Earth Sci Inform (2017) 10:247–256
DOI 10.1007/s12145-016-0285-z

METHODOLOGY ARTICLE

An architecture for consolidating multidimensional time-series data onto a common coordinate grid

Tim Shippert[1] · **Krista Gaustad**[1]

Received: 12 July 2016 / Accepted: 6 December 2016 / Published online: 16 December 2016

Abstract Consolidating measurement data for use by data models or in inter-comparison studies frequently requires transforming the data onto a common grid. Standard methods for interpolating multidimensional data are often not appropriate for data with non-homogenous dimensionality, and are hard to implement in a consistent manner for different datastreams. These challenges are increased when dealing with the automated procedures necessary for use with continuous, operational datastreams. In this paper we introduce a method of applying a series of one-dimensional transformations to merge data onto a common grid, examine the challenges of ensuring consistent application of data consolidation methods, present a framework for addressing those challenges, and describe the implementation of such a framework for the Atmospheric Radiation Measurement (ARM) program.

Keywords Datastream · Flattened arrays · Data slices · Data consolidation · Regridding · Data Quality

Background

The ARM program collects field measurements of atmospheric data from continuously operating, highly instrumented ground stations, and from mobile instrument stations whose locations change on an approximate yearly schedule (Mather and Voyles 2013). The instrument-level data is collected in a large number of individual native formats, and then converted to netCDF (network Common Data Format) for permanent archival and open distribution from the ARM Data Archive. Applying increasingly more advanced analysis techniques and quality analysis to existing data products creates higher-level data products with additional scientific value. To support its internal processing and to provide users with data in the format they need, the program developed the ARM Data Integrator (ADI) framework (Gaustad et al. 2014). ADI automates the process of retrieving and preparing data for analysis, and creating integrated data products through a module data integration workflow. This paper discusses the Serial 1D transformation method implemented in ADI, the metadata it uses and produces, and design considerations that can adversely affect the scientific validity of the underlying data.

Communicated by: H. A. Babaie

✉ Tim Shippert
Tim.Shippert@pnnl.gov

Krista Gaustad
Krista.Gaustad@pnnl.gov

[1] Data Sciences Group in Advanced Computing, Mathematics, and Data Division, Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O. Box 999, Richland, WA 99352, USA

Introduction

Operational instrument-based time-series datastreams are challenging to provide to end-users in a way that facilitates their analyses. Many scientific issues such as climate research require multiple, continuous, on-going datasets, collected by instruments that remain in the field for years. Such datastreams and their derived products like model runs and retrievals require automatic methods of production, analysis, and assimilation that can be applied consistently on a wide variety of data. By contrast, in a traditional case-study with a limited dataset, sophisticated analyses and manipulation could be applied manually (Miller et al. 1994).

One of the challenges of dealing with operational data is transforming it onto a different dimensional grid. This is

necessary to consolidate data from different instruments for use as part of the same analysis, or to generate a complete set of inputs in a consistent format for a model or retrieval. In a case-study mode, a variety of interpolation methods of appropriate dimensionality might be applied, examined, and tweaked, and quality control might be done manually. But in a production environment such as ARM (U.S. Department of Energy 2009), an automated way of consolidating datasets of arbitrary dimensionality and quality is necessary, and must function without any a priori knowledge of the nature of the data other than its original dimensionality.

A major complication is the fact that multidimensional instrument-based datastreams are generally inhomogeneous in their dimensionality – a water-vapor retrieval algorithm might be dimensioned by time and vertical height, or a spectral radiometer will have dimensions of time and wavelength, for example. Standard multidimensional interpolation methods such as a Delaunay tessellation require and are sensitive to scaling (Li and Heap 2011). Such specific analysis is impossible to apply generally and automatically on a continuous, on-going dataset. The Serial 1D approach described in this paper transforms each dimension individually, removing any such scaling requirement.

An additional requirement when transforming operational data is a general, automated way to use input quality information, and to generate output metadata that describes the quality and status of the transformation. Regions of missing or bad data should be interpolated over or skipped when performing integrations, and as much provenance as possible should be kept to help end user determine when a quality control (QC) event has occurred and what solutions were applied during coordinate transformation. In addition, supporting fields called *metrics* can be generated, providing additional information about the nature of the transformation or the original input data.

Finally, an automated data transformation framework must reconcile two contradictory requirements: it must be flexible and customizable enough to deal with any arbitrary dataset, but it also must provide a consistent user interface and output. For ADI, we developed a framework for implementing such automated, quality-based coordinate transformations, both in the context of a standalone tool and as a module for use with scientific analysis programs developed using ADI. The ADI *data_consolidator* will be compared to other available time-series data transformation tools.

The Transformation Methods section will discuss why a serial approach of transforming each dimension independently was chosen, and then examine the specific transformations methods supported. The design and implementation considerations that drove the architecture and the details of the parameter control methods that support the creation of high-quality standardized data products will be presented in the sections that follow.

Transformation methods

The general method is to apply a series of interpolation and integration methods on one-dimensional subsets (or *slices*) of each dataset. This allows each dimension to be handled independently, keeps a consistent methodology for application to one-dimensional datastreams, and can be automated for an arbitrary number of dimensions. We will refer to this method as the serial one-dimensional (or Serial 1D) method. It is straightforward to extend the method and transform multidimensional surfaces in a serial manner; this will be discussed further at the end of the next section.

Traditional methods and existing tools often implicitly implement a Serial 1D methodology, for example by averaging or interpolating time-series measurements onto common time grid before interpolating onto a common pressure or height grid. Thus, Serial 1D is simply a way of implementing these different transformations in a consistent and easy-to-use manner. To allow for flexibility, a number of standard transformation methods (linear or nearest-neighbor interpolation, and averaging) are described, while other methods could be implemented and applied in a similar manner.

The process of transforming an input variable to a new coordinate system consists of defining the output grid, selecting transformation methods for each dimension, identifying and setting the parameters of the input and output grids that will affect the transformation and filtering undesirable input data from the transformation using whatever QC information is available.

The resulting output includes not only data transformed to the new coordinate system, but also parallel QC and metric fields describing various conditions that occurred during the transformation. This provides additional information about the transform.

Serial 1D

The Serial 1D method is illustrated in Fig. 1. Each dimension is extracted and transformed in individual *slices* of that dimension. A slice in dimension d is defined as the set of all values of x_d with all the other indeces $x_{d'}$ fixed for $d' \neq d$. Each such slice is thererfore one-dimensional in d. For example, in the two-dimensional array illustrated in Fig. 2, each row and each column is a 1D slice. In Fig. 2a, the rows are defined by common values of the index a, and thus are one-dimensional in index b. Similarly, the columns are dimensioned by index a and all have common values of b.

If the size of each dimension is n_d and the total number of elements in our entire multidimensional data array is n_{tot}, then the number of slices in any dimension d is n_{tot}/n_d . For each of these slices we can apply a one-dimensional transformation to put that slice on the new data grid. Once all such slices have been transformed, we move to the next dimension and repeat the process, until each dimension has been transformed and the data has been fully regridded.

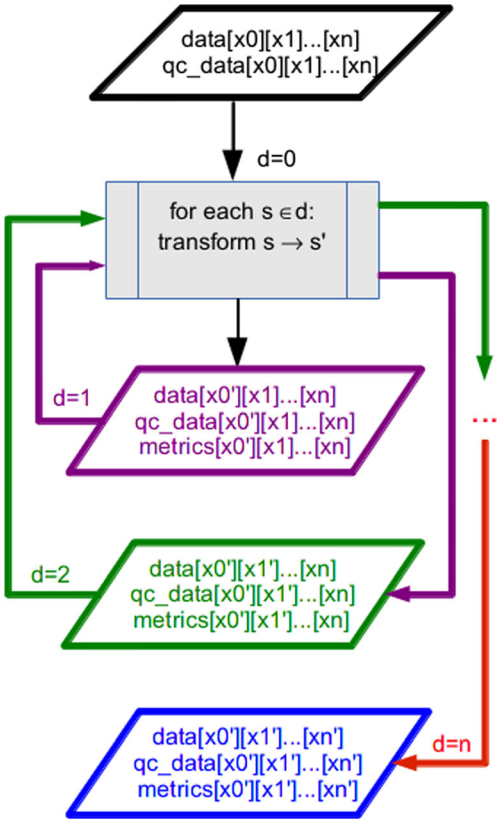

Fig. 1 Logic flow for transforming multiple dimensions in series. Each set of arrows represents the steps taken to transform a different dimension *d*, and the primed indices indicate dimensions that have been transformed at each stage. This process generates a series of intermediate data and QC arrays containing both transformed and untransformed dimensions, which are used as inputs to the next step until all dimensions have been transformed

Flattened arrays

We want to develop a general method of applying one-dimensional transformations onto each slice of an array of arbitrary dimensionality. The transformation library (like ADI) is written in C, and the C language syntax is different for arrays of different numbers of dimensions. For example, a one-dimensional array is accessed by syntax of the form *M[i]*, while a three-dimensional array would use syntax like *M[i][j][k]*. Therefore, to use regular C array addressing would require multiple functions written for each dimensionality, called in a conditional block or using some kind of overloading method, and even then it would not be a truly general method.

Instead, we will take advantage of the way C stores multi-dimensional arrays internally, as a single one-dimensional array stored in contiguous memory (Knuth 1997). We will call this one-dimensional representation a *flattened* array. Any regular array of any number of dimensions can be flattened and therefore addressed by using a single index, and thus this will allow us to develop a general method of extracting and transforming each slice of each dimension of any such array. We will first examine how to map M-dimensional data onto a flattened one-dimensional representation:

$$\text{array}^{(M)}[x_0]\ldots[x_{M-1}] \rightarrow array^{(1D)}[k] \tag{1}$$

The indices x_i are used when considering the data in the M-dimensional representation, while the index k will represent data in the flattened representation. We can relate these two indices by defining the *stride coefficients* D_d for each dimension d as[1]:

$$k \equiv \sum_{d=0}^{M-1} x_d D_d \tag{2}$$

Thus, to find the value of k corresponding to increasing an index x_d by one (i.e., adjacent elements in d), you have to jump (or "stride") D_d elements down our flattened array; see Fig. 2.1b.

To derive the appropriate values of D_d, we note that C stores data in *row-major* order, where the higher dimensions vary faster than the lower dimensions. If column-major ordering is desired (as it is in, for example, FORTRAN, IDL, or R) one must transpose the arrays and indices as they are described here.

By definition,

$$D_{M-1} \equiv 1 \tag{3}$$

as increasing the highest index x_{M-1} must always increment k by one. The other stride coefficients D_d are given by:

$$D_d = \prod_{i=d+1}^{M-1} n_i \tag{4}$$

where n_d are the lengths of each dimension. For example, for a 2D data set dimensioned by time and height ([t][h]), we have:

$$k = n_h t + h \tag{5}$$

and for 3D data in [x][y][z]:

$$k = n_y n_z x + n_z y + z \tag{6}$$

From Eqs. (3) and (4), we can see that:

$$D_d = n_{d+1} D_{d+1} \tag{7}$$

which provides us with a recursive relation for calculating our stride coefficients.

Data slices

With all data in flattened arrays and with known stride coefficients, we can develop a general method for applying transformations to each one-dimensional slice of data in the dataset. The number of such slices in d is $n_s = n_{tot}/n_d$.

[1] Note that all our indices will be zero offset, i.e., $x_i \in \{0,1,\ldots,n_i\text{-}1\}$

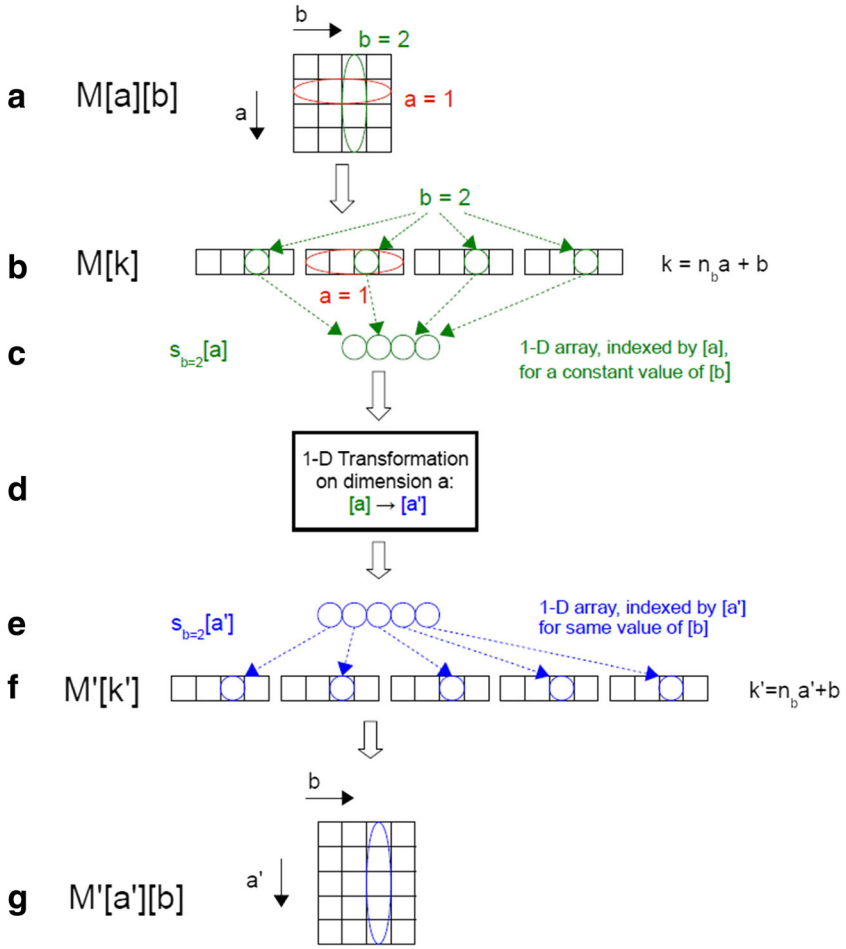

Fig. 2 Transforming the [a] dimension of a 2D array M[a][b]. (**a**) The original 2D array M[a][b], with dimensions a and b. The size of these dimensions are $n_a = 4$ and $n_b = 4$. The horizontal ellipse illustrates a slice in b with a constant value of a = 1, while the vertical ellipse illustrates a slice in a of constant b = 2. (**b**) The flattened 1D representation M[k] of the original 2D array. The slices from the 2D representation are located as indicated in the flattened representation. (**c**) Each column of M[a][b] corresponds to a 1D array of constant b, and therefore is indexed by dimension [a]. Each such slice s[a] is extracted from the flattened array; the initial b = 2 column from (a) is shown here. (**d**) Each slice is sent to the 1D transformation code, which creates a new slice s'[a'] on the output grid, as shown in (**e**). (**f**) This slice s'[a'] is then inserted into a new 1D flattened array M'[k']. (**g**) After all such slices s'[a'] are filled in M'[k'] is complete and represents a new intermediate 2D array M'[a'][b] containing a mixture of the newly transformed dimension a' (size $n_{a'} = 5$) and the untransformed dimension b. We now go on to transform dimension b - > b', using the flattened 1D array of (f) in step (b), this time extracting and transforming slices s[b] in b

Suppose that we have a flattened index k_0 that we know corresponds to the index value $x_d = 0$ (i.e. the initial value of our slice). Then by Eq. (2) we can find every element of this d-slice by the following algorithm:

$$slice[i] = array^{(1D)}[k_0 + i{\cdot}D_d] \quad i{\in}\{0, \ldots, n_d{-}1\} \tag{8}$$

Therefore, the problem of finding and looping over all the slices in dimension d has been reduced to finding the initial value $k_0(s)$ for each slice s, which is the same thing as finding all possible permutations of $\{x_{d'}\}$ for $d' \neq d$ while keeping $x_d = 0$.

We can assert that $k_0(0) = 0$; this corresponds to the case where all indices $x_i = 0$. To find all the remaining slices defined by $k_0(s)$, we note that adding a value between 1 and D_d-1 to any value of k corresponds to keeping the index x_d the same and changing just some indices $x_{d'}$ for $d' > d$. Similarly, adding a factor of D_{d-1} to k modifies only indices for $d' < d$. Thus, to iterate over all slices in d we need to run over the faster dimensions by adding one to each successive value of $k_0(s)$ until that value reaches D_d. We then add D_{d-1} to increment the slower dimensions and reset the faster dimensions back to 0 by subtracting D_d.

The following algorithm summarizes this method of deriving $k_0(s)$ for all slices s:

$$\begin{aligned} & k_0(0) \equiv 0 \\ & \textit{for } s \in \{1, \ldots, n_s{-}1\} : \\ & \quad k_0(s) = k_0(s{-}1) + 1 \\ & \quad \text{if } [k_0(s) \bmod D_d] = 0 : \\ & \quad\quad k_0(s) = k_0(s) + D_{d-1}{-}D_d \end{aligned} \tag{9}$$

Multidimensional surfaces

We can extend the above method to extract multidimensional surfaces simply by combining multiple indices in M-dimensional space into a single virtual 1D index using the same flattening technique. For example, you can treat an M-dimensional array as an M-1 dimensional array by

$$array^{(M)}[t][x][y][z] \rightarrow array^{(M-1)}[t][r][z] \quad (10)$$

where

$$r = y + n_y x \quad (11)$$

In this way we can apply the methods of the previous section to transform the *xy* surface in a Serial 1D context.

Intermediate structures

We have now reduced our transformation problem to a series of 1D transformations. The Serial 1D method involves looping over all dimensions *d* of our data, and applying the desired transformation to all 1D slices of *d*. This means we apply later transforms on data that has already been transformed in another dimension. As each transformation will (usually) change the size of the dimension in question, it is necessary to store the results in an intermediate array that contains a mixture of original and transformed dimensions, which is then used as input to the next transformation. For example, if you have a 2D array $data_0[t][z]$, dimensioned by time and height, and you want to put that on a new time and height grid *[t'][z']*, you first transform the time coordinate onto the new grid (taking $t \rightarrow t'$) for each original height *z*. The result of this is the intermediate array $data_1[t'][z]$, with a transformed time coordinate and the original height coordinate. To complete the transformation, you then transform the height coordinate for each new time *t'*, taking $z \rightarrow z'$ and generating the final array $data_2[t'][z']$. Figure 1 demonstrates this logic in the general case.

Of course, each transitional array is actually stored as a flattened array; the slicing methods of the previous section must be applied to both the input and output (transitional) data to make sure each slice is transformed correctly and ready for the next transformation.

Time complexity

To transform each dimension, we call one 1D transformation for each of the n_s slices in that dimension, an operation that is linear in n_s. Therefore, if the transformations are linear in time (as are the three basic transformations we describe in this paper), the complexity of transforming each dimension fully is $O(n_{tot})$, where n_{tot} is the total number of elements in the array. Thus, the complexity of transforming across all n_d dimensions is $O(n_d n_{tot})$, but since n_d is typically very small for instrumental datasets, this is still essentially linear in n_{tot}. On modern computers, even very high resolution multidimensional data from radars or models can be transformed in a matter of seconds.

If more complex nonlinear transformations were implemented in a Serial 1D framework, computational issues could arise while transforming large datasets. Similar problems could arise if this framework were applied to transforming very large high dimensional abstract datasets, where n_d approaches the size of n_{tot}. However, as each 1D slice for each dimension is independent of all the other slices, parallel or distributed computing methods could be applied in such cases.

Supported transformations

In this section, we will examine three one-dimensional transformations that have been implemented in ADI: *linear interpolation, bin averaging,* and *nearest-neighbor sampling*. For each transformation, we will also discuss the QC implications and the parameters that may be used to customize the nature of that transformation.

Linear interpolation

The standard interpolation transformation is linear (as opposed to more complicated polynomial or spline interpolation methods). We take the nearest bracketing input points around our target transformed coordinate index, draw a straight line through them, and take the value of that line corresponding to our target index. This is the default transformation when we try to transform data from a larger grid to a smaller one, and can also be used to (for example) shift every index in a grid half a bin over.

Bin averaging

Averaging is the most complicated standard transform, because it requires your input and output data to represent a region (or *span*) of your coordinate space, not just a single point. To emphasize this fact we call this method a *bin average*; the input and output data are represented by bins with a finite width. Thus, we need *two* numbers to index our variables: the front and back edge of each coordinate bin, or a single coordinate index and a bin width. Each input bin is then *weighted* by the fraction of the overlap with the span of the transformed bin. Most interior input bins will be completely covered by the transformed bin, so their weights will be 1.0. But bins on the edge may straddle two different transformed bins, and thus their contribution has to be split between them.

Nearest-neighbor sampling

Nearest-neighbor sampling is the simplest transform, and consists of simply taking the nearest good input point within our *range*.

The direction of the sampling doesn't matter - we take the value of the point with the least absolute distance to the target index.

Quality control methods

For data to be of use in scientific studies it needs to be collected and analyzed in context of an end-to-end quality assurance program that includes data QC and documentation. For continuous datastreams like those generated for ARM, this requires automated methods of both calculating and using QC in our analysis codes. The purpose of automated QC is to flag data which may be bad or require further (human) analysis; this is especially important in a production environment as human eyes may not evaluate the data until much later, and it may be used as an input to other automated procedures in the meantime (Timms et al. 2010).

The ARM standards require datastreams to store QC checks in auxiliary fields that are parallel to data fields and cover the same dimensions. For each value *data[×0][×1][.][xn]* we therefore have a companion value *qc_data[×0][×1][...][xn]*. These QC values are stored as integers, with each bit representing a particular state or condition. Depending on the nature of the test, failure (represented by setting that bit to 1) may indicate the data is bad and should not be used, or it could simply indicate an unusual or noteworthy condition (ARM Standards Committee 2015).

Filtering input data

Within the context of the ADI transformation library, these parallel QC fields allow the transformations to filter bad input data automatically. The inclusive nature of these integer flags allows the end user to customize the particular states he wants to reject simply by setting an appropriate mask, which will then be compared bitwise to each QC value. When a transformation encounters a data point that is flagged as bad, it will attempt to "go around" that data point in whatever way makes sense for that transformation. The interpolation transformation, for example, will not interpolate using a bad input point but will scan up or down the input data to find good data with which to perform the interpolations.

Output QC

The output of the ADI transformation process is meant to be a new ARM-standard datastream. Therefore, the transformations will also generate parallel output QC fields to describe the various states and conditions that occur during the transformation itself. When a transformation fails, it is important to document both the occurrence and the reason why it failed - if all the inputs were missing or if a required input had a value outside its valid range, for example. Storing this information can allow us to generate statistics or do correlations on when such conditions occur, and can be an important part of analyzing and improving a given datastream.

It is especially important to flag non-standard "indeterminate" conditions because they do not necessarily mean the data is flawed, just that the transformation occurred in special circumstances. An example of this would be when some but not all of the input values in an average were flagged as bad; we can still calculate a meaningful average value, but we are not using all the points we were expecting to.

Because the transformation library is designed to be consistent across all applications, the possible QC states that come out of a transformation are fixed, and all transformed data will have only these QC bits set. In this way the transformation process necessarily "washes out" any detailed QC information provided by the input datastream. We can no longer tell exactly which input point was bad, nor can we tell which test the data might have failed. All we can do is set the appropriate QC flags that declare that some (or all) of the points used to generate a given output data point were bad.

Under the Serial 1D method, the output data and QC fields generated by transforming the first dimension will be used as input to the transformation of the second dimension, and so on until all the dimensions have been transformed. Therefore, each intermediate QC field will be used to filter data for the next dimension's transformation, until we have transformed all dimensions. The final output QC fields will hold the QC states generated while transforming just the final dimension. For example, in our 2D case where $data_0[t][z]$ is dimensioned by time and height, after transforming the time coordinate we will have the new arrays $data_1[t'][z]$ and $qc\_data_1[t'][z]$. When transforming z, we use the QC values given by $qc\_data_1[t'][z]$ to filter bad values of $data_1[t'][z]$, in exactly the way we used the original input QC fields to filter bad data while transforming t. Figure 1 illustrates the same process in the general case.

This means that some intermediate QC information has been lost by the time we have transformed all dimensions. We cannot determine the value of $qc\_data_1[t'][z]$ at the end, because we do not save or store anything on this intermediate *[t'][z]* grid. But because $qc\_data_1[t'][z]$ has been used as input to a later transformation its impact is propagated through to the final output. Many of the QC flags as described in Table 1 reflect some qualitative QC information about the intermediate transformations. For example, QC_SOME_BAD_INPUTS upon output implies that the result of the penultimate transformation generated some bad data, and provides a starting point for further investigation if desired.

Table 1 QC States by transform method

Transform QC Bit	Average method	Interpolate method	Subsample method	Assessment
QC_BAD	X	X	X	Bad
QC_INDETERMINATE	X	X	X	Indeterminate
QC_INTERPOLATE		X		Indeterminate
QC_EXTRAPOLATE		X		Indeterminate
QC_NOT_USING_CLOSEST			X	Indeterminate
QC_SOME_BAD_INPUTS	X			Indeterminate
QC_ZERO_WEIGHT	X			Indeterminate
QC_OUTSIDE_RANGE	X	X	X	Bad
QC_ALL_BAD_INPUTS	X	X	X	Bad
QC_ESTIMATED_INPUT_BIN	X	X	X	Indeterminate
QC_ESTIMATED_OUTPUT_BIN	X	X	X	Indeterminate

Table 1 lists the possible QC states generated during transformation, and which of the initial three transformation methods apply:

About half of the quality states are general in that they apply to all transformation methods. These include a flag to denote the transformation was unsuccessful (QC_BAD), that the transformation included one or more input values with an *indeterminate* assessment (QC_INDETERMINATE).

The QC_ESTIMATED_INPUT_BIN and QC_ESTIMATED_OUTPUT_BIN refer to whether the transformation parameters "width" and "alignment" have been set externally by the user whether default values were calculated. Details of the parameters that can be externally set will be discussed in a later section.

The QC_OUTSIDE_RANGE state is the only QC state assigned a *bad* assessment other than the test test documenting whether all inputs were bad and the test noting that the transformation failed. When averaging data it is set if none of the input bins overlaps with any part of the output bin, or if an input dimension's values are more limited then its value in the output (i.e., if input dimension height goes up to 60 km, but output max height is 20 km, then all values above 20 km will have this flag set). For subsample and interpolation transformations where we use two input points to calculate every output point. If one of our inputs has been flagged, we scan up or down the input grid until we find the nearest good point in that direction that is still within our defined *range transform parameter*. If not found within the range then QC_OUTSIDE_RANGE is set.

Quality control states unique to the averaging method document whether some, but not all of the inputs in the averaging window were flagged as bad and thus excluded from the transform (QC_SOME_BAD_INPUTS), and if all the inputs to be averaged for this output bin were zero (QC_ZERO_WEIGHT). For nearest neighbor, if the nearest *good* point is not the nearest absolute point (i.e., the nearest point was flagged as bad), we flag that "indeterminate" status in the QC field. If a linear interpolation technique is being used, if no such good point exists, we scan down in the other direction until we find a good point to use; in that case, the transform actually becomes an *extrapolation* (which is mathematically identical to an interpolation; the only difference is that instead of bracketing our target index the two points we use are on the same side). If we do not use the two closest bracketing points to interpolate, we set a QC flag to indicate that a non-standard interpolation took place (QC_INTERPOLATE). We also set a flag to indicate if one of the bracketing points had been flagged as *indeterminate* (QC_INDETERMINATE).

Transform metrics

In a manner similar to the automated QC tests, each of the transformation methods can also create appropriate companion variables called "metrics" that provide additional details about the transformed data. Currently only the bin average transform does so, and provides two metrics: the standard deviation of the points used in the average and a fractional indicator of the number of good points available in the averaging window. The naming convention for these variables is to append a suffix to the transformed variable name; in the case of the averaging metrics, std. is used for the standard deviation and goodfraction for the factional test. In a similar manner to QC, only the metrics generated on the final dimensional transformation will be available on output.

Transform parameters

Transform parameters are variables embedded in the transformation methods that can be externally set by users, and that allow customization of how the transformations are applied. They can be used to characterize the coordinate system grid being created for the output data products and to describe relevant charactisics of the input data, and an additional class of parameters are provided that allow a user to alter how the input data is interpreted. The available parameters, the transformations to which they apply, and whether they apply to the input or output coordinate grids is documented in Table 2. The parameters that must always have a value include a parameter that documents the transformation method applied (interpolate, average, or nearest neighbor) and parameter's start and length, which denote the first value of, and the number of values that will comprise the coordinate variable respectively.

Table 2 Transformation parameters by transform method

Transform parameter	Transformation	Input grid	Output grid
Transform	All	N	Y
Interval	Interpolate, Average	Y	Y
Start	All	Y	Y
Length	All	Y	Y
Width	Average	Y	Y
Front_edge	Average	Y	Y
Back_edge	Average	Y	Y
Range	Interpolate, subsample	Y	N
QC_bad	Interpretation of input data	Y	N
Missing_value	Interpretation of input_data	Y	N
QC_mask	Interpretation of input data	Y	N

Parameters unique to the bin averaging method algorithm document the bin width, front edge, and back edge. Preferably the bin parameters of input data are documented within the data itself, but if not it can be supplied by the user, as can the output bin characteristics. If not supplied, the averaging routine will calculate the default bin width from the data with the individual front edge and back edges of the bins for each sample derived from the resulting bins. A related parameter interval documents the difference between two values of a grid as a single value, and as such only relate to a regular grid (i.e., a grid whose bin widths are equal for all samples). The remaining parameter that directly relates to how a transformation is applied, and affects subsequent QC states, is range. It allows users to set the maximum distance for a given dimension over which a set of data will be interpolated or subsampled. Details of the remaining parameters, how they are used, and their default values are presented in the ADI online documentation (Gaustad 2015). The documentation also describes how these parameters can be used to perform analysis such as creating an average of data and smoothing data using a running average.

Comparisons with other tools

Frequently scientists will implement their own transformation methods using netCDF libraries provided by their preferred software language such as Python, IDL, or MatLab. Because the intent of the ADI framework for which this transformation approach was developed was to eliminate the need for scientiests to write their own transformation algorithms, this section will focus on tools that provide higher level transformation capabilities than those available in common programming languages. A general advantage of the approach discussed over those commonly used is the extent of control provided by allowing users to set many of the parameters frequently embedded into existing techniques, such as limiting the range to look for nearest neighbors or to filter for specific QC conditions. Thus, a general transformation method can be fine tuned by the end user for their needs. Another advantage of the presented approach is the use of built in automated quality control and metrics. These capabilities allow a user to capture provenance and to fully understand how the data has been affected by the transformation process.

In the remainder of this section we will compare our framework with three existing tools which may be used to manipulate and interpolate netCDF files: The netCDF Operators (NCO), Climate Data Operators (CDO), and Python library wradlib (Pfaff et al. 2012). The individual sections that follow will focus on how the methods compare in terms of the approach, flexibility, and accuracy of the transformation methods themselves.

NCO

The NCO (Zender 2016) package is a series of UNIX executables designed to quickly and easily manipulate data in netCDF format. One of the functions of ADI data consolidator was to provide a simple tool to bring many different datastreams onto the same coordinate grid, so the comparison is a natural one.

Most of the NCO tools are designed to merge or perform statistics upon netCDF files of the same structure, as compared to our framework, which is designed to consolidate heterogeneous datasets into a common structure. There are some NCO functions (ncra, ncwa, nces) which are used to integrate over an entire dimension in a file (and thereby removing it), and an interpolation tool ncflint whose purpose is to generate data between that given by two files of the same structure, using the same record dimension. Specific applications like these simplify their individual use, but at the cost of overall flexibility.

The NCO tool ncap2 provides a robust processing environment which can be scripted to perform very complicated manipulations. It includes a native 2D bilinear-interpolation routine, and also allows interface to the GNU Scientific Library (GSL) (Galassi et al. 2009) 1D

interpolation routines, and averaging routines could probably be scripted for use with ncap2. However, such scripting is essentially a programming effort; some of the methods described in this paper might be implemented in such a task.

CDO

The Climate Data Operators (CDO) (Schulzweida et al. 2009) is another package of UNIX-based command-line routines designed for standard processing of netCDF files on climate model output and unlike the described technique may not necessarily be easily applied to non-model or other types of time-series data. The CDO package contains many tools for interpolating data from one standard climate grid to another and performing other scientific and statistical retrievals from such datasets. Such interpolation methods are applied to the spatial 3D grid, a 2D horizontal grid, a vertical coordinate, or over time values, and include standard methods such as multidimensional linear interpolation, distance weighted averaging, or nearest-neighbor interpolation.

For standard climate model data, CDO may be a good choice for data and grid manipulation, as specific issues related to those datasets can be dealt with. But CDO is limited to the three spatial dimensions and one temporal dimension listed, and thus would not be useful for spectral data or other multidimensional datasets. By contrast, the serial 1D approach described in this paper is a general approach for N-dimensional variables.

Wradlib

Wradlib is a Python library designed to facilitate the use of weather radar data. It provides collections of algorithms and functions that enable users to create customized data products for use in forecasting, research, development, or teaching (Pfaff et al. 2012). Because of its focus on radar data, wradlib supports a larger, more diverse set of transformation methods useful with radar data (such as Z to R conversions). It also supports spatial interpolation techniques such as Ordinary Kriging. However, it is only available in Python, although Jupyter Notebooks are provided to allow users to experiment and easily access code snippets. While the ARM ADI transformation library is being updated to support the Caracena grid transformation method (Caracena 1987), the wradlib is a good alternative for working with radar data and the special tools and methods that data needs. However, it is probably less useful for non-radar atmospheric data and non-atmospheric time-series data.

Conclusions and future work

ARM has developed and been successfully using the Serial 1D transformation method in over a dozen production algorithms and several dozen in house algorithms whose analysis required consolidation of temporally diverse datasets onto a common grid. ARM's data consolidation architecture has greatly increased the efficiency of implementing production algorithms, frequently shortening their development time by a factor of two or higher, improved the robustness of the code through the use of a heavily used and well tested library, standardized logging and provenance, and automated QCs applied.

The Serial 1D method capabilities are currently being updated to support the creation and evaluation of data that describes data sets that span across a grid of measurement locations. The core libraries have been updated to grid transformations, with the initial approach being the Carecena method (Caracena 1987). The gridded capabilities will facilitate the development of data sets that can be readily used to evaluate the model simulations of the atmosphere and better support the global climate modeling community's needs.

Acknowledgements This research was supported by the Office of Biological and Environmental Research of the U.S. Department of Energy under Contract No DE-AC05d76RL01830 as part of the Atmospheric Radiation Measurement Climate Research Facility.

This project took advantage of netCDF software developed by UCAR/Unidata (www.unidata.ucar.edu/software/netcdf/).

References

ARM Standards Committee (2015) ARM Data File Standards Version: 1.1 http://www.arm.gov/publications/programdocs/doe-sc-arm-15-004.pdf. Accessed 26 May 2016

Caracena F (1987) Analytic approximation of discrete field samples with weighted sums and the Gridless computation of field derivatives. J Atmos Sci 44(24):3753–3768

Galassi M, Davies J, Theiler J, Gough B, Jungman G, Alken P, Booth M, Rossi F (2009) GNU Scientific Library Reference Manual, The GSL Team. http://www.network-theory.co.uk/docs/gslref/. Accessed 29 Mar 2016

Gaustad K (2015) ARM Data Integrator (ADI) Documentation. https://engineering.arm.gov/ADI_doc. Accessed 24 Oct 2016

Gaustad K, Shippert T, Ermold B, Beus S, Daily J, Borsholm A, Fox K (2014) A scientific data processing framework for time series NetCDF data. Environ Model Softw 60:241–249

Knuth DE (1997) The art of computer programming. Addison-Wesley, USA

Li J, Heap AD (2011) A review of comparative studies of spatial interpolation methods in environmental sciences: performance and impact factors. Ecol Inform 6(3–4):228–241. doi:10.1016/j.ecoinf.2010.12.003

Mather JH, Voyles JW (2013) The ARM climate research facility: a review of structure and capabilities. B AM Meteorol Soc 94(3):377–392

Miller N, J Liljegren, T Shippert, S Clough, P Brown (1994) Quality measurement experiments within the atmospheric radiation measurement program. In Proceedings of 74th AMS Annual Meeting. Nashville, TN

Pfaff T, Heistermann M, Jacobi S (2012) wradlib – An Open Source Library for Weather Radar Data Processing. ERAD 2012 - The Seventh European Conference on Radar in Meteorology and Hydrology. http://www.meteo.fr/cic/meetings/2012/ERAD/extended_abs/DQ_305_ext_abs.pdf. Accessed 24 Oct 2016

Schulzweida U, Kornblueh L, Quast R (2009) CDO User's Guide, Colorado State University. http://vista.cira.colostate.edu/nco/documents/CDO/cdo.pdf. Accessed 29 Mar 2016

Timms, GP, PA Souza, L Reznik (2010) Automated assessment of data quality in marine sensor networks OCEANS 2010 I.E. - Sydney, Sydney, NSW, IEEE

U.S. Department of Energy (2009) Atmospheric radiation measurement program plan. Report No. DOE/ER-0441, Pacific Northwest National Laboatory https://www.arm.gov/publications/doe-er-0441.pdf. Accessed 30 March, 2016

Zender C (2016) NCO User Guide. University of California, Irvine https://code.zmaw.de/projects/cdo/files. Accessed 29 Mar 2016