A C Language-Based Modular System for Analyzing and Displaying Gridded Numerical Data

David J. Raymond

Physics Department and Geophysical Research Center, R&D Division
New Mexico Institute of Mining and Technology
Socorro, NM 87801

ABSTRACT

A system for analyzing and displaying gridded numerical data called Candis is described. The system is written in the C programming language, and is built on a standard way of representing such data. The analysis package is modular, hierarchical, and extensible. Facilities available on the UNIX® operating system enhance its ease of use.

11 January 1999

1. Introduction

Over the years, a number of formats have been introduced for representing numerical data produced by various observing systems and numerical models. These formats have the advantage of providing a well-defined standard for the benefit of analysis programs. Examples are the Block Data Set (BDS) of McPherron (1976), the Universal Format for meteorological radar data (Barnes, 1980), FLATDBMS of Smith and Clauer (1986), the Common Cartesian Format (CCF) of Mohr et al. (1986), and the Common Data Format (CDF) of Treinish and Gough (1987). These systems typically isolate the programmer from details of data representation by providing a standard set of access subroutines. Data files are generally self-defining, in that all the information required to interpret a data set is included in the data set.

A common feature of all these systems is that access routines are written in Fortran. One disadvantage of Fortran is that dynamic allocation of arrays is not available except possibly in non-portable extensions on particular operating systems. This makes the construction of general purpose analysis programs awkward, as the largest array ever expected must be allocated space at compile time.

With the spread of the UNIX operating system to a wide range of hardware, the C programming language (Kernighan and Ritchie, 1978) is becoming widely available. The C language is generally accompanied by a set of libraries to do mathematics, input and output, and other functions such as dynamic memory allocation. Though not strictly part of the language definition, these libraries are quite standardized. The ability to dynamically allocate memory plus the variable pointer facility of C allow the construction of compact and general analysis programs.

This paper describes a system written in the *C* language for the *an*alysis and *dis*play of gridded numerical data (Candis). As with the systems described above, Candis is based on a standard way of representing numerical data, with associated standard access methods. In addition, the system is modular, with individual modules reading and writing files in standard format. It may be extended by creating new modules. The system is also heirarchical, in that applications are constructed by writing shell scripts invoking modules or other shell scripts.

Unlike some of the above systems, access to data files is purely sequential. Random access to files facilitates many operations in data analysis. However, the advent of computers with large virtual memories allows rather large files to be read completely into "memory". Subsequent addressing of different parts of such files corresponds to a form of random disk access, and has the advantage of being completely transparent to the user. Some understanding of how paging works is needed to use this method intelligently, but a similar comment can be made of more conventional forms of random access.

One benefit of using only sequential access is the ability to construct applications as sequences of modules in which the output of one module is fed directly into the input of another. This so-called pipe mechanism first appeared on the UNIX operating system, but is becoming available on other systems as well. Advantages are that a proliferation of intermediate files is avoided, and the heirarchical construction of applications using a shell or command processor is facilitated. In addition, new applications may require the development of only a small number of new programs, the bulk of the processing being done by existing software. This speeds development and makes debugging easier.

The organization of this paper is as follows: Section 2 describes the data format used with Candis. Section 3 illustrates the operation of Candis using examples of existing analysis programs. The analysis and display of Doppler radar data is described in section 4. Section 5 is a summary and discussion.

2. Common data format

In this section I specify the format of data files used by the system, and then introduce some additional concepts outside the formal specification, but of great utility.

a. Formal specification

The file structure used here became locally known as the *common data format* before the publication of Treinish and Gough's (1987) system of the same name. The collision in terminology is unfortunate, but I will try to avoid confusion by using lower case letters for the common data format of the Candis system. Changing our terminology would be difficult, as it has become deeply embedded in the documentation.

Common data format files contain three parts, which occur in sequence (see figure 1). The first part is the *header*, which is a sequence of alphanumeric characters organized into lines. The second part is called the *static slice*, and contains data such as calibration fields. The third part contains one or more *variable slices*, each slice containing a particular instance of a set of data fields. This partitioning is internally defined so that the file is nothing more than a stream of bytes to the underlying operating system. In particular, no dependence is made upon, say, the physical record structure of some device such as a magnetic tape or a disk drive.

As mentioned above, the header is organized into lines. Each line must be terminated by a newline character (i. e., linefeed) and must not exceed 81 characters in length, including the newline. White space (i. e., spaces and tabs) before the final newline is ignored. The maximum number of lines in a header is normally 300.

Figure 2 shows the contents of a typical header. Since the header is totally alphanumeric, its contents may be examined by text editors. As figure 2 shows, the header contains 5 sections, namely, 1) comments, 2) parameters, 3) static field descriptions, 4) variable field descriptions, and 5) file format. The header is then terminated with a single line containing an asterisk in the first position.

The comment section is free form, subject only to the restrictions on line length, number, and termination discussed above. Each line of the parameter section contains a parameter name-parameter value pair separated by white space. The value is in alphanumeric form, and need not even be numeric. However, it can't contain white space. Following the parameter value, and separated by white space, is an optional comment. The comment must begin with a pound sign. This provides commenting capability in addition to the comments found in the first section of the header. Use of parameters is up to the programmer -- they play no direct role in defining the rest of the file.

The static field section describes the data that occur in the static slice. Each line describes a *field*. A field is an array of numbers in zero, one, two, three, or four dimensions, and is the basic unit of data in the common data format representation. Fields described in this section occur in the same order in the static slice.

A field description contains a series of alphanumeric strings separated by white space. These strings have the following meaning in sequence:

1) The name of the field.
2) The multiplicative scaling constant, *smul*.
3) The additive scaling constant, *sadd*.
4) The precision in packed integer format (*c*, *s*, or *l*).
5) The dimensionality of the field (0, 1, 2, 3, or 4).
6) Name of the first dimension, *dname*1.
7) Array size of the first dimension, *dsize*1.
8) Name of the second dimension, *dname*2.
9) Array size of the second dimension, *dsize*2.
10) Name of the third dimension, *dname*3.
11) Array size of the third dimension, *dsize*3.
12) Name of the fourth dimension, *dname*4.
13) Array size of the fourth dimension, *dsize*4.
14) An optional comment as in the parameter description lines.

Only as many of entries 6-13 need be included as is justified by the dimensionality of the field, e. g., a two dimensional field would require 6-9, while a zero dimensional field (a scalar field) would require none. The meanings of entries 2-4 will be explained below in the description of the format section.

The variable field section describes the structure of variable slices in the same format used for the static field section. Multiple variable slices can occur, but they all must have the same structure. This restriction makes the storage of variable length data somewhat awkward, but greatly simplifies analysis programs.

The format section contains just one line that has a string equal to "float", "int", or "ascii". This indicates the way in which data elements in the static and variable fields are represented. In the float format, data elements are stored sequentially in the internal single precision floating point format used by the subject computer. In ascii format data elements are in ASCII character set floating point form (g format in Fortran or C) separated by white space. In this case, white space can include newlines as well as space characters and tabs. In int format the field elements are stored sequentially as integers in binary representation at a level of precision specified by entry 4 in the associated field definition line. The one character codes here refer to associated types in the C language, namely, c = char, or nominally 8 bits, s = short, or nominally 16 bits, and l = long, or nominally 32 bits. The quoted numbers of bits refer to the values used in C compilers on commonly available computers, but there is no guarantee that these values will always hold.

The integer format actually contains data that are scaled so as to be representable as an integer. The scaling parameters *smul* and *sadd* are defined in the header for each field, and it is up to the creator of the file to give these sensible values. The integer representation is obtained from the equation $I = F * smul + sadd + 0.5 * s$ where F is the floating point value and I is the integer value. The inverse transformation is $F = (I - sadd)/smul$. The term 0.5*s in the first equation is to enforce rounding rather than truncation in the float to integer conversion. If $F * smul + sadd$ is positive, $s = +1$. If negative, $s = -1$ for computers that truncate negative floats toward zero, and +1 otherwise. The different integer precisions allow tradeoffs between precision and dynamic range on one hand, and data storage space on the other hand. Since a separate precision is defined for each field, this tradeoff can be made on a field-by-field basis.

A consequence of the above definitions is that the only file format that can generally be expected to transport from one computer type to another without modification is the ascii form. The float format will very rarely transport. Storage of data as 16 bit integers is frequently used to facilitate transport between computers, but even this can be tricky, as different computers may represent integers with different byte order. The main criterion with the ascii format is that both computers do indeed use the 7 bit ASCII standard to represent characters. Issues having to do with parity bits and extraneous characters such as carriage returns and nulls at the ends of lines need also to be considered.

Each slice has the following structure. At the beginning there is an 8 byte sub-header that contains the *element count*, or the number of *elements* in the slice. The elements from each field then occur in sequence. The element sequence for multi-dimensional fields is the same as in the C language, i. e., the last dimension mentioned in the field definition is iterated most rapidly.

The element count is obtained by adding up the elements from all constituent fields. The number of elements in a field is simply unity times $dsize1$ times $dsize2$ ..., where as many dimension sizes are included as there are dimensions. For example, a scalar, or zero dimensional field would have an element size of one, whereas a two dimensional field would have $dsize1$ times $dsize2$. The element count is represented in the slice sub-header as an ASCII-coded decimal integer.

In the header, all section sub-headers (e. g., ***parameters*** -- see figure 2) must be present, even if there are no entries for that section. There must be one or more variable slices, and if there are no static fields, the static slice element count must still be present, albeit with a value of zero.

b. Useful constructs

I now discuss a number of concepts that are not a formal part of the common data format specification, but that turn out to be quite useful.

Even though successive variable slices are typically envisioned to represent fields at successive times, no special mechanism is provided to specify the time of each slice. Instead, one simply defines a scalar variable field called, for instance, "time", that contains the time information. This *sequence field* need not even be time -- it could be, for instance, elevation, with the idea that successive slices represent fields at different levels rather than different times. For use with certain software, it should be monotonically increasing with position in the file.

Another useful construct is the *index field*. Index fields are one dimensional static fields with the same dimension name and field name. They are useful for specifying the domain over which data are defined. For instance, if data are defined for x between 0 and 10 km at intervals of 2 km, then one would define an index field named "x" with *dsize*1 = 6. The successive elements of this field would be assigned the values 0, 2, 4, 6, 8, 10. Index fields aid plotting routines, and it is good practice to define an index field for each dimension used in a common data format file. Note that the elements of index fields need not be equally spaced. For instance, if data points were closer together for small x in the above example, the index field might take on values 0, 0.5, 1, 2, 5, 10.

A more economical way of defining a domain is to specify *index parameters*. These are typically used when equally spaced data points must be guaranteed. Some programs search the parameter section of the header for parameters of the form *dname*0 and d*dname*, where *dname* is a dimension name that occurs in a field definition. For instance, if a dimension name "x" is found, parameters with the names "x0" and "dx" are sought. These are the index parameters for the dimension x, and are interpreted respectively as the starting value and increment for points in the x domain on which field values are defined. For example, if x0 = 3 and dx = 2.5, then the x values 3, 5.5, 8, ... are implied. Most programs that use index parameters assume default values of 0 and 1 respectively for the starting value and the increment if the corresponding parameters are not found.

Many observational data sets have regions of bad or missing data. The prime example is radar data, wherein data are only defined in regions containing precipitation particles. Many of the programs written for common data format files look for parameters called "bad" and "badlim". The value of the latter parameter is assumed to define the range of valid numerical data. Values larger than "badlim" in absolute magnitude are assumed to indicate that the datum is bad or missing. The parameter "bad" suggests a value (greater than the value of "badlim") to be used to indicate bad data. If these parameters are missing, default values of 1.e30 and 9.99e29 are respectively assumed.

3. Hierarchical approach to data analysis

Programs in the Candis system can be classified into one of three levels, namely *primitive functions*, *filters*, and *shell scripts*. Casual users should be able to obtain considerable utility from the system by programming at the highest and simplest level, namely the shell script level. However, new projects will often require the creation of new analysis programs, or filters. Generally these can be kept quite simple, as many standard functions will already be available to solve standard parts of the analysis problem. The shell script and pipe mechanisms provide an effective way of combining standard and non-standard operations. Recourse to the lowest level should rarely if ever be necessary. This is the level of direct manipulation of common data format files, and is adequately done by a library of primitive C language functions.

The primary exception to this rule occurs when a common data format file is created by a program in a language different from C. In this case the C primitives can’t be used. However, creation of a particular common data format file is much easier than interpreting an arbitrary file, so this presents no particular problems. (A word of caution: It is generally not safe to assume that languages other than C produce binary data in a form that is compatible with the C representation. For instance, Fortran implementations sometimes write binary data in a record structure with embedded byte counts, checksums, etc. Thus, translation between *languages*, even on the same computer, can cause problems. Use of ascii format should minimize these problems. Recall, however, the different order in which multi-dimensional arrays are stored in Fortran and in most other languages, including C.)

I now describe the approaches used and specific functions developed for the three levels of programming.

a. Primitive functions

The required primitive functions fall into four categories, namely functions to create common data format headers, to interpret these headers, to read and write headers and data slices, and to access fields within slices. Header information may be needed throughout the period in which a data file is being accessed, so the philosophy is to read it into a user-defined buffer that can be retained. The imposition of a maximum number of header lines allows static allocation of buffer memory. This is memory-inefficient for small headers, but simplifies programming.

The technique for creating a new common data format header is to create a null header (with just the section labels) and then add comments, parameters, and fields on a line-by-line basis. Functions exist to perform each of these tasks. If a new header consists of an old header plus additions, it can be created by copying the old header to a new buffer and then applying the above functions. Partial copies of individual sections of headers can also be accomplished.

Interpretation of headers is perhaps the most difficult task. Functions exist to extract the comment section from a common data format header and to determine the format of the data file. In addition, there are functions to extract parameter values and field characteristics by either name or position in the header. If requested parameters or fields don't exist, a special code is returned, and appropriate action can be taken by the calling program.

There are functions to read and write headers and slices, as well as a function to query the header as to the expected number of elements in a static or variable slice. This information is needed so that consistency with the element count in the appropriate slice may be checked. As mentioned above, it is customary to statically allocate memory for header buffers. However, data slices of unpredictable size can occur, and dynamic allocation of memory is important here. Once the header is read, the memory required for static and dynamic slices is readily computed. Space can then be allocated using a C language library function. There is a Candis routine that simplifies this operation.

Accessing particular fields within a data slice is done by assigning a pointer to the start of the field of interest. Two functions exist to accomplish this task, one of which also returns additional information about the field in question.

Important declaration information is kept in a file named "cdfhdr.h". Any program calling the primitive functions needs to include this file. Visual inspection of the file can also enhance one's understanding of their operation. Table 1 gives the names and summarizes the uses of available primitive functions.

Name	Use
gethdr	read header from designated stream to a header buffer
getelcnt	read an element count from designated stream
getslice	read data slice from designated stream
puthdr	write header to designated stream from a header buffer
putslice	write data slice to designated stream
nullhdr	create null header in specified buffer
copycmt	copy comment section to new header buffer
copypar	copy parameter section to new buffer
copyfld	copy static or variable field section to new buffer
addcline	add comment line to specified header buffer
addpar	add parameter entry to specified header buffer
addfld	add field description to specified header buffer
getcmt	extract comment from specified header buffer
getpar	extract parameter name and value by position from a header buffer
seekpar	extract parameter value by name from a header buffer
getfld	extract field characteristics by position from a header buffer
seekfld	extract field characteristics by name from a header buffer
getfmt	extract file format from a header buffer
elemcnt	extract expected element count from a header buffer
getbuff	allocate memory for a slice buffer
getptr	compute a pointer for a specified field
getptr2	compute a pointer and return field information

Table 1. Primitive functions and their use. Functions that deal with fields can be directed either to the static or variable section of the header. Getslice and putslice only work on files in float format. Special handling is required for slices in other formats.

b. Standard filters

In the context of UNIX, a filter is a program that reads data from the *standard input*, transforms it in some way, and sends the result to the *standard output*. Standard input and output are pre-defined ports that normally read from and write to the user's terminal, but may be redirected to a file or another program. The operation of all but the simplest filters is controlled by command line arguments. If errors occur, error messages are sent to *standard error*, another pre-defined output port. The standard error writes to the user's terminal, and normally isn't redirected like the standard output. This provides a mechanism for separating data from error messages. When filters are invoked with an incorrect number of arguments, a "usage" statement is printed to standard error, and the filter exits. This provides a simple form of on-line help. Candis filters typically read and write common data format files and record their actions in the comment section of the output file. Translating filters convert foreign data formats to common data format. Programs that read or write more than one data file don't fit into the filter paradigm and must obtain names of desired files from the command line. However, most desired operations on data can indeed be regarded as filters. A naming convention has been adopted, in which all general purpose filters begin with the prefix "cdf".

The natural data format for most numerical work is the float format. Most filters therefore only work on common data format files in this format. A filter (cdftrans) is provided to transform files from any format into any other.

One of the most commonly needed operations is to determine the contents of a common data format file. Cdflook fulfills this need by displaying the header and selected information about each slice on the standard output.

Several filters are available to limit the domain over which data are passed to the output file. Cdfwindow passes only data within requested limits for specified dimension names. Cdftsel passes only variable slices that have values of a specified sequence field within a particular range. Cdfextr passes only those fields specified as command line arguments. Cdfrdim passes only data defined at a particular value of a specified dimension name, and thus reduces the dimensionality of those fields with a dimension of that name.

Three programs combine data into bigger chunks. Cdfcat combines all variable slices of a particular file into a single variable slice. A specified sequence field is turned into an index field, and the dimensionality of all other fields is increased, with the new dimension being given the name of the new index field. Cdfcatf merges homogeneous files such as successive radar volumes into a single file. Cdfcatf is not a filter, because it obtains its input from files listed on the command line. Compatibility between files is checked. Cdfmerge combines heterogeneous files, each with only a single variable slice. Static and variable slices from each file are merged into a single file. This allows, for instance, the merging of aircraft and radar data for common display. (Unlike the CCF system mentioned in the Introduction, no conversion to a common Cartesian coordinate system is done by this program.) Collisions between field names are prevented by appending a unique suffix to fields from each input file.

Rtape, as its name suggests, reads magnetic tape files onto disk. Since some existing data formats depend on a particular record structure for their decoding, rtape passes on information about the physical record structure on the tape by prepending each record with a byte count for that record. Tape records of arbitrary size can be read, and different record sizes can be mixed within a particular tape file. A number of common data format filters use rtape output.

Numerous other filters have been written, but the above examples give the flavor of the Candis system. One filter not mentioned so far is a plotting filter called cdfplot. This is sufficiently complex that it is discussed separately. UNIX-style documentation exists for all of the above filters, and on the primitive functions as well.

c. Shell scripts

All operating systems have some form of command interpreter. Many are suitable for constructing complex applications by combining calls to programs in a script or text file which is read by the interpreter. The UNIX command interpreter is called the "shell", and I illustrate its use with an example taken from the analysis of the output of a time-dependent, two-dimensional numerical model.

The sample shell script, named "slice", reads as follows:

```
: make arbitrary slice and plot
if test $# -lt 4
then echo "Usage: slice testnumber reducevar reduceval cmdlist ..."
else
  if test -f txz$1
  then
    echo -n ""
  else
    echo -n "making txz file ... "
    expand 0 < test_$1 | cdfcat time 0 201 51 > txz$1
    echo "done"
  fi
  for i in $4 $5 $6 $7 $8 $9
    do list="$list $i"
  done
  cdfrdim $2 $3 < txz$1 | cdfplot $list
  $PG
fi
```

The function of slice is to make contour and vector plots over two-dimensional sub-spaces of the three-dimensional space of the model, x, z, and time. These sub-spaces include snapshots at a given time and time sections at constant x and constant z. The input to slice consists of a common data format file containing multiple variable slices, one per time level. This file is created by the numerical model, and is assumed to have the name "test_N" where N is the number of the test run. The first function of slice is to create a file named "txzN", in which all variable slices have been combined into one using cdfcat, and in which certain auxiliary fields have been computed by a special purpose program "expand". The vertical bar in the line containing these programs indicates that the output of expand is piped into the input of cdfcat. This sequence is only invoked if the file txzN doesn't already exist, so that this relatively time consuming operation need not be repeated. The sub-space extraction and plot generation is accomplished by the line containing cdfrdim and cdfplot.

The symbols "<" and ">" respectively indicate redirection of standard input and output from keyboard and terminal to the indicated files. Character sequences consisting of a dollar sign and a number are dummy variables replaced by the corresponding command line arguments to slice. Thus, $1 refers to the desired test, $2 to the dimension to be held constant, and $3 to the desired value of that dimension. Subsequent dummy variables contain instructions to the plotting routine, cdfplot. These are concatenated into a single string by the looping construct that begins with "for". $PG is a variable that is set previous to the invocation of slice indicating which graphics device should receive the plots. One of the features of slice is that if less than four command line arguments are typed, a usage statement is printed and the shell script exits.

Though relatively simple, this shell script illustrates the features needed to make a command processor useful in the context of Candis, namely dummy variable replacement, looping, and branching. It is also useful if shell scripts can invoke other shell scripts. It is evident from the above example that individual filters *must not* operate in an interactive manner through the user's terminal. All control over filter operation must be via command line arguments. This is necessary to avoid collisions between terminal input and output from different filters. If interactivity is desired, it should be limited to the uppermost level, i. e., the shell script itself.

d. The portable graphics system

The filter cdfplot referred to above invokes a locally developed graphics system called *Pgraf*. Candis makes no commitment to any particular graphics system. However, Pgraf (for "portable graphics") provides some useful lessons that are worth describing, even though its capabilities are relatively primitive.

Pgraf, as the name implies, was developed to port easily from one graphics device and computer to another. To facilitate this, a main program interfaces to hardware through six simple, low level subroutines.

These are easily rewritten for each graphics device, and versions could be made to drive various graphics standards as well. User programs actually invoke subroutines that create a file containing a device-independent graphics metacode. A separate program then reads the metacode file and draws the graphics images on the desired device. Available graphics functions include station plots, line graphs, scatter plots, contour plots with optional hatching for emphasis, and vector plots.

The filter cdfplot serves as a general purpose link between Candis and the portable graphics system, allowing arbitrary one and two-dimensional fields to be graphed, contoured, etc. It therefore largely eliminates the need to write special purpose programs to generate plots. A complete description of the operation of cdfplot is beyond the scope of this paper, but examples of its use will be cited in the next section.

One important feature of cdfplot is that by default it prints out a copy of the entire comment section of the common data format header adjacent to the actual plot. Since Candis filters record their actions as comments, this provides a complete history of significant operations on the data with every plot.

4. Synthesized radar data

The National Center for Atmospheric Research (NCAR) has developed programs to synthesize multiple Doppler radar data and extract three dimensional particle and wind velocities (Mohr et al., 1986). One of the first major uses of the Candis system has been to further analyze output data from these programs. In this section I present our efforts in this area as a hopefully non-trivial example of the use of Candis.

Several specialized filters were developed to handle the output of NCAR's programs. The first, called radcedric, simply converts the output of the CEDRIC program into a common data format file. Radcedric requires that NCAR tapes be read onto disk by rtape, which is discussed in the previous section. CEDRIC presents data fields as a sequence of two dimensional arrays at different levels. Radcedric converts these into a single three dimensional field in x, y, and z for each variable at each analysis time. Typically one then has Cartesian components of particle velocities, reflectivities from one or more radars, and possibly vertical air motion obtained from integration of the continuity equation. CEDRIC field names are retained, but converted to lower case. The prefix "rad" indicates programs dealing exclusively with radar data.

Radvert recomputes the vertical air motion based on a locally-derived algorithm (Krehbiel, personal communication). The ambient pressure and density fields are computed and stored in the static slice in the course of these computations.

Figure 3 shows a contour plot of vertical and horizontal winds for a thunderstorm that occurred over Langmuir Laboratory in central New Mexico. It was created with the following command:

```
cdfrdim z 7 < b16 | cdfplot 6,6,t/u,v,3,3,v/wi,4,1,c/wi,-4,4,1,f ; $PG
```

Radcedric, radvert, and cdfwindow were used separately to create the file "b16". Cdfrdim then extracted a slice through the data at an elevation of 7 km. The result was passed to cdfplot, which made a vector plot of the horizontal wind components, u and v, and 4 m/s contours of the vertical wind, wi. Vertical hatching indicates wi > 4 m/s, while horizontal hatching indicates wi < -4 m/s. Horizontal wind vectors have a cross at their tail indicating the analysis point. Vector components one grid interval in length equal 3 m/s.

A filter called cdfocut can substitute for cdfrdim with the result that the sub-space is a vertical plane with arbitrary azimuth and location. Bilinear interpolation is made to the desired plane, and horizontal velocity components in and normal to the plane are also computed. Replacement of cdfrdim by cdfocut in the above script makes possible the examination of data along non-cardinal directions.

CEDRIC works with data at one analysis time (actually, the range of times over which all radars complete a single volume scan) and combines it into a single file called a volume. The resulting common data format files thus contain a single variable slice. Certain analyses such as the computation of Lagrangian parcel trajectories require data at different times. Cdfcatf provides a way to combine multiple volumes into a single file. Cdflagr then uses such a file to compute trajectories from specified starting points in space and time. Integrations can proceed both forward and backward in time, and either air parcel or particle trajectories can be computed. In addition to the actual trajectories, cdflagr interpolates and stores the values of all fields along the trajectories.

As an example of the use of cdflagr, we compute the trajectories of air parcels reaching z = 10 km at x = 4 km at the time of figure 3. Figure 4 shows air velocities and reflectivity in a vertical plane defined by x = 4 km, while figure 5 shows the projection into this plane of trajectories reaching the above-defined line at time = 40.9 ks (kiloseconds after midnight). Successive diamonds indicate parcel positions at intervals of 100 s. The results clearly show that some parcels originated from near cloud base at 4 km, even though vertical velocities at lower levels are quite small by the analysis time. Figure 5 was created with the following command:

```
cdflagr u v wi time 40.9 -0.1 21 x 4 z 10 < b0 | cdfplot -4,6,x/4,13,y/6,10,t/y,z,1,p/y,z,4,m ; $PG
```

The input file "b0" was created by concatenating several successive volumes with cdfcatf.

Cdflagr works by creating a new common data format file consisting of two dimensional fields, the two dimensions being parcel number and time. Three of these fields are called "x", "y", and "z", and in the above example, "y" and "z" are plotted against each other. In addition to these fields, cdflagr creates additional fields with the same names as fields in the input file. These contain the input fields sampled along the trajectories of the associated parcels.

One of the recognized problems in meteorological data analysis is combining data of disparate types. Mohr, et al.'s (1986) CCF system solves this problem by interpolating everything to a Cartesian grid. The Candis system offers the alternative of combining heterogeneous data without modification in the same common data format file using the cdfmerge utility mentioned above. Cdfplot can then be used to present these data sets on the same plot. Figure 6 shows an example of this procedure, wherein the trajectories of figure 5 are overlayed on the reflectivity field shown in figure 4. This was done by diverting the output cdflagr into a temporary file "lagr1" and then merging this file with the radar volume of interest, "b16":

```
cdfmerge b16 "" lagr1 ".l" > b16.l
```

The output file was given the name "b16.l", and all fields from the parcel trajectory file were given the suffix ".l". Fields from b16 were given a null suffix. Figure 6 was then made using the command

```
cdfrdim x 4 < b16.l | cdfplot 6,10,t/zcp4,10,3,c/zcp4,-30,30,3,f/y.l,z.l,1,p/y.l,z.l,4,m ; $PG
```

The merger process in this case is only graphical, whereas the CCF system is capable of, say, combining different sources of wind information on a common grid using an objective analysis scheme. The latter approach is also possible using the Candis system, but sometimes graphics overlays are all that is required.

5. Summary and discussion

The key feature of Candis is use of the C programming language for its implementation. Though this a potential barrier to use by programmers experienced only in Fortran, the ability to dynamically allocate memory and to manipulate pointers provide substantial advantages in the construction of general purpose analysis programs. The general purpose nature of Candis filters derives directly from these characteristics.

Another important feature of Candis is that applications can be built up in a heirarchical fashion using a mix of old and new filters in conjunction with a shell script. The key to this capability is the restriction of programs to strict serial access of data in the standard format. The availability of large virtual memory computers alleviates this restriction in cases where random access is needed; the entire file is simply read into memory.

The final feature that enhances the utility of Candis is the flexibility of the common data format. All data are represented on grids of from zero to four dimensions, with data defined over different spaces being kept separate by judicious use of index fields. Unlike CCF or the CDF system of Treinish and Gough (1987), heterogeneous data can easily be stored in the same file. The advantage is that different types of data can be made available to a common analysis program in a structured fashion without having to access more than one file.

One important consequence of the structure of the common data format is that variable length data are difficult to store efficiently. By allowing successive variable slices to be variable in length, this goal could be accomplished. However, the added complexity was not judged to be worth the gain in generality. One way variable length data, e. g., significant level sounding data from a collection of stations, could be handled is to allocate a variable slice of maximum plausible size and define all unused storage as bad. This approach does, of course, make files larger than they strictly need to be.

Efficiency of operation, narrowly defined in terms of minimizing demands on processor time and disk storage and access, was *not* a goal of this project. The point was to define a system that minimized the effort required to develop new applications. In light of the ever-increasing costs of software and the ever-decreasing costs of hardware, this seems like a reasonable point of view. In spite of this, response time in data-intensive applications such as Doppler radar is tolerable on modern workstations. There will, of course, be applications in which absolute maximum efficiency must be extracted from the hardware. Candis may not be suitable for such applications.

One area in which the Candis system may be useful is in the direct *generation* of data sets in the field. The format is simple enough that very little overhead is imposed on the data collection system, and the ability to write an indefinite number of variable slices without knowing how many there will be beforehand is essential to field-generated data. Since it is easier to *create* common data format files than it is to interpret them, the real time system need not be restricted to use of the C language or the UNIX operating system.

Candis software will be made available to other users through the University Corporation for Atmospheric Research's UNIDATA project.

Acknowledgments. The critical comments of Bill Winn resulted in clearer and more useful constructs at all stages of this project. The comments of anonymous reviewers were also appreciated. Student programmers Sarah Bottomley, Dale Harris, Robert Solomon, and Dinh Ton That made significant contributions. This work was supported by National Science Foundation grants ATM-8311017, ATM-8611364, and ATM-8605136.

REFERENCES

Barnes, S. L., 1980: Report on a meeting to establish a common Doppler radar exchange format. *Bull. Amer. Meteor. Soc.,* **61**, 1401-1404.

Kernighan, B. W., and D. M. Ritchie, 1978: The C programming language. Prentice-Hall, Inc., Englewood Cliffs, NJ, 228 pp.

McPherron, R. L., 1976: A self-documenting source-independent data format for computer processing of tensor time series. *Phys. Earth Planet. Inter.,* **12**, 103-111.

Mohr, C. G., L. J. Miller, R. L. Vaughan, and H. W. Frank, 1986: The merger of mesoscale datasets into a common Cartesian format for efficient and systematic analyses. *J. Atmos. Oceanic Tech.,* **3**, 143-161.

Smith, A. Q., and C. R. Clauer, 1986: A versatile source-independent system for digital data management. *EOS, Trans. AGU,* **67**, 188.

Treinish, L. A., and M. L. Gough, 1987: A software package for the data-independent management of multidimensional data. *EOS, Trans. AGU,* **68**, 633-635.

FIGURE CAPTIONS

Figure 1. Schematic layout of a common data format file. Variable fields have the same structure as the static field.

Figure 2. Example of a common data format header.

Figure 3. Example of a plot produced by the filter cdfplot. Radar-derived horizontal and vertical winds are shown in a horizontal plane at 7 km. A vector equal to a grid interval in length represents 3 m/s in horizontal wind. The cross defines the analysis point and indicates the tail of the vector. Vertical wind contours are at 4 m/s intervals, with horizontal hatching indicating vertical winds less than -4 m/s and vertical hatching for vertical winds greater than +4 m/s.

Figure 4. Vertical section at x = 4 km at the same time as figure 3. Wind components in the y-z plane are shown. Vector components equal to a horizontal grid interval represent 3 m/s, while the

corresponding vertical scaling is 6 m/s. The difference reflects the different grid intervals in the vertical (500 m) and horizontal (250 m). Radar reflectivities exceeding 30 dBZ are indicated by vertical hatching.

Figure 5. Projection into the y-z plane of trajectories of parcels reaching the line x = 4 km, z = 10 km at the time of figures 3 and 4. Integration is backward in time for a maximum of 2000 s. Parcel positions at 100 s intervals are denoted by the diamonds. Note that some parcels originated from as low as 4 km.

Figure 6. Regions of high radar reflectivity as in figure 4, with the trajectories of figure 5 overlayed. Reflectivity is contoured at 10 dBZ intervals, and regions with reflectivity exceeding 30 dBZ are hatched.