variable, which by default on the ECMWF Cray computers is set to **/etc/sh_env**. If it is set and the file it references is readable then the shell executes the commands stored in the file.

/etc/profile is set up by ECMWF systems staff and contains commands to set up such global shell variables as TERM - terminal type, DISPLAY - display console name, ECLIB - the location of the eclib library, TEMP - the name of the user's directory in /tmp etc.

$HOME/.profile is owned by the user and therefore can be modified by him. However in general it should not be modified, as the next file is available for that purpose.

$HOME/.user_profile is owned by the user and can be tailored by him to contain any commands he wishes to execute automatically at login time or at the beginning of batch job execution. This may be used to set up other global shell variables, modify the PATH parameter, or set up private aliases and shell functions, though these are better done in an "ENV" file.

/etc/sh_env is set up by ECMWF systems staff and this is the value of the ENV shell variable. The file contains alias and function definitions which are deemed to be useful for all users. There is a reference within this file to the file **$HOME/.sh_env**. If this file exists and is readable then the commands within it are also executed.

$HOME/.sh_env is owned by the user and can be tailored by him to contain any private aliases and functions or global shell variables.

As can be seen there are quite a few files to read and execute. The "ENV" files would appear to be surplus to requirements, but these files, /etc/sh_env and $HOME/.sh_env are special in that not only are they read at login or start of job time, but they are also read and the commands executed whenever a new shell is invoked, whereas the various "profile" files are only read once. To see how this matters we need to look at the steps involved when a new shell is invoked and when a child shell is invoked.

If a script is executed by name then it normally runs as a forked process of the current shell and all exported variables, aliases and functions are available to this forked process. If the the script is run via the "sh" (or "ksh") command, or if it has execute but not read permission, or if it has a line saying "#! /bin/sh" (or "/! /bin/ksh") as its first record, then a new shell (not a forked shell process) is started and although the exported variables are available, the exported aliases and functions are not. However in this case, commands in the file specified by the ENV variable are executed when the new shell starts up. From this one can see that it is best to define exported

aliases and functions in the file $HOME/.sh_env, rather than in $HOME/.user_profile, so that they are available to all subsequent scripts, not just those executed by forked shell processes.

If you have a lot of shell functions, putting them in the $HOME/.sh_env file becomes unwieldy. It may also affect the time it takes to start new shells, as the file has to be read each time and the functions loaded. The command "**autoload *function***" circumvents these problems. "autoload" is a standard alias for the ubiquitous "typeset" special command, the alias being "autoload='typeset -fu'". This tells the shell that the function exists but is undefined. When the POSIX shell encounters a reference to this function it then searches the set of directories specified by the **FPATH** environment variable (whose format is the same as PATH), for a file with the same name as the function. This function is then loaded into the shell. If, for example, your FPATH variable is set up in your $HOME/.user_profile as:

FPATH="$HOME/sh_functions"

Then the first reference to any function specified on an "autoload" command will cause the shell to look into the directory $HOME/sh_functions to see if a file with the same name as the function exists. The function definition will then be loaded into the shell and will be used directly from the shell from then on. Initially this variable FPATH is set in /etc/sh_env to:

FPATH="/usr/local/bin/sh_functions"

This directory contains some functions which are of general use and which are defined as "autoload" functions in /etc/sh_env also. If you want to add your own functions, it is suggested that you put each function definition in a separate file in a directory called $HOME/sh_functions, define each function as an "autoload" function (or if you wish them to be exported use: **autoload -x**) in your $HOME/.sh_env file, and define the FPATH variable in your $HOME/.user_profile file as

FPATH="$FPATH:$HOME/sh_functions"

To see what functions are defined use the **functions** command (an alias for typeset -f), and to delete a function definition use **unset -f *function***. This will erase the function from the shell's memory; however, autoloaded functions are still remembered and any reference to one will cause it to be reloaded.

Functions help to simplify scripts by making them more modular. Since they reside in the shell's memory (however see the description of autoloaded functions above), they execute quickly and do not run as a separate process, which is the case with script files. A function which has the same name as a script or executable binary takes precedence over these. The command precedence of the POSIX shell is:

Keywords such as "if", "while", "else" etc.

Aliases

Built-in special commands such as "cd", "print", "set" etc.

Functions

Scripts and executable binaries

Integer Arithmetic and Array Variables

The ability to do integer arithmetic is provided by the POSIX shell with the special command "**let**". Constants are of the form [*base#*]*n*, where *base* is a decimal number between 2 and 36 representing the arithmetic base and *n* is a number in that base. *base#* is optional and if omitted then base 10 is assumed, e.g. 16#1f is decimal 31 and 2#111 is 7.

An arithmetic expression uses the same syntax, precedence and associativity of expression as in the C language. All integral operators, other than **++, - -, ?:** and **,** are supported, including **+, -, *, /, %, >>, <<, !, ~, &, |, ^, &&, | |, >, <, >=, <=, !=, ==, +=, -=, *=, /=, %=**. Named parameters can be referenced by name within an expression without using the parameter substitution (**$**) syntax. When a named parameter is referenced, its value is evaluated as an arithmetic expression, unless it has been specified to be of integer type using the "-i" parameter of the "typeset" special command.

Because many of the arithmetic operators need quotation marks, to prevent them from being interpreted by the shell, an alternative form of **let " .... "** is available, in the form of **((....))**. The result of an arithmetic expression can be obtained using the form **$((....))**. Do not confuse this with ***$(command)***, which is analagous to using back-quotes (`) to substitute the output produced by running *command*. One possible problem with using "let" or "(....)", which also exists with the **expr** command, is that an expression which evaluates to 0 (zero) sets the shell status variable "**$?**" to a non-zero value, which will abort the shell if the "**set -e**" command has been used. This problem does not occur with the "$((...)) construct. Some examples should make this clearer.

```
# Create a function to see if a value is an odd number
# ----------------------------------------------------
is_odd()
{
# Return TRUE status (0) if argument is an odd number
      ((($1 % 2) == 1))
}

# Show different ways of setting c to 4 and printing
# --------------------------------------------------
expand="(a + b)"
typeset -i a=8#12
b=2
let "c = expand % 8"
echo $c
      4
((c = expand % 8))
echo $c
      4
c=$((expand % 8))
echo $c
      4
echo $((expand % 8))
      4

# Show how status depends on value and construct
# ----------------------------------------------
((n = 1 + 1)) && print "n = $n, status = $?"
      n = 2, status = 0
((n = 1 - 1)) || print "n = $n, status = $?"
      n = 0, status = 1
n=$((1 - 1)) && print "n = $n, status = $?"
      n = 0, status = 0

# Obtain the time and print minutes since midnight
# ------------------------------------------------
hr=$(date +%H)
mn=$(date +%M)
sc=$(date +%S)
secs=$((sc + 60 * ( mn + 60 * hr )))
print $hr $mn $sc $secs
      15 21 04 55264
```

The POSIX shell supports the use of one dimensional array variables. Array subscripts can be constants, variables or expressions in the range 0 to 1023. Arrays need not be declared. Whole arrays can be assigned values using the "-A" parameter of the special command "set", while individual elements can be set as normal. Any reference to a named parameter with a valid subscript (0 - 1023) is legal and an array will be created if necessary. Referencing an array without a subscript is equivalent to referencing element 0 of the array. For example, exporting **array** is equivalent to exporting **array[0]**, i.e. only the first element, not all elements of the array. To reference an array element, other than in an expression, it is necessary to use the construct **${*array*[*element*]}**. The construct **${*array*[*]}** or **${*array*[@]}** gives the value of each element separated by a space. The construct **${#*array*[*]}** gives the number of valid elements of *array*. A strange feature of POSIX shell arrays is that if an array element has not been assigned, then it does not exist. If this value were, for example, 4, then this would not necessarily mean that elements 0, 1, 2 and 3 of the array existed, just that 4 elements in the range 0 to 1023 existed. Normally a script would be written to use consecutive elements, starting at 0. Some examples should make this clearer:

```
# Run 4 resolutions with associated "n" and "s" options
# ----------------------------------------------------
set -A nvals 1 22 53 75
set -A svals 7 16 31 96
set -A resolution T21 T63 T106 T213
code[0]="OK - no problem"
code[1]="ERROR - correct and rerun"

n=0
while [ $n -lt ${#resolution[*]} ]
do
      ./run_spm -n ${nvals[$n]} -s ${svals[$n]} ${resolution[$n]}
      echo ${code[$?]}
      ((n += 1))
done

# Create a 10 element array of random numbers in range 0 - 31
# -----------------------------------------------------------
RANDOM=$$               # set the random number generator seed
integer i=0             # integer is an alias for "typeset -i "
while ((i < 10))        # note - arithmetic expression used
do
      ((ran[i] = RANDOM % 32))
      print "Random number $i = ${ran[$i]}"
      ((i += 1))
done
```

```
# Create a 3 element array with "funny" element numbers
# ------------------------------------------------------
odd_val[3]="value 0"
odd_val[13]="value 13"
odd_val[23]="value 23"
echo "number of elements of odd_value = ${#odd_value[*]}"
      number of elements of odd_value = 3
echo "number of characters in element 13 = ${#odd_val[13]}"
      number of characters in element 13 = 8
echo "all = ${odd_val[@]}"    # @ is like * but preserves spaces
      all = value 0 value 13 value 23
```

Regular Expressions and Pattern-matching Operators

The POSIX shell, like the Bourne shell has the concept of "wildcard" characters, (*, ? and [ ]). However the POSIX shell goes much further, in that it allows **regular expressions** to be used. This gives it very powerful string matching capabilities, similar to those of UNIX utilities such as **awk(1)**, **egrep(1)**, **sed(1)** and others, although the syntax is different. In fact since the shell allows *extended* regular expressions, it is more powerful in this respect than "sed", which only allows *normal* regular expressions. Regular expressions can be difficult to understand and it is beyond the scope of this article to explain them. The books mentioned at the end give detailed explanations. A few simple examples are given here, just to show what may be done:

```
# Return a good status (0) if the argument contains only digits
# -------------------------------------------------------------
isnumber()
{
      [[ $1 = +([0-9]) ]]
}

# Return a good status (0) if the argument contains only letters
# --------------------------------------------------------------
isalpha()          # Return TRUE status if argument is a word
{
      [[ $1 = +([a-zA-Z]) ]]
}

read data
if isnumber  $data ; then
      print  $data is a number
```

```
elif isalpha $data ; then
      if [[  $data = [A-Z]* ]] ; then
            print "$data begins with an upper case letter"
      else
            print "$data begins with a lower case letter"
      fi
else
      print "$data is neither a number nor a word"
fi
```

Notice in the above example that the test format used above is "**if [[*condition*]]**", rather than the old format "**if [*condition*]**". This new POSIX shell test format allows regular expressions to be used on the right-hand side. The old format does not allow regular expressions, but allows for Bourne shell compatibility.

There are a set of pattern-matching operators **#**, **##**, **%** and **%%**,which allow patterns to be stripped from strings, contained within shell variables. The classic use of these are to strip off components of pathnames. The operator **${*variable#pattern*}** will, if *pattern* matches the beginning of the value of *variable*, delete the shortest part that matches and return the rest. Operator **${*variable##pattern*}** is similar, but this time deletes the longest part that matches. **${*variable%pattern*}** and **${*variable%%pattern*}** are also similar, but this time the match is at the end of *variable*'s value not the beginning. Again, to learn more in detail about these consult one of the books listed below. The following are examples of how they can be used:

```
# Obtain the last element of a pathname (i.e. the filename)
# ----------------------------------------------------------
basename()                    # see basename(1)
{
      print ${1##*/}          # to remove all but the last element
}

# Obtain the directory part of a file's pathname
# ----------------------------------------------
dirname()                     # see dirname(1)
{
      print ${1%/*}           # to remove last element only
}

# For Fortran (.f) or C (.c) files return the object name (.o)
# ------------------------------------------------------------
objname()
{
```

```
        print ${1%@(.f|.c)}.o   # to produce an object filename
}

SCRIPT=$(basename $0)
for file in *.f *.c
do
        if [[ $file = *.f ]] ; then
                cf77 $file
        else
                cc   $file
        fi
        print "$SCRIPT: SRC = $file, OBJ = $(objname $file)"
done
```

Notice in this example that the "*.f" in the "for" statement expands to *all filenames in the current working directory which have a ".f" suffix*". However the "*.f" in the "if" statement does not. It is a regular expression and means *any pattern that ends with a ".f" suffix.*

The "select" Control Structure

The "select" control structure provides a simple method for creating menus for interactive scripts. The shell sends menu items to the standard error file and prompts the user for a selection. The structure is as follows:

```
select identifier in words
do
        commands
done
```

Typically the *words* are the names of the menu items. Select presents each *word* (which can be a phrase) preceded by a menu number and then prompts the user for input. The prompt used is defined by shell variable PS3, which defaults to "#?" if not set. If the user enters a number corresponding to one of the menu items, the POSIX shell sets *identifier* to the *word* corresponding to that item number. If the user presses the *RETURN* key without typing anything, the menu and prompt are displayed again. If the user types something other than a menu number, the *identifier* is set to null. The user's input is saved in the shell variable **REPLY** so that if the user entered an invalid response it is possible to find out what was typed.

A simple example should make this clear:

```
PS3="Enter your selection: "

echo "        COMMAND MENU \n"

select choice in "Date & Time" "Users Logged-in" "Files" "Exit"
do
      case $choice in
            "Date & Time")
                        date
                        ;;
            "Users Logged-in")
                        who
                        ;;
            "Files")
                        ls
                        ;;
            "Exit")
                        print "End of selection menu"
                        exit
                        ;;
            *)
                        echo "Invalid selection: $REPLY"
                        ;;
      esac
done
```

Executing this produces the following:

```
                COMMAND MENU

          1) Date & Time
          2) Users Logged-in
          3) Files
          4) Exit Menu
          Enter your selection: 1
          Tue May 10 07:43:33 GMT 1994
          Enter your selection: 4
          End of selection menu
```

New Shell Variables and Built-in Commands

The POSIX shell has several new variables that can be used in scripts, including:

ERRNO	The value of *errno* as set by the most recently failed system call.
LINENO	The number of the current line within the script or function being executed.
OLDPWD	The previous working directory set up by the *cd* command.
RANDOM	A random number, uniformly distributed between 0 and 32767.
REPLY	The input given to the *select* statement and to the *read* statement, when no arguments are supplied.
SECONDS	The number of seconds that have elapsed since the shell was invoked.

New built-in commands include:

cd [*arg*]
cd *old new*

The first form is the "normal" change directory command, if *arg* is "-" then the directory is changed to the previous directory.
In the the second form *cd* substitutes the string *new* for the string *old* in the current directory name and tries to change to this new directory.

fc [-e *ename*] [-n] [-l] [-r] [*first* [*last*]]
fc -e - [*old=new*] [*command*]

In the first form, a range of commands from *first* to *last* is selected from the last HISTSIZE commands that were typed at the terminal. The arguments *first* and *last* may be specified as a number or a string. A string is used to locate the most recent command beginning with the string. A negative number is used as an offset from the current command number. The "-l" flag causes the command(s) to be listed at the terminal. Otherwise the editor *ename* is invoked on a file containing these commands. If *ename* is not supplied then the value of FCEDIT is used. When editing is complete, the edited commands are executed. Flag "-r" reverses the order of the commands and "-n" omits the line numbers when listing them.

In the second form the *command*, which can be a string or a number, is re-executed after the substitution *old=new* is performed (see earlier).

print [-R] [-n] [-p] [-r] [-s] [-u [*n*]] [*arg* ....]

This is the shell output mechanism and can be used instead of **echo(1)**. For more information see the manual page for **ksh(1)**.

read [-p] [-r] [-s] [-u [*n*]] [*name?prompt*] [*name* ....]

This is the shell input mechanism. For more information see the manual page for **ksh(1).**

typeset [±H] [±L [*n*]] [±R [*n*]] [±Z [*n*]] [±f] [±i [*n*]] [±l] [±r] [±t] [±u] [±x] [*name*[=*value*]] ....

This sets attributes and values for shell variables.
Flag "-L" left justifies and removes leading blanks from *value*. If *n* is nonzero, it defines the width of the field, otherwise the first assignment determines the width. When *name* is assigned a value, it is filled with trailing blanks, or truncated if necessary. Leading zeros are removed if flag "-Z" is set.
Flag "-R" right-justifies the field.
Flag "-f" indicates that *name* is a function and no assignment may be specified. In this case the only other valid flags are "-t" to turn on tracing for the functions, "-u" to undefine (autoload) the function and "-x" to export the function.
Flag "-i" specifies that *name* is an integer of base *n*.
Flag "-l" converts all uppercase characters to lowercase.
Flag "-u" converts all lowercase characters to uppercase.
Flag "-r" specifies that *name* is read-only.
Flag "-x" marks the given *name(s)*, for automatic export to the environment of subsequently executed commands.
Using a "+" instead of a "-" turns the flag off.

The following examples demonstrates some of these new commands:

```
# Print (the number 12 in Decimal, Octal and Binary
# ------------------------------------------------
typeset -i8 octal_number
typeset -i2 binary_number=12
decimal_number=12
octal_number=$decimal_number
print "$decimal_number    $octal_number   $binary_number"
      12    8#14    2#1100

# Input a username, get the corresponding id and print both
# --------------------------------------------------------
typeset -R10 username
typeset -R6  id
typeset -u   upper
while :                 # Loop until "exit" is input.
do
      read uid?"Username? "
```

```
        username="$uid"
        upper="$uid"
        if [ "$upper" = "EXIT" ] ; then
              print "Finished"
              break
        fi
        id=$(grep "^$uid:" /etc/passwd | cut -f3 -d:)
        print "$username $id"
done

        Username? abc
               abc    123
        Username? vwxyz
             vwxyz   1370
        Username? Exit
        Finished
```

Conclusion

The POSIX shell has many new features over and above those of the Bourne shell. It contains features which make it more suitable for interactive use than the C shell. In essence there is now one shell which is suited both to interactive and batch use and which will continue to run existing Bourne shell scripts. However it is not downwardly compatible with the Bourne shell and any use in a script of the new features described here will make that script unusable under the Bourne shell, unless, of course, the system posesses a Korn shell, in which case you should be able to prefix the script with a record stating "#! /bin/ksh", causing the Bourne shell to start a Korn shell to execute the script.

UNICOS 8 is running on the YMP-EL at ECMWF. It will also be the operating system running on the YMP-2E/T3D machine and will eventually become the production system on the C90.

To use the POSIX shell as described here you should copy the file "**/usr/local/src/public/profile**" to the file "**$HOME/.profile**" and tailor "**$HOME/.user_profile**" and "**$HOME/.sh_env**" to your requirements. You can print the /etc/profile and /etc/sh_env files to see what has been set up by default and you can print any of the files in /usr/local/bin/sh_functions to see what functions are available. Under UNICOS 8 to make the POSIX shell your default interactive shell you should use the command:

chsh $USER /bin/sh

Although this article has concentrated on the POSIX shell under UNICOS 8, most of this information also applies to the Korn shell, "**/bin/ksh**", under UNICOS 7 and under the IRIX operating system on the Silicon Graphics workstations at ECMWF. The Korn shell is not available on the SUN workstations at ECMWF.

One slight problem with the Korn shell under UNICOS 7 is that it does not support the "-S" flag of the "set" special command, whereas the POSIX shell under UNICOS 8 does.

The purpose of this article is to show some of the new features that can be used with the POSIX shell. It is not intended as a reference manual. To obtain further information, read the "man" page for **ksh(1)**, or better still, read one of the books describing the Korn shell such as:

"The Korn Shell User & Programming Manual" by Anatole Olczak (Addison-Wesley)
"The Kornshell Command & Programming Language" by Morris I. Bolsky & David G. Korn
"Learning the Korn Shell" by Bill Rosenblatt (O'Reilly & Associates)
"A Practical Guide to Unix System V" by Mark G. Sobell (Benjamin/Cummings)

- Neil Storer

* * * * * * * * * * * *

The following article is reprinted here by courtesy of SCD Computing News, January/February 1993 issue. Jeanne Adams chaired the International Programming Languages Committee of the ISO (International Standards Organisation) and chaired the ANSI Committee which developed Fortran 90. She also co-authored the books "The Fortran 90 Handbook: Complete ANSI/ISO Reference", and "Fortran 90 Handbook".

COMPARING POINTERS IN CRAY FORTRAN AND FORTRAN 90

Editor's note: This is the eleventh and last in a series of articles about Fortran 90 features.

Most scientific programs involve large amounts of data that the programmer is called upon to manage. If space is a problem, and it often is, using space efficiently becomes important. A pointer facility makes it possible to do this with efficiency and clarity. Versions of Fortran at most installations running large data codes have some form of pointer and a set of related memory-management functions as extensions to FORTRAN 77; the Cray pointer facility is one of these extensions. A pointer facility is also included in the Fortran 90 standard.

A pointer facility makes it possible to use space efficiently.

However, Cray pointers are *not* like Fortran 90 pointers. While converting from Cray pointers to Fortran 90 pointers is possible, you will have to modify your existing programs to conform with the Fortran 90 standard. FORTRAN 77 did not have a pointer facility, and Cray pointers were added as an extension responding to user needs at the time.

Cray pointers are supported in version 5.0 of CF77, the Cray compiling system, and will be supported in Cray's Fortran 90 compiler (tentatively scheduled for release in early 1994). However, Cray pointers may not be available on new computer architectures. The Cray Fortran 90 compiler will support both Cray and Fortran 90 pointers. This will be possible because there is no ambiguity in having both types of pointers in the same compiler.

Fortran 90 pointers are not currently available in CF77, but are available on the IBM RS/6000 cluster and will be available on most computer architectures in the near future. The Fortran 90 pointer facility fits nicely into the new array features and extensions for declaring data in the new standard.

Because Cray pointers and Fortran 90 pointers are based on two different standards (FORTRAN 77 and Fortran 90), the syntax and semantics are different. In general, a Cray pointer is a new data type that does absolute addressing and address arithmetic. A Fortran 90 pointer is a data attribute and is a descriptor pointing to a named variable.

Cray pointers

The most common use of Cray pointers is to manage a workspace efficiently and conveniently. A Cray pointer is an absolute memory-addressing facility that points to or addresses various work areas of an array, making it possible to use another name for space in memory. The pointer and its target, the pointee, are declared in the following statement:

```
 POINTER (pointer,
$                pointee [(dimensions)])  [,...]
```

Examples might be:

```
 POINTER (P1, A),  (P2, B)
 POINTER (PX, X(1:40))
```

A, B, and X(1:40) are pointees - that is, targets pointed to by P1, P2, and PX. A pointee may be a variable name, an array declarator, or an array name.

Example 1 demonstrates a simple memory-management scheme for the array WORK. Simple pointer integer arithmetic changes the starting address of the array associated with the pointer, and therefore, the placement of the pointee in the array WORK. Example 2 shows output for the program in Example 1.

Example 1. A simple memory-management scheme for the array WORK

```
 1      PROGRAM CRAY_POINTERS
 2       COMMON WORK (20)
 3       REAL PNTEE (5,2), X (5,2), Y (5,2), WORK
 4 C       The type and shape of pointer targets PNTEE, X, Y established.
 5       POINTER (PTR, PNTEE), (PX, X), (PY, Y)
 6 C       PNTEE, X and Y are pointees; PTR, PX and PY are pointers.
 7 C
 8        N = 10
 9 C       Next make PTR point to the WORK array in Common; that is,
10 C       PNTEE (1,1) will be equivalenced to WORK(1), and effectively
11 C       assigning space. LOC is a Cray function to obtain the address
12 C       of the argument.
13       PTR = LOC(WORK)
14 C       Next make PX point to 10 address locations beyond LOC(WORK).
15       PX = PTR + N
16 C
17       PNTEE = 1.0
18       X = 2.0
19       PRINT "(A/,(5F5.1))", "The WORK array", WORK
20       PRINT "(A/,(5F5.1))", "Arrays PNTEE and X", PNTEE, X
21       PRINT "(A,2I10)", "Value of Pointers", PTR, PX
22 C
23       PTR = PX
24       PRINT "(A/,(5F5.1))", "The WORK array", WORK
25       PRINT "(A/,(5F5.1))", "Arrays PNTEE and X after PTR=PX", PNTEE, X
26       PRINT "(A,2I10)", "Value of Pointers", PTR, PX
27 C
28       PNTEE (1:5, 1) = 3.0
29       PY = PX - 5
30 C
31       PRINT "(A/,(5F5.1))", "The WORK array", WORK
32       PRINT "(A/,(5F5.1))", "Arrays PNTEE and X after PNTEE reset", PNTEE, X
33       PRINT "(A/,(5F5.1))", "The Y array", Y
34       PRINT "(A,3I10)", "Final Value of PTR, PX, and PY", PTR, PX, PY
35      END
```

Example 2. Output for the program in Example 1

```
The WORK array
  1.0   1.0   1.0   1.0   1.0
  1.0   1.0   1.0   1.0   1.0
  2.0   2.0   2.0   2.0   2.0
  2.0   2.0   2.0   2.0   2.0

Arrays PNTEE and X
  1.0   1.0   1.0   1.0   1.0
  1.0   1.0   1.0   1.0   1.0
  2.0   2.0   2.0   2.0   2.0
  2.0   2.0   2.0   2.0   2.0

Value of Pointers              85872          85882

The WORK array
  1.0   1.0   1.0   1.0   1.0
  1.0   1.0   1.0   1.0   1.0
  2.0   2.0   2.0   2.0   2.0
  2.0   2.0   2.0   2.0   2.0

Arrays PNTEE and X after PTR=PX
  2.0   2.0   2.0   2.0   2.0
  2.0   2.0   2.0   2.0   2.0
  2.0   2.0   2.0   2.0   2.0
  2.0   2.0   2.0   2.0   2.0

Value of Pointers              85882          85882

The WORK array
  1.0   1.0   1.0   1.0   1.0
  1.0   1.0   1.0   1.0   1.0
  3.0   3.0   3.0   3.0   3.0
  2.0   2.0   2.0   2.0   2.0

Arrays PNTEE and X after PNTEE reset
  3.0   3.0   3.0   3.0   3.0
  2.0   2.0   2.0   2.0   2.0
  3.0   3.0   3.0   3.0   3.0
  2.0   2.0   2.0   2.0   2.0

The Y array
  1.0   1.0   1.0   1.0   1.0
  3.0   3.0   3.0   3.0   3.0

Final Value of PTR, PX, and PY    85882          85882          85877
```

Fig. 1 illustrates the targets PNTEE and X in the work array WORK after line 18 in Example 1. Fig. 2 shows the targets PNTEE and X after the assignment statement PTR = PX (line 23 in Example 1). Fig. 3 shows the targets Y, PNTEE, and X after the assignment statement PY=PX-5 (line 29 in Example 1).

Figure 1

```
POINTERS    ADDRESSES   WORK SPACE IN MEMORY          NAMES OF POINTEES
-------------------------------------------------------------------------

                        WORK (20)
                        ---------------------
PTR         85872 ----> 1.0  1.0  1.0  1.0  1.0       PNTEE (5,2)
            85877       1.0  1.0  1.0  1.0  1.0
                        ---------------------
PX          85882 ----> 2.0  2.0  2.0  2.0  2.0       X (5,2)
            85887       2.0  2.0  2.0  2.0  2.0
                        ____________________
```

Figure 2

```
                        WORK (20)
                        ---------------------
            85872       1.0  1.0  1.0  1.0  1.0
            85877       1.0  1.0  1.0  1.0  1.0
                        ---------------------
PTR, PX     85882 ----> 2.0  2.0  2.0  2.0  2.0     PNTEE (5,2), X (5,2)
            85887       2.0  2.0  2.0  2.0  2.0
                        ____________________
```

Figure 3

```
                        WORK (20)
                        ---------------------
            85872       1.0  1.0  1.0  1.0  1.0
PY          85877 ----> 1.0  1.0  1.0  1.0  1.0     Y (5,2)
                        ---------------------
PTR, PX     85882       3.0  3.0  3.0  3.0  3.0     PNTEE (5,2), X (5,2)
            85887       2.0  2.0  2.0  2.0  2.0
                        ____________________
```

Cray pointers may point to variables of different types during execution. They are assigned absolute addresses, and while address arithmetic can be performed, there is no way to nullify the pointer status.

When porting code from the Cray Y-MP8/864 (shavano) to the cluster, note that the IBM RS/6000 is byte- and not work-oriented, so that in the program in Example 1, N should be 40 and not 10. If these pointers are used on the IBM compiler XLF, DO loops must be used instead of whole-array assignment.

Fortran 90 pointers

In Fortran 90, a pointer is a variable that has the POINTER attribute. A pointer may be associated with or aliased to various data objects (targets) during execution, or it may be undefined or null (not aliased to any data object). The pointer and the target attributes must be declared in the specification part of the program in order for data to be used in this way. The pointer and its target must be of the same type; the pointer must have the POINTER attribute, and the target data must have the TARGET attribute.

A Cray pointer is a data type; a Fortran 90 pointer is a data attribute.

For example:

```
REAL, TARGET :: POINTEE
REAL, POINTER :: PTR

PTR => POINTEE
```

The pointer PTR points to the target POINTEE. This is done with the pointer-assignment statement and the new pointer-assignment operator =>. The target may be a scalar, an array, or an array section.

A new statement, the ALLOCATE statement, creates space for variables with the POINTER attribute or for arrays with the ALLOCATABLE attribute. Space may subsequently be deallocated. For example:

```
REAL, POINTER :: P1(:)
ALLOCATE (P1(100))
...
...
DEALLOCATE (P1)
```

A pointer variable may be nullified - that is, cleared so that it does not point to anything. A NULLIFY statement removes the association of a pointer and a target. The pointer may then point to a different target later in the program. For example:

```
PRT => POINTEE
...
NULLIFY (PTR)
...
PTR => A
```

An important additional use of Fortran 90 pointers will be in processing linked lists.

Differences between Cray and Fortran 90 pointers

An informal report, "Fortran 90 Pointers vs. Cray Pointers" (Jeanne Martin, Lawrence Livermore National Laboratory, UCRL-ID 108534), provides additional insight into the differences between Cray and Fortran 90 pointers. The following discussion is taken from that report, with permission of the author.

Cray pointers and Fortran 90 pointers differ in the following ways:

There are two names associated with a Cray pointer: the name of the pointer and the name used to refer to the pointer target. There is only one name associated with a Fortran pointer. The interpretation of this name is determined by context; that is, certain statements can refer only to pointers, while others can refer only to target objects.

Cray pointers are memory locations. The pointer is treated as an integer, and the type of the target can change during execution. Fortran 90 pointers, on the other hand, are descriptors containing both type and rank information. They point to specific kinds of data objects; that is, Fortran 90 pointers are "strongly typed". For example, if a given pointer is declared to point to a two-dimensional real array, it can never point to any other kind of data object. It may point to several different arrays, and the dimensions may vary; but it can never point to a scalar integer object, for instance.

A pointer is an attribute in Fortran 90. This attribute may be given to any data object, including an object of user-defined type. In some implementaions of Cray pointers, it is not possible to point to objects of type CHARACTER. An implementer (that is, a writer of a Fortran 90 compiler) might find it equally difficult to extend Cray pointers to point to some of the new Fortran 90 data objects of user-defined type or nondefault kind. An implementer who added Cray pointers to a standard Fortran 90 compiler would have to decide whether to make the pointers pervasive throughout the language or to provide them only to aid code migration (that is, applying only to FORTRAN 77 features).

Cray pointers can be assigned absolute addresses; Fortran 90 pointers cannot.

A Fortran 90 pointer can be set to point to no object by a NULLIFY statement. There is no language-provided way to nullify a Cray pointer.

A compiler may assume that a Cray pointer target has no storage in common with another variable. This provides optimisation at the expense of possibly unreliably code. A Fortran 90 compiler assumes that pointers and objects with the TARGET attribute may overlap. This provides reliable code at the expense of some optimisations.

Note

Differences between the two types of pointers can cause significant problems when porting code from Cray pointers to Fortran 90 pointers; however, a discussion of these problems is beyond the scope of this article.

- Jeanne Adams

* * * * * * * * * * * *

NAG LIBRARY - MARK 16 NEWS

Introduction

Implementations of the NAG Library at Mark 16 are now starting to appear. These will gradually be made available at ECMWF over the coming months. As yet only the SUN implementation has been received. The following summary, provided by NAG Ltd., gives information at this Mark on the new features, changed algorithms, and routines that have been removed.

For more detailed information, including routines due for withdrawal at later Marks please see the entry in the ECMWF on-line help system "echelp". It is available in the file "NAG 16 preview" under "Local facilities -documentation".

New Features of Mark 16

Mark 16 represents a further considerable expansion of the NAG Fortran Library. It contains a total of 1134 documented routines, of which 126 are new at this Mark. Two new chapters have been introduced:

F08 - Least-squares and Eigenvalue Problems (LAPACK)
G10 - Smoothing

Of the 126 new routines, 72 are in the new chapter F08. This chapter includes routines to compute the solution of linear least-squares, symmetric eigenvalue, unsymmetric eigenvalue, singular value and generalized symmetric-definite eigenvalue problems. In addition there are routines to perform various matrix factorizations associated with the above problems, estimate condition numbers of eigenvalues and eigenvectors, estimate the rank of a matrix, and to solve the Sylvester matrix equation.

Twenty-two of the new routines are in the Statistics chapters. They include facilities (in the stated chapters) for:

- further statistical distribution functions (G01)

- cluster analysis (G03)

- improved analysis of variance routines (G04)

- further random number generators, including matrices (G05)

- computation of robust confidence intervals (G07)

- smoothing techniques (G10)

- two way contingency table analysis (G11)

- computation of partial autoregressive matrices (G13).

Coverage in the differential and integral equations chapters has been improved and extended with (in the stated chapters):

- new Runge-Kutta methods for the initial value problem in ordinary differential equations (D02)

- a Keller box discretization technique for first order partial differential equations (with coupled differential algebraic systems) in one spatial dimension and remeshing facilities (D03)

- routines to solve weakly singular nonlinear convolution Volterra-Abel equations (D05).

New, more robust, techniques have been incorporated in the optimization Chapter (E04) for the solution of linear programming and quadratic programming problems.

New black-box eigenvalue problem routines which exploit software in the new F08 (LAPACK) Chapter have been included in the eigenvalues and eigenvectors Chapter (F02).

Routines Revised at Mark 16

A new algorithm has been adopted in the Random Number Generator routines G05DHF, G05DJF and G05DKF. Results produced by these routines prior to this Mark will not be repeatable.

Routines withdrawn at Mark 16

The following routines have been withdrawn from the NAG Fortran Library at Mark 16. Warning of their withdrawal was included in the Mark 15 Library Manual, together with advice on which routines to use instead. The relevant Chapter Introduction documents give more detailed guidance.

Withdrawn Routine	Recommended Replacement
C02AEF	C02AGF
E02DBF	E02DEF
E04HBF	not needed except with E04JBF
E04JBF	E04UCF
E04KBF	E04UCF
F01ACF	F01ABF
F01BQF	F07GDF (SPPTRF/DPPTRF) or F07PDF (SSPTRF/DSPTRF)
F01CLF	F06YAF (SGEMM/DGEMM)
F02WAF	F02WEF
F04AQF	F07GEF (SPPTRS/DPPTRS) or F07PEF (SSPTRS/DSPTRS)
F06QGF	F06RAF, F06RCF and F06RJF
F06VGF	F06UAF, F06UCF and F06UJF
G01BAF	G01EBF
G01BBF	G01EDF
G01BCF	G01ECF
G01BDF	G01EEF
G01CAF	G01FBF
G01CBF	G01FDF
G01CCF	G01FCF
G01CDF	G01FEF
G02CJF	G02DAF and G02DGF
G05DGF	G05FFF
G05DLF	G05FEF
G05DMF	G05FEF
G08ABF	G08AGF
G08ADF	G08AHF, G08AKF and G08AJF
G08CAF	G08CBF
M01AJF	M01DAF, M01ZAF and M01CAF
M01AKF	M01DAF, M01ZAF and M01CAF
M01APF	M01CAF
X02AAF	X02AJF
X02ABF	X02AKF
X02ACF	X02ALF
X02AGF	X02AMF

- John Greenaway

* * * * * * * * * * * *

ECMWF ANNUAL SEMINAR

The ECMWF Annual Seminar for this year will be on "The Parametrization of Physical Processes in Models". It will be held at ECMWF from 5 to 9 September 1994. The seminar will deal with a wide range of problems associated with surface and boundary layer processes, radiation and cloud processes, cumulus convection and gravity wave drag. There will be a review of recent developments including the formulation of affordable representations of these processes, and the understanding of interactions both mutually and with basic model dynamics.

Posters and application forms have been mailed to the national meteorological services and major universities in the ECMWF Member States.

- Els Kooij-Connally

* * * * * * * * * * * *

SIXTH WORKSHOP ON USE OF PARALLEL PROCESSORS IN METEOROLOGY

21 - 25 NOVEMBER 1994

Every second year ECMWF organises a workshop on the use of parallel processing in meteorology. This year's workshop has the overall title "*Coming of Age - The Use of Parallel Processors in Meteorology*".

Once again we expect participants and speakers from the national weather services of our Member States and other meteorological centres, as well as from project teams working on this subject. Manufacturers of parallel supercomputers have been invited to present their latest developments.

Invited speakers at the workshop will comprise, amongst others, Tor Bloch from the Advanced Computer Research Institute at Lyon (France), Horst Förster from the Commission of the European Communities, Friedel Hossfeld from KFA Jülich (Germany), Jean-Pierre Peltier from ONERA, Paris, and Hans Zima from the University of Vienna.

It is planned that the proceedings of the workshop will be published by World Scientific Publishing Co.

- Norbert Kreitz

* * * * * * * * * * * *

ECMWF CALENDAR 1994

29 August	*ECMWF HOLIDAY*
5 - 9 September	Seminar - Parametrization of the physical processes in models
26 - 28 September	Scientific Advisory Committee, 23rd session
28 - 30 September	Technical Advisory Committee, 20th session
4 - 5 October	Finance Committee, 53rd session
25 - 26 October	Policy Advisory Committee, 3rd session
14 - 16 November	Workshop - Modelling and assimilation of clouds
21 - 25 November	Workshop - Parallel processing in meteorology
1 - 2 December	Council, 41st session
23 - 27 December	*ECMWF HOLIDAY*

* * * * * * * * * * * * *

ECMWF PUBLICATIONS

Forecast and Verification Charts to 31 May 1994

* * * * * * * * * * * * *

INDEX OF STILL VALID NEWSLETTER ARTICLES

This is an index of the major articles published in the ECMWF Newsletter series. As one goes back in time, some points in these articles may have been superseded. When in doubt, contact the author or User Support.

* * * * * * * * * * * * *

USEFUL NAMES AND 'PHONE NUMBERS WITHIN ECMWF

		Room*	Ext.**
DIRECTOR	- David Burridge	OB 202	2001
DEPUTY DIRECTOR and HEAD OF OPERATIONS DEPARTMENT	- Michel Jarraud	OB 010A	2003
ADVISORY: Available 9-12, 14-17 Monday to Friday			2801
Other methods of quick contact:	- Telefax (+44 734 869450) - VMS MAIL addressed to ADVISORY - Internet mail addressed to Advisory@ecmwf.co.uk		
REGISTRATION			
Project Identifiers	- Pam Prior	OB 225	2384
User Identifiers	- Tape Librarian	CB Hall	2315
COMPUTER OPERATIONS			
Console	- Shift Leaders	CB Hall	2803
Console fax number	- +44 734 499 840		
Reception Counter	- Tape Librarian	CB Hall	2315
Tape Requests	- Tape Librarian	CB Hall	2315
Terminal Queries	- Norman Wiggins	CB 026	2308
Telecoms Fault Reporting	- Michael O'Brien	CB 028	2306
ECMWF LIBRARY & DOCUMENTATION - DISTRIBUTION	- Els Kooij-Connally	Library	2751
LIBRARIES (ECLIB, NAG, etc.)	- John Greenaway	OB 226	2385
METEOROLOGICAL DIVISION			
Division Head	- Horst Böttger	OB 007	2060
Applications Section Head	- John Hennessy (acting)	OB 014	2400
Operations Section Head	- Bernard Strauss	OB 328	2420
Meteorological Analysts	- Andreas Lanzinger	OB 314	2425
	- Ray McGrath	OB 329	2424
	- Anders Persson	OB 315	2421
Meteorological Operations Room	-	CB Hall	2426

		Room*	Ext.**	Bleeper
COMPUTER DIVISION				
Division Head	- Geerd-R. Hoffmann	OB 009A	2050	150
Systems Software Sect.Head	- Claus Hilberg	OB 104A	2350	115
User Support Section Head	- Andrew Lea	OB 227	2380	138
User Support Staff	- Antoinette Alias	OB 224	2382	154
	- John Greenaway	OB 226	2385	155
	- Norbert Kreitz	OB 207	2381	156
	- Dominique Lucas	OB 206	2386	139
	- Pam Prior	OB 225	2384	158
Computer Operations Section Head	- Peter Gray	CB 023	2300	114
Security, Internal Networks and Workstation Section Head	- Walter Zwieflhofer	OB 140	2352	145
GRAPHICS GROUP				
Group Leader	- Jens Daabeck	CB 133	2375	159
RESEARCH DEPARTMENT				
Head of Research Department	- Anthony Hollingsworth	OB 119A	2005	
Computer Co-ordinator	- David Dent	OB 123	2702	

---

* CB - Computer Block
OB - Office Block

** The ECMWF telephone number is READING (0734) 499000, international +44 734 499000, or direct dial to (0734) 499 + last three digits of individual extension number, e.g. the Director's direct number is (0734) 499001.

DEC MAIL: Contact scientific and technical staff via VMS MAIL, addressed to surname.

Internet: The ECMWF address on Internet is **ecmwf.co.uk**
Individual staff addresses are firstname.lastname, e.g. the Director's address is David.Burridge@ecmwf.co.uk