Candis Tutorial

David J. Raymond

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

1. Introduction

Candis (C language ANalysis and DISplay) is a system for analyzing and displaying gridded numerical data. Raymond (1988) provides a general discription of the system. The purpose of this document is to teach the reader how to use existing Candis programs and how to create new ones. The paper is organized as follows: Section 2 is a primer that introduces the reader to Candis. Section 3 briefly describes existing Candis programs. Section 4 illustrates how to create new programs.

2. Primer

In this section Candis is introduced by means of specific examples. Covered first is the Candis data format, with associated concepts and vocabulary. Then examples of use of the system are presented to give the reader an idea of what is possible. Familiarity with some of the more basic concepts of UNIX is assumed, i. e., the UNIX file system, redirection of input and output, and the notion of pipes.

2.1. Candis data format

The basis for Candis is a format for representing gridded numerical data. This format assumes that data of interest can be represented as rectangular arrays of floating point numbers with from zero to four dimensions. A zero dimensional array is, of course, just a single number, or scalar. An example of a one dimensional array might be a time series of a variable such as temperature, obtained, say, from an aircraft or a surface station. Alternatively, it might be all the values of radar reflectivity along a single ray of radar data. A two or three dimensional array might be a single field from the output of a two or three dimensional numerical model. Candis allows for the existence of successive instances of the same field, e. g., the velocity field from a model at successive times, by implementing a record-like structure, with successive records, or *slices*, as they are called containing data from successive instances. "Successive" often refers to time, but need not. Successive slices may, for instance, represent data on horizontal planes at successive elevations.

Candis slices can contain fields of different dimensionality and size. For instance, a slice may contain three dimensional fields from a numerical model as well as one dimensional vertical profiles and zero dimensional fields that might represent certain constant or average values. There are two types of slices, *static slices*, which occur only once in a Candis file, and *variable slices*, which can occur an arbitrary number of times. The static and variable slices will in general contain different fields, with the static slice containing those fields that only need to be presented once. The structures of variable slices in a given file are all the same (i. e., they contain the same number, size, and arrangement of fields).

Candis files are self-describing, in that there is an ASCII header that informs users and programs about the file. The structure of Candis files is as follows:

Header

Static slice

Variable slice 1

Variable slice 2

Etc.

Below is an example of a Candis header. It corresponds to the header of the sample file in candis/data.

```
***comments***
fwave33: test_21a
cdfcat: time 0 201 51
cdfthin: time 3
***parameters***
badlim 1.e30
bad 0.999e30
dx 1
x0 0
dz 0.100000
z0 0
dtime 60
time0 0
mu 5
***static_fields***
qs 10000 0 s 1 z 11
time 100 0 s 1 time 4
x 100 0 s 1 x 21
z 1000 0 s 1 z 11
***variable_fields***
h 10000 10000 s 3 time 4 x 21 z 11
u 1000 1000 s 3 time 4 x 21 z 11
w 10000 10000 s 3 time 4 x 21 z 11
b 10000 10000 s 3 time 4 x 21 z 11
psi 10000 10000 s 3 time 4 x 21 z 11
***format***
float
*
```

Notice that the header is divided into five sections. The purpose of the "comments" section is to keep a record of what has happened to the file since its creation. The format of entries here for each processing program is the name of the program followed by a colon, followed by the command line. The command line may continue onto additional comment lines if necessary. In this case the file was created by the program "fwave33", which happens to be a two dimensional, time dependent numerical model. The Candis programs "cdfcat" and "cdfthin" were then applied. The functions of these programs will be discussed later. It is sufficient here to realize that most Candis programs operate as UNIX filters, i. e., they accept a Candis file on the standard input and write a modified Candis file to the standard output.

The "parameters" section contains the names and values of parameters useful to subsequent programs or users. Two classes of parameters have particular meaning to most Candis filters. The parameters "bad" and "badlim" indicate the valid range of data in the file. If field data take on values greater than badlim in absolute value, this is considered to represent bad or missing data associated with that field. "Bad" is a suggested value for indicating bad data. Note that the same badlim value holds for all fields in a given file. If no bad data parameters appear, default values of 0.999e30 and 1.e30 are respectively assumed for badlim and bad.

The other parameters with special meaning are called index parameters. These are discussed later in this section. All other parameters have meanings private to the file in which they appear.

The static and variable field sections describe the characteristics of the fields in the static and variable slices. The formats of these two sections are the same, with one field described per line. Each line consists of words or numbers separated by white space. The first word is the name of the field. The second, third, and fourth words give scaling information, which allows floating point data in the field to be packed into scaled integers. In particular, the packed integer form is obtained by multiplying the floating point form by word #2 and adding word #3. Word #4 is one of "c", "s", or "l", indicating "character", "short", or "long". This specifies the length of the receiving integer in terms of C language integer lengths. These generally correspond to 8, 16, and 32 bits respectively, but may differ on machines of uncommon architecture. The fifth word gives the dimensionality of the field, and may range from 0 to 4. The remaining words give a dimension name and dimension size for each dimension. For instance, the static field "qs" in the above example has one dimension named "z", with 11 points. The variable field "h" has three dimensions, "time" of size 4, "x" of size 21, and "z" of size 11.

Note that each dimension has an associated one dimensional field in the static slice with a field name the same as the dimension name. Such a field is called an index field. The purpose of index fields is to specify the grid over which data are defined. Most Candis programs depend on there being index fields for each dimension. Some also require that the dimensions occur in the same order in all field definitions, and that the same order is observed in the arrangement of the index fields in the static slice.

There is no requirement that grid points, as defined by index fields, must be equally spaced. However, if they are, grid information can be characterized more compactly by starting values and increments. This is the purpose of index parameters. For instance, in the above example, the parameters "x0" and "dx" respectively represent the starting value and increment in the dimension "x". Some Candis programs require that there be index parameters, and are hence limited to uniformly spaced grids.

The final section of the header is the format section. Candis files can take any one of three formats, "float", "int", and "ascii". The working format of Candis is "float". In the "int" format, data are converted to packed integers as specified for each field. The purpose of this format is to conserve storage space by making the file as small as possible. The "ascii" format converts each data element to a formated ASCII string, and provides a portable representation for moving data between machines of different architectures. The ascii format tends to increase the size of data files. However, applying the UNIX "compress" utility after conversion to ascii form sometimes results in significant data compression. As more and more computers adopt IEEE floating point format, the float format itself will become more portable, requiring no data conversion at all.

The slice format is relatively simple. Each slice has an eight byte decimal ASCII integer at the beginning, called the *element count*, which gives the number of elements in that slice. An element is a single number, whether it be in float, int, or ascii form. Fields are made up of elements, and follow the element count in the order specified in the header. Within each field, the last dimension is iterated most rapidly, as in C language arrays. An empty static slice must still contain the element count, which, of course, will be zero in this case.

2.2. Using Candis

This section leads you through some simple examples of the use of Candis on a real data file. The file is in the directory candis/data, and has the header given in the previous section. To execute the following commands, you must first translate this file to float format. Do the following:

```
cdftrans -f < example.a > example
```

Then, to look at the file header, execute

```
cdflook < example | more
```

Piping the output through "more" keeps the output from scrolling off the page.

Note that the data are contained in a number of three dimensional fields, "h", "u", etc. To obtain a two dimensional cut through these data, use the Candis program "cdfrdim":

    cdfrdim time 60 < example > tempfile

This results in a new file, which we have chosen to name "tempfile", which contains the two dimensional cut. You could use cdflook on this file to confirm that cdfrdim had the desired effect.

The time value "60" was chosen after examining the index parameters "dtime" and "time0" in the header, which suggested that data existed for time = 0, 60, 120, etc. The size of the time dimension is 4, so the maximum time represented is 60*(4 - 1) = 180. If the time requested in cdfrdim isn't precisely on an existing value, the nearest time value will be taken. Similarly, if a cut at x = 8 rather than constant t were desired, the command

    cdfrdim x 8 < example > tempfile

could be executed.

To examine the results of the time = 60 cut, the plotting program "cdfplot" could be invoked. For instance, to obtain contour plots of w and b, simply execute (after reading the following paragraph!)

    cdfplot w,1,c b,1,c < tempfile && pg

This will produce two separate contour plots with default contour intervals.

A word of explanation is required about the operation of cdfplot. This program produces an intermediate file in the current directory called "pgraf.out". This is a "metacode" file that contains the plotting information for interpretation by a Portable Graphics System (Pgraf for short) metacode reader. In order for this to happen, you must have access to suitable graphics hardware, and a metacode reader program that draws pictures on that hardware. By convention, these are named pg..., where the "..." indicate the type of hardware, e. g., pghp for Hewlett Packard graphics terminals, pgtek for Tektronix 4010 graphics, etc. If you include (for instance) the lines

    setenv PG pgtek
    alias pg $PG

in your ".cshrc" file (assuming you use C shell), then "pg" really means "pgtek". Thus, you don't have to remember that your graphics terminal uses Tektronix graphics every time you make a plot. The reason for the intermediate "PG" will be discussed below.

The reason for the "&&" in the cdfplot command is to make execution of the metacode reader conditional upon the successful execution of cdfplot. Otherwise, pg will try to plot whatever exists in pgraf.out, regardless of whether it is garbage or not!

Notice that cdfplot produces two boxes, the left one with the actual plot, the right one with information about the input file and the plot. This information consists of 1) the comment section of the input file, 2) plot information proper, and 3) the values of scalar fields. From "contour(w*1,0.0016,1)" we deduce that the field w is contoured with a scale factor of 1, the contour interval is 0.0016, and the line type is 1. Also confirmed is that time = 60. (The time index field is reduced to a scalar field by cdfrdim.)

Recall that the above example makes separate plots for the fields w and b. To overlay the contours with separate line types, execute instead

    cdfplot w,1,c/b,2,c < tempfile && pg

This makes only one plot (actually, one per variable slice) containing solid contours of w and dashed contours of b. The number indicates line type, which ranges from 1 to 4, indicating respectively solid, dashed, dotted, and dot-dashed lines.

Cdfplot doesn't label contours, but there are two ways to extract this information. The command

```
cdfplot w,1,c/w,g < tempfile && pg
```

overlays a grid of numbers on the contour plot, while

```
cdfplot w,1,c/w,1,f < tempfile && pg
```

hatches regions of the plot in which w exceeds a certain value in absolute magnitude. The notation "fill(w*1,-0.0016,0.0016,1)" on the plot tells us that horizontal hatching of line type 1 occurs for w < -0.0016, and vertical hatching occurs for w > 0.0016. If you desire to change the default contour interval, and control the extent of the hatched regions as well, the command

```
cdfplot w,0.001,1,c/w,-0.001,0.001,1,f < tempfile && pg
```

sets the contour interval to 0.001 and makes horizontal hatching for w < -0.001 and vertical hatching for w > 0.001.

Suppose that instead of a contour plot, you desired to obtain a vertical profile at x = 8 of w. A second application of cdfrdim to tempfile would isolate the desired data, and cdfplot could then be used to make the plot. Instead of creating another intermediate file, let's pipe the results of cdfrdim directly into cdfplot:

```
cdfrdim x 8 < tempfile | cdfplot w,z,p && pg
```

Suppose after experimentation, it was clear that w was limited to the range [-.01,.02] and z was limited to [0,1]. Prettier axes could be created with the following command:

```
cdfrdim x 8 < tempfile | cdfplot -.01,.02,x/0,1,y/4,6,t/w,z,p && pg
```

The "-.01,.02,x/0,1,y" sets the the horizontal and vertical axes to the specified ranges. Don't confuse the "x" here with the "x" dimension -- in this context, "x" means the horizontal axis (whatever the name of the variable plotted) and "y" means the vertical axis. The "4,6,t" sets the number of tic marks on the two axes.

Full documentation will be found for cdfplot in its manual page. Also, typing cdfplot without any arguments will give you a brief usage summary. This is also true of most other Candis commands.

Sometimes it is desirable to put a long series of commands into a shell script so that the whole sequence doesn't have to be retyped each time. As an example, one might create a shell script to make vertical profiles of w (as in the above example) at arbitrary values of x and time. The script named "profile" might appear as follows:

```
#!/bin/sh
# profile -- plot vertical profile of w
#
if test $# != 2
then
      echo 'Usage: profile x time'
else
      cdfrdim x $1 | cdfrdim time $2 | \
      cdfplot -.01,.02,x/0,1,y/4,6,t/w,z,p/x=$1_time=$2,-.005,.9,l && $PG
fi
```

The first line of this script indicates that the Bourne shell is to be used to interpret it. If the number of arguments is not two, a usage message is typed, and the script exits. Otherwise, the desired cuts are made and the plot is performed. The "x=$1_time=$2,-.005,.9,l" puts a label on the plot at (-.005, .9) so that the results for different values of x and time won't be confused. The metacode reader is invoked with "$PG" rather than "pg", because the Bourne shell doesn't know about aliases. Since no specific file is redirected into the initial cdfrdim, the script has to be invoked with redirection, e. g.,

```
profile 8 60 < example
```

(Don't forget to change the script's mode to "executable", or it won't run.) Note that no intermediate file is created in this case.

This concludes the primer. We have only scratched the surface on the functionality available with Candis. However, most operations can be performed in the same style as indicated above. The next section lists and briefly describes the most important Candis programs.

3. Existing programs

Leserman (1988) introduced a classification of Candis programs based on the general sort of thing that they did. A slightly modified classification is used here, namely, "selectors", "constructors", "math operators", "display utilities", "input-output utilities", "translators", and "special purpose data converters". The most important programs in each class are discussed below. For more complete descriptions, see the manual pages for each program. In all cases, "[ ]" indicates an optional argument or flag, "..." indicates additional optional arguments in the same form as previous arguments, and "|" indicates alternative options.

3.1. Selectors

Selectors extract a subset of information from the input Candis file and transfer it to the output file. There are numerous selector programs.

Cdfextr [-ps] entry1 entry2 ... < infile > outfile: This program passes only the indicated parameters (-p), static fields (-s), or variable fields (no option flag) from the input to the output file.

Cdfrdim dimension low [high] < infile > outfile: This program reduces the dimensionality of the output file by averaging the specified dimension over the range [low, high]. If the "high" argument is missing, it is assumed to take the value of "low", and the operation reduces to the extraction of a subspace of the input file.

Cdfwindow dim1 low1 high1 dim2 low2 high2 ... < infile > outfile: This program passes only that region of the input file specified by the range on each dimension. The entire ranges of dimensions not specified on the command line are passed. The program thus "windows" regions of interest.

Cdfisocut -t|-b dimension test_field test_value < infile > outfile: This program reduces the dimensionality of a file by extracting field values of all fields on the subspace defined by "test_field = test_value". The dimension indicated is the one eliminated. The flag indicates whether the search for equality is from the top (-t) or bottom (-b) of the indicated dimension.

Cdfocut x y x_val y_val theta xplow xphigh [u v] < infile > outfile: This program takes a cut through the space defined by the dimensions of the input file at an angle "theta" to the "x" "y" plane. "x" and "y" are the names of two dimensions. The cut passes through the point "x_val", "y_val", and extends from "xplow" to "xphigh" along the cut. Optionally, the components of a vector, "u", "v", are rotated so they are respectively parallel and perpendicular to the cut direction.

Cdfthin dimension1 i1 ... < infile > outfile: This program thins out data in the directions specified by the dimensions. Every "i1"th point is retained for dimension1, etc.

Cdfdefint dimension < infile > outfile: This program integrates all fields with dimensionality "dimension" in the specified direction, thus reducing the dimensionality of the output file by one.

Cdftsel record_field beginning_value ending_value < infile > outfile: This program keeps only variable slices with a scalar field "record_field" with a value in the range ["beginning_value", "ending_value"]. The values of the field don't have to be ordered monotonically through the file.

Cdfuniq < infile > outfile: This program looks for fields in which all values are identical. If found any are found, they are reduced to scalar fields. This program only works on files with one variable slice.

Cdfcat record_field beginning_value ending_value max < infile > outfile: This program turns a file with multiple variable slices into a file with a single variable slice and increased dimensionality. "Record_field" is a scalar field that becomes the index field for the new dimension in the output file. Only those slices with values of this field in the range ["beginning_value", "ending_value"] are incorporated into the new file. The value of "record_field" must increase monotonically through the file. "Max" should be greater than or equal to the maximum number of variable slices expected.

3.2. Constructors

Constructors are programs that combine two or more Candis files into a single output file. Only two constructors currently exist.

Cdfcatf infile1 infile2 ... > outfile: This program copies the first input file to the output. The variable slices of the other input files are then appended to the output file. This procedure only works if the input files are homogeneous in the sense that the variable slices have the same field names and sizes.

Cdfmerge infile1 suffix1 infile2 suffix2 ... > outfile: This program merges heterogeneous input files into a single output file. The process only works if all input files have only a single variable slice. All static fields are put in a single static slice, and all variable fields are put into a single variable slice. The specified suffixes are added to the field names of the associated input files in order to avoid name clashes. If, in spite of everything, a name clash occurs, special rules are followed, which are discussed in detail in the manual page for this program.

3.3. Math operators

Math operators perform mathematical transformations on fields in the input Candis file, possibly creating new fields in the output file.

Cdfmath 'expression' < infile > outfile: This program does point-by-point mathematical operations. The results are placed either in an existing field or in a new field. The expression is in reverse Polish notation, and should be quoted to protect it from the shell, as many math operators are also shell metacharacters. A typical expression might be 'a b + 2 * sin c ='. This means 'c = sin(2*(a + b))' in more conventional notation. The fields "a" and "b" must exist in the input file, but "c" may be a new field. The operation is repeated for each point in the subspace defined by the union of the dimensions of all specified fields. If "c" is new, its dimensionality is the union of the dimensions of the input fields.

Cdforder indexfield1 indexfield2 ... < infile > outfile: This program reorders index fields in the static slice in the indicated order. It also rearranges the order of elements in fields so that the dimensions in each field are ordered as indicated. There is no requirement that the order of index fields must be the same as the order of dimensions, or even that the ordering of dimensions in different fields must be consistent in Candis. However, cdfmath (see above) fails if these conditions are not met. Cdforder is a way of fixing non-conforming Candis files.

Cdfderiv derived_field input_field dimension < infile > outfile: This program takes the partial derivative of "input_field" with respect to the indicated dimension. The results are put in a new field, specified as "derived_field" on the command line.

Cdfsmooth dimension1 lambda1 ... < infile > outfile: This program applies a low pass filter to all fields over the dimension indicated. The half-amplitude wavelength is 2*pi*lambda. Lambda is called the smoothing length. Smoothing over multiple dimensions may be accomplished in the same invocation of cdfsmooth by specifying additional dimension-smoothing length pairs on the command line.

Cdfthresh 'logical_expression' ... < infile > outfile: This program evaluates one or more logical expressions on a point by point basis. For each point at which one or more of the logical expressions is false, all fields in the variable slice are set to the bad data value. Each logical expression is of the form 'field_name > value' or 'field_name < value'. "Value" may be an actual number or the word badlim. 'Field_name > badlim' means that the expression is true at points where the specified field contains bad data, whereas 'field_name < badlim' means the expression is false under these circumstances. Cdfthresh allows one to remove data at points where a test field doesn't meet some data quality criterion.

3.4. Display utilities

Display utilities allow one to look at data from a Candis file in various ways. The graphics display routines all use the portable graphics system, Pgraf.

Cdflook < infile: This program lists the header of the input file on the standard input. It then lists the number of elements in each slice and the values of any scalar fields.

Cdfplot 'command_list' ... < infile: This program makes a Pgraf metacode file named "pgraf.out" according to the commands on the command line, taking the input from the input file. The graphics can then be displayed by invoking the Pgraf metacode interpreter appropriate to the available graphics hardware. Typical usage might be "cdfplot 'w,5,1,c' < infile && pgtek", where the two-dimensional field "w" is contoured at intervals of 5 units. The graphics is displayed on a Tektronix-compatible graphics terminal. The "&&" means that pgtek only executes if cdfplot succeeds. For details of usage, see the manual page on cdfplot.

Cdfuaplot pt t dp pw u v ['comment'] < infile: This program creates a Pgraf metacode file that displays a skew-T log-p chart of an atmospheric sounding. Also displayed are wind component profiles. The field "pt" gives the pressure levels for the thermodynamic fields temperature, "t", and dewpoint, "dp". The field "pw" gives the pressure levels for the westerly and southerly wind fields, "u" and "v". An optional comment is displayed on the output plot.

3.5. Input-output utilities

These utilities assist in getting data in and out of a UNIX computer system.

Rtape /dev/raw_tape_device > outfile: This program reads a tape on a UNIX system. The tape drive is specified by its special raw file name, e. g., "/dev/rmt0". Rtape reads one physical tape record at a time and counts the number of bytes in the record. This number is written to the output file as an 8 byte ASCII decimal integer. The tape record is then written to the output. In this manner information about the physical record structure on tape is retained. This is necessary for interpreting the data on some tapes. No assumptions are made about the record structure on the tape, i. e., any mix of record sizes may be read.

Wtape /dev/raw_tape_device < infile: This program recreates a tape written to disk using rtape.

Zin linename baudrate [error_file] > outfile: This program reads from a serial port on a UNIX system and sends the result to the output file. Eight bit bytes are read, and no input editing is performed. "Linename" is the special file name of the serial line, e. g., "/dev/ttya". "Baudrate" is the desired baud rate of the serial line, one of 300, 1200, 2400, 4800, or 9600. Error messages are sent to an optional error file. If this is not specified, they are sent to the standard error.

3.6. Translators

Translators convert files in some foreign format to Candis format, or vice versa.

Nwssa station_file < infile > outfile: This program converts National Weather Service surface observations (SAs) to Candis format, one variable slice per observation. Data on surface stations is obtained from the file specified on the command line. See the manual page for more information. This program doesn't extract all information, and breaks on mis-coded data.

Nwsua < infile > outfile: This program converts National Weather Service significant level upper air observations (TTBBs and PPBBs) to Candis format. See the manual page for more information.

Radcedric -r|-d [-s] < infile > outfile: This program converts CEDRIC files to Candis format (Mohr, Miller, Vaughan, and Frank, 1986). Either the -r or the -d flag must be specified.

The former indicates that the CEDRIC file was read from tape using the rtape utility, and thus has embedded record counts. The latter indicates that it was somehow produced without these record counts. The optional -s swaps bytes on all 16 bit integers. This is needed only if the CEDRIC file was created on a machine with different byte ordering than the machine executing radcedric. The two dimensional horizontal slices of CEDRIC are stacked into three dimensional fields in the Candis file.

Raduf < infile > outfile: This program converts radar Universal Format tapes (Barnes, 1980) to Candis format, one ray per variable slice. The Universal Format tape must have been transferred to disk using the rtape utility.

Rafread headerfile [datafile|-] > outfile: This program converts NCAR Research Aviation Facility GENPRO II tapes to Candis format. For each dataset GENPRO II produces a header file and a data file. If "-" is specified for the data file, rafread expects the data file on the standard input. If no data file is specified, an ASCII table containing decoded header information is written to the standard output. GENPRO II headers are hard to decode by computer, so this program may fail on new datasets with slightly different headers. Rafread expects files to have been transferred to disk using the rtape utility. The output file contains one second's worth of data per variable slice.

Uniget netcdf_file > outfile: This program converts a Unidata netCDF file to Candis format. All netCDF data types are converted to float.

Uniput netcdf_file < infile: This program converts a Candis file to Unidata netCDF format.

Cdftrans -f|-i|-a < infile > outfile: This program converts between float, int, and ascii forms of Candis files. The type of the input file is deduced automatically. The type of the output file is specified by the option flag, which may be -f (float), -i (int), or -a (ascii).

Select < infile: This is a lex program that reads the National Weather Service Domestic Data stream (typically piped from zin) and selects out various components, such as upper air data, selected surface stations, selected warnings and summaries, radar reports, etc. When data of a particular type appears, it is appended to the file defined for that type. Files are closed between appends. The user will desire to customize this program for his own use. The version given should help the user construct his own version.

3.7. Special purpose data converters

This diverse collection of programs performs operations on data known to be of a specific type, e. g., radar data, data from a National Weather Service feed, etc. Selected programs of this type are described very briefly here. See the manual pages for more detail.

Nwsuafix pres temp dewpt upres u v station_file < infile > outfile: This program accepts the output of nwsua and produces upper air soundings interpolated to a 50 mb grid.

Nwsgrid sname xname x0 dx nx yname y0 dy ny radius < infile > outfile: This program grids surface or upper air station data in latitude and longitude.

Nwsts refday u v year month day hour < infile > outfile: This program computes the Julian day from year, month, day, and hour, and includes this new scalar field in the output file. See also the utility cdfj in the manual pages.

Radcart x0 dx nx y0 dy ny zo dz nz x_origin y_origin z_origin < infile > outfile: This program interpolates radar data to a Cartesian grid. See the manual pages for more detail.

Radsynth minradars suffix1 suffix2 ... < infile > outfile: This program synthesizes Cartesian components of particle velocity, given the output of radcart for two or more Doppler radars. The two radcart output files should be merged into a single file using cdfmerge for input to radsynth.

Rafhirate tape_time_name < infile > outfile: This program reduces the rather complex output that rafread produces for high rate (> 1 Hz) data to a usable format.

4. Creating new Candis programs

In this section I describe how to write new Candis programs. A library of subroutines (libcdf.a) exists for accessing Candis files and performing various functions. These are described in detail in the manual page cdf3. To compile and link Candis programs, include a call to this library. In addition, include the file "cdfhdr.h", noting that cdfhdr.h itself includes nmimt-copyright.h. A typical compilation command line would look like

```
cc -Llibrary_directory -Iinclude_directory -o prog1 prog1.c -lcdf
```

with other possible libraries, such as the math library, being added if necessary.

Candis differs from what most people are used to doing in Fortran in that all data buffers are dynamically allocated. Fields are accessed by pointers to the beginning of each field. Multidimensional array indexing is slightly more difficult than when arrays are allocated statically. However, the increase in flexibility that results from dynamic allocation more than offsets this awkwardness.

Headers are set up by a sequence of calls to subroutines that define various elements of the header. All headers start as null headers, and each subsequent call adds a field definition, a parameter, or a comment line. When the header is complete, data buffers and field pointers are allocated.

In the following subsections various types of Candis programs are given in skeleton code form.

4.1. Create a new Candis file

In this program a new Candis file is created, field values are computed, and the file is written to the standard output. The program is couched in terms of a two dimensional, time dependent numerical model, but it can serve as a skeleton for any number of uses.

```
/* prog1.c -- This program provides a skeleton structure for a two dimensional
 * numerical model.  The x-z grid as well as the time step and the number
 * of time levels are obtained from the command line.  One time level
 * is presented per variable slice.  The two dimensional fields recorded
 * are u, w, and buoy.
 */

/* include statements */
#include <stdio.h>
#include <math.h>
#include "cdfhdr.h"

/* The following define can be used to access the elements of a 2-D array
   in a sensible way.  In Candis, one has the pointer to a field
   (say, u), which is defined in the following way: "float *u;"
   If this represents a 2-D nx by ny array, the (ix,iy)th element of this
   array can be accessed with the expression u[I(ix,iy)].  This is only
   slightly more verbose than the usual way of accessing multidimensional
   arrays in Fortran.  Generalization to 3 and 4 dimensional arrays
   is obvious.  For a one dimensional array (say float *x;), simply use x[ix].
   For a zero dimensional field (say, float *time;), just use *time. */
#define I(ix,iy) ((iy) + ny*(ix))

/* Variables having to do with Candis. */
char hbuff[HBMAX][LINE];        /* This is the buffer that contains the
                                header.  It consists of HBMAX lines
                                each LINE - 2 characters long.  The
                                number of lines is a conventional
                                maximum (= 300), but may be increased
                                or decreased if necessary.  By convention
                                LINE = 82.  These quantities are
                                defined in cdfhdr.h. */
char c1[LINE],c2[LINE];         /* These are two character buffers that
                                are used to construct entries for the
                                comment and parameter sections of the
                                header. */
float *sbuff,*vbuff;            /* Pointers to the static and variable
                                field buffers. */
long nsbuff,nvbuff;             /* Sizes (in elements) of static and
                                variable buffers */
float *u,*w,*buoy;              /* Pointers to the 2-D fields produced
                                by the numerical simulation. */
float *time;                    /* A zero dimensional variable field
                                pointer to represent time. */
float *x,*z;                    /* Index field pointers. */

/* Variables needed for the calculation. */
int ix,iz,itime;                /* Looping variables. */
int nx,nz,ntime;                /* Grid size and number of time levels. */
float dx,dz,dtime;              /* Grid dimensions and time step. */

/* Main program entry point. */
main(argc,argv)
int argc;
```

```
char *argv[];
{

/* Check command line arguments, print a usage statement, and exit if the
   number of arguments is incorrect. */
  if (argc != 7) {
    fprintf(stderr,"Usage: prog dx nx dz nz dtime ntime\n");
    exit(1);
  }

/* Otherwise get the values. */
  dx = atof(argv[1]);
  nx = atoi(argv[2]);
  dz = atof(argv[3]);
  nz = atoi(argv[4]);
  dtime = atof(argv[5]);
  ntime = atoi(argv[6]);

/* Make a null header in float format. */
  nullhdr(hbuff,HBMAX,"float");

/* Add a comment. */
  sprintf(c1,"%s: %s %s %s\n",argv[0],argv[1],argv[2],argv[3]);
  addcline(hbuff,HBMAX,c1);
  sprintf(c1,"  %s %s %s\n",argv[4],argv[5],argv[6]);
  addcline(hbuff,HBMAX,c1);

/* Add index parameters. */
  addpar(hbuff,HBMAX,"x0","0");
  addpar(hbuff,HBMAX,"z0","0");
  addpar(hbuff,HBMAX,"dx",argv[1]);
  addpar(hbuff,HBMAX,"dz",argv[3]);

/* Add bad data parameters (this isn't necessary if default values
   will do). */
  addpar(hbuff,HBMAX,"badlim","9.99e9");
  addpar(hbuff,HBMAX,"bad","1.e10");

/* Add index fields to static slice. */
  addfld(hbuff,HBMAX,'s',"x",1000.,0.,'s',1,"x",nx);
  addfld(hbuff,HBMAX,'s',"z",1000.,0.,'s',1,"z",nz);

/* Add time field to variable slice. */
  addfld(hbuff,HBMAX,'v',"time",1000.,0.,'s',0);

/* Add 2-D output fields. */
  addfld(hbuff,HBMAX,'v',"u",1000.,0.,'s',2,"x",nx,"z",nz);
  addfld(hbuff,HBMAX,'v',"w",1000.,0.,'s',2,"x",nx,"z",nz);
  addfld(hbuff,HBMAX,'v',"buoy",1000.,0.,'s',2,"x",nx,"z",nz);

/* The header is now complete.  Allocate space for slice buffers.
   Information comes from the completed header. */
  nsbuff = elemcnt(hbuff,HBMAX,'s');  /* Get size of static slice ... */
  sbuff = getbuff(nsbuff);            /* ... allocate the space. */
```

```
  nvbuff = elemcnt(hbuff,HBMAX,'v');  /* Same for variable slice. */
  vbuff = getbuff(nvbuff);

/* Allocate pointers to the variables. */
  x = getptr(hbuff,HBMAX,sbuff,'s','d',"x");    /* Goes in static slice. */
  z = getptr(hbuff,HBMAX,sbuff,'s','d',"z");
  time = getptr(hbuff,HBMAX,vbuff,'v','d',"time"); /* Goes in variable slice */
  u = getptr(hbuff,HBMAX,vbuff,'v','d',"u");
  w = getptr(hbuff,HBMAX,vbuff,'v','d',"w");
  buoy = getptr(hbuff,HBMAX,vbuff,'v','d',"buoy");

/* Fill in values of index fields. */
  for (ix = 0; ix < nx; ix++) x[ix] = ix*dx;
  for (iz = 0; iz < nz; iz++) z[iz] = iz*dz;

/* Write header to standard output. (If this were some other file, it
   would have to be opened first using fopen.) */
  puthdr(stdout,hbuff,HBMAX);

/* Write static slice to standard output. */
  putslice(stdout,nsbuff,sbuff);

/* Do numerical model initialization. */
  *time = 0.;
  initialize();

/* Write initial variable slice. */
  putslice(stdout,nvbuff,vbuff);

/* Loop on time. */
  for (itime = 1; itime < ntime; itime++) {
    *time = itime*dtime;

/* Do numerical time step. */
    stepit();

/* Write variable slice. */
    putslice(stdout,nvbuff,vbuff);

/* End of time loop. */
  }
}
initialize()
{
}
stepit()
{
}
```

All of the numerical work is contained in the subroutines "initialize" and "stepit", which are not included here since this is skeleton code. The "boiler plate" may seem excessive in this program, but the effort put in at this stage is amply rewarded during the analysis of the output.

4.2. Read an existing file about which something is known

In this section I present an example of a program that reads an existing Candis file. The program knows the name and dimensionality of each field, but obtains information on dimension sizes from the input file.

```
/* prog2.c -- This skeleton program reads a Candis file of known structure
 * and hands the information to a subroutine that does something with
 * the data.  Refer to prog1.c for more complete explanations of
 * code in common.
 */

#include <stdio.h>
#include <math.h>
#include "cdfhdr.h"

/* For use in accessing 2-D arrays. */
#define I(ix,iz) ((iz) + nz*(ix))

/* Candis stuff -- mostly the same as in prog1. */
char hbuff[HBMAX][LINE];
char c1[LINE];
float *sbuff,*vbuff;
long nsbuff,nvbuff;
float *u,*w,*buoy,*time,*x,*z;
float bad,badlim;                  /* These are the values of the bad
                                  data parameters.  They are obtained
                                  either from the parameter section
                                  of the input file, or if they don't
                                  exist there, from default values
                                  in cdfhdr.h. */
struct field *fp;                 /* This structure is defined in cdfhdr.h.
                                  It contains information about a field,
                                  and is returned by calls to getfld,
                                  seekfld, and getptr2. */
int nx,nz;                        /* Sizes of dimensions, to be determined
                                  from examining input file. */

main(argc,argv)
int argc;
char *argv[];
{

/* Check command line arguments. */
  ...

/* Read header from standard input. */
  gethdr(stdin,hbuff,HBMAX);

/* Check input file format to be sure that it is float. */
  if (getfmt(hbuff,HBMAX) != 'f') {
    fprintf(stderr,"prog2: Input file format must be float!\n");
    exit(1);
  }

/* Get values of bad data parameters -- use defaults BAD and BADLIM,
   defined in cdfhdr.h, if there are no bad data parameters defined.
   (OK is also defined in cdfhdr.h.) */
  if (seekpar(hbuff,HBMAX,"bad",c1) == OK) bad = atof(c1);
  else bad = BAD;
```

```
  if (seekpar(hbuff,HBMAX,"badlim",c1) == OK) badlim = atof(c1);
  else badlim = BADLIM;

/* Allocate slice buffers */
  nsbuff = elemcnt(hbuff,HBMAX,'s');
  sbuff = getbuff(nsbuff);
  nvbuff = elemcnt(hbuff,HBMAX,'v');
  vbuff = getbuff(nvbuff);

/* Get pointers to static fields -- use getptr2 so that the field structure
   is returned for each field -- this yields (among other things)
   dimension size information.  The "die" return is used, so if the
   desired field isn't in the input, getptr2 dies with an error
   message.  The sizes of the x and z dimensions are checked, and x and
   z are checked to see if they are really index fields.  Note that *fp
   is in static storage, and is hence overwritten by each new call to
   getptr2.  */
  x = getptr2(hbuff,HBMAX,sbuff,'s','d',"x",&fp);
  if ((fp->dim !=1) || (strcmp(fp->fname,fp->dname1) != 0)) {
    fprintf(stderr,"prog2: %s not an index field\n",fp->fname);
    exit(1);
  }
  nx = fp->dsize1;
  z = getptr2(hbuff,HBMAX,sbuff,'s','d',"z",&fp);
  if ((fp->dim !=1) || (strcmp(fp->fname,fp->dname1) != 0)) {
    fprintf(stderr,"prog2: %s not an index field\n",fp->fname);
    exit(1);
  }
  nz = fp->dsize1;

/* Get pointers to variable fields.  More consistency checks could be
   done here if desired. */
  time = getptr2(hbuff,HBMAX,vbuff,'v','d',"time",&fp);
  u = getptr2(hbuff,HBMAX,vbuff,'v','d',"u",&fp);
  w = getptr2(hbuff,HBMAX,vbuff,'v','d',"w",&fp);
  buoy = getptr2(hbuff,HBMAX,vbuff,'v','d',"buoy",&fp);

/* Read the static slice. */
  getslice(stdin,nsbuff,sbuff);

/* Read variable slices until end of file. */
  while (getslice(stdin,nvbuff,vbuff) != EOF) {

/* Do what needs to be done. */
    ...

/* End of slice read loop. */
  }
}
```

4.3. Read a Candis file, modify it, and write it

Reading a Candis file and then writing a modified version is the most common thing Candis programs do. Such programs are largely combinations of the above two programs. In general, of course, the input and output buffers, headers, and pointers will have to take on different names.

In constructing the new header from the old, several subroutines are of help, namely, copycmt, copypar, and copyfld. These respectively copy the entire comment, parameter, and static or variable field section from one header buffer to another. If this isn't desired, one parameter or field description at a time can be obtained from the input header buffer using getpar and getfld.

In most cases, shortcuts are available if the output file isn't too different from the input file. If the field and parameter definitions are all the same, then the two headers are the same, except that a line may be added to the comment section for the output. Thus, the same header, static slice, and variable slice buffers, as well as field pointers may be used. If one or more fields are added to the output, the same buffers can be used as well, as long as the following sequence of actions is followed: 1) Read the input header buffer and obtain the lengths of the input static and variable slices using elemcnt. 2) Use addfld to add the desired fields to the header buffer. 3) Obtain the sizes of the expanded slice buffers using elemcnt and allocate the buffers based on these sizes. 4) Compute pointers to the desired fields using getptr or getptr2 as before. 5) Read input slices using the input buffer sizes and write output slices using the output buffer sizes. This procedure is possible because new fields added using addfld are added to the end of the field description section, and hence to the end of the slice buffers. The addresses of existing fields in each slice are thus not disrupted.

4.4. Other considerations

General purpose Candis programs assume very little about the character of the input file. Thus, more checking has to be done to determine the nature of each field. The field structure, which is returned by getfld, seekfld, and getptr2, provides the number of dimensions of each field, the dimension names, the dimension sizes, and various other pieces of information. Thus, general purpose programs can be made to respond appropriately to any input file, albeit at the cost of considerable checking code. Note that getfld and seekfld return a pointer to a field structure. Getptr2's last argument is a pointer to a pointer to a field structure. (This is made necessary by the C language convention that passed arguments are read-only.) The actual structure in each case is stored in static memory, so that if the information is to be retained beyond a subsequent call to any of these routines it needs to be transferred out, e. g.,

```
struct field *field_pointer,field_buffer;
  ...
  field_pointer = seekfld(...);
  field_buffer = *field_pointer;
  ...
  field_pointer = seekfld(...);
  ...
```

or

```
struct field *field_pointer,field_buffer;
  ...
  field1 = getptr2(...,&field_pointer);
  field_buffer = *field_pointer);
  ...
  field2 = getptr2(...,&field_pointer);
  ...
```

It is possible to read and write files in non-float format using the routines gislice, gaslice, pislice, and paslice. The auxiliary routine getilist is useful when reading and writing files in integer format. All these routines work to and from float format images in memory. Thus, once an integer file (for instance) is read into memory using gislice, all the usual operations can be performed on it without further conversion. The header read and write routines are the same for non-float files. It is good practice after reading the header to check to see if the input file format is as expected.

When reading or writing Candis files from other than the standard input and output, the file needs to be opened, e. g., file_pointer = fopen(...)), for reading or writing beforehand. Then the various input-output calls can be made, with the "stream" argument set to the file pointer returned by fopen, e. g., gethdr(file_pointer,...). The standard error output should be reserved for error messages, and error exits should return a non-zero exit code, e. g., exit(1), so that shell scripts can determine whether the program failed or succeeded. Since a shell script can contain many programs, it is good practice to include the name of the failing program in any error messages, e. g., "cdfrdim: float format expected!". Another good practice is to make the program print out a usage statement when the structure of the argument list is incorrect. The verbosity of this statement can vary, but in general it should remind a person generally familar with the program how it is invoked. It shouldn't be too cryptic, but it shouldn't reproduce the manual page! The general form of a usage statement should be

```
Usage: program_name argument1 argument2 ...
   Optional additional explanation
```

Unless otherwise noted, Candis programs are expected to read a file on the standard input, and write a file on the standard output.

A subroutine useful for extracting information from fields in general purpose Candis programs is subspace1. This extracts data along a particular dimension of a field. For instance,

```
struct field *fp;
float *field1;
float element
long instance,start,incr,size;
long loop;
char dimname[LINE];
  ...

/* Get field pointer for field1. */
  field1 = getptr2(...,&fp);
  ...

/* Loop over possible penetrations through field in "dimname" direction. */
  instance = 0;
  while (subspace1(fp,dimname,instance++,&start,&incr,&size) != FAIL) {

/* For each penetration, access the field elements in that direction. */
    for (loop = 0; loop < size; loop++) {
      element = field1[start + incr*loop];
      ...
    }
  }
  ...
```

Thus, for example, if a field has dimensions x, y, and z, setting dimname to y would yield values of the field along the y axis. Different values of "instance" would give all possible combinations of x and z. The value of subspace1 becomes apparent when trying to access a field whose structure is not known beforehand.

5. References

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

Leserman, D. H., 1988: Feasibility of an intellegent user interface to Candis. Report to the Unidata Program Center, UCAR, Boulder, CO.

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. Ocean. Tech.,* **3**, 144-161

Raymond, D. J., 1988: A C language-based modular system for analyzing and displaying gridded numerical data. *J. Atmos. Oceanic Tech.,* **5**, 501-511.