NOAA Technical Memorandum ERL WPL-212

QUALITY CONTROL ALGORITHM FOR PROFILER MEASUREMENTS OF WINDS AND TEMPERATURES

Bob L. Weber
David B. Wuertz

Wave Propagation Laboratory
Boulder, Colorado
October 1991

UNITED STATES DEPARTMENT OF COMMERCE

Robert A. Mosbacher
Secretary

NATIONAL OCEANIC AND ATMOSPHERIC ADMINISTRATION

John A. Knauss
Under Secretary for Oceans and Atmosphere/Administrator

Environmental Research Laboratories

Joseph O. Fletcher
Director

NOTICE

Mention of a commercial company or product does not constitute an endorsement by NOAA/ERL. Use of information from this publication concerning proprietary products or the tests of such products for publicity or advertising purposes is not authorized.

For sale by the National Technical Information Service, 5285 Port Royal Road
Springfield, VA 22161

CONTENTS

Quality Control Algorithm for Profiler Measurements of Winds and Temperatures

Bob L. Weber and David B. Wuertz

ABSTRACT. This document describes a computer algorithm that was developed to control the quality of wind and RASS temperature measurements from remote sensing radars. It is a manual that gives detailed instructions on how to implement the computer program and then how to apply the algorithm to data analysis.

1. INTRODUCTION

This document describes a computer algorithm, called the continuity algorithm, that we have used to control the quality of wind and Radio Acoustic Sounding System (RASS) temperature measurements for the new NOAA profiling Doppler radars (Weber et al., 1990).

This document is intended for a narrow audience primarily of computer programmers and scientists who are interested in data quality control. It therefore contains details and technicalities that will be esoteric to the general reader, who would better benefit from demonstrations of this algorithm (Weber, 1991). The critical part is contained in Section 2, which explains how to implement and how to use the algorithm. Sections 3, 4, and 5 document the code and explain the concepts embodied in that code. These three sections, thus, will not be of interest to most readers.

The basic idea behind the algorithm is simple and has general application (Wuertz and Weber, 1989). That is, data are checked for continuity or consistency, e.g., over height and over time. In many cases, data found to be inconsistent (by the continuity algorithm) can be reliably eliminated from further consideration in data analysis. Since the data are compared against themselves, the continuity algorithm operates on the assumption that some of the data are good.

We must emphasize that consistency alone is insufficient for determining accuracy because some errors may be consistent over height and/or over time. Therefore, it is important to understand the method of measurement. With profilers, the measurements of winds and RASS temperatures begin with the measurements of Doppler frequency shifts for backscattered atmospheric signals in radar Doppler spectra. Wind measurements use clear-air backscattering from refractive index fluctuations caused by turbulence, but clouds and precipitation also cause backscattering (Gossard and Strauch, 1983; Strauch et al.,

1984). RASS temperature measurements use backscattering from acoustic waves (Strauch et al., 1988; May et al., 1990). The natural variability in these scattering mechanisms causes Doppler broadening of the atmospheric signals, limiting the precision of the radar measurements. But meteorological variability limits the accuracy (Weber and Wuertz, 1990). These limits on precision and accuracy are fundamental to the profiler instrument and consistency checks can do nothing to remove these limits.

There are other errors that may be continuous over height and over time that, thus, are not detectable by a consistency check. For example, Strauch et al. (1987) found that horizontal wind estimates contained errors when they were not corrected for the effects of large vertical motion. And more errors are introduced when the wrong vertical velocities are used to make those corrections. That can happen in convective precipitation (Wuertz et al., 1988) and in the presence of wave motion (Weber et al., 1991). It can also happen when ground clutter (even with clutter suppression) biases the vertical velocity estimates. When these errors are made consistently over height and time, a consistency check does not detect them.

2. APPLICATION

In this section, we describe how to use and how to control the algorithm. We recommend that, if nothing else, this section be read carefully in its entirety.

2.1. Implementation

We start at the beginning with the computer code, which is contained in three source files:

qc.f
qcio.f
qcwork.f

which were written using (nearly) standard FORTRAN. We assume that the user has access to these files and that he does not have to produce the code from the listings provided in this document. The user's driving program is compiled along with the sources in these files.

The first file, *qc.f*, contains the main routine *qc* (Section 3.1), which is accessed by the driving program using *call qc*. This file also contains all subroutines and function routines used by the main routine *qc*. Changes in these routines should be unnecessary unless the code is found to be incompatible with the user's compiler.

The second file, *qcio.f*, contains the common region *io* (Section 3.2), through which the driving program controls the algorithm (Section 2.2) and receives the results (Section 2.3). The only change that may be required in this file is made for the parameter *nmax* (Section 2.2).

The third file, *qcwork.f*, contains the common region *work* (Section 3.3); work arrays are used only internally by the algorithm. For computers with serious memory constraints, we suggest some changes that can be made in this file (see the end of Section 3.3).

2.2. Input and Control

The user inputs data and controls the algorithm by providing values for the following.

nmax
nxmax
n
nx
nmin
dy
gd
y
dx
x

All input and output variables are declared in the file *qcio.f*. Except for *nmax* and *nxmax*, all input control variables are established in the driving program.

nmax

The maximum number of data points *nmax* is set in a parameter statement in the file *qcio.f*. It is used to dimension the data arrays in the *io* common region and the work arrays in the *work* common region. It should be no larger than necessary in order to minimize memory requirements. It is set equal to the expected largest number of data points.

nxmax

The maximum number of independent variables *nxmax* is set in a parameter statement in the file *qcio.f*. For the present version of this algorithm, always set $nxmax = 2$ even if the data are one-dimensional. Future versions will allow other values.

n

The driver determines the actual total number n of data points being sent to the algorithm. Keep in mind that n must not exceed $nmax$. If aliased data (Section 2.4) are unfolded, n equals the total of the original plus the unfolded data.

nx

The number of independent coordinates for the data is indicated by nx, which must not exceed $nxmax$. For the present version of this algorithm, nx and $nxmax$ should always be kept at 2, because the algorithm uses planar interpolation for determining continuity. This is appropriate for profiler data that are distributed over height and over time. We plan a future version of the algorithm that uses an arbitrary number of dimensions nx. However, the present version will handle either one or two-dimensional data. With one-dimensional data, simply set the second independent coordinates to the same value for all data. Then the interpolation automatically reduces to linear.

$nmin$

Confidence in any measurement point is established by the number of other measurment points with which it exhibits continuity. The minimum number required for confidence is the quota $nmin$, the size of the smallest allowable pattern detected by the pattern recognition in this algorithm. Data in smaller patterns will be indicated as unreliable by the algorithm. This number is somewhat subjective, but the performance of the algorithm should not be very sensitive to its actual value. We have used values for $nmin$ as small as 4 and as large as 64, but it should never be less than 1. Start with $nmin = n/10$.

dy

The continuity of the measurement data is established by using some standard measurement interval dy, whose value represents a reasonable change in the measurements over the neighborhood interval dx. One should view dy/dx as the maximum allowable derivative for continuous data. Therefore, smaller values are more restrictive, flagging more data and making the measurements look smoother. Larger values allow more structure in the data. In the pattern recognition part of the algorithm (Section 4), the differences in the data values y for neighboring points are compared with dy. A pair of neighboring points whose difference is less than or equal to dy is said to be continuous, and thus they tend to fall in the same pattern. A pair of neighboring points whose difference is greater than dy is said to be discontinuous; therefore, they tend to fall in different patterns. In the quality control part of the algorithm (Section 5), each point may be compared with the value interpolated from neighboring data points using a least-squares planar interpolation. Then, a point is flagged when the difference between it and the interpolated value exceeds dy. The physical units of dy must be the same as those for the data y.

gd

Gross or obvious discontinuities can be readily identified using the gross difference gd, which must be larger than dy. We have typically used values of $gd = 8\ dy$. This parameter is used to minimize computation time by allowing entire patterns (e.g., of aliased data) to be flagged all at once instead of point by point. The physical units of gd must be the same as those for the data y.

y

The data (e.g., RASS temperatures, radial velocities, or wind components) are contained in the array

$$y(i) \qquad \text{for} \qquad i = 1, \ldots, n$$

The algorithm evaluates the consistency of these measurements, the dependent variables, over the independent variables x. The physical units for y, dy, and gd must be the same.

dx

The algorithm compares each point with its neighbors, where the neighborhood size is determined by

$$dx(jx) \qquad \text{for} \qquad jx = 1, \ldots, nx$$

which should exceed the minimum spacing (e.g., in height or in time) between data points. We typically allow at least two neighbors on either side in each dimension. When applying this algorithm to hourly wind measurements by the new NOAA Wind Profiler Network (reported at 250-m height intervals), we usually set $dx(1) = 500$ m and $dx(2) = 2$ h. The physical units for $dx(1)$ must be the same as those for $x(1...n)$ and the physical units for $dx(2)$ must be the same as those for $x(n+1...2n)$. Never set $dx = 0$ even for one dimensional data.

x

The independent variables are the coordinates

$$x(ix) \qquad \text{for} \qquad ix = i + n\,(\,jx - 1\,)$$
$$\text{where} \qquad jx = 1, \ldots, nx$$

whose quality is not questioned by the algorithm. For example, $x(1, \ldots, n)$ may be the time and $x(n+1, \ldots, 2n)$ may be the heights of the measurements $y(1, \ldots, n)$. The data need not be entered in any particular order so long as the corresponding x and y values have the same appropriate indices. For one-dimensional data, set all coordinates x for $jx = 2$ to the same value.

2.3. Output

The algorithm returns only one output result, indicating the quality (or confidence) for each data point $y(i)$,

$$qcy(i) \qquad \text{for} \qquad i = 1, \ldots, n$$

with values from 0 to 100, where lower values indicate greater consistency or quality. We recommend that, as a rule, data be retained when $qcy \leq 10$ and be rejected otherwise. However, more data may be rejected by lowering this cutoff below 10 or more data may be retained by raising this cutoff above 10. Lowering the cutoff requires greater consistency of the data. Raising the cutoff requires less.

2.4. Example

The following subroutine *example* is given the radial velocities vr and returns quality controls qr (0 for good and 1 for bad). The velocities are represented on a uniform grid over 36 heights and over 10 times. Thus, the height coordinates $x(1...n)$ and the time coordinates $x(n+1...2n)$ can be represented in terms of dimensionless indices. (See the first two *do loops*.) Of course, these coordinates need not be uniformly spaced. The height and time intervals $dx(1)$ and $dx(2)$ are assigned dimensionless values of 2, giving 24 neighbors for each point. We have been careful to set $nx = 2$. The radial velocities vr have values between $\pm vrn$, where $vrn = 24$ m s^{-1}. We set $dy = 3$ m s^{-1} and $gd = 12$ m s^{-1}, as fractions of vrn. The radial velocities vr are transfered to y in the first two *do loops*. In addition, since some of the larger radial velocities are suspected of being aliased, we unfold all velocities whose magnitudes are greater than gd and also include those unfolded values in y. Note that the original data and the unfolded data have the same height and time coordinates x. Note also that, without unfolded data, the number of data points $n = 360$. With unfolded data included, n is larger. Hence, the offset $nmax$ is used for the time coordinates x in the first two *do loops*. Later, in the third *do loop* the indices are shifted. This is important. Finally, $nmin$ is equated with one-tenth of the total number of points n. Then the subroutine qc is called.

```
subroutine example( vr, qr )
include 'qcio.f'
real vr(36,10), qr(36,10)
```

```
      vrn = 24
      dy = vrn / 8
      gd = vrn / 2
      nx = 2
      dx(1) = 2
      dx(2) = 2
      n = 0
      do 1 i = 1, 36
       do 2 j = 1, 10
        n = n + 1
        x(n) = i
        x(n+nmax) = j
        y(n) = vr(i,j)
        if( vr(i,j) .gt. + gd ) then
         n = n + 1
         x(n) = i
         x(n+nmax) = j
         y(n) = vr(i,j) - 2 * vrn
        else if( vr(i,j) .lt. - gd ) then
         n = n + 1
         x(n) = i
         x(n+nmax) = j
         y(n) = vr(i,j) + 2 * vrn
        end if
2      continue
1     continue
      do 3 i = 1, n
       x(i+n) = x(i+nmax)
3     continue
      nmin = n / 10
      call qc
      n = 0
      do 4 i = 1, 36
       do 5 j = 1, 10
        n = n + 1
        qr(i,j) = 1
        if( qcy(n) .le. 10 ) qr(i,j) = 0
        if( vr(i,j) .gt. + gd ) then
         n = n + 1
         if( qcy(n) .le. 10 ) then
          vr(i,j) = y(n)
          qr(i,j) = 0
         end if
```

```
        else if( vr(i,j) .lt. - gd ) then
         n = n + 1
         if( qcy(n) .le. 10 ) then
          vr(i,j) = y(n)
          qr(i,j) = 0
         end if
        end if
5       continue
4      continue
       return
       end
       include 'qc.f'
```

Upon returning from the subroutine *qc*, we use the quality controls *qcy* to flag some of the data and to replace aliased data with unfolded data. The last two *do loops* reverse the logic in first two *do loops*.

This example illustrates how aliased data can be easily and automatically unfolded. The gross parameter *gd* is used to quickly flag all aliased points (Section 5.1). In fact, this algorithm can be used to unfold aliased data without controlling the quality of other data by setting $dy = gd$. Normally, *dy* is set much smaller than *gd*. It is important to realize though that we can never unfold all the data using this method alone. If all the data are unfolded and included with the original data, then the algorithm has no valid basis for determining which data are to be retained. The algorithm attempts to keep the maximum number of data points that are continuous with one another. This is the basis on which the algorithm decides to keep some of the unfolded data and to reject some of the original aliased data. Those rejected aliased data will be discontinuous with nonaliased data, whereas their unfolded counterparts that are accepted will be continuous.

3. ALGORITHM

The reader need venture beyond this point only if driven by curiosity or implementation difficulties. This section describes the main routine, which serves as an outline of the algorithm (Section 3.1). This section also discusses memory requirements, since that can be the biggest limitation in the application of this algorithm (Sections 3.2 and 3.3). Finally, this section presents the continuity model, the fundamental idea behind the algorithm (Section 3.4). All subroutines used by the main routine are presented in Sections 4 and 5.

3.1. Main Subroutine

The main subroutine *qc* is contained in the file *qc.f* and is listed below.

```
      subroutine qc
      include 'qcio.f'
      include 'qcwork.f'
c     pattern recognition
      call neighbors
      call nodes
      call branches
      call grow
      call connections
      call patterns
c     quality control
      call cut
      call prune
      call trim
      call weed
      return
      end
```

This routine calls 10 subroutines whose functions are outlined below.

I. The pattern recognition is accomplished in the first six subroutines (Section4).

1. The *neighbors* subroutine (Section 4.1) identifies which points are within dx of each other (Section 2.2). It also determines which pairs of neighbors are connected within dy using the continuity model (Section 3.4.2).

2. The *nodes* subroutine (Section 4.2) identifies those points where branches are likely to merge and orders them based on the degree of branching.

3. The *branches* subroutine (Section 4.3) establishes initial pattern units that consist of points that are interconnected to all other neighbors in the same branch. Nodes are excluded from branches at this point.

4. The *grow* subroutine (Section 4.4) assimilates nodes into existing branches if they are connected to all neighbors belonging to those branches. Otherwise, new branches are created.

5. The *connections* subroutine (Section 4.5) computes the average connection between neighboring branches.

6. The *patterns* subroutine (Section 4.6) establishes patterns of branches, which are interconnected. Two branches can belong to the same pattern without being directly connected, but they cannot be grossly disconnected (gd).

II. The quality control is accomplished in the last four subroutines (Section 5).

1. The *cut* subroutine (Section 5.1) flags all points in branches that are grossly disconnected from the majority of the data. This step is intended to minimize time spent in the algorithm. Otherwise it is unnecessary because latter subroutines would flag these points one at a time.

2. The *prune* subroutine (Section 5.2) flags discontinuous points (section 5.5) one at a time in the smaller patterns next to the largest patterns. This is done because the largest patterns presumably are those in which we have the greatest confidence.

3. The *trim* subroutine (Section 5.3) flags discontinuous points (section 5.5) one at a time in the smaller patterns starting with the largest discontinuities.

4. The *weed* routine (Section 5.4) flags remnants of patterns that were reduced in size by steps 2 and 3.

Note that *include* statements are used here (and in all other subroutines) so that the contents of the files *qcio.f* (Section 3.2) and *qcwork.f* (Section 3.3) need not be duplicated many times. Most, but not all, FORTRAN compilers should accept these *include* statements.

3.2. Input/Output

The input to and output from the algorithm are done through the common region *io*, contained in the file *qcio.f*, which is listed below. The parameters *nmax* and *nxmax* (Section 2.2) are set here for purposes of dimensioning the data arrays and the work arrays (Section 3.3). All other variables in this common region are established by the driving program (Section 2.2). With $nmax = 1440$ and with 4-byte real variables, the data array y requires 5,760 bytes of memory. With 4-byte integer variables, the quality array *qcy* requires the same memory. With $nxmax = 2$, the array x requires twice that memory, bringing the total memory in this common region to over 23,000 bytes. That is still small compared with the work arrays in the next section. With some FORTRAN compilers, *qcy* can be declared a byte rather than an integer variable, saving some memory.

```
c     nmax = maximum number of points
c     nxmax = maximum dimension of independent variable
c     n = number of points
c     nx = dimension of independent variable
c     nmin = minimum number of points in pattern
c     qcy(i) = quality
c            = 0–10 : retain
c            = 11–100 : reject
```

```
c     dy = standard difference between points
c     gd = gross difference between points
c     y(i) = dependent variable ( i = 1, ... , n )
c     dx(jx) = maximum separation of points ( jx = 1, ... , nx )
c     x(ix) = independent variable ( ix = i + n ( jx-1 ) )
c     byte qcy
      integer qcy
      integer nmax, nxmax, n, nx, nmin
      real dy, gd, y, dx, x
      parameter ( nmax = 1440, nxmax = 2 )
      common /io/ n, nx, nmin, qcy(nmax),
     &            dy, gd, y(nmax), dx(nxmax), x(nxmax*nmax)
```

3.3. Work Space

The work arrays used internally by the algorithm are in the common region *work*, contained in the file *qcwork.f*, which is listed below:

```
c     nn(i) = number of neighbors
c     ni(i,j) = neighbors for i ( j = 1, ... , nn(i) )
c     ic(i,ii) = neighbor connection
c           = 0-10 : connected
c           = 11-100 : disconnected
c     no = number of nodes
c     io(i) = node number for i = 1, ... , n
c     ioi(j) = i ( node order, smallest first, j = 1, ... , no )
c     nb = number of branches
c     npb(ib) = number of points in branch
c     ib(i) = branch number for i
c           = 1 , ... , nb
c     ibi(ib(i),j) = i ( j = 1, ... , npb(i) )
c     iob(j) = ib ( branch order, largest first, j = 1, ... , nb )
c     np = number of patterns
c     npp(ip) = number of points in pattern
c     ip(i) = pattern number for i
c        = 1, ... , np
c     iop(j) = ip ( pattern order, largest first, j = 1, ... , np )
c     bc(i,ii) = branch connection
c          = 0-10 : connected
c          = 11-100 : disconnected
c     qbc(i) = branch quality
c          = 0-10 : retain
c          = 11-100 : reject
c     byte ic, bc, qbc
```

```
      integer ic, bc, qbc
      integer nn, ni, no, io, ioi, nb, npb, ib, ibi, iob
      integer np, npp, ip, iop, gqc
      common /work/
     &    nn(nmax), ni(nmax,nmax), ic(nmax,nmax), no, io(nmax),
     &    ioi(nmax), nb, npb(nmax), ib(nmax), ibi(nmax,nmax),
     &    iob(nmax), np, npp(nmax), ip(nmax), iop(nmax),
     &    bc(nmax,nmax), qbc(nmax), gqc
```

Most of the work arrays are used for pattern recognition, although that information is utilized also for quality control. The two-dimensional arrays require large computer memory when the number of points is large. If time is not an issue, the algorithm could be rewritten to use less memory by simply recomputing certain quantities every time they are needed. For the present, we merely note the memory requirements based on the value $nmax = 1440$ (Section 3.2). The largest memory requirement is for the two-dimensional arrays *ni*, *ic*, *ibi*, and *bc*. With 4-byte integer variables, these four arrays require more than 8 megabytes each. The reason we use these arrays is to minimize compute time in real-time applications. That memory is readily available on many virtual memory computers.

Where memory must be conserved, we offer these remedies:

(1) Keep *nmax* to a minimum. In our sample application, we used $nmax = 1440$ to allow all points $n = 720$ to be aliased. However, one should never attempt to unfold all of the data. (See the discussion at the end of Section 2.4.) Use a smaller *nmax* and check in the driving program to see if n exceeds that value. If *nmax* is reduced from 1440 to 1000, there is better than a factor of 2 memory reduction in the four two-dimensional arrays.

(2) On those machines that allow it, use byte (8-bit) variables instead of integer (32-bit) variables for *qbc*, *ic*, and *bc*. For example, we developed this algorithm on a Digital Equipment Corporation (DEC) MicroVAX III using the nonstandard VAX VMS FORTRAN. We used byte variables for the two-dimensional arrays *ic* and *bc* and the one-dimensional arrays *qbc* and *qcy* (Section 3.2), since their values can be stored in 1 byte. In this way, the memory requirement for those arrays is reduced by a factor of 4.

(3) Reduce the size of the second dimension of *ni*, i.e., use $ni(nmax, mmax)$, where *mmax* is the expected number of neighbors for each point. We like to use about 2 neighbors on either side in each dimension ($nx = 2$), giving about 25 neighbors. Therefore, unless the neighborhood size *dx* is so large as to include all points n (which should not be done!), $mmax = 144$ should be sufficient. This gives a factor of 10 memory reduction. A word of caution: the code contains no checks for memory violations.

(4) Reduce the size of the first one dimension of $ibi(mmax, nmax)$, where $mmax$ is the expected number of branches. That number is larger for smaller dy and smaller for larger dy. If we use $mmax = 288$, there is a factor of 5 memory reduction. A word of caution: the code contains no checks for memory violations, so do not make this change casually.

In future versions of this algorithm, we shall pay closer attention to memory reduction techniques. In developing the present version of this algorithm, we emphasized features in this order: functionality, speed, standards, and memory.

3.4. Continuity Model

We present the continuity model because a competent user may wish to substitute another model. This is the way in which the user can most easily and most effectively change the performance of the algorithm. However, we caution the user so inclined to do two things before making changes. (1) Carefully examine the simple model presented and (2) thoroughly test the algorithm using this model.

This algorithm uses two models. One continuity model is used to identify continuous and discontinuous points and another model is used in the pattern recognition. Both are contained in the following function routine:

```
integer function link( y1, y2, dy, model )
yd = abs( y2 - y1 )
rd = 10 * ( yd / dy )
if( model .eq. 1 ) then
 ym = min( abs( y1 ), abs( y2 ) )
 if( yd .lt. ym ) rd = rd * ( yd / ym )
end if
link = min( 100., rd )
return
end
```

Continuity information is stored in the *link* parameter, whose values are restricted to be between 0 and 100 so that they can be stored in nonstandard byte arrays if desired in order to preserve memory (Section 3.3). Values from 0 to 10 are used to indicate continuity and values from 11 to 100 indicate discontinuities in varying degrees. Values larger than 100 are truncated to 100.

3.4.1. Point continuity

The continuity model for identifying continuous and discontinuous points is straightforward. That is, the actual data value y at some location is compared with its value y_i interpolated from neighboring points (Section 5.4). If the difference $|y - y_i|$ is less than or

equal to the standard interval dy, then the point is said to be continuous. If this difference is greater than dy, then the point is said to be discontinuous. Remember that the user arbitrarily sets the standard value dy and in this way determines which points are identified as continuous. However, the interpolation makes this model insensitive to the actual value of dy when the data are continuous.

3.4.2. Pattern continuity

Pattern recognition (Section 4.) starts with the comparisons of all pairs of neighboring points. If those comparisons used the same model as that used in identifying discontinuities, then the pattern recognition would be too sensitive to the value of dy set by the user. For example, this can cause large wind velocities to be placed in different patterns when their differences exceed dy even though those differences represent only small fractional changes. Hence, with this model, two neighboring points with values y_1 and y_2 are said to be continuously connected if $|y_2 - y_1|^2$ is less than or equal to $dy\ y_m$, where y_m is the minimum of $|y_1|$ and $|y_2|$, but it is never smaller than the difference $|y_2 - y_1|$. Thus, this model is not as restrictive as the other model.

Consider two radial velocities of 20 and 22 m s^{-1} and a standard interval of $dy = 1$ m s^{-1}. If we used the first model, then $link = 20$ and the algorithm would treat them as if they were disconnected and in different patterns. But the second model gives $link = 2$, and the algorithm will treat them as connected in the same pattern. Then consider two other radial velocities of 2 and 4 m s^{-1}, whose absolute difference is the same as the previous example. Both models give $link = 20$, and the algorithm will treat them as if they are disconnected and in different patterns. However, the fact that two measurements are disconnected does not mean that they are automatically discarded as discontinuous. To be identified as discontinuous, a point must not be linked with its interpolated value.

4. PATTERN RECOGNITION

This section describes the subroutines that accomplish pattern recognition. Patterns are recognized by identifying subsets of the data that are continuously connected in the sense of the continuity model (Section 3.4). We define two points to be continuously connected if their derivative is less than or equal to dy/dx, where dy and dx are parameters input by the user (Section 2.2).

4.1. Neighborhood

Only neighbors are checked for continuity. Two points are considered to be neighbors when the differences in their independent coordinates x are less than the intervals dx set by the user. The following subroutine stores the neighborhood information for each point in the number of neighbors nn and the list of neighbors ni:

```
      subroutine neighbors
      include 'qcio.f'
      include 'qcwork.f'
      integer link
      model = 1
      do 1 i = 1, n
       nn(i) = 0
       ic(i,i) = 0
       qcy(i) = 0
1      continue
      do 2 i = 1, n-1
       do 3 ii = i+1, n
       ic(i,ii) = -111
       ic(ii,i) = -111
        do 4 jx = 1, nx
         ix = i + n * (jx - 1)
         iix = ii + n * (jx - 1)
         if ( abs( x(ix) - x(iix) ) .gt. dx(jx) ) go to 3
4       continue
        nn(i) = nn(i) + 1
        nn(ii) = nn(ii) + 1
        ni(i,nn(i)) = ii
        ni(ii,nn(ii)) = i
        ic(i,ii) = link( y(i), y(ii), dy, model )
        ic(ii,i) = ic(i,ii)
3      continue
2     continue
      return
      end
```

This routine also stores continuity or *link* (Section 3.4) information in the *ic* array. For nonneighbors, *ic*=−111. For neighbors, *ic* has values between 0 and 100, which are within the limits −128 and +127 for integers that can be stored in a byte variable (nonstandard) if desired in order to reduce memory requirements. Smaller *ic* values indicate greater continuity or connectivity.

Finally, the quality controls *qcy* are initialized. Later, values ≤ 10 are used to indicate acceptable, high-quality points and values from 11 to 100 are used to indicate unacceptable, low-quality points, with higher values meaning poorer quality.

4.2. Nodes

The starting points for pattern recognition are the points (nodes) where branches meet or join. The subroutine below define nodes to be points that are connected to two

or more points that are themselves disconnected from each other:

```
      subroutine nodes
      include 'qcio.f'
      include 'qcwork.f'
      integer link
      model = 1
      no = 0
      do 1 i = 1, n
       ib(i) = 0
       io(i) = 0
       do 2 j = 1, nn(i)-1
        ii = ni(i,j)
        if( ic(i,ii) .gt. 10 ) go to 2
        do 3 k = j+1, nn(i)
         iii = ni(i,k)
         if( ic(i,iii) .gt. 10 ) go to 3
         if( ic(ii,iii) .eq. -111 ) then
          if( link( y(ii), y(iii), dy, model ) .le. 10 ) go to 3
         else
          if( ic(ii,iii) .le. 10 ) go to 3
         end if
         io(i) = io(i) + 1
3       continue
2      continue
       if( io(i) .eq. 0 ) go to 1
       no = no + 1
       ioi(no) = i
       jo = no
       do 4 ko = no -1, 1, -1
        if( io(ioi(jo)) .ge. io(ioi(ko)) ) go to 1
        ioi(jo) = ioi(ko)
        ioi(ko) = i
        jo = ko
4      continue
1     continue
      return
      end
```

Node information is stored in *io*. Points that are not nodes have $io = 0$. These non-nodes form the core of branches (Section 4.3), but initially the branch indicator $ib = 0$ for all points. Nodes have values $io \geq 1$, with larger values indicating greater branching or more structure. This subroutine orders the nodes according to the number of possible branches diverging from them. The order is a measure of local pattern complexity. Higher

orders indicate the convergence of larger numbers of branches. The order is stored in *ioi*, with points having smaller *io* values entering first. Finally, the number of nodes is *no*. If *no* is small compared with the total number of points *n*, then either the data are very smooth or the standard interval *dy* is too large compared with the variance in the data values. On the other hand, if *dy* is set too small, then *no* will be large, approaching the limit *n*. Near that limit, pattern recognition is difficult.

4.3. Branches

If one thinks of a pattern as a tree, then one can identify branches in each pattern. Branches are initially composed only of points that are connected with all neighboring points of the same branch. As a result, nodes are initially excluded from any branch. Later (Section 4.4), nodes are assimilated into existing branches or they become seeds for new branches. Initially branches identify the most recognizable features in patterns. Then every point in a pattern will belong to a branch or it will be a node. A pattern may be composed of a number of branches and nodes, but no branch and no node can belong to more than one pattern. The number of branches is *nb*, the number of points in a branch is *npb*, the number of the branch to which a point belongs is *ib*, and the list of branch members is contained in *ibi*. The branches are ordered *iob* according to their size, from largest to smallest. The branches subroutine is as follows:

```
      subroutine branches
      include 'qcio.f'
      include 'qcwork.f'
      nb = 0
      do 1 i = 1, n
       if( io(i) .gt. 0 ) go to 1
       if( ib(i) .gt. 0 ) go to 1
       nb = nb + 1
       ib(i) = nb
       npb(nb) = 1
       ibi(nb,1) = i
       lpb = 0
2      mpb = lpb + 1
       lpb = npb(nb)
       do 3 j = mpb, lpb
        ii = ibi(nb,j)
        do 4 jj = 1, nn(ii)
         iii = ni(ii,jj)
         if( io(iii) .gt. 0 ) go to 4
         if( ib(iii) .gt. 0 ) go to 4
         do 5 jjj = 1, nn(iii)
          iv = ni(iii,jjj)
          if( ib(iv) .ne. nb ) go to 5
```

```
        if( ic(iv,iii) .gt. 10 ) go to 4
5       continue
        ib(iii) = nb
        npb(nb) = npb(nb) + 1
        ibi(nb,npb(nb)) = iii
4       continue
3       continue
      if( npb(nb) .gt. lpb ) go to 2
      job = nb
      iob(job) = nb
      do 6 kob = nb - 1, 1, -1
       if( npb(iob(job)) .le. npb(iob(kob)) ) go to 1
       iob(job) = iob(kob)
       iob(kob) = nb
       job = kob
6     continue
1     continue
      return
      end
```

4.4. Grow Branches

If *dy* is sufficiently small, then the branches will all be small and there will be a large number of nodes. This situation is not conducive to clearly recognizing patterns. Therefore, branches are allowed to grow, assimilating nodes. Care must be exercised in this process, though, because it may not always be obvious which node should be attached to which branch. The routine starts with the lowest order nodes because they presumably are in regions with less branching. Furthermore, a node can become a member of a branch only if it is connected to every one of its neighbors that belongs to that branch. When a node has more than one branch for which it qualifies, then it goes to the branch with the most connections. New branches are created for any remaining nodes that cannot qualify for membership in existing branches according to the above criteria. The following is used to grow branches:

```
      subroutine grow
      include 'qcio.f'
      include 'qcwork.f'
1     ifirst = 0
      do 2 jo = 1, no
       i = ioi(jo)
       if( ib(i) .gt. 0 ) go to 2
       if( ifirst .eq. 0 ) ifirst = i
       jobmax = 0
```

```
      jbmax = 0
      ncmax = 0
      do 3 job = 1, nb
       jb = iob(job)
       nc = 0
       do 4 j = 1, nn(i)
        ii = ni(i,j)
        if( ib(ii) .ne. jb ) go to 4
        if( ic(i,ii) .gt. 10 ) go to 3
        nc = nc + 1
4      continue
       if( nc .le. ncmax ) go to 3
       jobmax = job
       jbmax = jb
       ncmax = nc
3     continue
      if( jobmax .eq. 0 ) go to 2
      job = jobmax
      jb = jbmax
      ib(i) = jb
      npb(jb) = npb(jb) + 1
      ibi(jb,npb(jb)) = i
      iob(job) = jb
      do 5 kob = jobmax - 1, 1, -1
       if( npb(iob(job)) .le. npb(iob(kob)) ) go to 2
       iob(job) = iob(kob)
       iob(kob) = jb
       job = kob
5     continue
2    continue
     if( ifirst .gt. 0 ) then
      i = ifirst
      nb = nb + 1
      ib(i) = nb
      npb(nb) = 1
      ibi(nb,1) = i
      iob(nb) = nb
      go to 1
     end if
     return
     end
```

4.5. Branch Connections

When branch growth is completed, the connection between branches is computed as the average connection between neighbors in different branches. The link or bond between pairs of individual points was important in the recognition of branches. The link or bond between individual branches is important in pattern recognition. The following subroutine computes branch connections and initializes the quality *qbc* of each branch:

```
      subroutine connections
      include 'qcio.f'
      include 'qcwork.f'
      do 1 jb = 1, nb
       bc(jb,jb) = 0
       qbc(jb) = 0
1     continue
      do 2 jb = 1, nb-1
       do 3 kb = jb+1, nb
        bc(jb,kb) = -111
        bc(kb,jb) = -111
        nd = 0
        yd = 0.0
        do 4 j = 1, npb(jb)
         i = ibi(jb,j)
         do 5 k = 1, npb(kb)
          ii = ibi(kb,k)
          if( ic(i,ii) .lt. -111 ) go to 5
          nd = nd + 1
          yd = yd + ic(i,ii)
5        continue
4       continue
        if( nd .eq. 0 ) go to 3
        bc(jb,kb) = yd / float( nd )
        bc(kb,jb) = bc(jb,kb)
3      continue
2     continue
      return
      end
```

4.6. Branch Patterns

Patterns are defined here to be the set of all branches that are directly or indirectly connected to one another. Neighboring connected branches (with *bc* less than or equal to 10) are directly connected and, hence, in the same pattern. But neighboring branches that are disconnected from one another may also be in the same pattern if they are connected

to other branches that are in that pattern. However, neighboring branches with gross connections greater than *gd* cannot belong to the same pattern. Thus, patterns as defined here can be quite convoluted. This subroutine stores the pattern information in the number of patterns *np*, the number of points in a pattern *npp*, the number *ip* of the pattern to which a branch belongs; the patterns are ordered *iop* according to size, with the largest having lowest order. The subroutine follows:

```
      subroutine patterns
      include 'qcio.f'
      include 'qcwork.f'
      logical find, fail
      gqc = min( 100., 10 * gd / dy )
      np = 0
      do 1 jb = 1, nb
       ip(jb) = 0
       npp(jb) = 0
1     continue
      do 2 job = 1, nb
       jb = iob(job)
       if( ip(jb) .gt. 0 ) go to 2
       np = np + 1
       ip(jb) = np
       npp(np) = npb(jb)
3      find = .false.
       do 4 kob = 1, nb
        kb = iob(kob)
        if( ip(kb) .gt. 0 ) go to 4
        fail = .true.
        do 5 lob = 1, nb
         lb = iob(lob)
         if( ip(lb) .ne. np ) go to 5
         if( bc(lb,kb) .eq. -111 ) go to 5
         if( bc(lb,kb) .gt. gqc ) go to 4
         if( bc(lb,kb) .gt. 10 ) go to 5
         fail = .false.
5       continue
        if( fail ) go to 4
        find = .true.
        ip(kb) = np
        npp(np) = npp(np) + npb(kb)
4      continue
       if( find ) go to 3
       iop(np) = np
```

```
      jop = np
      do 6 kop = np - 1, 1, -1
       if( npp(iop(jop)) .le. npp(iop(kop)) ) go to 2
       iop(jop) = iop(kop)
       iop(kop) = np
       jop = kop
6     continue
2     continue
      return
      end
```

5. QUALITY CONTROL

This section describes those subroutines that assign quality (or confidence) to each data point. Several steps (subroutines) are involved because quality depends on a number of factors. The first step flags entire branches that are found to be clearly inconsistent with the majority of data (Section 5.1). The next two steps are more cautious, flagging one point at a time (Sections 5.2 and 5.3). If the first step (Section 5.1) were not taken, then these two steps would flag points, only more slowly one at a time. That first step is a time saver because it flags entire branches all at once. Finally, insignificantly small branches are flagged (Section 5.3).

5.1. Cutting Branches

When two branches in two different patterns exhibit obviously large discontinuities, with branch connection *bc* larger than the gross value *gqc*, one or the other of the two branches is assigned poor quality. The gross value *gqc* is derived from the gross difference *gd* (Section 2.1) in the *patterns* subroutine (Section 4.6) and generally has a value much larger than 10. All points in the branch of the smaller pattern are assigned poor quality equal to the branch connection *bc*. The following is used for cutting branches.

```
      subroutine cut
      include 'qcio.f'
      include 'qcwork.f'
      logical choose(2)
      integer jb(2), jp(2)
1     do 2 jop = 1, np-1
       jp(1) = iop(jop)
       if( npp(jp(1)) .eq. 0 ) go to 2
       do 3 kop = jop+1, np
        jp(2) = iop(kop)
        if( npp(jp(2)) .eq. 0 ) go to 3
```

```
      do 4 job = 1, nb
       jb(1) = iob(job)
       if( ip(jb(1)) .ne. jp(1) ) go to 4
       if( qbc(jb(1)) .gt. gqc ) go to 4
       do 5 kob = 1, nb
        jb(2) = iob(kob)
        if( ip(jb(2)) .ne. jp(2) ) go to 5
        if( qbc(jb(2)) .gt. gqc ) go to 5
        if( bc(jb(1),jb(2)) .eq. -111 ) go to 5
        if( bc(jb(1),jb(2)) .le. gqc ) go to 5
        if( npp(jp(1)) .eq. npp(jp(2)) ) then
         if( npb(jb(1)) .lt. npb(jb(2)) ) then
          choose(1) = .true.
          choose(2) = .false.
         else if( npb(jb(1)) .eq. npb(jb(2)) ) then
          choose(1) = .true.
          choose(2) = .true.
         else if( npb(jb(1)) .gt. npb(jb(2)) ) then
          choose(1) = .false.
          choose(2) = .true.
         end if
        else if( npp(jp(1)) .gt. npp(jp(2)) ) then
         choose(1) = .false.
         choose(2) = .true.
        end if
        do 6 k = 1, 2
         if( .not. choose(k) ) go to 6
         qbc(jb(k)) = bc(jb(1),jb(2))
         do 7 j = 1, npb(jb(k))
          qcy(ibi(jb(k),j)) = qbc(jb(k))
7        continue
         npp(jp(k)) = npp(jp(k)) - npb(jb(k))
6       continue
        do 8 lop = jop, np-1
         do 9 mop = lop+1, np
          if( npp(iop(mop)) .le. npp(iop(lop)) ) go to 9
          lp = iop(lop)
          iop(lop) = iop(mop)
          iop(mop) = lp
9         continue
8        continue
        go to 1
5       continue
```

```
4       continue
3      continue
2     continue
      return
      end
```

5.2. Pruning Branches

Only the *cut* subroutine (Section 5.1) controls the quality of entire branches, marking all points in branches with the same quality. The next two subroutines (this section and Section 5.3) look for point discontinuities (Section 5.5) in a special order. The present *prune* subroutine identifies discontinuities only in smaller patterns next to larger patterns. It will never identify a discontinuity in the largest pattern unless there is another pattern of the same size. Therefore, this subroutine attempts to preserve the largest patterns. The following gives this subroutine:

```
      subroutine prune
      include 'qcio.f'
      include 'qcwork.f'
      logical choose(2)
      integer jb(2), jp(2)
      do 1 jop = 1, np-1
       jp(1) = iop(jop)
       if( npp(jp(1)) .eq. 0 ) go to 1
       do 2 kop = jop+1, np
        jp(2) = iop(kop)
        if( npp(jp(2)) .eq. 0 ) go to 2
        do 3 job = 1, nb
         jb(1) = iob(job)
         if( ip(jb(1)) .ne. jp(1) ) go to 3
         if( qbc(jb(1)) .gt. gqc ) go to 3
         do 4 kob = 1, nb
          jb(2) = iob(kob)
          if( ip(jb(2)) .ne. jp(2) ) go to 4
          if( qbc(jb(2)) .gt. gqc ) go to 4
          if( bc(jb(1),jb(2)) .eq. -111 ) go to 4
          if( npp(jp(1)) .eq. npp(jp(2)) ) then
           if( npb(jb(1)) .lt. npb(jb(2)) ) then
            choose(1) = .true.
            choose(2) = .false.
           else if( npb(jb(1)) .eq. npb(jb(2)) ) then
            choose(1) = .true.
            choose(2) = .true.
           else if( npb(jb(1)) .gt. npb(jb(2)) ) then
```

```
          choose(1) = .false.
          choose(2) = .true.
         end if
        else if( npp(jp(1)) .gt. npp(jp(2)) ) then
         choose(1) = .false.
         choose(2) = .true.
        end if
        do 5 j = 1, 2
         if( choose(j) ) call discontinuities( jb(j), jb(3-j) )
5       continue
4      continue
3     continue
2    continue
1   continue
    return
    end
```

5.3. Trimming Branches

This subroutine looks for discontinuous points in the branch that exhibits the worst connection (i.e., with the largest *bc* value) with an adjacent larger branch. This procedure is repeated for all branches except the largest branch in the largest pattern.

```
    subroutine trim
    include 'qcio.f'
    include 'qcwork.f'
    logical choose(2)
    integer jb(2), jbmax(2), bcmax
1   jbmax(1) = 0
    jbmax(2) = 0
    bcmax = 0
    do 2 job = 1, nb-1
     jb(1) = iob(job)
     if( qbc(jb(1)) .gt. gqc ) go to 2
     do 3 kob = job+1, nb
      jb(2) = iob(kob)
      if( qbc(jb(2)) .gt. gqc ) go to 3
      if( bc(jb(1),jb(2)) .eq. -111 ) go to 3
      if( bc(jb(1),jb(2)) .le. bcmax ) go to 3
      jbmax(1) = jb(1)
      jbmax(2) = jb(2)
      bcmax = bc(jb(1),jb(2))
3     continue
```

```
2     continue
      if( bcmax .eq. 0 ) return
      jb(1) = jbmax(1)
      jb(2) = jbmax(2)
      if( npp(ip(jb(1))) .lt. npp(ip(jb(2))) ) then
       choose(1) = .true.
       choose(2) = .false.
      else if( npp(ip(jb(1))) .eq. npp(ip(jb(2))) ) then
       if( npb(jb(1)) .lt. npb(jb(2)) ) then
        choose(1) = .true.
        choose(2) = .false.
       else if( npb(jb(1)) .eq. npb(jb(2)) ) then
        choose(1) = .true.
        choose(2) = .true.
       else if( npb(jb(1)) .gt. npb(jb(2)) ) then
        choose(1) = .false.
        choose(2) = .true.
       end if
      else if( npp(ip(jb(1))) .gt. npp(ip(jb(2))) ) then
       choose(1) = .false.
       choose(2) = .true.
      end if
      do 4 j = 1, 2
       if( choose(j) ) call discontinuities( jb(j), jb(3-j) )
4     continue
      bc(jb(1),jb(2)) = - bc(jb(1),jb(2))
      bc(jb(2),jb(1)) = - bc(jb(2),jb(1))
      go to 1
      end
```

5.4. Minimum Pattern

After marking discontinuous branches and individual discontinuous points, some patterns may have been so reduced in size that the remaining points should have reduced quality. Hence, patterns smaller than *nmin* are flagged. This weed operation cleans up scattered fragments left over by the previous operations:

```
      subroutine weed
      include 'qcio.f'
      include 'qcwork.f'
      do 1 jop = 1, np
       jp = iop(jop)
       if( npp(jp) .ge. nmin ) go to 1
       do 2 job = 1, nb
```

```
      jb = iob(job)
      if( ip(jb) .ne. jp ) go to 2
      if( qbc(jb) .gt. gqc ) go to 2
      qbc(jb) = 111
      do 3 j = 1, npb(jb)
       i =ibi(jb,j)
       if( qcy(i) .gt. 10 ) go to 3
       qcy(i) = 111
3     continue
2    continue
1   continue
    return
    end
```

5.5. Point Discontinuities

The *prune* (Section 5.2) and *trim* (Section 5.3) subroutines call this subroutine to identify point discontinuities. Individual discontinuous points are found by comparing their data values with all neighboring points whose quality equals or exceeds the quality of the point in question. The data value of each point is compared with its interpolated value (Section 5.6) derived from all neighboring points that have not been previously flagged using the continuity model (Section 3.4). This method for ascertaining continuity is used because connectivity is determined by the arbitrary control *dy*. Suppose that data lie along a straight line with intervals exceeding *dy*. The points are not connected with each other but they are certainly continuous because they are connected with their interpolated values. The following subroutine finds point discontinuities:

```
      subroutine discontinuities( jb, kb )
c     this subroutine assumes nx = 2 and does planar interpolation
      include 'qcio.f'
      include 'qcwork.f'
      logical find, near
      integer link
      real interpolate, xn(256), yn(256), zn(256)
      model = 0
1     find = .false.
      near = .false.
      do 2 j = 1, npb(jb)
       i = ibi(jb,j)
       if( qcy(i) .gt. 10 ) go to 2
       xo = x(i) / dx(1)
       yo = x(i+n) / dx(2)
       zo = y(i)
       in = 0
```

```
      do 3 k = 1, nn(i)
       ii = ni(i,k)
       if( qcy(ii) .gt. 10 ) go to 3
       if( ib(ii) .eq. kb ) near = .true.
       in = in + 1
       xn(in) = x(ii) / dx(1)
       yn(in) = x(ii+n) / dx(2)
       zn(in) = y(ii)
3     continue
      if( in .eq. 0 ) go to 2
      if( .not. near ) go to 2
      zi = interpolate(xo,yo,in,xn,yn,zn,dzz)
      iq = qcy(i)
      qcy(i) = max( iq, link( zo, zi, dy, model ) )
      if( qcy(i) .le. 10 ) go to 2
      find = .true.
      npp(ip(jb)) = npp(ip(jb)) - 1
2     continue
     if( find ) go to 1
     return
     end
```

5.6. Interpolation

The *discontinuities* subroutine (Section 5.5) calls the present function routine, which does a least-squares fit of a linear model to the data. Then that model is used to interpolate the data to the point in question. The model is a plane for two-dimensional data and a straight line for one-dimensional data. The present routine assumes $nx = 2$. It will work for one-dimensional domains x, though, so long as $dx(2)$ is non-zero and the x values are the same for the second dimension.

```
      real function interpolate(xo,yo,nn,xn,yn,zn,dzz)
      real xn(*), yn(*), zn(*)
      x1 = xo
      x2 = xo
      y1 = yo
      y2 = yo
      do 1 i = 1, nn
       x1 = min( x1, xn(i) )
       x2 = max( x2, xn(i) )
       y1 = min( y1, yn(i) )
       y2 = max( y2, yn(i) )
1     continue
      if( x1 .eq. x2 ) x2 = x1 + 1
```

```
      if( y1 .eq. y2 ) y2 = y1 + 1
      xo = ( xo - x1 ) / ( x2 - x1 )
      yo = ( yo - y1 ) / ( y2 - y1 )
      do 2 i = 1, nn
       xn(i) = ( xn(i) - x1 ) / ( x2 - x1 )
       yn(i) = ( yn(i) - y1 ) / ( y2 - y1 )
    2 continue
      x = 0.0
      y = 0.0
      z = 0.0
      xx = 0.0
      yy = 0.0
      zz = 0.0
      xy = 0.0
      xz = 0.0
      yz = 0.0
      xm = 0.0
      ym = 0.0
      zm = 0.0
      do 3 i = 1, nn
       x = x + xn(i)
       y = y + yn(i)
       z = z + zn(i)
       xx = xx + xn(i) * xn(i)
       yy = yy + yn(i) * yn(i)
       zz = zz + zn(i) * zn(i)
       xy = xy + xn(i) * yn(i)
       xz = xz + xn(i) * zn(i)
       yz = yz + yn(i) * zn(i)
       xm = amax1( xm, abs( xn(i) ) )
       ym = amax1( ym, abs( yn(i) ) )
       zm = amax1( zm, abs( zn(i) ) )
    3 continue
      if( xm .eq. 0.0 ) xm = 1.0
      if( ym .eq. 0.0 ) ym = 1.0
      if( zm .eq. 0.0 ) zm = 1.0
      x = x / ( nn * xm )
      y = y / ( nn * ym )
      z = z / ( nn * zm )
      xx = xx / ( nn * xm * xm )
      yy = yy / ( nn * ym * ym )
      zz = zz / ( nn * zm * zm )
      xy = xy / ( nn * xm * ym )
```

```
      xz = xz / ( nn * xm * zm )
      yz = yz / ( nn * ym * zm )
      zo = z
      eo = zz - z * z
      d = xx * yy - xy * xy - x * x * yy + 2 * x * xy * y - xx * y * y
      if( d .eq. 0.0 ) go to 4
c      x and y vary, planar interpolation
      a = ( ( xx * yy - xy * xy ) * z + ( xy * y - x * yy ) * xz
     &      + ( x * xy - xx * y ) * yz ) / d
      b = ( ( xy * y - x * yy ) * z + ( yy - y * y ) * xz
     &      + ( x * y - xy ) * yz ) / d
      c = ( ( x * xy - xx * y ) * z + ( x * y - xy ) * xz
     &      + ( xx - x * x ) * yz ) / d
      e = a * a + b * b * xx + c * c * yy + zz + 2 * a * b * x
     &      + 2 * a * c * y - 2 * a * z + 2 * b * c * xy - 2 * b * xz - 2 * c * yz
      if( abs(eo) .le. abs(e) ) go to 4
      eo = e
      zo = a + b * xo / xm + c * yo / ym
4     d = xx - x * x
      if( d .eq. 0.0 ) go to 5
c      x vary only
      a = ( xx * z - x * xz ) / d
      b = ( xz - x * z ) / d
      e = a * a + b * b * xx + zz + 2 * a * b * x - 2 * a * z - 2 * b * xz
      if( abs(eo) .le. abs(e) ) go to 5
      eo = e
      zo = a + b * xo / xm
5     d = yy - y * y
      if( d .eq. 0.0 ) go to 6
c      y vary only
      a = ( yy * z - y * yz ) / d
      b = ( yz - y * z ) / d
      e = a * a + b * b * yy + zz + 2 * a * b * y - 2 * a * z - 2 * b * yz
      if( abs(eo) .le. abs(e) ) go to 6
c      linear interpolation
      eo = e
      zo = a + b * yo / ym
6     interpolate = zo * zm
      dzz = sqrt( abs(eo) ) * zm
      return
      end
```

6. SUMMARY

The algorithm presented here uses pattern recognition and a continuity model to control the quality of data. The algorithm provides external controls which the user can use to affect the results. However, the algorithm was designed to be insensitive to the precise setting of those external controls.

Weber (1991) achieved very impressive results in applications of the algorithm to profiler winds and RASS temperature measurements under a wide range of meteorological conditions over many months of observations. The input control parameters (Section 2.2) were adjusted to be consistent with the radar operation parameters and with the reasonable range and resolution of the measurements. The parameters were not adjusted separately for each individual case, but rather the same parameters were used throughout entire data sets. No independent meteorological information was used in the case of profiler winds and RASS temperature measurements.

The computation time is also an important factor. Weber (1991) timed this algorithm when it was applied to 31 continuous days of wind profiler data. The algorithm was applied independently to the radial velocities on each of three antenna beams during each hour, for which there were 10 samples at each of 72 heights. Thus, there was a minimum of 720 points supplied to the algorithm in every case. Whenever the radial velocity exeeded half the maximum measureable value, the velocity was unfolded and included with the original data, bringing the total number of points to more than 720. Using a a Digital Equipment Corporation (DEC) Micro-VAX III running the VMS operating system, the algorithm averaged less than 20 seconds to process one hour of data for one antenna beam.

Nevertheless, quality control should not be based on continuity alone because many different processes (and not just the desired one) can exhibit consistency. For example, if ground clutter or sea echo is mistaken for the atmospheric signals over many heights and over time, then wind profiler measurements will be very consistent but they will also be very wrong. This algorithm can be a useful and effective tool when used intelligently in combination with other quality controls based on the physics of the phenomnea being observed.

7. REFERENCES

Gossard, E.E., and R.G. Strauch, 1983: *Radar Observations of Clear Air and Clouds*, Developments in Atmospheric Science, 14, Elsevier, New York, 280 pp.

May, P.T., R.G. Strauch, K.P. Moran, and W.L. Ecklund, 1990: Temperature sounding by RASS with wind profiler radars: A preliminary study. *IEEE Trans. Geosci. Remote Sens.*, **28**, 19-28.

Strauch, R.G., D.A. Merritt, K.P. Moran, K.B. Earnshaw, D. van de Kamp, 1984: The Colorado wind-profiling network. *J. Atmos. Oceanic Tech.*, **1**, 37-49.

Strauch, R.G., B.L. Weber, A.S. Frisch, C.G. Little, D.A. Merritt, K.P. Moran, and D.C. Welsh, 1987: The precision and relative accuracy of profiler wind measurements, *J. Atmos. Oceanic Tech.*, **4**, 563-571.

Strauch, R.G., K.P. Moran, P.T. May, A.J. Bedard, and W.L. Ecklund, 1988: RASS temperature sounding techniques. NOAA Technical Memorandum ERL WPL-158, Wave Propagation Laboratory, Boulder, CO, 12 pp.

Weber, B.L., 1991: Quality controls for profiler measurements of winds and RASS temperatures. *J. Atmos. Oceanic Tech.* (submitted).

Weber, B.L., and D.B. Wuertz, 1990: Comparison of rawinsonde and wind profiler radar measurements. *J. Atmos. Oceanic Tech.*, **7**, 157-174.

Weber, B.L., D.B. Wuertz, R.G. Strauch, D.A. Merritt, and K.P. Moran, 1990: Preliminary evaluation of the first NOAA demonstration network wind profiler. *J. Atmos. Oceanic Tech.*, **7**, 909-918.

Weber, B.L., D.B. Wuertz, D.C. Law, A.S. Frisch, and J.M. Brown, 1991: Effects of small scale vertical motion on radar measurements of wind and temperature. *J. Atmos. Oceanic Tech.* (accepted June 1992).

Wuertz, D.B., B.L. Weber, R.G. Strauch, A.S. Frisch, C.G. Little, D.A. Merritt, K.P. Moran, and D.C. Welsh, 1988: Effects of precipitation on UHF wind profiler measurements. *J. Atmos. Oceanic Tech.*, **5**, 450-465.

Wuertz, D.B., and B.L. Weber, 1989: Editing wind profiler measurements. NOAA Technical Report ERL 438-WPL 62, Wave Propagation Laboratory, Boulder, CO, 78 pp.

☆U.S. GOVERNMENT PRINTING OFFICE: 1991 - 673-002/41013