NetCDF Climate and Forecast (CF) Metadata Conventions

Brian Eaton, Jonathan Gregory, Bob Drach, Karl Taylor, Steve Hankin, John Caron, Rich Signell, Phil Bentley, Greg Rappa, Heinke Höck, Alison Pamment, Martin Juckes, Martin Raspaud, Jon Blower, Randy Horne, Timothy Whiteaker, David Blodgett, Charlie Zender, Daniel Lee, David Hassell, Alan D. Snow, Tobias Kölling, Dave Allured, Aleksandar Jelenak, Anders Meier Soerensen, Lucile Gaultier, Sylvain Herlédan, Fernando Manzano, Lars Bärring, Christopher Barker, Sadie L. Bartholomew, Thomas Lavergne, Bryan Lawrence, Neil Massey, Antonio Cofiño

Version v1.12.0-rc7-24-gb724218, 16 December, 2025 23:08:53Z: main

Climate and Forecast Conventions version 1.13-draft has no DOI yet: 10.5281/zenodo.FFFFFF

The Climate and Forecasting Conventions website https://cfconventions.org/ contains additional resources and provides further information.

DON'T use the following reference to cite this version of the document, as it is only shown as a draft:
Eaton, B., Gregory, J., Drach, B., Taylor, K., Hankin, S. et al. (2024). NetCDF Climate and Forecast (CF) Metadata Conventions (1.13-draft). CF Community. https://doi.org/10.5281/zenodo.FFFFFF

Table of Contents

List of Tables

List of Figures

List of Examples

About the authors

Original Authors

- Brian Eaton, NCAR
- Jonathan Gregory, University of Reading and UK Met Office Hadley Centre
- Bob Drach, PCMDI, LLNL
- Karl Taylor, PCMDI, LLNL
- Steve Hankin, PMEL, NOAA

Additional Authors

- John Caron, Unidata
- Rich Signell, USGS
- Phil Bentley, UK Met Office
- Greg Rappa, MIT
- Heinke Höck, DKRZ
- Alison Pamment, NCAS and CEDA
- Martin Juckes, NCAS
- Martin Raspaud, SMHI
- Jon Blower, University of Reading
- Randy Horne, Excalibur Laboratories Inc
- Timothy Whiteaker, University of Texas
- David Blodgett, USGS
- Charlie Zender, University of California, Irvine
- Daniel Lee, EUMETSAT
- David Hassell, NCAS and University of Reading
- Alan D. Snow, Corteva Agriscience
- Tobias Kölling, Max Planck Institute for Meteorology
- Dave Allured, NOAA/CIRES/University of Colorado
- Aleksandar Jelenak, HDF Group
- Anders Meier Soerensen, EUMETSAT
- Lucile Gaultier, OceanDataLab
- Sylvain Herlédan, OceanDataLab
- Fernando Manzano, Puertos del Estado, Madrid
- Lars Bärring, SMHI and Lund University
- Christopher Barker, NOAA
- Sadie L. Bartholomew, NCAS and University of Reading

- Thomas Lavergne, MET Norway
- Bryan Lawrence, NCAS and University of Reading
- Neil Massey, NCAS and STFC
- Antonio Cofiño, University of Cantabria

Many others have contributed to the development of CF through their participation in discussions about proposed changes.

Abstract

This document describes the CF conventions for climate and forecast metadata designed to promote the processing and sharing of files created with the netCDF Application Programmer Interface [NetCDF]. The conventions define metadata that provide a definitive description of what the data in each variable represents, and of the spatial and temporal properties of the data. This enables users of data from different sources to decide which quantities are comparable, and facilitates building applications with powerful extraction, regridding, and display capabilities.

The CF conventions generalize and extend the COARDS conventions [COARDS]. The extensions include metadata that provides a precise definition of each variable via specification of a standard name, describes the vertical locations corresponding to dimensionless vertical coordinate values, and provides the spatial coordinates of non-rectilinear gridded data. Since climate and forecast data are often not simply representative of points in space/time, other extensions provide for the description of coordinate intervals, multidimensional cells and climatological time coordinates, and indicate how a data value is representative of an interval or cell. This standard also relaxes the COARDS constraints on dimension order and specifies methods for reducing the size of datasets.

Chapter 1. Introduction

1.1. Goals

The NetCDF library [NetCDF] is designed to read and write data that has been structured according to well-defined rules and is easily ported across various computer platforms. The netCDF interface enables but does not require the creation of *self-describing* datasets. The purpose of the CF conventions is to require conforming datasets to contain sufficient metadata that they are self-describing in the sense that each variable in the file has an associated description of what it represents, including physical units if appropriate, and that each value can be located in space (relative to earth-based coordinates) and time.

An important benefit of a convention is that it enables software tools to display data and perform operations on specified subsets of the data with minimal user intervention. It is possible to provide the metadata describing how a field is located in time and space in many different ways that a human would immediately recognize as equivalent. The purpose in restricting how the metadata is represented is to make it practical to write software that allows a machine to parse that metadata and to automatically associate each data value with its location in time and space. It is equally important that the metadata be easy for human users to write and to understand.

This standard is intended for use with climate and forecast data, for atmosphere, surface and ocean, and was designed with model-generated data particularly in mind. We recognise that there are limits to what a standard can practically cover; we restrict ourselves to issues that we believe to be of common and frequent concern in the design of climate and forecast metadata. Our main purpose therefore, is to propose a clear, adequate and flexible definition of the metadata needed for climate and forecast data. Although this is specifically a netCDF standard, we feel that most of the ideas are of wider application. The metadata objects could be contained in file formats other than netCDF. Conversion of the metadata between files of different formats will be facilitated if conventions for all formats are based on similar ideas.

This convention is designed to be backward compatible with the COARDS conventions [COARDS], by which we mean that a conforming COARDS dataset also conforms to the CF standard. Thus new applications that implement the CF conventions will be able to process COARDS datasets.

We have also striven to maximize conformance to the COARDS standard, that is, wherever the COARDS metadata conventions provide an adequate description we require their use. Extensions to COARDS are implemented in a manner such that the content that doesn't depend on the extensions is still accessible to applications that adhere to the COARDS standard.

1.2. Principles for design

The following principles are followed in the design of these conventions:

1. CF-netCDF metadata is designed to make datasets self-describing as far as practically possible. A self-describing dataset is one which can be interpreted without need for reference to resources outside itself, and the CF principle is to minimise that need. Therefore CF-netCDF does not use codes, but instead relies on controlled vocabularies containing terms that are chosen to be self-

explanatory (but more detailed definitions of them are provided in CF documents).

2. The conventions are changed only as actually required by common use-cases, and not for needs which cannot be anticipated with certainty.
3. In order to keep them logical, consistent in approach and as simple as possible, the netCDF conventions are devised with and within the conceptual framework of the CF data model, and new standard names are constructed as far as possible to follow the syntax and vocabulary of existing standard names.
4. The conventions should be practicable for both producers and users of data.
5. The metadata should be both easily readable by humans and easily parsable by programs.
6. To avoid potential inconsistency within the metadata, the conventions should minimise redundancy.
7. The conventions should minimise the possibility for mistakes by data-writers and data-readers.
8. Conventions are provided to allow data-producers to describe the data they wish to produce, rather than attempting to prescribe what data they should produce; consequently most CF conventions are optional.
9. Because many datasets remain in use for a long time after production, it is desirable that metadata written according to previous versions of the convention should also be compliant with and have the same interpretation under later versions.
10. Because all previous versions must generally continue to be supported in software for the sake of archived datasets, and in order to limit the complexity of the conventions, there is a strong preference against introducing any new capability to the conventions when there is already some method that can adequately serve the same purpose (even if a different method would arguably be better than the existing one).

1.3. Terminology

The terms in this document that refer to components of a netCDF file are defined in the NetCDF User's Guide (NUG) [NUG]. Some of those definitions are repeated below for convenience.

aggregated data

The data of an aggregation variable, after it has been created in memory by an application program (see Section 2.8, "Aggregation Variables").

aggregated dimension

One of the dimensions of the aggregated data of an aggregation variable (see Section 2.8, "Aggregation Variables").

aggregation variable

A variable containing no data, but which enables the formation its data (i.e. its aggregated data) by the combination of data arrays found in one or more fragments (see Section 2.8, "Aggregation Variables").

ancestor group

A group from which the referring group is descended via direct parent-child relationships

auxiliary coordinate variable

Any netCDF variable that contains coordinate data, but is not a coordinate variable (in the sense of that term defined by the [NUG] and used by this standard - see below). Unlike coordinate variables, there is no relationship between the name of an auxiliary coordinate variable and the name(s) of its dimension(s).

boundary variable

A boundary variable is associated with a variable that contains coordinate data. When a data value provides information about conditions in a cell occupying a region of space/time or some other dimension, the boundary variable provides a description of cell extent.

CDL syntax

The ascii format used to describe the contents of a netCDF file is called CDL (network Common Data form Language). This format represents arrays using the indexing conventions of the C programming language, i.e., index values start at 0, and in multidimensional arrays, when indexing over the elements of the array, it is the last declared dimension that is the fastest varying in terms of file storage order. The netCDF utilities ncdump and ncgen use this format (see NUG section on CDL syntax). All examples in this document use CDL syntax.

cell

A region in one or more dimensions whose boundary can be described by a set of vertices recorded in boundary variables. The term *interval* is sometimes used for one-dimensional cells. A two-dimensional cell is analogous to a pixel in a raster graphic, but is a more general concept (see Section 1.4, "Overview").

calendar

A CF calendar defines an ordered set of valid datetimes with integer seconds.

coordinate variable

A coordinate variable is a one-dimensional variable with the same name as its dimension e.g., `time(time)`. In CF, a coordinate variable must be of a numeric data type (note that NUG section on coordinate variables does not have this requirement). The coordinate values must be in strict monotonic order (all values are different, and they are arranged in either consistently increasing or consistently decreasing order). Missing values are not allowed in coordinate variables. To avoid confusion with coordinate variables, CF does not permit a one-dimensional string-valued variable to have the same name as its dimension. Note that an aggregation coordinate variable is stored as a scalar and has the same name as its aggregated dimension (see Section 2.8, "Aggregation Variables").

fragment

Data, generally found in an external dataset, that contributes to the formation of the aggregated data of an aggregation variable (see Section 2.8, "Aggregation Variables").

datetime

The set of numbers which together identify an instant of time, namely its year, month, day, hour, minute and second, where the second may have a fraction but the others are all integer.

grid mapping variable

A variable used as a container for attributes that define a specific grid mapping. The type of the variable is arbitrary since it contains no data.

interpolation variable

A variable used as a container for attributes that define a specific interpolation method for uncompressing tie point variables. The type of the variable is arbitrary since it contains no data.

latitude dimension

A dimension of a netCDF variable that has an associated latitude coordinate variable.

local apex group

The nearest (to a referring group) ancestor group in which a dimension of an out-of-group coordinate is defined. The word "apex" refers to position of this group at the vertex of the tree of groups formed by it, the referring group, and the group where a coordinate is located.

longitude dimension

A dimension of a netCDF variable that has an associated longitude coordinate variable.

most rapidly varying dimension

The dimension of a multidimensional variable for which elements are adjacent in storage. When a netCDF dataset is represented in CDL, the most rapidly varying dimension is the last one e.g. `x` in `float data(z,y,x)`. C and Python NumPy use the same order as CDL, also called "row-major order", while Fortran and R use the alternative order, also called "column-major order", so that when netCDF variables are accessed in Fortran or R the most rapidly varying dimension is the first one.

multidimensional coordinate variable

An auxiliary coordinate variable that is multidimensional.

nearest item

The item (variable or group) that can be reached via the shortest traversal of the file from the referring group following the rules set forth in the Section 2.7, "Groups".

out-of-group reference

A reference to a variable or dimension that is not contained in the referring group.

path

Paths must follow the UNIX style path convention and may begin with either a '/', '..', or a word.

quantization variable

A variable used as a container for attributes that define a specific quantization algorithm. The type of the variable is arbitrary since it contains no data.

recommendation

Recommendations in this convention are meant to provide advice that may be helpful for reducing common mistakes. In some cases we have recommended rather than required particular attributes in order to maintain backwards compatibility with COARDS. An application

must not depend on a dataset's adherence to recommendations.

referring group

The group in which a reference to a variable or dimension occurs.

scalar coordinate variable

A scalar variable (i.e. one with no dimensions) that contains coordinate data. Depending on context, it may be functionally equivalent either to a size-one coordinate variable (Section 5.7, "Scalar Coordinate Variables") or to a size-one auxiliary coordinate variable (Section 6.1, "Labels" and Section 9.2, "Collections, instances, and elements").

sibling group

Any group with the same parent group as the referring group

spatiotemporal dimension

A dimension of a netCDF variable that is used to identify a location in time and/or space.

tie point variable

A netCDF variable that contains coordinates that have been compressed by sampling. There is no relationship between the name of a tie point variable and the name(s) of its dimension(s).

time dimension

A dimension of a netCDF variable that has an associated time coordinate variable.

vertex dimension

The dimension of a boundary variable along which the vertices of each cell are ordered.

vertical dimension

A dimension of a netCDF variable that has an associated vertical coordinate variable.

1.4. Overview

No variable or dimension names are standardized by this convention. Instead we follow the lead of the [NUG] and standardize only the names of attributes and some of the values taken by those attributes. Variable or dimension names can either be a single variable name or a path to a variable. The overview provided in this section will be followed with more complete descriptions in following sections. Appendix A, Attributes contains a summary of all the attributes used in this convention.

Files using this version of the CF Conventions must set the [NUG] defined attribute `Conventions` to contain the string value "`CF-1.13-draft`" to identify datasets that conform to these conventions.

The general description of a file's contents should be contained in the following attributes: `title`, `history`, `institution`, `source`, `comment` and `references` (Section 2.6.2, "Description of file contents"). For backwards compatibility with COARDS none of these attributes is required, but their use is recommended to provide human readable documentation of the file contents.

Each variable in a netCDF file has an associated description which is provided by the attributes

`units`, `long_name`, and `standard_name`. The `units`, and `long_name` attributes are defined in the [NUG] and the `standard_name` attribute is defined in this document.

The `units` attribute is required for all variables that represent dimensional quantities (except for boundary variables defined in Section 7.1, "Cell Boundaries"). The values of the `units` attributes are character strings that are recognized by UNIDATA's UDUNITS package [UDUNITS] (with exceptions allowed as discussed in Section 3.1, "Units").

The `long_name` and `standard_name` attributes are used to describe the content of each variable. For backwards compatibility with COARDS neither is required, but use of at least one of them is strongly recommended. The use of standard names will facilitate the exchange of climate and forecast data by providing unambiguous identification of variables most commonly analyzed.

Four types of coordinates receive special treatment by these conventions: latitude, longitude, vertical, and time. Every variable must have associated metadata that allows identification of each such coordinate that is relevant. Two independent parts of the convention allow this to be done. There are conventions that identify the variables that contain the coordinate data, and there are conventions that identify the type of coordinate represented by that data.

There are two methods used to identify variables that contain coordinate data. The first is to use the [NUG]-defined "coordinate variables." *The use of coordinate variables is required for all dimensions that correspond to one dimensional space or time coordinates*. In cases where coordinate variables are not applicable, the variables containing coordinate data are identified by the `coordinates` attribute.

Once the variables containing coordinate data are identified, further conventions are required to determine the type of coordinate represented by each of these variables. Latitude, longitude, and time coordinates are identified solely by the value of their `units` attribute. Vertical coordinates with units of pressure may also be identified by the `units` attribute. Other vertical coordinates must use the attribute `positive` which determines whether the direction of increasing coordinate value is up or down. Because identification of a coordinate type by its units involves the use of an external package [UDUNITS], we provide the optional attribute `axis` for a direct identification of coordinates that correspond to latitude, longitude, vertical, or time axes.

Latitude, longitude, and time are defined by internationally recognized standards, and hence, identifying the coordinates of these types is sufficient to locate data values uniquely with respect to time and a point on the earth's surface. On the other hand identifying the vertical coordinate is not necessarily sufficient to locate a data value vertically with respect to the earth's surface. In particular a model may output data on the parametric (usually dimensionless) vertical coordinate used in its mathematical formulation. To achieve the goal of being able to spatially locate all data values, this convention provides a mapping, via the `standard_name` and `formula_terms` attributes of a parametric vertical coordinate variable, between its values and dimensional vertical coordinate values that can be uniquely located with respect to a point on the earth's surface (Section 4.3.3, "Parametric Vertical Coordinate"; Appendix D, Parametric Vertical Coordinates).

It is often the case that data values are not representative of single points in time, space and other dimensions, but rather of intervals or multidimensional cells. CF defines a `bounds` attribute to specify the extent of intervals or cells. Because both the [NUG] and [COARDS] define coordinate variables but not cells or bounds, many applications assume that gridpoints are always located at

the centers of their cells. This assumption does not hold in CF. If bounds are not provided, the location of the gridpoint within the cell is undefined, and nothing can be assumed about the location and extent of the cell.

A two-dimensional cell is analogous to a pixel in a raster graphic, but is a more general concept. Pixels in a raster are evenly spaced in each dimension and arranged in a logically rectangular array. Two-dimensional cells in a CF field do not necessarily satisfy either of those conditions, though they commonly do. Furthermore, as an alternative to cells in two dimensions, CF defines a convention for the case where each data value is associated with a geographical feature that is described by one or more points, lines or polygons.

When data that is representative of cells can be described by simple statistical methods (for instance, mean or maximum), those methods can be indicated using the `cell_methods` attribute. An important application of this attribute is to describe climatological and diurnal statistics.

Methods for reducing the total volume of data include both packing and compression. Packing reduces the data volume by reducing the precision of the stored numbers. It is implemented using the attributes `add_offset` and `scale_factor` which are defined in the [NUG]. Compression on the other hand loses no precision, but reduces the volume by not storing missing data. The attribute `compress` is defined for this purpose.

1.5. Relationship to the COARDS Conventions

These conventions generalize and extend the COARDS conventions [COARDS]. A major design goal has been to maintain *backward compatibility* with COARDS. Hence applications written to process datasets that conform to these conventions will also be able to process COARDS conforming datasets. We have also striven to maximize *conformance* to the COARDS standard so that datasets that only require the metadata that was available under COARDS will still be able to be processed by COARDS conforming applications. But because of the extensions that provide new metadata content, and the relaxation of some COARDS requirements, datasets that conform to these conventions will not necessarily be recognized by applications that adhere to the COARDS conventions. The features of these conventions that allow writing netCDF files that are not COARDS conforming are summarized below.

COARDS standardizes the description of grids composed of independent latitude, longitude, vertical, and time axes. In addition to standardizing the metadata required to identify each of these axis types, COARDS requires (*time*, *vertical*, *latitude*, *longitude*) as the CDL order for the dimensions of a variable, with longitude being the most rapidly varying dimension (the last dimension in CDL order). Because of I/O performance considerations it may not be possible for models to output their data in conformance with the COARDS requirement. The CF convention places no rigid restrictions on the order of dimensions, however we encourage data producers to make the extra effort to stay within the COARDS standard order. The use of non-COARDS axis ordering will render files inaccessible to some applications and limit interoperability. Often a buffering operation can be used to miminize performance penalties when axis ordering in model code does not match the axis ordering of a COARDS file.

COARDS addresses the issue of identifying dimensionless vertical coordinates, but does not provide any mechanism for mapping the dimensionless values to dimensional ones that can be located with

respect to the earth's surface. For backwards compatibility we continue to allow (but do not require) the `units` attribute of dimensionless vertical coordinates to take the values "level", "layer", or "sigma_level." But we recommend that the `standard_name` and `formula_terms` attributes be used to identify the appropriate definition of the dimensionless vertical coordinate (see Section 4.3.3, "Parametric Vertical Coordinate").

The CF conventions define attributes which enable the description of data properties that are outside the scope of the COARDS conventions. These new attributes do not violate the COARDS conventions, but applications that only recognize COARDS conforming datasets will not have the capabilities that the new attributes are meant to enable. Briefly the new attributes allow:

- Identification of quantities using standard names.
- Description of dimensionless vertical coordinates.
- Associating dimensions with auxiliary coordinate variables.
- Linking data variables to scalar coordinate variables.
- Associating dimensions with labels.
- Description of intervals and cells.
- Description of properties of data defined on intervals and cells.
- Description of climatological statistics.
- Data compression for variables with missing values.

1.6. UGRID Conventions

These conventions implicitly incorporate parts of the UGRID conventions for storing unstructured (or flexible mesh) data in netCDF files using mesh topologies [UGRID]. Only version 1.0 of the UGRID conventions is allowed. The UGRID conventions description is referenced from, rather than rewritten into, this document and the canonical description of how to store mesh topologies is only to be found at [UGRID]. A summary indicating how UGRID relates to other parts of the CF conventions, and which features of UGRID are excluded from CF, can be found in Section 5.9, "Mesh Topology Variables". To reduce the chance of ambiguities arising from their accidental re-use, all of the UGRID standardized attributes are specified in Appendix K, Mesh Topology Attributes and Appendix A, Attributes.

The UGRID conventions have their own conformance document, which should be used in conjunction with the CF conformance document when checking the validity of datasets.

Chapter 2. NetCDF Files and Components

The components of a netCDF file are described in section 2 of the [NUG]. In this section we describe conventions associated with filenames and the basic components of a netCDF file. We also introduce new attributes for describing the contents of a file.

2.1. Filename

NetCDF files should have the file name extension "`.nc`".

2.2. Data Types

Data variables must be one of the following data types: `string`, `char`, `byte`, `unsigned byte`, `short`, `unsigned short`, `int`, `unsigned int`, `int64`, `unsigned int64`, `float` or `real`, and `double` (which are all the netCDF external data types supported by netCDF-4). The `string` type, which has variable length, is only available in files using the netCDF version 4 (netCDF-4) format. The `char` and `string` types are not intended for numeric data. One byte numeric data should be stored using the `byte` or `unsigned byte` data types. It is possible to treat the `byte` and `short` types as unsigned by using the [NUG] convention of indicating the unsigned range using the `valid_min`, `valid_max`, or `valid_range` attributes. In many situations, any integer type may be used. When the phrase "integer type" is used in this document, it should be understood to mean `byte`, `unsigned byte`, `short`, `unsigned short`, `int`, `unsigned int`, `int64`, or `unsigned int64`.

A text string can be stored either in a variable-length `string` or in a fixed-length `char` array. In both cases, text strings must be represented in Unicode Normalization Form C (NFC, section 3.11 and Annex 15 of the Unicode standard) and encoded according to UTF-8. A text string consisting only of ASCII characters is guaranteed to conform with this requirement, because the ASCII characters are a subset of Unicode, and their NFC UTF-8 encodings are the same as their one-byte ASCII codes (decimal 0-127, hexadecimal `00`-`7F`).

Before version 1.12, CF did not require UTF-8 encoding, and did not provide or endorse any convention to record what encoding was used. However, if the text string is stored in a `char` variable, the encoding might be recorded by the `_Encoding` attribute, although this is not a CF or NUG convention.

An *n*-dimensional array of strings may be implemented as a variable or an attribute of type `string` with *n* dimensions (only *n*=1 is allowed for an attribute) or as a variable of type `char` with *n*+1 dimensions, where the most rapidly varying dimension (the last dimension in CDL order) is large enough to contain the longest string in the variable. For example, a `char` variable containing the names of the months would be dimensioned (12,9) in order to accommodate "September", the month with the longest name. The other strings, such as "May", would be padded with trailing NULL or space characters so that every array element is filled. A `string` variable to store the same information would be dimensioned (12), with each element of the array containing a string of the appropriate length. The CDL example below shows one variable of each type.

Example 2.1. String Variable Representations

```
dimensions:
  strings = 30 ;
  strlen = 10 ;
variables:
  char char_variable(strings,strlen) ;
    char_variable:long_name = "strings of type char" ;
  string str_variable(strings) ;
    str_variable:long_name = "strings of type string" ;
```

The examples in this document that use string-valued variables alternate between these two forms.

2.3. Naming Conventions

It is recommended that variable, dimension, attribute and group names begin with a letter and be composed of letters, digits, and underscores. By the word *letters* we mean the standard ASCII letters uppercase `A` to `Z` and lowercase `a` to `z`. By the word *digits* we mean the standard ASCII digits `0` to `9`, and similarly *underscores* means the standard ASCII underscore `_`. Note that this is in conformance with the COARDS conventions, but is more restrictive than the netCDF interface which allows almost all Unicode characters encoded as multibyte UTF-8 characters (NUG Appendix B). The netCDF interface also allows leading underscores in names, but the NUG states that this is reserved for system use.

Case is significant in netCDF names, but it is recommended that names should not be distinguished purely by case, i.e., if case is disregarded, no two names should be the same. It is also recommended that names should be obviously meaningful, if possible, as this renders the file more effectively self-describing.

This convention does not standardize any variable or dimension names. Attribute names and their contents, where standardized, are given in English in this document and should appear in English in conforming netCDF files for the sake of portability. Languages other than English are permitted for variables, dimensions, and non-standardized attributes. The content of some standardized attributes are string values that are not standardized, and thus are not required to be in English. For example, a description of what a variable represents may be given in a non-English language using the `long_name` attribute (see Section 3.2, "Long Name") whose contents are not standardized, but a description given by the `standard_name` attribute (see Section 3.3, "Standard Name") must be taken from the standard name table which is in English.

2.4. Dimensions

A variable may have any number of dimensions, including zero, and the dimensions must all have different names. *COARDS strongly recommends limiting the number of dimensions to four, but we wish to allow greater flexibility*. The dimensions of the variable define the axes of the quantity it contains. Dimensions other than those of space and time may be included. Several examples can be found in this document. Under certain circumstances, one may need more than one dimension in a

particular quantity. For instance, a variable containing a two-dimensional probability density function might correlate the temperature at two different vertical levels, and hence would have temperature on both axes.

If any or all of the dimensions of a variable have the interpretations of "date or time" (`T`), "height or depth" (`Z`), "latitude" (`Y`), or "longitude" (`X`) then we recommend, but do not require (see Section 1.5, "Relationship to the COARDS Conventions"), those dimensions to appear in the relative order `T`, then `Z`, then `Y`, then `X` in the CDL definition corresponding to the file. All other dimensions should, whenever possible, be placed to the left of the spatiotemporal dimensions.

Dimensions may be of any size, including unity. When a single value of some coordinate applies to all the values in a variable, the recommended means of attaching this information to the variable is by use of a dimension of size unity with a one-element coordinate variable. It is also acceptable to use a scalar coordinate variable which eliminates the need for an associated size one dimension in the data variable. The advantage of using either a coordinate variable or an auxiliary coordinate variable is that all its attributes can be used to describe the single-valued quantity, including boundaries. For example, a variable containing data for temperature at 1.5 m above the ground has a single-valued coordinate supplying a height of 1.5 m, and a time-mean quantity has a single-valued time coordinate with an associated boundary variable to record the start and end of the averaging period.

2.5. Variables

This convention does not standardize variable names.

NetCDF variables that contain coordinate data are referred to as *coordinate variables*, *auxiliary coordinate variables*, *scalar coordinate variables*, or *multidimensional coordinate variables*.

2.5.1. Missing data, valid and actual range of data

NUG Appendix A, Attribute Conventions provide the `_FillValue`, `missing_value`, `valid_min`, `valid_max`, and `valid_range` attributes to indicate missing data. Missing data is allowed in data variables and auxiliary coordinate variables. Generic applications should treat the data as missing where any auxiliary coordinate variables have missing values; special-purpose applications might be able to make use of the data. Missing data is not allowed in coordinate variables.

The NUG conventions for missing data changed significantly between version 2.3 and version 2.4. Since version 2.4 the NUG defines missing data as all values outside of the `valid_range`, and specifies how the `valid_range` should be defined from the `_FillValue` (which has library specified default values) if it hasn't been explicitly specified. If only one missing value is needed for a variable then we recommend that this value be specified using the `_FillValue` attribute. Doing this guarantees that the missing value will be recognized by generic applications that follow either the before or after version 2.4 conventions.

The scalar attribute with the name `_FillValue` and of the same type as its variable is recognized by the netCDF library as the value used to pre-fill disk space allocated to the variable. This value is considered to be a special value that indicates undefined or missing data, and is returned when reading values that were not written. The `_FillValue` should be outside the range specified by `valid_range` (if used) for a variable. The netCDF library defines a default fill value for each data type

(See the "Note on fill values" in NUG Appendix B, File Format Specifications).

The missing values of a variable with `scale_factor` and/or `add_offset` attributes (see Section 8.1, "Packed Data") are interpreted relative to the variable's external values (a.k.a. the packed values, the raw values, the values stored in the netCDF file), not the values that result after the scale and offset are applied. Applications that process variables that have attributes to indicate both a transformation (via a scale and/or offset) and missing values should first check that a data value is valid, and then apply the transformation. Note that values that are identified as missing should not be transformed. Since the missing value is outside the valid range it is possible that applying a transformation to it could result in an invalid operation. For example, the default `_FillValue` is very close to the maximum representable value of IEEE single precision floats, and multiplying it by 100 produces an "Infinity" (using single precision arithmetic).

This convention defines a two-element vector attribute `actual_range` for variables containing numeric data. If the variable is packed using the `scale_factor` and `add_offset` attributes (see Section 8.1, "Packed Data"), the elements of the `actual_range` should have the type intended for the unpacked data. The elements of `actual_range` must be exactly equal to the minimum and the maximum data values which occur in the variable (when unpacked if packing is used), and both must be within the `valid_range` if specified. If the data is all missing or invalid, the `actual_range` attribute cannot be used.

2.6. Attributes

This standard describes many attributes (some mandatory, others optional), but a file may also contain non-standard attributes. Such attributes do not represent a violation of this standard. Application programs should ignore attributes that they do not recognise or which are irrelevant for their purposes. Conventional attribute names should be used wherever applicable. Non-standard names should be as meaningful as possible. Before introducing an attribute, consideration should be given to whether the information would be better represented as a variable. In general, if a proposed attribute requires ancillary data to describe it, is multidimensional, requires any of the defined netCDF dimensions to index its values, or requires a significant amount of storage, a variable should be used instead. When this standard defines string attributes that may take various prescribed values, the possible values are generally given in lower case. However, applications programs should not be sensitive to case in these attributes. Several string attributes are defined by this standard to contain "blank-separated lists". Consecutive words in such a list are separated by one or more adjacent spaces. The list may begin and end with any number of spaces. See Appendix A, Attributes for a list of attributes described by this standard.

2.6.1. Identification of Conventions

Files that follow this version of the CF Conventions must indicate this by setting the [NUG] defined global attribute `Conventions` to a string value that contains "`CF-1.13-draft`". The Conventions version number contained in that string can be used to find the web based versions of this document are from the netCDF Conventions web page. Subsequent versions of the CF Conventions will not make invalid a compliant usage of this or earlier versions of the CF terms and forms.

It is possible for a netCDF file to adhere to more than one set of conventions, even when there is no inheritance relationship among the conventions. In this case, the value of the Conventions attribute

may be a single text string containing a list of the convention names separated by blank space (recommended) or commas (if a convention name contains blanks). This is the Unidata recommended syntax from NetCDF Users Guide, Appendix A. If the string contains any commas, it is assumed to be a comma-separated list.

When CF is listed with other conventions, this asserts the same full compliance with CF requirements and interpretations as if CF was the sole convention. It is the responsibility of the data-writer to ensure that all common metadata is used with consistent meaning between conventions.

The UGRID conventions, which are fully incorporated into the CF conventions, do not need to be included in the `Conventions` attribute.

2.6.2. Description of file contents

The following attributes are intended to provide information about where the data came from and what has been done to it. This information is mainly for the benefit of human readers. The attribute values are all character strings. For readability in ncdump outputs it is recommended to embed newline characters into long strings to break them into lines. For backwards compatibility with COARDS none of these global attributes is required.

The [NUG] defines `title` and `history` to be global attributes. We wish to allow the newly defined attributes, i.e., `institution`, `source`, `references`, and `comment`, to be either global or assigned to individual variables. When an attribute appears both globally and as a variable attribute, the variable's version has precedence.

`title`

A succinct description of what is in the dataset.

`institution`

Specifies where the original data was produced.

`source`

The method of production of the original data. If it was model-generated, `source` should name the model and its version, as specifically as could be useful. If it is observational, `source` should characterize it (e.g., "`surface observation`" or "`radiosonde`").

`history`

Provides an audit trail for modifications to the original data. Well-behaved generic netCDF filters will automatically append their name and the parameters with which they were invoked to the global history attribute of an input netCDF file. We recommend that each line begin by indicating the date and time of day that the program was executed.

`references`

Published or web-based references that describe the data or methods used to produce it.

`comment`

Miscellaneous information about the data or methods used to produce it.

2.6.3. External Variables

The global `external_variables` attribute is a blank-separated list of the names of variables which are named by attributes in the file but which are not present in the file. These variables are to be found in other files (called "external files") but CF does not provide conventions for identifying the files concerned. The only attribute for which CF standardises the use of external variables is `cell_measures`.

2.7. Groups

Groups provide a powerful mechanism to structure data hierarchically. This convention does not standardize group names. It may be of benefit to name groups in such a way that human readers can interpret them. However, files that conform to this standard shall not require software to interpret or decode information from group names. References to out-of-group variable and dimensions shall be found by applying the scoping rules outlined below.

2.7.1. Scope

The scoping mechanism is in keeping with the following principle:

> "Dimensions are scoped such that they are visible to all child groups. For example, you can define a dimension in the root group, and use its dimension id when defining a variable in a sub-group."
>
> — The NetCDF Data Model: Groups

Any variable or dimension can be referred to, as long as it can be found with one of the following search strategies:

- Search by absolute path
- Search by relative path
- Search by proximity

These strategies are explained in detail in the following sections.

If any dimension of an out-of-group variable has the same name as a dimension of the referring variable, the two must be the same dimension (i.e. they must have the same netCDF dimension ID).

Search by absolute path

A variable or dimension specified with an absolute path (i.e., with a leading slash "/") is at the indicated location relative to the root group, as in a UNIX-style file convention. For example, a `coordinates` attribute of `/g1/lat` refers to the `lat` variable in group `/g1`.

Search by relative path

As in a UNIX-style file convention, a variable or dimension specified with a relative path (i.e., containing a slash but not with a leading slash, e.g. `child/lat`) is at the location obtained by affixing

the relative path to the absolute path of the referring attribute. For example, a `coordinates` attribute of `g1/lat` refers to the `lat` variable in subgroup `g1` of the current (referring) group. Upward path traversals from the current group are indicated with the UNIX convention. For example, `../g1/lat` refers to the `lat` variable in the sibling group `g1` of the current (referring) group.

Search by proximity

A variable or dimension specified with no path (for example, `lat`) refers to the variable or dimension of that name, if there is one, in the referring group. If not, the ancestors of the referring group are searched for it, starting from the direct ancestor and proceeding toward the root group, until it is found.

A special case exists for coordinate variables. Because coordinate variables must share dimensions with the variables that reference them, the ancestor search is executed only until the local apex group is reached. For coordinate variables that are not found in the referring group or its ancestors, a further strategy is provided, called lateral search. The lateral search proceeds downwards from the local apex group width-wise through each level of groups until the sought coordinate is found. The lateral search algorithm may only be used for [NUG] coordinate variables; it shall not be used for auxiliary coordinate variables.

NOTE	This use of the lateral search strategy to find them is discouraged. They are allowed mainly for backwards-compatibility with existing datasets, and may be deprecated in future versions of the standard.

2.7.2. Application of attributes

The following attributes are optional for non-root groups. They are allowed in order to provide additional provenance and description of the subsidiary data. They do not override attributes from parent groups.

- `title`
- `history`

If these attributes are present, they may be applied additively to the parent attributes of the same name. If a file containing groups is modified, the user or application need only update these attributes in the root group, rather than traversing all groups and updating all attributes that are found with the same name. In the case of conflicts, the root group attribute takes precedence over per-group instances of these attributes.

The following attributes may only be used in the root group and shall not be duplicated or overridden in child groups:

- `Conventions`
- `external_variables`

Furthermore, per-variable attributes must be attached to the variables to which they refer. They may not be attached to a group, even if all variables within that group use the same attribute and value.

If attributes are present within groups without being attached to a variable, these attributes apply to the group where they are defined, and to that group's descendants, but not to ancestor or sibling groups. If a group attribute is defined in a parent group, and one of the child group redefines the same attribute, the definition within the child group applies for the child and all of its descendants.

2.8. Aggregation Variables

An *aggregation variable* is a variable which has been formed by combining (i.e. aggregating) multiple *fragments* that are generally stored in *fragment datasets* that are external to the dataset containing the aggregation variable, i.e. the *aggregation dataset*. A fragment contains data with sufficient metadata for it to be correctly interpreted in the context of the aggregation. The aggregation variable does not contain any actual data, instead it contains instructions on how to create its *aggregated data* in memory as an aggregation of the data from each fragment. The aggregated data is identical to that which would be stored in the dataset if the variable were encoded in usual (i.e. non-aggregated) manner.

Aggregation provides the utility of being able to view, as a single entity, a dataset that has been partitioned across multiple other datasets. This view takes up very little extra space on disk, since the aggregation dataset contains no copies of the data in the fragments. Fragment datasets may be CF-compliant or have any other format, thereby allowing an aggregation variable to act as a CF-compliant view of non-CF datasets. Aggregations can facilitate a range of activities such as data analysis, by avoiding the computational expense of deriving the aggregation at the time of analysis; archive curation, by acting as a metadata-rich archive index; and the post-processing of model simulation outputs, by spanning multiple datasets written at run time that together constitute a more cohesive and useful product.

An aggregation variable must be a scalar (i.e. it has no dimensions). It acts as a container for all of the usual attributes that describe a variable, with the addition of two special attributes: one that defines its *aggregated dimensions* (i.e. the dimensions of the aggregated data, which in turn define the aggregated data shape); and one that provides the instructions on how the aggregated data is to be created. These attributes are described in Section 2.8.1, "Aggregated Dimensions and Data". The data type of the aggregation variable indicates the data type of the aggregated data, and the value of the aggregation variable's single element is immaterial.

Aggregation variables may be used as any kind of variable (data variable, coordinate variable, cell measures variable, etc.), but it is recommended that container variables whose data are immaterial (such as grid mapping variables) not be encoded as aggregation variables. A dataset may contain both aggregation and non-aggregation variables, and it is up to the data-writer to decide which variables, if any, are stored as aggregation variables.

Any rules that apply to a variable in the CF conventions apply in exactly the same way to an aggregation variable in the same role; and any reference to the dimensions or data of a variable applies to the aggregated dimensions or aggregated data, respectively, of an aggregation variable. For instance:

- The dimension of a coordinate variable of an aggregation data variable is included as one of the aggregated dimensions of the aggregation data variable.
- The name of an aggregation coordinate variable (which is a scalar) is the same as the name of

its single aggregated dimension (identified by its `aggregated_dimensions` attribute), just as the name of a coordinate variable (which is one-dimensional) is the same as the name of its single dimension.

The details of how to encode and decode aggregation variables are given in this section, with extra examples provided in Appendix L, Aggregation Variable Examples.

2.8.1. Aggregated Dimensions and Data

If a variable has an `aggregated_dimensions` attribute then it must be an aggregation variable. This attribute records the names of the aggregated dimensions as a blank-separated list, in the order of the dimensions of the aggregated data. If the aggregated data is scalar then there are no aggregated dimensions and the `aggregated_dimensions` attribute must be an empty string. Any aggregated dimensions must exist as dimensions in the aggregation dataset.

The aggregated dimensions are partitioned by the fragments (in their canonical forms, see Section 2.8.2 "Fragment Interpretation"), and this partitioning is consistent across all of the fragments, i.e. any two fragments either span the same part of a given aggregated dimension, or else do not overlap along that same dimension. In addition, each fragment data value provides exactly one aggregated data value, and each aggregated data value comes from exactly one fragment. With these constraints, the fragments can be organised into a fully-populated orthogonal multidimensional *array of fragments*, for which the size of each dimension is equal to the number of fragments that span its corresponding aggregated dimension. See Example 2.2 for a schematic representation of an array of fragments.

The aggregated data is formed by combining the fragments in the same relative positions as they appear in the array of fragments, and with no gaps or overlaps between neighbouring fragments.

Example 2.2. Schematic representation of an array of fragments for an aggregation variable

Array of fragments

Position `[0, 0, 0]` Fragment dataset name: `file_A.nc` Fragment data shape: `(17, 90, 180)` `17` vertical levels `[90, 0]` degrees north `[0, 180]` degrees east	**Position `[0, 0, 1]`** Fragment dataset name: `file_B.nc` Fragment data shape: `(17, 90, 180)` `17` vertical levels `[90, 0]` degrees north `[180, 360]` degrees east
Position `[0, 1, 0]` Fragment dataset name: `file_C.nc` Fragment data shape: `(17, 45, 180)` `17` vertical levels `[0, -45]` degrees north `[0, 180]` degrees east	**Position `[0, 1, 1]`** Fragment dataset name: `file_D.nc` Fragment data shape: `(17, 45, 180)` `17` vertical levels `[0, -45]` degrees north `[180, 360]` degrees east

Position `[0, 2, 0]`	**Position `[0, 2, 1]`**
Fragment dataset name: `file_E.nc` Fragment data shape: `(17, 45, 180)` `17` vertical levels `[-45, -90]` degrees north `[0, 180]` degrees east	Fragment dataset name: `file_F.nc` Fragment data shape: `(17, 45, 180)` `17` vertical levels `[-45, -90]` degrees north `[180, 360]` degrees east

The six fragment datasets are arranged in a three-dimensional array of fragments with shape `(1, 3, 2)`. Each fragment spans the entirety of the Z dimension, but only a part of the Y-X plane, which has 1 degree resolution. The fragments combine to create three-dimensional aggregated data that have global Z-Y-X coverage, with shape `(17, 180, 360)`. The Z aggregated dimension is spanned by one fragment, the Y aggregated dimension is spanned by three fragments, and the X aggregated dimension is spanned by two fragments. Note that, since this example is a schematic representation, the C or Fortran order of the dimensions is of no consequence.

See Example 2.3 for an encoding of an aggregation variable that uses these fragments.

The array of fragments must be defined by an aggregation variable's `aggregated_data` attribute. This attribute must take a string value comprising blank-separated elements of the form "*feature: variable*", where *feature* is a case-sensitive keyword that specifies a feature of the array of fragments, and *variable* is a variable in the aggregation dataset that provides values for that feature. The order of elements in the `aggregated_data` attribute is not significant.

The feature keywords must comprise either all three of `map`, `uris`, and `identifiers`; or else both of `map` and `unique_values`. No other combination of feature keywords is allowed. The variables that correspond to these features are defined as follows:

map

The integer-valued `map` variable maps each fragment (in its canonical form, see Section 2.8.2 "Fragment Interpretation") to a part of the aggregated data. The `map` variable data provides the sizes of the fragments along each of the aggregated dimensions. The `map` variable is two-dimensional: the rows (i.e. the slowest-varying dimension, and the first dimension in CDL order) correspond to the aggregated dimensions in the same order; and the columns correspond to the fragments along the aggregated dimensions. Since the aggregated dimensions can be spanned by differing numbers of fragments, the rows of the `map` variable are padded with missing values to create a rectangular array. The part of each aggregated dimension that is occupied by a given fragment is defined by that fragment's size along that dimension, offset by the sum of the fragment sizes that precede it.

For instance, in Example 2.2, the corresponding `map` variable has 3 rows (one for each of the Z, Y, and X aggregated dimensions), and 3 columns (to allow space for the largest number of fragments along any of the aggregated dimensions). Each of the rows contains the sizes of the fragments along that dimension, padded with missing values (denoted by _), to create a rectangular array:

```
17   _   _
90  45  45
```

```
180 180   _
```

From this array we can deduce, for instance, that the shape of the fragment (in its canonical form, see Section 2.8.2 "Fragment Interpretation") at position `[0, 1, 1]` of the array of fragments is `(17, 45, 180)`; and that this fragment occupies zero-based indices 0 to 16 of the Z aggregated dimension, 90 to 134 of the Y aggregated dimension, and 180 to 359 of the X aggregated dimension. See Example 2.3.

In the special case that aggregated data is scalar, the `map` variable must also be scalar and contain the value `1`. See Example L.6.

uris

The string-valued `uris` variable defines the name of each fragment dataset. Its dimensions are those of the array of fragments; and its data provides each fragment dataset name in the form of a URI (Uniform Resource Identifier) [URI] . Each URI must be either an *absolute URI* (a URI that begins with a scheme component followed by a `:` character, such as `file:///data/file.nc`, `https://remote.host/data/file.nc`, `s3://remote.host/data/file.nc`, or `locally_meaningful_protocol:///UID`), or else a *relative-path URI reference* (a URI that is not an absolute URI and which does not begin with a `/` or `#` character, such as `file.nc`, `../file.nc`, or `data/file.nc`). A relative-path URI reference is taken as being relative to the location of the aggregation dataset. If the aggregation dataset is moved to another location, then a fragment dataset identified by an absolute URI will still be accessible, whereas a fragment dataset identified by a relative-path URI reference will also need be moved to preserve the relative reference. Not all fragment dataset names need be of the same URI type. See Example L.1 and Example L.2.

identifiers

The `identifiers` variable defines how to identify each fragment within its fragment dataset. In general, the dimensions of the `identifiers` variable are the same, and in the same order, as those of the `uris` variable, and its data contain an identifier corresponding to each fragment dataset. If, however, the identifiers are the same for all fragments then the `identifiers` variable may be a scalar whose single data value is the identifier common to all fragments. The identifier for a netCDF fragment dataset is the string-valued variable name of the fragment, which does not need to be the same as the name of the aggregation variable. See Example L.1 and Example L.4.

unique_values

When the data values within each fragment are all identical, the `unique_values` variable allows these unique values to be explicitly stored in the aggregation dataset, without reference to external fragment datasets via `uris` and `identifiers` variables. The `unique_values` variable dimensions are those of the array of fragments, and the data provide the unique value for each fragment. The fragment implied by a unique value has dimensions corresponding to the aggregated dimensions, and the fragment shape is defined by the `map` variable. When a fragment contains wholly missing data, its unique value is specified as any missing value defined by the aggregation variable. See Example L.5.

Example 2.3. An aggregation variable

```
dimensions:
  level = 17 ;
  latitude = 180 ;
  longitude = 360 ;
  // Array of fragments dimensions
  f_level = 1 ;
  f_latitude = 3 ;
  f_longitude = 2 ;
  // Map variable dimensions
  j = 3 ;        // Number of aggregated dimensions
  i = 3 ;        // Largest number of fragments along any aggregated dimension

variables:
  // Data aggregation variable
  double temperature ;
    temperature:standard_name = "air_temperature" ;
    temperature:units = "K" ;
    temperature:cell_methods = "time: mean" ;
    temperature:aggregated_dimensions = "level latitude longitude" ;
    temperature:aggregated_data = "map: fragment_map
                                   uris: fragment_uris
                                   identifiers: fragment_identifiers" ;
  // Coordinate variables
  double level(level) ;
    level:standard_name = "air_pressure" ;
    level:units = "hPa" ;
  double latitude(latitude) ;
    latitude:standard_name = "latitude" ;
    latitude:units = "degrees_north" ;
  double longitude(longitude) ;
    longitude:standard_name = "longitude" ;
    longitude:units = "degrees_east" ;
  // Array of fragments variables
  int fragment_map(j, i) ;
  string fragment_uris(f_level, f_latitude, f_longitude) ;
  string fragment_identifiers ;

data:
  temperature = _ ;
  level = ... ;
  latitude = ... ;
  longitude = ... ;
  fragment_map = 17, _, _,
                 90, 45, 45,
                 180, 180, _ ;
  fragment_uris = "file_A.nc", "file_B.nc",
                  "file_C.nc", "file_D.nc",
                  "file_E.nc", "file_F.nc" ;
```

```
   fragment_identifiers = "tmp" ;
```

An encoding for the aggregated data defined by the fragments described in Example 2.2. The `temperature` variable's data is an aggregation of six fragments. The non-missing values in each row of the `fragment_map` variable's data indicate that the `level` aggregated dimension is spanned by one fragment, the `latitude` aggregated dimension is spanned by three fragments, and the `longitude` aggregated dimension is spanned by two fragments. Therefore the shape of the array of fragments is `(1, 3, 2)`. Note that the row sums of the `fragment_map` variable are `17`, `180`, and `360`, which equal the sizes of the `level`, `latitude`, and `longitude` aggregated dimensions, respectively.

The data for the `level`, `latitude` and `longitude` variables are omitted for clarity.

2.8.2. Fragment Interpretation

Fragment datasets can be encoded in many different but equivalent ways, so we define a *canonical form* of a fragment that provides a view of the fragment for which its data are consistent with the data from other fragments, as well as with the attributes of the aggregation variable. When constructing the aggregated data, it is assumed that each fragment's data has been transformed to its canonical form. The canonical form of a fragment's data is such that:

- The fragment's data have the same number of dimensions, and in the same order, as the aggregated data.
- The fragment's data have the same units as the aggregation variable.
- The fragment's data have the same data type as the aggregation variable.
- Missing values in the fragment's data the same as those defined by the aggregation variable.
- The fragment's data are unpacked (as described in Section 8.1, "Packed Data").

A fragment dataset can deviate from any of these requirements, provided that it is possible to convert the fragment to its canonical form without changing the meaning of the data. For instance, if the aggregation variable had units of kg m^{-2}, then the fragment data within its dataset could have any of the equivalent units kg m^{-2}, g cm^{-2}, etc., but it would be an error if the fragment had units for which its values can not be converted to kg m^{-2} (such m s^{-1}). Similarly, if the aggregation variable had a data type of `float`, then the fragment data could have any numerical data type.

The conversion of the fragment's data to its canonical form is carried out by the application program which is creating the aggregated data in memory. The application program can ignore any metadata and variables in a fragment dataset that are not needed for the conversion to the canonical form. When transforming a fragment's data to its canonical form, note that:

- A fragment can have fewer dimensions than the aggregated data, provided that the missing dimensions have size 1 (e.g. as could be the case when aggregating two-dimensional fragments into three-dimensional aggregated data); but a fragment can not have more dimensions than the aggregated data.
- It is the responsibility of the creator of the aggregation dataset to ensure that all valid values in a fragment's data are different from any of the missing values defined by the aggregation

variable.

- It is up to the application program to decide if any modifications to the values in the fragment dataset are acceptable, in terms of information loss (e.g. whether or not to create aggregated data with data type `int` when some of the fragments have data type `float`).
- The aggregated data is identical to the data that would be stored within a dataset that contained the equivalent non-aggregation variable. A consequence of this is that when the aggregation variable indicates that its data are packed or compressed (such as by techniques described in Chapter 8, Reduction of Dataset Size) then the aggregated data, after its creation, is subject to the aggregation variable's unpacking or decompression procedures.

Chapter 3. Description of the Data

The attributes described in this section are used to provide a description of the content and the units of measurement for each variable. We continue to support the use of the `units` and `long_name` attributes as defined in COARDS. We extend COARDS by adding the optional `standard_name` attribute which is used to provide unique identifiers for variables. This is important for data exchange since one cannot necessarily identify a particular variable based on the name assigned to it by the institution that provided the data.

The `standard_name` attribute can be used to identify variables that contain coordinate data. But since it is an optional attribute, applications that implement these standards must continue to be able to identify coordinate types based on the COARDS conventions.

3.1. Units

The `units` attribute is required for all variables that represent dimensional quantities (except for boundary variables defined in Section 7.1, "Cell Boundaries" and climatology boundary variables defined in Section 7.4, "Climatological Statistics"). The `units` attribute is permitted but not required for dimensionless quantities (see Section 3.1.1, "Dimensionless units").

The value of the `units` attribute is a string that can be recognized by the UDUNITS package [UDUNITS], with the exceptions that are given in Section 3.1.1, "Dimensionless units" and Section 3.1.3, "Scale factors and offsets". Note that case is significant in the `units` strings. Note also that CF depends on UDUNITS only for the definition of legal `units` strings. CF does not assume or require that the UDUNITS software will be used for `units` conversion. In most `units` conversions, the sole operation on the data is multiplication by a scale factor. Special treatment is required in converting the `units` of variables that involve temperature (Section 3.1.2, "Temperature units") and the `units` of time coordinate variables (Section 4.4, "Time Coordinate").

The COARDS convention prohibits the unit `degrees` altogether, but this unit is not forbidden by the CF convention because it may in fact be appropriate for a variable containing, say, solar zenith angle. The unit `degrees` is also allowed on coordinate variables such as the latitude and longitude coordinates of a transformed grid. In this case the coordinate values are not true latitudes and longitudes, which must always be identified using the more specific forms of `degrees` as described in Section 4.1, "Latitude Coordinate" and Section 4.2, "Longitude Coordinate".

3.1.1. Dimensionless units

A variable with no `units` attribute is assumed to be dimensionless. However, a `units` attribute specifying a dimensionless unit may optionally be included. The canonical unit (see also Section 3.3, "Standard Name") for dimensionless quantities that represent fractions, or parts of a whole, is `1`. The UDUNITS package defines a few dimensionless units, such as `percent`, `ppm` (parts per million, 1e-6), and `ppb` (parts per billion, 1e-9). As an alternative to the canonical `units` of `1` or some other unitless number, the `units` for a dimensionless quantity may be given as a ratio of dimensional units, for instance `mg kg-1` for a mass ratio of 1e-6, or `microlitre litre-1` for a volume ratio of 1e-6. Data-producers are invited to consider whether this alternative would be more helpful to the users of their data.

The CF convention supports dimensionless units that are UDUNITS compatible, with one exception, concerning the dimensionless units defined by UDUNITS for volume ratios, such as `ppmv` and `ppbv`. These units are allowed in the **`units`** attribute by CF only if the data variable has no **`standard_name`**. These units are prohibited by CF if there is a **`standard_name`**, because the **`standard_name`** defines whether the quantity is a volume ratio, so the **`units`** are needed only to indicate a dimensionless number.

Information describing a dimensionless physical quantity itself (e.g. "area fraction" or "probability") does not belong in the **`units`** attribute, but should be given in the **`long_name`** or **`standard_name`** attributes (see Section 3.2, "Long Name" and Section 3.3, "Standard Name"), in the same way as for physical quantities with dimensional units. As an exception, to maintain backwards compatibility with COARDS, the text strings `level`, `layer`, and `sigma_level` are allowed in the **`units`** attribute, in order to indicate dimensionless vertical coordinates. This use of **`units`** is not compatible with UDUNITS, and is deprecated by this standard because conventions for more precisely identifying dimensionless vertical coordinates are available (see Section 4.3.2, "Dimensionless Vertical Coordinate").

The UDUNITS syntax that allows scale factors and offsets to be applied to a unit is not supported by this standard, except for case of specifying reference time, see section Section 4.4, "Time Coordinate". The application of any scale factors or offsets to data should be indicated by the **`scale_factor`** and **`add_offset`** attributes. Use of these attributes for data packing, which is their most important application, is discussed in detail in Section 8.1, "Packed Data".

3.1.2. Temperature units

The **`units`** of temperature imply an origin (i.e. zero point) for the associated measurement scale. When the temperature value is the degree of warmth with respect to the origin of the measurement scale, we call it an *on-scale temperature*. When **`units`** of on-scale temperature are converted, the data may require the addition of an offset as well as multiplication by a scale factor, because the physical meaning of a numerical value of zero for an on-scale temperature depends on the unit of measurement. On-scale temperature is *unique* among quantities in the respect that the origin and the unit of measurement are both defined by the **`units`** and therefore cannot be chosen independently. For all other quantities, the origin and the unit of measurement are independent. Converting the unit of measurement alone, without changing the origin, does not change the meaning of zero. For example (using **bold** to indicate a numerical data value), **0** `kilogram` is the same mass as **0** `pound`, and **0** `seconds since 1970-1-1` means the same as **0** `days since 1970-1-1`, but **0** `degC` is not the same temperature as **0** `degF` (= **-17.8** `degC`), because these two temperature **`units`** implicitly refer to measurement scales which have different origins.

On the other hand, when the temperature value is a *temperature difference*, which compares two on-scale temperatures with the same origin, the value of that origin is irrelevant as it cancels out when taking the difference. Therefore to convert the **`units`** of a temperature difference requires only multiplication by a scale factor, without the addition of an offset.

The **`units`** attribute does not distinguish between on-scale temperatures and temperature differences. This ambiguity also affects units of temperature raised to some power e.g. `K^2` or multiplied by other units e.g. `W m-2 K-1`, `degF/foot` or `degC m s-1`. A **`standard_name`** (Section 3.3, "Standard Name") or **`standard_name`** modifier (Appendix C, Standard Name Modifiers) may clarify the intention, but they are optional. Some statistical operations described by the **`cell_methods`**

attribute (Section 7.3, "Cell Methods"; Appendix E, Cell Methods) imply that temperature must be interpreted as temperature difference, but this attribute is optional too.

In order to convert the **units** correctly, it is essential to know whether a temperature is on-scale or a difference. Therefore this standard strongly recommends that any variable whose **units** involve a temperature unit should also have a **units_metadata** attribute to make the distinction. This attribute must have one of the following three values: `temperature: on_scale`, `temperature: difference`, `temperature: unknown`. The **units_metadata** attribute, **standard_name** modifier (Appendix C, Standard Name Modifiers) and **cell_methods** attribute (Appendix E, Cell Methods) must be consistent if present. A variable must not have a **units_metadata** attribute if it has no **units** attribute or if its **units** do not involve a temperature unit.

Example 3.1. Use of **units_metadata** *to distinguish temperature quantities*

```
variables:
  float Tonscale;
    Tonscale:long_name="global-mean surface temperature";
    Tonscale:standard_name="surface_temperature";
    Tonscale:units="degC";
    Tonscale:units_metadata="temperature: on_scale";
    Tonscale:cell_methods="area: mean";
  float Tdifference;
    Tdifference:long_name="change in global-mean surface temperature relative to
pre-industrial";
    Tdifference:standard_name="surface_temperature";
    Tdifference:units="degC";
    Tdifference:units_metadata="temperature: difference";
    Tdifference:cell_methods="area: mean";
```

With `temperature: unknown`, correct conversion of the **units** cannot be guaranteed. This value of **units_metadata** indicates that the data-writer does not know whether the temperature is on-scale or a difference. If the **units_metadata** attribute is not present, the data-reader should assume `temperature: unknown`. The **units_metadata** attribute was introduced in CF 1.11. In data written according to versions before 1.11, `temperature: unknown` should be assumed for all **units** involving temperature, if it cannot be deduced from other metadata. We note (for guidance *only* regarding `temperature: unknown`, *not* as a CF convention) that the UDUNITS software assumes `temperature: on_scale` for **units** strings containing only a unit of temperature, and `temperature: difference` for **units** strings in which a unit of temperature is raised to any power other than unity, or multiplied or divided by any other unit.

With `temperature: on_scale`, correct conversion can be guaranteed only for pure temperature **units**. If the quantity is an on-scale temperature multiplied by some other quantity, it is not possible to convert the data from the **units** given to any other **units** that involve a temperature with a different origin, given only the **units**. For instance, when temperature is on-scale, a value in `kg degree_C m-2` can be converted to a value in `kg K m-2` only if we know separately the values in `degree_C` and `kg m-2` of which it is the product.

3.1.3. Scale factors and offsets

UDUNITS recognises the SI prefixes shown in Prefixes for decimal multiples and submultiples of units for decimal multiples and submultiples of units, and allows them to be applied to non-SI units as well. UDUNITS offers a syntax for indicating arbitrary scale factors and offsets to be applied to a unit. (Note that this is different from the scale factors and offsets used for converting between `units`, as discussed for temperature in Section 3.1.2, "Temperature units".) This UDUNITS syntax for arbitrary transformation of `units` is not supported by the CF standard, except for the case of specifying reference time (Section 4.4, "Time Coordinate"). The application of any scale factors or offsets to data should be indicated by the `scale_factor` and `add_offset` attributes. Use of these attributes for data packing, which is their most important application, is discussed in detail in Section 8.1, "Packed Data".

Table 3.1. Prefixes for decimal multiples and submultiples of units

Factor	Prefix	Abbreviation		Factor	Prefix	Abbreviation
1e1	deca,deka	da		1e-1	deci	d
1e2	hecto	h		1e-2	centi	c
1e3	kilo	k		1e-3	milli	m
1e6	mega	M		1e-6	micro	u
1e9	giga	G		1e-9	nano	n
1e12	tera	T		1e-12	pico	p
1e15	peta	P		1e-15	femto	f
1e18	exa	E		1e-18	atto	a
1e21	zetta	Z		1e-21	zepto	z
1e24	yotta	Y		1e-24	yocto	y

3.2. Long Name

The `long_name` attribute is defined by the [NUG] to contain a long descriptive name which may, for example, be used for labeling plots. For backwards compatibility with COARDS this attribute is optional. But it is highly recommended that either this or the `standard_name` attribute defined in the next section be provided for all data variables and variables containing coordinate data, in order to make the file self-describing. If a variable has no `long_name` attribute then an application may use, as a default, the `standard_name` if it exists, or the variable name itself.

3.3. Standard Name

A fundamental requirement for exchange of scientific data is the ability to describe precisely the physical quantities being represented. To some extent this is the role of the `long_name` attribute as defined in the [NUG]. However, usage of `long_name` is completely ad-hoc. For many applications it is desirable to have a more definitive description of the quantity, which allows users of data from different sources (some of which might be models and others observational) to determine whether

quantities are in fact comparable. For this reason each variable may optionally be given a "standard name", whose meaning is defined by this convention. There may be several variables in a dataset with any given standard name, and these may be distinguished by other metadata, such as coordinates (Chapter 4, Coordinate Types) and `cell_methods` (Section 7.3, "Cell Methods").

A standard name is associated with a variable via the attribute `standard_name` which takes a string value comprised of a standard name optionally followed by one or more blanks and a standard name modifier (a string value from Appendix C, Standard Name Modifiers).

The set of permissible standard names is contained in the standard name table. The table entry for each standard name contains the following:

standard name

The name used to identify the physical quantity. A standard name contains no whitespace and is case sensitive.

canonical units

Representative units of the physical quantity. Unless it is dimensionless, a variable with a `standard_name` attribute must have units which are physically equivalent (not necessarily identical) to the canonical units, possibly modified by an operation specified by the standard name modifier (see below and Appendix C, Standard Name Modifiers) or by the `cell_methods` attribute (see Section 7.3, "Cell Methods" and Appendix E, Cell Methods) or both.

Units of time coordinates (Section 4.4, "Time Coordinate"), whose `units` attribute includes the word `since`, are *not* physically equivalent to time units that do not include `since` in the `units`. To mark this distinction, the canonical unit given for quantities used for time coordinates is `s since 1958-1-1`. The reference datetime in the canonical unit (the beginning of the day i.e. midnight on 1st January 1958 at 0 `degrees_east`) is not restrictive; the time coordinate variable's own `units` may contain any reference datetime (after `since`) that is valid in its calendar. (We use `1958-1-1` because it is the beginning of International Atomic Time, and a valid datetime in all CF calendars; see also Section 4.4.3, "Leap Seconds".) In both kinds of time `units` attribute (with or without `since`), any unit for measuring time can be used i.e. any unit which is physically equivalent to the SI base unit of time, namely the second.

description

The description is meant to clarify the qualifiers of the fundamental quantities such as which surface a quantity is defined on or what the flux sign conventions are. We don't attempt to provide precise definitions of fundumental physical quantities (e.g., temperature) which may be found in the literature. The description may define rules on the variable type, attributes and coordinates which must be complied with by any variable carrying that standard name (such as in Example 3.5).

The standard name table is located at https://cfconventions.org/Data/cf-standard-names/current/src/cf-standard-name-table.xml, written in compliance with the XML format, as described in Appendix B, Standard Name Table Format. Knowledge of the XML format is only necessary for application writers who plan to directly access the table. A formatted text version of the table is provided at https://cfconventions.org/Data/cf-standard-names/current/build/cf-standard-name-table.html, and this table may be consulted in order to find the standard name that should be assigned to a

variable. Some standard names (e.g. `region`, Section 6.1.1, "Geographic Regions", and `area_type`, Statistics applying to portions of cells) are used to indicate quantities which are permitted to take only certain standard values. This is indicated in the definition of the quantity in the standard name table, accompanied by a list or a link to a list of the permitted values.

Standard names by themselves are not always sufficient to describe a quantity. For example, a variable may contain data to which spatial or temporal operations have been applied. Or the data may represent an uncertainty in the measurement of a quantity. These quantity attributes are expressed as modifiers of the standard name. Modifications due to common statistical operations are expressed via the `cell_methods` attribute (see Section 7.3, "Cell Methods" and Appendix E, Cell Methods). Other types of quantity modifiers are expressed using the optional modifier part of the `standard_name` attribute. The permissible values of these modifiers are given in Appendix C, Standard Name Modifiers.

Example 3.2. Use of `standard_name`

```
float psl(lat,lon) ;
  psl:long_name = "mean sea level pressure" ;
  psl:units = "hPa" ;
  psl:standard_name = "air_pressure_at_sea_level" ;
```

The description in the standard name table entry for `air_pressure_at_sea_level` clarifies that "sea level" refers to the mean sea level, which is close to the geoid in sea areas.

3.4. Ancillary Data

When one data variable provides metadata about the individual values of another data variable it may be desirable to express this association by providing a link between the variables. For example, instrument data may have associated measures of uncertainty. The attribute `ancillary_variables` is used to express these types of relationships. It is a string attribute whose value is a blank separated list of variable names. The nature of the relationship between variables associated via `ancillary_variables` must be determined by other attributes. The variables listed by the `ancillary_variables` attribute will often have the standard name of the variable which points to them including a modifier (Appendix C, Standard Name Modifiers) to indicate the relationship. The dimensions of an ancillary variable must be the same as or a subset of the dimensions of the variable to which it is related, but their order is not restricted, and with one exception: If an ancillary variable of a data variable that has been compressed by gathering (Section 8.2, "Lossless Compression by Gathering") does not span the compressed dimension, then its dimensions may be any subset of the data variable's uncompressed dimensions, i.e. any of the dimensions of the data variable except the compressed dimension, and any of the dimensions listed by the `compress` attribute of the compressed coordinate variable.

Example 3.3. Ancillary instrument data

```
  float q(time) ;
    q:standard_name = "specific_humidity" ;
```

```
    q:units = "g/g" ;
    q:ancillary_variables = "q_error_limit q_detection_limit" ;
  float q_error_limit(time)
    q_error_limit:standard_name = "specific_humidity standard_error" ;
    q_error_limit:units = "g/g" ;
  float q_detection_limit(time)
    q_detection_limit:standard_name = "specific_humidity detection_minimum" ;
    q_detection_limit:units = "g/g" ;
```

Alternatively, `ancillary_variables` may be used as status flags indicating the operational status of an instrument producing the data or as quality flags indicating the results of a quality control test, or some other quantitative quality assessment, performed against the measurements contained in the source variable. In these cases, the flag variable will include a standard name that differs from that of the source variable and indicates the specific type of flag the variable represents.

The standard names table includes many names intended to be used in this situation, both general names meant to be used to flexibly represent any type of status or quality assessment, as well as names for specific quality control tests commonly applied to geophysical phenomena timeseries data. Several examples are listed below:

Sample flag variable standard names:

- `status_flag` and `quality_flag`: general flag categories for instrument status or quality assessment
- `climatology_test_quality_flag`, `flat_line_test_quality_flag`, `gap_test_quality_flag`, `spike_test_quality_flag`: a subset of standard name flags used to indicate the results of commonly-used geophysical timeseries data quality control tests (consult the standard names table for a full list of published flags)
- `aggregate_quality_flag`: flag indicating an aggregate summary of all quality tests performed on the data variable, both automated and manual (i.e. a master quality flag for a particular variable)

The following example illustrates the use of three of these flags to represent two independent quality control tests and an aggregate flag that combines the results of the two tests.

Example 3.4. Ancillary quality flag data

```
float salinity(time, z);
        salinity:units = "1";
        salinity:long_name = "Salinity";
        salinity:standard_name = "sea_water_practical_salinity";
        salinity:ancillary_variables = "salinity_qc_generic
salinity_qc_flat_line_test salinity_qc_agg";

    int salinity_qc_generic(time, z);
        salinity_qc_generic:long_name = "Salinity Generic QC Process Flag";
        salinity_qc_generic:standard_name = "quality_flag";
```

```
    int salinity_qc_flat_line_test(time, z);
        salinity_qc_flat_line_test:long_name = "Salinity Flat Line Test Flag";
        salinity_qc_flat_line_test:standard_name = "flat_line_test_quality_flag";

    int salinity_qc_agg(time, z);
        salinity_qc_agg:long_name = "Salinity Aggregate Flag";
        salinity_qc_agg:standard_name = "aggregate_quality_flag";
```

Note that the ancillary variables in this example are simplified to exclude `flag_values`, `flag_masks` and `flag_meanings` attributes described in Section 3.5, "Flags" that they would ordinarily require

3.5. Flags

The attributes `flag_values`, `flag_masks` and `flag_meanings` are intended to make variables that contain flag values self describing. Status codes and Boolean (binary) condition flags may be expressed with different combinations of `flag_values` and `flag_masks` attribute definitions.

The `flag_values` and `flag_meanings` attributes describe a status flag consisting of mutually exclusive coded values. The `flag_values` attribute is the same type as the variable to which it is attached, and contains a list of the possible flag values. The `flag_meanings` attribute is a string whose value is a blank separated list of descriptive words or phrases, one for each flag value. Each word or phrase should consist of characters from the alphanumeric set and the following five: '_', '-', '.', '+', '@'. If multi-word phrases are used to describe the flag values, then the words within a phrase should be connected with underscores. The following example illustrates the use of flag values to express a speed quality with an enumerated status code.

Example 3.5. A flag variable, using `flag_values`

```
byte current_speed_qc(time, depth, lat, lon) ;
  current_speed_qc:long_name = "Current Speed Quality" ;
  current_speed_qc:standard_name = "status_flag" ;
  current_speed_qc:_FillValue = -128b ;
  current_speed_qc:valid_range = 0b, 2b ;
  current_speed_qc:flag_values = 0b, 1b, 2b ;
  current_speed_qc:flag_meanings = "quality_good sensor_nonfunctional
                                    outside_valid_range" ;
```

Note that the data variable containing current speed has an ancillary_variables attribute with a value containing current_speed_qc.

The flag_masks and flag_meanings attributes describe a number of independent Boolean conditions using bit field notation by setting unique bits in each flag_masks value. The flag_masks attribute is the same type as the variable to which it is attached, and contains a list of values matching unique bit fields. The flag_meanings attribute is defined as above, one for each flag_masks value. A flagged condition is identified by performing a bitwise AND of the variable value and each flag_masks

value; a non-zero result indicates a true condition. Thus, any or all of the flagged conditions may be true, depending on the variable bit settings. The following example illustrates the use of flag_masks to express six sensor status conditions.

Example 3.6. A flag variable, using `flag_masks`

```
byte sensor_status_qc(time, depth, lat, lon) ;
  sensor_status_qc:long_name = "Sensor Status" ;
  sensor_status_qc:standard_name = "status_flag" ;
  sensor_status_qc:_FillValue = 0b ;
  sensor_status_qc:valid_range = 1b, 63b ;
  sensor_status_qc:flag_masks = 1b, 2b, 4b, 8b, 16b, 32b ;
  sensor_status_qc:flag_meanings = "low_battery processor_fault
                                    memory_fault disk_fault
                                    software_fault
                                    maintenance_required" ;
```

A variable with standard name of `region`, `area_type` or any other standard name which requires string-valued values from a defined list may use flags together with `flag_values` and `flag_meanings` attributes to record the translation to the string values. The following example illustrates this using integer flag values for a variable with standard name `region` and `flag_values` selected from the standardized region names (see section 6.1.1).

Example 3.7. A region variable, using `flag_values`

```
int basin(lat, lon);
       standard_name: region;
       flag_values: 1, 2, 3;
       flag_meanings:"atlantic_arctic_ocean indo_pacific_ocean global_ocean";
data:
   basin: 1, 1, 1, 1, 2, ..... ;
```

The `flag_masks`, `flag_values` and `flag_meanings` attributes, used together, describe a blend of independent Boolean conditions and enumerated status codes. The `flag_masks` and `flag_values` attributes are both the same type as the variable to which they are attached. A flagged condition is identified by a bitwise AND of the variable value and each `flag_masks` value; a result that matches the `flag_values` value indicates a `true` condition. Repeated `flag_masks` define a bit field mask that identifies a number of status conditions with different `flag_values`. The `flag_meanings` attribute is defined as above, one for each `flag_masks` bit field and `flag_values` definition. Each `flag_values` and `flag_masks` value must coincide with a `flag_meanings` value. The following example illustrates the use of `flag_masks` and `flag_values` to express two sensor status conditions and one enumerated status code.

Example 3.8. A flag variable, using `flag_masks` and `flag_values`

```
byte sensor_status_qc(time, depth, lat, lon) ;
  sensor_status_qc:long_name = "Sensor Status" ;
  sensor_status_qc:standard_name = "status_flag" ;
  sensor_status_qc:_FillValue = 0b ;
  sensor_status_qc:valid_range = 1b, 15b ;
  sensor_status_qc:flag_masks = 1b, 2b, 12b, 12b, 12b ;
  sensor_status_qc:flag_values = 1b, 2b, 4b, 8b, 12b ;
  sensor_status_qc:flag_meanings =
       "low_battery
        hardware_fault
        offline_mode calibration_mode maintenance_mode" ;
```

In this case, mutually exclusive values are blended with Boolean values to maximize use of the available bits in a flag value. The table below represents the four binary digits (bits) expressed by the `sensor_status_qc` variable in the previous example.

Bit 0 and Bit 1 are Boolean values indicating a low battery condition and a hardware fault, respectively. The next two bits (Bit 2 and Bit 3) express an enumeration indicating abnormal sensor operating modes. Thus, if Bit 0 is set, the battery is low and if Bit 1 is set, there is a hardware fault - independent of the current sensor operating mode.

Table 3.2. Flag Variable Bits (from Example)

Bit 3 (MSB)	**Bit 2**	**Bit 1**	**Bit 0 (LSB)**
		H/W Fault	Low Batt

The remaining bits (Bit 2 and Bit 3) are decoded as follows:

Table 3.3. Flag Variable Bit 2 and Bit 3 (from Example)

Bit 3	**Bit 2**	**Mode**
0	1	offline_mode
1	0	calibration_mode
1	1	maintenance_mode

The "12b" flag mask is repeated in the `sensor_status_qc` `flag_masks` definition to explicitly declare the recommended bit field masks to repeatedly AND with the variable value while searching for matching enumerated values. An application determines if any of the conditions declared in the `flag_meanings` list are `true` by simply iterating through each of the `flag_masks` and AND'ing them with the variable. When a result is equal to the corresponding `flag_values` element, that condition is `true`. The repeated `flag_masks` enable a simple mechanism for clients to detect all possible conditions.

Chapter 4. Coordinate Types

The commonest use of coordinate variables is to locate the data in space and time, but coordinates may be provided for any other continuous geophysical quantity (e.g. density, temperature, radiation wavelength, zenith angle of radiance, sea surface wave frequency) or discrete category (see Section 4.5, "Discrete Axis", e.g. area type, model level number, ensemble member number) on which the data variable depends.

Four types of coordinates receive special treatment by these conventions: latitude, longitude, vertical, and time. We continue to support the special role that the `units` and `positive` attributes play in the COARDS convention to identify coordinate type. As an extension to COARDS, we strongly recommend that a parametric (usually dimensionless) vertical coordinate variable should be associated, via `standard_name` and `formula_terms` attributes, with its explicit definition, which provides a mapping between its values and dimensional vertical coordinate values that can be uniquely located with respect to a point on the earth's surface.

Because identification of a coordinate type by its units is complicated by requiring the use of an external package [UDUNITS], we provide two optional methods that yield a direct identification. The attribute `axis` may be attached to a coordinate variable and given one of the values `X`, `Y`, `Z` or `T` which stand for a longitude, latitude, vertical, or time axis respectively. Alternatively the `standard_name` attribute may be used for direct identification. But note that these optional attributes are in addition to the required COARDS metadata.

To identify generic spatial coordinates we recommend that the `axis` attribute be attached to these coordinates and given one of the values `X`, `Y` or `Z`. The values `X` and `Y` for the axis attribute should be used to identify horizontal coordinate variables. If both X- and Y-axis are identified, `X-Y-up` should define a right-handed coordinate system, i.e. rotation from the positive X direction to the positive Y direction is anticlockwise if viewed from above. We strongly recommend that coordinate variables be used for all coordinate types whenever they are applicable.

The methods of identifying coordinate types described in this section apply both to coordinate variables and to auxiliary coordinate variables named by the `coordinates` attribute (see Chapter 5, Coordinate Systems and Domain).

The values of a coordinate variable or auxiliary coordinate variable indicate the locations of the gridpoints. The locations of the boundaries between cells are indicated by bounds variables (see Section 7.1, "Cell Boundaries").

4.1. Latitude Coordinate

Variables representing latitude must always explicitly include the `units` attribute; there is no default value. The recommended value of the `units` attribute is the string `degrees_north`. Also accepted are `degree_north`, `degree_N`, `degrees_N`, `degreeN`, and `degreesN`.

Example 4.1. Latitude axis

```
float lat(lat) ;
  lat:long_name = "latitude" ;
```

```
  lat:units = "degrees_north" ;
  lat:standard_name = "latitude" ;
```

Application writers should note that the UDUNITS package does not recognize the directionality implied by the "north" part of the unit specification. It only recognizes its size, i.e., 1 degree is defined to be pi/180 radians. Hence, determination that a coordinate is a latitude type should be done via a string match between the given unit and one of the acceptable forms of `degrees_north`.

Optionally, the latitude type may be indicated additionally by providing the `standard_name` attribute with the value `latitude`, and/or the `axis` attribute with the value `Y`.

Coordinates of latitude with respect to a rotated pole should be given units of `degrees`, not `degrees_north` or equivalents, because applications which use the units to identify axes would have no means of distinguishing such an axis from real latitude, and might draw incorrect coastlines, for instance.

4.2. Longitude Coordinate

Variables representing longitude must always explicitly include the `units` attribute; there is no default value. The recommended value of the `units` attribute is the string `degrees_east`. Also accepted are `degree_east`, `degree_E`, `degrees_E`, `degreeE`, and `degreesE`.

Example 4.2. Longitude axis

```
float lon(lon) ;
  lon:long_name = "longitude" ;
  lon:units = "degrees_east" ;
  lon:standard_name = "longitude" ;
```

Application writers should note that the UDUNITS package has limited recognition of the directionality implied by the "east" part of the unit specification. It defines `degrees_east` to be pi/180 radians, and hence equivalent to `degrees_north`. We recommend the determination that a coordinate is a longitude type should be done via a string match between the given unit and one of the acceptable forms of `degrees_east`.

Optionally, the longitude type may be indicated additionally by providing the `standard_name` attribute with the value `longitude`, and/or the `axis` attribute with the value `X`.

Coordinates of longitude with respect to a rotated pole should be given units of `degrees`, not `degrees_east` or equivalents, because applications which use the units to identify axes would have no means of distinguishing such an axis from real longitude, and might draw incorrect coastlines, for instance.

4.3. Vertical (Height or Depth) Coordinate

Variables representing dimensional height or depth axes must always explicitly include the `units` attribute; there is no default value.

The direction of positive (i.e., the direction in which the coordinate values are increasing), whether up or down, cannot in all cases be inferred from the units. The direction of positive is useful for applications displaying the data. For this reason the attribute `positive` as defined in the COARDS standard is required if the vertical axis units are not a valid unit of pressure (as determined by the UDUNITS package [UDUNITS]) — otherwise its inclusion is optional. The `positive` attribute may have the value `up` or `down` (case insensitive). This attribute may be applied to either coordinate variables or auxiliary coordinate variables that contain vertical coordinate data.

For example, if an oceanographic netCDF file encodes the depth of the surface as 0 and the depth of 1000 meters as 1000 then the axis would use attributes as follows:

```
axis_name:units = "meters" ;
axis_name:positive = "down" ;
```

If, on the other hand, the depth of 1000 meters were represented as -1000 then the value of the `positive` attribute would have been `up`. If the `units` attribute value is a valid pressure unit the default value of the `positive` attribute is `down`.

A vertical coordinate will be identifiable by:

- units of pressure; or
- the presence of the `positive` attribute with a value of `up` or `down` (case insensitive).

Optionally, the vertical type may be indicated additionally by providing the `standard_name` attribute with an appropriate value, and/or the `axis` attribute with the value `Z`. If both `positive` and `standard_name` are provided, it is recommended that they should be consistent. For instance, if a depth of 1000 metres is represented by -1000 and `positive` is `up`, it would be inconsistent to give the `standard_name` as `depth`, whose definition (vertical distance below the surface) implies positive down. If an application detects such an inconsistency, the user should be warned, and the `positive` attribute should be used to determine the sign convention.

Recommendations: The `positive` attribute should be consistent with the sign convention implied by the definition of the `standard_name`, if both are provided.

4.3.1. Dimensional Vertical Coordinate

Variables representing dimensional vertical coordinates for depth or height must always explicitly include the `units` attribute. The acceptable units for a vertical (depth or height) coordinate variable must a UDUNITS [UDUNITS] representation of one of the following:

- units of pressure. For vertical axes the most commonly used of these include `bar`, `millibar`, `decibar`, `atmosphere (atm)`, `pascal (Pa)`, and `hPa`.
- units of length. For vertical axes the most commonly used of these include `meter (metre, m)`, and

`kilometer (km)`.

- other units that may under certain circumstances reference vertical position such as units of density or temperature.

Plural forms are also acceptable.

4.3.2. Dimensionless Vertical Coordinate

The `units` attribute is not required for dimensionless coordinates. For backwards compatibility with COARDS we continue to allow the `units` attribute to take one of the values: `level`, `layer`, or `sigma_level`. These values are not recognized by the UDUNITS package, and are considered a deprecated feature in the CF standard.

4.3.3. Parametric Vertical Coordinate

In some cases dimensional vertical coordinates are a function of horizontal location as well as parameters which depend on vertical location, and therefore cannot be stored in the one-dimensional vertical coordinate variable, which is in most of these cases is dimensionless. The `standard_name` of the parametric (usually dimensionless) vertical coordinate variable can be used to find the definition of the associated computed (always dimensional) vertical coordinate in Appendix D, Parametric Vertical Coordinates. The definition provides a mapping between the parametric vertical coordinate values and computed values that can positively and uniquely indicate the location of the data. The `formula_terms` attribute can be used to associate terms in the definitions with variables in a netCDF file, and the `computed_standard_name` attribute can be used to supply the `standard_name` of the computed vertical coordinate values computed according to the definition. To maintain backwards compatibility with COARDS the use of these attributes is not required, but is strongly recommended. Some of the definitions may be supplemented with information stored in the `grid_mapping` variable about the datum used as a vertical reference (e.g. geoid, other geopotential datum or reference ellipsoid; see Section 5.6, "Horizontal Coordinate Reference Systems, Grid Mappings, and Projections" and Appendix F, Grid Mappings).

Example 4.3. Atmosphere sigma coordinate

```
float lev(lev) ;
  lev:long_name = "sigma at layer midpoints" ;
  lev:positive = "down" ;
  lev:standard_name = "atmosphere_sigma_coordinate" ;
  lev:formula_terms = "sigma: lev ps: PS ptop: PTOP" ;
  lev:computed_standard_name = "air_pressure" ;
```

In this example the `standard_name` value `atmosphere_sigma_coordinate` identifies the following definition from Appendix D, Parametric Vertical Coordinates which specifies how to compute pressure at gridpoint `(n,k,j,i)` where `j` and `i` are horizontal indices, `k` is a vertical index, and `n` is a time index:

```
p(n,k,j,i) = ptop + sigma(k)*(ps(n,j,i)-ptop)
```

The `formula_terms` attribute associates the variable `lev` with the term `sigma`, the variable `PS` with the term `ps`, and the variable `PTOP` with the term `ptop`. Thus the pressure at gridpoint `(n,k,j,i)` would be calculated by

```
p(n,k,j,i) = PTOP + lev(k)*(PS(n,j,i)-PTOP)
```

The `computed_standard_name` attribute indicates that the values in variable `p` would have a `standard_name` of `air_pressure`.

4.4. Time Coordinate

A time coordinate is a number which identifies an instant along the continuous physical dimension of time, whether in reality or in a model. The instant can equivalently be identified by its datetime, which is a set of numbers comprising year, month, day, hour, minute and second, where the second may have a fraction but the others are all integer. The time coordinate and the datetime are interconvertible given the `calendar` attribute of the time coordinate variable (Section 4.4.2, "Calendar") and its `units` attribute (containing the time unit of the coordinate values and the reference datetime, Section 4.4.1, "Time Coordinate Units").

Variables containing time coordinates must always explicitly include the `units` attribute, formatted as described in Section 4.4.1, "Time Coordinate Units". There is no default value for the `units`. A coordinate variable is identifiable as a time coordinate variable from its `units` alone. Optionally, a time coordinate variable may be indicated additionally by providing the `standard_name` attribute with an appropriate value, and/or the `axis` attribute with the value `T`.

Example 4.4. Example of a time coordinate variable

```
double time(time) ;
  time:axis = "T"; // optional
  time:standard_name = "time" ; // optional
  time:units = "days since 1990-1-1 0:0:0" ; // mandatory
```

4.4.1. Time Coordinate Units

The `units` attribute of a time coordinate variable takes a string value that follows the formatting requirements of the [UDUNITS] package (e.g. Example of a time coordinate variable). It must comprise a unit of measure that is physically equivalent to the SI base unit of time (i.e. the second), followed by the word `since` and a reference datetime. The format of the `units` string implies that the time coordinate equals the length of the time interval from the instant identified by the reference datetime to the instant identified by the time coordinate. This is exactly true in all cases except when leap seconds occur between the two intervals in the `standard`, `proleptic_gregorian`, and `julian` calendars. See Section 4.4.3, "Leap Seconds".

The acceptable units of measure for time are given by UDUNITS. The most commonly used of these strings (and their abbreviations) are `day` (`d`), `hour` (`hr`, `h`), `minute` (`min`) and `second` (`sec`, `s`). Plural forms are also acceptable.

UDUNITS defines a `year` to be exactly 365.242198781 days (the interval between 2 successive passages of the sun through vernal equinox). *It is not a calendar year*. UDUNITS defines a `month` to be exactly `year/12`, which is *not a calendar month*. The CF standard follows UDUNITS in the definition of units, but we recommend that `year` and `month` should not be used, because of the potential for mistakes and confusion.

UDUNITS defines a `minute` as 60 `seconds`, an `hour` as 3600 `seconds` and a `day` as 86400 `seconds`. These are not calendar units. When a leap second is inserted into UTC, the minute, hour and day affected differ by one second from their usual durations according to clock time, but the UDUNITS and CF `minute`, `hour` and `day` do not; they are fixed units of measure. See also Section 4.4.3, "Leap Seconds".

UDUNITS permits a number of alternatives to the word `since` in the units of time coordinates. All the alternatives have exactly the same meaning in UDUNITS. For compatibility with other software, CF strongly recommends that `since` should be used.

The reference datetime string (appearing after the identifier `since`) is required. It must include the date, which may optionally be followed by time or time zone offset or both. Its format is *y-m-d* [*H*:*M*:*S*] [*Z*], where [...] indicates an optional element:

- *y* is year, *m* month, *d* day, *H* hour and *M* minute, which are all integers of one or more digits, and *y* may be prefixed with a sign (but note that some CF calendars do not permit negative years; see Section 4.4.2, "Calendar"),
- *S* is second, which may be integer or floating point (see Section 4.4.3, "Leap Seconds" regarding *S*>59),
- *Z* is the time zone offset. This is an interval of time, specified in one of the formats described below. Only numbers (digits, `+`, `-` and `:`) or the letter "Z" are allowed in *Z*, not time zone names or acronyms.

The default time zone offset is zero. In a time zone with zero offset, time (approximately) equals mean solar time for 0 `degrees_east` of longitude. (Although this may be exact in a model, in reality the time with zero time zone offset differs by some seconds from mean solar time; see the discussion of UTC and leap seconds in Section 4.4.2, "Calendar".) If both time and time zone offset are omitted the time is 00:00:00 (the beginning of the day i.e. midnight at 0 `degrees_east`). Thus, `units = "days since 1990-1-1"` means the same as `units = "days since 1990-1-1 00:00:00"`.

The time zone offset *Z* must be in one of the following five formats, where numeric hours may optionally be prefixed with a `+` or `-` sign:

- The letter `Z` indicating zero offset, sometimes referred to as "Zulu Time".
- *H*, the hour alone, of one or two digits e.g. `-6`, `2`, `+11`, which is sufficient for many time zones.
- *H*:*M*, where *H* is hour and *M* minute, each of one or two digits, e.g. `5:30`.
- four digits, of which the first pair are the hours and the second the minutes e.g. `0530`.
- three digits, of which the first is the hour (0—9) e.g. `530`.

If the time zone offset is the letter `Z` or begins with a sign, the space before it may be omitted.

While the default (of omitting the *Z* component) is an offset of zero, we suggest that a zero offset be specified to avoid any confusion where omitting it might be misunderstood as indicating local time.

For example, `seconds since 1992-10-8 15:15:42.5 -6:00` indicates seconds since October 8th, 1992 at 3 hours, 15 minutes and 42.5 seconds in the afternoon, in a time zone where the datetime is six hours behind the default. Subtracting the time zone offset from a given datetime converts it to the equivalent datetime with zero time zone offset e.g. `1989-12-31 18:00:00 -6` identifies the same instant as `1990-1-1 0:0:0`.

4.4.2. Calendar

The calendar defines the set of valid datetimes and their order. Note that the CF meaning of "calendar" refers to datetimes, not to dates alone. Datetimes which are not permitted in a given calendar are prohibited both in the time coordinate values and in the reference datetime string in the `units`. It is recommended that the calendar be specified by the `calendar` attribute of the time coordinate variable. The values currently defined for `calendar` are listed below.

Because the calendars have different sets of valid dates, and different treatments of leap seconds (see below in this section, and Section 4.4.3, "Leap Seconds"), a given time coordinate value with given `units` can represent different datetimes in different calendars; conversely, a given datetime is represented by different time coordinate values in different calendars. Moreover, in different calendars a given datetime can identify a different instant in the continuous physical dimension of time.

The lengths of the months in the Gregorian calendar are used in all calendars except `360_day`, `none` (see Section 4.4.4, "Time Coordinates with no Annual Cycle") and explicitly defined calendars (see Section 4.4.5, "Explicitly Defined Calendar"). The calendars differ in their treatment of leap years (when there are 29 days in February instead of 28).

Leap seconds are adjustments made at irregular and unpredictable intervals in Coordinated Universal Time (UTC). In response to slight variations in the Earth's rotation speed, positive or negative leap seconds are inserted in order to keep UTC close to mean solar time at 0 `degrees_east` i.e. the time zone with the default (zero) time zone offset in UDUNITS and CF (see Section 4.4.1, "Time Coordinate Units"). When a single positive leap second is introduced at the end of a minute, that minute contains 61 seconds. The net number of leap seconds added to UTC between 1958-1-1 and 2025-1-1 is 37. The CF calendars differ in their treatment of leap seconds (see Section 4.4.3, "Leap Seconds").

In the `julian` and the default `standard` calendar, dates in years before year 0 (i.e. before 0-1-1 0:0:0) are not allowed, and the year in the reference datetime of the units must not be negative. In these calendars, year zero has a special use to indicate a climatology (see Section 7.4, "Climatological Statistics"), but this use of year zero is deprecated. In other calendars, year 0 is the year before year 1, and negative years are allowed.

`standard`

Mixed Gregorian/Julian calendar as defined by UDUNITS. This is the default. A deprecated alternative name for this calendar is `gregorian`. The Gregorian and Julian calendars have the

same lengths of their months; they differ only in respect of the rules that decide which years are leap years. In the `standard` calendar, datetimes after and including 1582-10-15 0:0:0 are in the Gregorian calendar, in which a year is a leap year if either (i) it is divisible by 4 but not by 100 or (ii) it is divisible by 400. Datetimes before (and excluding) 1582-10-5 0:0:0 are in the Julian calendar, in which any year that is divisible by 4 is a leap year. Year 1 AD or CE in the `standard` calendar is also year 1 of the `julian` calendar. Negative years are invalid in time coordinates and reference datetimes in the `standard` calendar. In the `standard` calendar, 1582-10-15 0:0:0 is exactly 1 day later than 1582-10-4 0:0:0. Therefore datetimes in the range from (and including) 1582-10-5 0:0:0 until (but excluding) 1582-10-15 0:0:0 are invalid, and must not be used as reference in `units`. It is recommended that a reference datetime before the discontinuity should not be used for datetimes after the discontinuity, and vice-versa. See also Section 4.4.3, "Leap Seconds".

`proleptic_gregorian`

A calendar with the Gregorian rules for leap years extended to dates before 1582-10-15. All dates consistent with these rules are allowed, both before and after 1582-10-15 0:0:0. See also Section 4.4.3, "Leap Seconds".

`julian`

Julian calendar, in which a year is a leap year if it is divisible by 4, even if it is also divisible by 100. Year 1 AD or CE in the `julian` calendar is also year 1 of the `standard` calendar. Negative years are invalid in time coordinates and reference datetimes in the `julian` calendar. See also Section 4.4.3, "Leap Seconds".

`utc`

A Gregorian calendar *with* leap seconds as prescribed by UTC. Datetimes before 1958-01-01 0:0:0 are not allowed in this calendar. Datetimes in the future are not allowed in this calendar, because it is unknown when future leap seconds will occur. When a datetime is converted to a time coordinate value or vice-versa in this calendar, any leap seconds (positive or negative) must be counted that occurred in the interval between the datetime and the reference datetime in the `units`. For any given instant, the `utc` datetime is behind the `tai` datetime, where "behind" means the same as it does when describing a timezone to the west as being behind one to the east. The difference between the two datetimes for a given instant of time is the net number of leap seconds introduced since 1958-01-01. The difference was zero on that instant, when both calendars began. This means that a given datetime in the `utc` calendar represents an instant that is earlier than the same datetime in the `tai` calendar. See also Section 4.4.3, "Leap Seconds".

`tai`

A Gregorian calendar *without* leap seconds that is based on International Atomic Time (TAI). Datetimes before 1958-01-01 0:0:0 are not allowed in this calendar. For any given instant, the `tai` datetime is ahead of the `utc` datetime, where "ahead" means the same as it does when describing a timezone to the east as being ahead of one to the west. The difference between the two datetimes for a given instant of time is the net number of leap seconds introduced since 1958-01-01. The difference was zero on that instant, when both calendars began. This means that a given datetime in the `tai` calendar represents an instant that is later than the same datetime in the `utc` calendar. See also Section 4.4.3, "Leap Seconds".

`noleap` **or** `365_day`

A calendar with no leap years, i.e., all years are 365 days long, and there are no leap seconds.

`all_leap` **or** `366_day`

A calendar in which every year is a leap year, i.e., all years are 366 days long, and there are no leap seconds.

`360_day`

A calendar in which all years are 360 days, and divided into 30 day months, and there are no leap seconds.

`none`

To be used when there is no annual cycle. See Section 4.4.4, "Time Coordinates with no Annual Cycle".

Any other value may be given to the `calendar` attribute to describe an explicitly defined calendar. See Section 4.4.5, "Explicitly Defined Calendar".

4.4.3. Leap Seconds

This section describes how to deal properly with leap seconds. Most people ignore the existence of leap seconds, including many data producers and the CF standard before version 1.12. As a result, the time coordinates of two real-world observational datasets could disagree by some number of seconds if one has taken leap seconds into account and the other has not. Practically speaking, this means that if you are working with real-world data, and if it's important for your time coordinates to be accurate to the second, you need to care about leap seconds. Otherwise, you need only to be aware that the difference between two time coordinates might not exactly equal the duration of the time interval between the two instants, but could be inaccurate by a number of seconds, if leap seconds are involved. Relatedly, two instants with the same time of day on different days, which would always be separated by a multiple of 86400 seconds if there were no leap seconds, will have a few more seconds between them if leap seconds intervene.

Each calendar defines a set of valid combinations of the six numbers year-month-day-hour-minute-second. We refer to this set as the calendar's "set of datetimes". Fractions of seconds are allowed in all calendars in addition to the integer number of seconds. In this section, we use the word *timeline* to mean "continuous physical dimension of time". The valid datetimes identify discrete instants along the timeline, in that sense.

You need to know the set of datetimes defined by the calendar in order to compute time coordinate values from datetimes and vice-versa. Ignoring fractional seconds in datetimes, a time coordinate value expressed in seconds equals the number of valid (integer-second) datetimes *after* (not including) the reference datetime in the `units` *up to* (and including) the datetime that the time coordinate represents. For instance, in `units` of `seconds since 2024-9-14 11:12:00`, the time coordinate for the datetime `2024-9-14 11:12:03` is `3`, because there are three datetimes (`2024-9-14 11:12:01`, `2024-9-14 11:12:02`, `2024-9-14 11:12:03`) following `2024-9-14 11:12:00` up to and including `2024-9-14 11:12:03`. The coordinate for `2024-9-14 11:11:58` is `-2`, because there are two valid datetimes (`2024-9-14 11:11:59`, `2024-9-14 11:11:58`) from `2024-9-14 11:12:00` to (and including) `2024-9-14 11:11:58`, and the count is negative because it goes backwards. The signed difference between the fractional seconds of the datetime and the reference is added to the time coordinate after counting the seconds. This paragraph may appear to be excessively elaborate in describing a usually obvious procedure, but it is necessary to be very careful about it when there are leap

seconds.

The `utc` calendar is the *only* calendar which includes leap seconds in its set of datetimes. In all other calendars, datetimes within leap seconds are not valid. Therefore reference datetimes in the `units` attribute must not contain seconds equal to or greater than 60 unless the `calendar` is `utc`.

The `standard`, `proleptic_gregorian`, and `julian` calendars each have two variants. In one variant the timeline does not include leap seconds. In the other variant, the timeline includes leap seconds, even though they are *not* included in the valid set of datetimes. To resolve the ambiguity between the variants of these calendars, the `units_metadata` attribute should be defined as well as the `calendar` attribute, as described later in this section.

For `standard`, `proleptic_gregorian`, and `julian` calendars, there are the following cases:

1. **The calendar is being used for a timeline in which leap seconds do not exist.** This is the case for a model simulation that defines every day as having a constant length of 86400 seconds.
2. **The calendar is being used for a timeline in which leap seconds exist, and they are correctly accounted for in the datetimes represented by the time coordinates.** This could be the case for observations from a platform with equipment which records UTC datetimes and has prior knowledge of when new leap seconds are to be introduced, so that it is able to apply a new leap second at the appropriate time. It could equally be the case for model whose timesteps include leap seconds.
3. **The calendar is being used for a timeline in which leap seconds exist, but some or all leap seconds might not have not been correctly accounted for in the datetimes.** This could be the case for observations from a platform whose time recording equipment has a delay in applying a new leap second.
4. **It may be unknown whether leap seconds exist in the timeline.**

Except in the `utc` calendar, when a time coordinate value is calculated from a datetime, or the reverse, it is assumed that the coordinate value increases by exactly 60 seconds from the start of any minute (identified by year, month, day, hour, minute, all being integers) to the start of the next minute, because leap seconds are not valid datetimes. In other words, leap seconds (positive or negative) are never counted in the `standard`, `proleptic_gregorian`, and `julian` calendars. When these calendars are being be used for timelines *with* leap seconds (i.e. cases 2 and 3 and perhaps case 4), the assumption of 60-second minutes has the following consequences:

- It is impossible to identify any instant during a leap second (i.e. between the end of the 60th second of the last minute of one hour and the start of the first second of the next hour) by a time coordinate e.g. `2016-12-31 23:59:60.5` cannot be represented by a time coordinate value.
- A datetime in the excluded range must not be used as a reference datetime e.g. `seconds since 2016-12-31 23:59:60` is not a permitted value for `units`.
- The coordinate value does not count any leap seconds which occurred between the reference datetime and the datetime represented by the coordinate. For instance, 60 `seconds after 23:59:00` always means 0:0:0 on the next day, even if there is a leap second at 23:59:60, which makes the actual interval 61 seconds between 23:59:00 and 0:0:0 on the next day.

Because of the last point, the difference between two coordinate values with the same `units` string

does not exactly equal the length of the interval between instants they represent if there were any leap seconds between them. This discrepancy can happen in cases 2, 3 and 4 of the `standard`, `proleptic_gregorian`, and `julian` calendars. By contrast, in case 1 of those calendars (i.e. a timeline without leap seconds), and in all other calendars, the difference between two time coordinate values with the same `units` string is always equal to the length of time between the instants they represent. Furthermore, an inaccuracy results from converting a time coordinate to a datetime if the interval includes leap seconds which were *not* known when the time coordinate was calculated (possible in case 3 or 4). It is important to be aware of these disadvantages of the `standard`, `proleptic_gregorian` and `julian` calendars when used with timelines including leap seconds.

If it is essential for leap seconds to be counted in time coordinates, so that they exactly equal time intervals, you must use the `utc` calendar. For many applications of the `standard`, `proleptic_gregorian`, and `julian` calendars, these inaccuracies are too small to matter, but there are some applications where it is necessary to know about them. Therefore it is recommended that for the `standard`, `proleptic_gregorian`, and `julian` calendars the appropriate treatment of leap seconds should be indicated by giving the time coordinate variable a `units_metadata` attribute containing a `leap_seconds` keyword with one of the permitted values `none`, `utc` or `unknown`. `none` means that leap seconds do not exist in the timeline (i.e. case 1), `utc` means that leap seconds exist in the timeline and the time coordinates correctly represent the datetimes (i.e. case 2), and `unknown` means that the data-writer did not know or did not record whether the leap seconds exist in the timeline, nor how they are treated if they did exist (i.e. cases 3 and 4). If the `units_metadata` attribute is not present, or does not contain the `leap_seconds` keyword, the data-reader should assume `leap_seconds: unknown`. A variable's `units_metadata` attribute may only contain the `leap_seconds` keyword if the variable's calendar is one of `standard` , `proleptic_gregorian`, or `julian`.

Example 4.5. Use of `units_metadata` and `calendar` to define the treatment of leap seconds

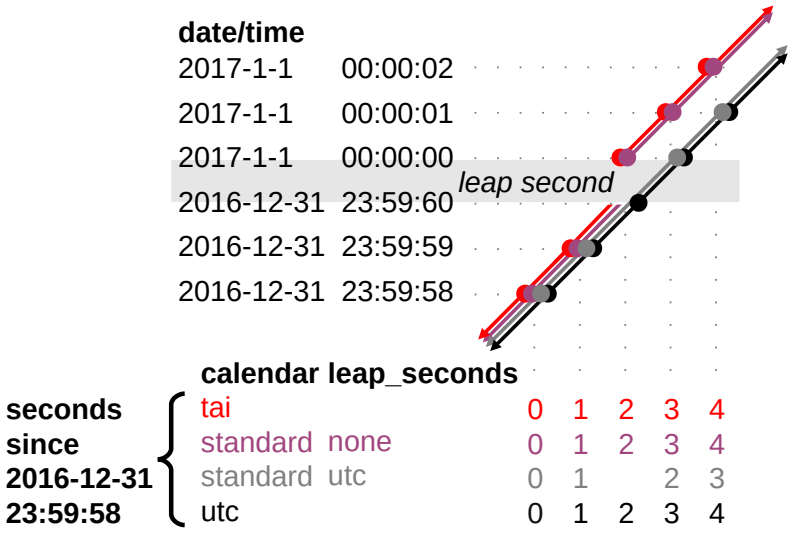
```
variables:
  float time_tai ;
    time_tai:standard_name = "time" ;
    time_tai:long_name = "Satellite data" ;
    time_tai:calendar = "tai" ;
    time_tai:units = "seconds since 2016-12-31 23:59:58" ;
  float time_stdnone ;
    time_stdnone:standard_name = "time" ;
    time_stdnone:long_name = "Model data with no leap seconds" ;
    time_stdnone:calendar = "standard" ;
    time_stdnone:units = "seconds since 2016-12-31 23:59:58" ;
    time_stdnone:units_metadata = "leap_seconds: none" ;
  float time_stdutc ;
    time_stdutc:standard_name = "time" ;
    time_stdutc:long_name = "Model data with leap seconds or obs data with
accurate UTC" ;
    time_stdutc:calendar = "standard" ;
    time_stdutc:units = "seconds since 2016-12-31 23:59:58" ;
    time_stdutc:units_metadata = "leap_seconds: utc" ;
  float time_utc ;
    time_utc:standard_name = "time" ;
    time_utc:long_name = "Time signal from UK National Physical Laboratory" ;
```

```
    time_utc:calendar = "utc" ;
    time_utc:units = "seconds since 2016-12-31 23:59:58" ;
  float time_unknown ;
    time_unknown:standard_name = "time" ;
    time_unknown:long_name = "Obs data with unreliable information on leap
seconds" ;
    time_unknown:calendar = "standard" ;
    time_unknown:units = "seconds since 2016-12-31 23:59:58" ;
    time_unknown:units_metadata = "leap_seconds: unknown" ;
data: // time coordinate variable and the datetime it represents
  time_tai = 2; // 2017-1-1 0:0:0 because no leap seconds in the timeline
  time_stdnone = 2; // 2017-1-1 0:0:0 because no leap seconds in the timeline
  time_stdutc = 2; // 2017-1-1 0:0:0 because the leap second is not counted
  time_utc = 2; // leap second 2016-12-31 23:59:60
  time_unknown = 2; // unknown whether 2016-12-31 23:59:60 or 2017-1-1 0:0:0
```

This example shows five scalar time coordinate variables. Although they all have the value 2 and the same `units` attribute, they do not all refer to the same datetime, as shown in the comments on their data values, because they have different treatments of the leap second that was added to the UTC calendar at the end of 2016. The first four of them correspond to the instants marked 2 `seconds since 2016-12-31 23:59:58` in Figure 4.1.

The value of `2` seconds for `time_stdnone`, `time_utc` and `time_tai` can be correctly interpreted as the length of the interval from the reference datetime 2016-12-31 23:59:58 to the datetime indicated in the comment. In both `time_stdnone` and `time_stdutc`, the time coordinate represents 2017-1-1 0:0:0, because 2016-12-31 23:59:60 is not permitted in the `standard` calendar, hence only two valid datetimes with integer seconds are counted (2016-12-31 23:59:59 and 2017-1-1 0:0:0). However, the *timeline* for `time_stdutc` *does* include the leap second, so the time interval from the reference datetime 2016-12-31 23:59:58 to 2017-1-1 0:0:0 is actually three seconds, not two as indicated by the time coordinate value. This is an example of the `standard` calendar not counting a leap second in the coordinate value, with the consequence that the difference between time coordinates does not exactly equal the duration of the interval. An application may choose either to ignore this inaccuracy or to correct for it when calculating the length of intervals which include the leap second. In the case of `time_unknown`, we cannot convert the time coordinate to a datetime with certainty, because we do not know whether 2017-1-1 0:0:0 is two or three seconds after 2016-12-31 23:59:58.

Figure 4.1. Illustration of the equivalence between datetimes and time coordinate values with `units="seconds since 2016-12-31 23:59:58"` *for various choices of the* `calendar` *attribute and* `leap_seconds` *keyword.*



This illustration shows that a given time coordinate value (the numbers in columns at the bottom right) can represent different datetimes in different calendars. However, the illustration cannot show another important point to keep in mind, that a given datetime may identify different instants in different calendars.

The diagonal lines depict the timelines of the calendars. Along each line, a filled circle marks the instant on the timeline that begins each second in the set of datetimes allowed by the calendar. There is no meaning in the slight left-right displacement of the circles at each second, which is done only so they can all be seen; they are supposed to be exactly coincident. As explained in the text of this section, the time coordinate in seconds is the count of valid datetimes (= the number of circles) that occur along the timeline *after* the reference datetime `2016-12-31 23:59:58` (which is the first circle on the line in every case, hence with a count of zero as shown in the column below its group of circles), up to and including the datetime represented. The instants marked 2 `seconds since 2016-12-31 23:59:58` are the ones represented by the first four time coordinate variables of Example 4.5.

A leap second was added to the UTC calendar at the end of 2016. The duration of the leap second is shown by the shading. The `utc` calendar is the only one in which datetimes in the leap second are valid; hence the black circle is the only marker of `2016-12-31 23:59:60`. The grey timeline of the `utc` variant of the `standard` calendar includes the the leap second as well, but datetimes in the leap second are not valid in that calendar, so there is no grey circle for it. The leap second does not appear in the timelines of the `tai` calendar and the `none` variant of the `standard` calendar. Their timelines (red and purple) skip over the leap second, and they have no circle for it. For those timelines, please imagine the digram having the shaded rectangle cut out, and the cut edges joined, making the red and purple lines continuous, passing smoothly from 2016-12-31 23:59:00 to 2017-1-1 00:00:00 as for all the other seconds.

4.4.4. Time Coordinates with no Annual Cycle

The `calendar` attribute may be set to `none` in climate experiments that simulate a fixed time of year. The time of year is indicated by the date in the reference time of the `units` attribute. The time coordinates that might apply in a perpetual July experiment are given in the following example.

Example 4.6. Perpetual time axis

```
variables:
  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1-7-15 0:0:0" ;
    time:calendar = "none" ;
data:
  time = 0., 1., 2., ...;
```

Here, all days simulate the conditions of 15th July, so it does not make sense to give them different dates. The time coordinates are interpreted as 0, 1, 2, etc. days since the start of the experiment.

4.4.5. Explicitly Defined Calendar

If none of the calendars defined in Section 4.4.2, "Calendar" applies (e.g., calendars appropriate to a different paleoclimate era), a calendar can be explicitly defined, in terms of permissible year-month-day combinations. To do this, the lengths of each month are explicitly defined with the `month_lengths` attribute of the time axis:

`month_lengths`

A vector of size 12, specifying the number of days in the months from January to December (in a non-leap year).

If leap years are included, then two other attributes of the time axis must also be defined:

`leap_year`

An example of a leap year. It is assumed that all years that differ from this year by a multiple of four are also leap years. If this attribute is absent, it is assumed there are no leap years.

`leap_month`

A value in the range 1-12, specifying which month is lengthened by a day in leap years (1=January). If this attribute is not present, February (2) is assumed. This attribute is ignored if `leap_year` is not specified.

When an explicitly defined calendar is being used, the calendar may be described by giving a value not defined in Section 4.4.2, "Calendar" to the `calendar` attribute; alternatively, the attribute may be omitted.

Example 4.7. Paleoclimate time axis

```
double time(time) ;
  time:long_name = "time" ;
  time:units = "days since 1-1-1 0:0:0" ;
  time:calendar = "126 kyr B.P." ;
  time:month_lengths = 34, 31, 32, 30, 29, 27, 28, 28, 28, 32, 32, 34 ;
```

4.5. Discrete Axis

The spatiotemporal coordinates described in sections 4.1-4.4 are continuous variables, and other geophysical quantities may likewise serve as continuous coordinate variables, for instance density, temperature or radiation wavelength. By contrast, for some purposes there is a need for an axis of a data variable which indicates either an ordered list or an unordered collection, and does not correspond to any continuous coordinate variable. Consequently such an axis may be called “discrete”. A discrete axis has a dimension but might not have a coordinate variable. Instead, there might be one or more auxiliary coordinate variables with this dimension (see preamble to section 5). Following sections define various applications of discrete axes, for instance section 6.1.1 “Geographical regions”, section 7.3.3 “Statistics applying to portions of cells”, section 9.3 “Representation of collections of features in data variables”.

Chapter 5. Coordinate Systems and Domain

A data variable's dimensions are used to locate data values in time and space or as a function of other independent variables. This is accomplished by associating these dimensions with the relevant set of latitude, longitude, vertical, time and any non-spatiotemporal coordinates. This section presents two methods for making that association: the use of *coordinate variables*, and the use of *auxiliary coordinate variables*.

Any of a variable's dimensions that is an independently varying latitude, longitude, vertical, or time dimension (see Section 1.3, "Terminology") and that has a size greater than one must have a corresponding coordinate variable, i.e., a one-dimensional variable with the same name as the dimension (see examples in Chapter 4, Coordinate Types). This is the only method of associating dimensions with coordinates that is supported by [COARDS].

Any longitude, latitude, vertical or time coordinate which depends on more than one spatiotemporal dimension must be identified by the `coordinates` attribute of the data variable. The value of the `coordinates` attribute is *a blank separated list of the names of auxiliary coordinate variables*. There is no restriction on the order in which the auxiliary coordinate variables appear in the `coordinates` attribute string. The dimensions of an auxiliary coordinate variable must be a subset of the dimensions of the variable with which the coordinate is associated, with three exceptions. First, string-valued coordinates (Section 6.1, "Labels") will have a dimension for maximum string length if the coordinate variable has a type of `char` rather than a type of `string`. Second, if an auxiliary coordinate variable of a data variable that has been compressed by gathering (Section 8.2, "Lossless Compression by Gathering") does not span the compressed dimension, then its dimensions may be any subset of the data variable's uncompressed dimensions, i.e. any of the dimensions of the data variable except the compressed dimension, and any of the dimensions listed by the `compress` attribute of the compressed coordinate variable. Third, in the ragged array representations of data (Chapter 9, Discrete Sampling Geometries), special methods are needed to connect the data and coordinates.

We recommend that the name of a multidimensional coordinate variable should not match the name of any of its dimensions because that precludes supplying a coordinate variable for the dimension. This practice also avoids potential bugs in applications that determine coordinate variables by only checking for a name match between a dimension and a variable and not checking that the variable is one dimensional.

If the longitude, latitude, vertical or time coordinate is multi-valued, varies in only one dimension, and varies independently of other spatiotemporal coordinates, it is not permitted to store it as an auxiliary coordinate variable. This is both to enhance conformance to COARDS and to facilitate the use of generic applications that recognize the [NUG] convention for coordinate variables. An application that is trying to find the latitude coordinate of a variable should always look first to see if any of the variable's dimensions correspond to a latitude coordinate variable. If the latitude coordinate is not found this way, then the auxiliary coordinate variables listed by the `coordinates` attribute should be checked. Note that it is permissible, but optional, to list coordinate variables as well as auxiliary coordinate variables in the `coordinates` attribute. If the longitude, latitude, vertical or time coordinate is single-valued, it may be stored either as a coordinate variable with a dimension of size one, or as a scalar coordinate variable (Section 5.7, "Scalar Coordinate Variables").

If an `axis` attribute is attached to an auxiliary coordinate variable, it can be used by applications in the same way the `axis` attribute attached to a coordinate variable is used. However, it is not permissible for a data variable to have both a coordinate variable and an auxiliary coordinate variable, or more than one of either type of variable, having an `axis` attribute with any given value e.g. there must be no more than one `axis` attribute for `X` for any data variable. Note that if the `axis` attribute is not specified for an auxiliary coordinate variable, it may still be possible to determine if it is a spatiotemporal dimension from its own units or `standard_name`, or from the units and `standard_name` of the coordinate variable corresponding to its dimensions (see Chapter 4, Coordinate Types). For instance, auxiliary coordinate variables which lie on the horizontal surface can be identified as such by their dimensions being horizontal. Horizontal dimensions are those whose coordinate variables have an `axis` attribute of `X` or `Y`, or a `units` attribute indicating latitude and longitude.

To geo-reference data horizontally with respect to the Earth, a *grid mapping variable* may be provided by the data variable, using the `grid_mapping` attribute. If the coordinate variables for a horizontal grid are not longitude and latitude, then a grid_mapping variable provides the information required to derive longitude and latitude values for each grid location. If no *grid mapping variable* is referenced by a data variable, then longitude and latitude coordinate values shall be supplied *in addition* to the required coordinates. For example, the Cartesian coordinates of a map projection may be supplied as coordinate variables and, in addition, two-dimensional latitude and longitude variables may be supplied via the `coordinates` attribute on a data variable. The use of the `axis` attribute with values `X` and `Y` is recommended for the coordinate variables (see Chapter 4, Coordinate Types).

It is sometimes not practical to specify the latitude-longitude location of data which is representative of geographic regions with complex boundaries. For this purpose, provision is made in Section 6.1.1, "Geographic Regions" for indicating the region by a standardized name.

5.1. Independent Latitude, Longitude, Vertical, and Time Axes

When each of a variable's spatiotemporal dimensions is a latitude, longitude, vertical, or time dimension, then each axis is identified by a coordinate variable.

Example 5.1. Independent coordinate variables

```
dimensions:
  lat = 18 ;
  lon = 36 ;
  pres = 15 ;
  time = 4 ;
variables:
  float xwind(time,pres,lat,lon) ;
    xwind:long_name = "zonal wind" ;
    xwind:units = "m/s" ;
  float lon(lon) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
```

```
  float lat(lat) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
  float pres(pres) ;
    pres:long_name = "pressure" ;
    pres:units = "hPa" ;
  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1990-1-1 0:0:0" ;
```

`xwind(n,k,j,i)` is associated with the coordinate values `lon(i)`, `lat(j)`, `pres(k)`, and `time(n)`.

5.2. Two-Dimensional Latitude, Longitude, Coordinate Variables

The latitude and longitude coordinates of a horizontal grid that was not defined as a Cartesian product of latitude and longitude axes, can sometimes be represented using two-dimensional coordinate variables. These variables are identified as coordinates by use of the `coordinates` attribute.

Example 5.2. Two-dimensional coordinate variables

```
dimensions:
  xc = 128 ;
  yc = 64 ;
  lev = 18 ;
variables:
  float T(lev,yc,xc) ;
    T:long_name = "temperature" ;
    T:units = "K" ;
    T:coordinates = "lon lat" ;
  float xc(xc) ;
    xc:axis = "X" ;
    xc:long_name = "x-coordinate in Cartesian system" ;
    xc:units = "m" ;
  float yc(yc) ;
    yc:axis = "Y" ;
    yc:long_name = "y-coordinate in Cartesian system" ;
    yc:units = "m" ;
  float lev(lev) ;
    lev:long_name = "pressure level" ;
    lev:units = "hPa" ;
  float lon(yc,xc) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
  float lat(yc,xc) ;
    lat:long_name = "latitude" ;
```

```
    lat:units = "degrees_north" ;
```

`T(k,j,i)` is associated with the coordinate values `lon(j,i)`, `lat(j,i)`, and `lev(k)`. The vertical coordinate is represented by the coordinate variable `lev(lev)` and the latitude and longitude coordinates are represented by the auxiliary coordinate variables `lat(yc,xc)` and `lon(yc,xc)` which are identified by the `coordinates` attribute.

Note that coordinate variables are also defined for the `xc` and `yc` dimensions. This faciliates processing of this data by generic applications that don't recognize the multidimensional latitude and longitude coordinates.

5.3. Reduced Horizontal Grid

A "reduced" longitude-latitude grid is one in which the points are arranged along constant latitude lines with the number of points on a latitude line decreasing toward the poles. Storing this type of gridded data in two-dimensional arrays wastes space, and results in the presence of missing values in the 2D coordinate variables. We recommend that this type of gridded data be stored using the compression scheme described in Section 8.2, "Lossless Compression by Gathering". Compression by gathering preserves structure by storing a set of indices that allows an application to easily scatter the compressed data back to two-dimensional arrays. The compressed latitude and longitude auxiliary coordinate variables are identified by the `coordinates` attribute.

Example 5.3. Reduced horizontal grid

```
dimensions:
  londim = 128 ;
  latdim = 64 ;
  rgrid = 6144 ;
variables:
  float PS(rgrid) ;
    PS:long_name = "surface pressure" ;
    PS:units = "Pa" ;
    PS:coordinates = "lon lat" ;
  float lon(rgrid) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
  float lat(rgrid) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
  int rgrid(rgrid);
    rgrid:compress = "latdim londim";
```

`PS(n)` is associated with the coordinate values `lon(n)`, `lat(n)`. Compressed grid index `(n)` would be assigned to 2D index `(j,i)` (C index conventions) where

```
j = rgrid(n) / 128
i = rgrid(n) - 128*j
```

Notice that even if an application does not recognize the `compress` attribute, the grids stored in this format can still be handled, by an application that recognizes the `coordinates` attribute.

5.4. Timeseries of Station Data

This section has been superseded by the treatment of time series as a type of discrete sampling geometry in Chapter 9.

5.5. Trajectories

This section has been superseded by the treatment of time series as a type of discrete sampling geometry in Chapter 9.

5.6. Horizontal Coordinate Reference Systems, Grid Mappings, and Projections

A *grid mapping variable* may be referenced by a data variable in order to explicitly declare the coordinate reference system (CRS) used for the horizontal spatial coordinate values. For example, if the horizontal spatial coordinates are latitude and longitude, the grid mapping variable can be used to declare the figure of the earth (WGS84 ellipsoid, sphere, etc.) they are based on. If the horizontal spatial coordinates are easting and northing in a map projection, the grid mapping variable declares the map projection CRS used and provides the information needed to calculate latitude and longitude from easting and northing.

When the horizontal spatial coordinate variables are not longitude and latitude, it is required that further information is provided to geo-locate the horizontal position. A *grid mapping variable* provides this information.

If no *grid mapping variable* is provided and the coordinate variables for a horizontal grid are not longitude and latitude, then it is required that the latitude and longitude coordinates are supplied via the `coordinates` attribute. Such coordinates may be provided in addition to the provision of a *grid mapping variable*, but that is not required.

When a data variable is representative of cells of non-zero size, and the coordinate variables are not longitude and latitude, *bounds* variables should be provided for vertices of the cell boundaries in the horizontal coordinates of the grid (see Section 7.1, "Cell Boundaries"). The *grid mapping variable* provides then the information to convert bounds in latitude and longitude coordinates, and it is optional for the dataset to provide these. If no *grid mapping variable* is provided, then the cell extents in latitude and longitude coordinate system should be provided (see Section 7.1, "Cell Boundaries" and especially Example 7.2 for the common case of four-sided cells).

A grid mapping variable provides the description of the mapping via a collection of attached attributes. It is of arbitrary type since it contains no data. Its purpose is to act as a container for the

attributes that define the mapping. The one attribute that all grid mapping variables must have is `grid_mapping_name`, which takes a string value that contains the mapping's name. The other attributes that define a specific mapping depend on the value of `grid_mapping_name`. The valid values of `grid_mapping_name` along with the attributes that provide specific map parameter values are described in Appendix F, Grid Mappings.

The grid mapping variables are associated with the data and coordinate variables by the `grid_mapping` attribute. This attribute is attached to data variables so that variables with different mappings may be present in a single file. The attribute takes a string value with two possible formats. In the first format, it is a single word, which names a grid mapping variable. In the second format, it is a blank-separated list of words `<gridMappingVariable>: <coordinatesVariable> [<coordinatesVariable> ⋯] [<gridMappingVariable>: <coordinatesVariable>⋯]`, which identifies one or more grid mapping variables, and with each grid mapping associates one or more coordinatesVariables, i.e. coordinate variables or auxiliary coordinate variables.

Where an extended `<gridMappingVariable>: <coordinatesVariable> [<coordinatesVariable>]` entity is defined, then the order of the `<coordinatesVariable>` references within the definition provides an explicit order for these coordinate value variables, which is used if they are to be combined into individual coordinate tuples.

This order is only significant if `crs_wkt` is also specified within the referenced grid mapping variable. Explicit 'axis order' is important when the *grid mapping variable* contains an attribute `crs_wkt` as it is mandated by the OGC CRS-WKT standard that coordinate tuples with correct axis order are provided as part of the reference to a Coordinate Reference System.

Using the simple form, where the `grid_mapping` attribute is only the name of a grid mapping variable, 2D latitude and longitude coordinates for a projected coordinate reference system use the same geographic coordinate reference system (ellipsoid and prime meridian) as the projection is projected from.

The `grid_mapping` variable may identify datums (such as the reference ellipsoid, the geoid or the prime meridian) for horizontal or vertical coordinates. Therefore a grid mapping variable may be needed when the coordinate variables for a horizontal grid are longitude and latitude. The `grid_mapping_name` of `latitude_longitude` should be used in this case.

The expanded form of the `grid_mapping` attribute is required if one wants to store coordinate information for more than one coordinate reference system. In this case each coordinate or auxiliary coordinate is defined explicitly with respect to no more than one `grid_mapping` variable. This syntax may be used to explicitly link coordinates and grid mapping variables where only one coordinate reference system is used. In this case, all coordinates and auxiliary coordinates of the data variable not named in the `grid_mapping` attribute are unrelated to any grid mapping variable. All coordinate names listed in the `grid_mapping` attribute must be coordinate variables or auxiliary coordinates of the data variable.

In order to make use of a grid mapping to directly calculate latitude and longitude values it is necessary to associate the coordinate variables with the independent variables of the mapping. This is done by assigning a `standard_name` to the coordinate variable. The appropriate values of the `standard_name` depend on the grid mapping and are given in Appendix F, Grid Mappings.

Example 5.6. Rotated pole grid

```
dimensions:
  rlon = 128 ;
  rlat = 64 ;
  lev = 18 ;
variables:
  float T(lev,rlat,rlon) ;
    T:long_name = "temperature" ;
    T:units = "K" ;
    T:coordinates = "lon lat" ;
    T:grid_mapping = "rotated_pole" ;
  char rotated_pole ;
    rotated_pole:grid_mapping_name = "rotated_latitude_longitude" ;
    rotated_pole:grid_north_pole_latitude = 32.5 ;
    rotated_pole:grid_north_pole_longitude = 170. ;
  float rlon(rlon) ;
    rlon:long_name = "longitude in rotated pole grid" ;
    rlon:units = "degrees" ;
    rlon:standard_name = "grid_longitude";
  float rlat(rlat) ;
    rlat:long_name = "latitude in rotated pole grid" ;
    rlat:units = "degrees" ;
    rlat:standard_name = "grid_latitude";
  float lev(lev) ;
    lev:long_name = "pressure level" ;
    lev:units = "hPa" ;
  float lon(rlat,rlon) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
  float lat(rlat,rlon) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
```

A CF compliant application can determine that rlon and rlat are longitude and latitude values in the rotated grid by recognizing the standard names `grid_longitude` and `grid_latitude`. Note that the units of the rotated longitude and latitude axes are given as `degrees`. This should prevent a COARDS compliant application from mistaking the variables `rlon` and `rlat` to be actual longitude and latitude coordinates. The entries for these names in the standard name table indicate the appropriate sign conventions for the units of `degrees`.

Example 5.7. Lambert conformal projection

```
dimensions:
  y = 228;
  x = 306;
  time = 41;
```

```
variables:
  int Lambert_Conformal;
    Lambert_Conformal:grid_mapping_name = "lambert_conformal_conic";
    Lambert_Conformal:standard_parallel = 25.0;
    Lambert_Conformal:longitude_of_central_meridian = 265.0;
    Lambert_Conformal:latitude_of_projection_origin = 25.0;
  double y(y);
    y:units = "km";
    y:long_name = "y coordinate of projection";
    y:standard_name = "projection_y_coordinate";
  double x(x);
    x:units = "km";
    x:long_name = "x coordinate of projection";
    x:standard_name = "projection_x_coordinate";
  double lat(y, x);
    lat:units = "degrees_north";
    lat:long_name = "latitude coordinate";
    lat:standard_name = "latitude";
  double lon(y, x);
    lon:units = "degrees_east";
    lon:long_name = "longitude coordinate";
    lon:standard_name = "longitude";
  int time(time);
    time:long_name = "forecast time";
    time:units = "hours since 2004-06-23T22:00:00Z";
  float Temperature(time, y, x);
    Temperature:units = "K";
    Temperature:long_name = "Temperature @ surface";
    Temperature:missing_value = 9999.0;
    Temperature:coordinates = "lat lon";
    Temperature:grid_mapping = "Lambert_Conformal";
```

An application can determine that `x` and `y` are the projection coordinates by recognizing the standard names `projection_x_coordinate` and `projection_y_coordinate`. The grid mapping variable `Lambert_Conformal` contains the mapping parameters as attributes, and is associated with the `Temperature` variable via its `grid_mapping` attribute.

Example 5.8. Latitude and longitude on a spherical Earth

```
dimensions:
  lat = 18 ;
  lon = 36 ;
variables:
  double lat(lat) ;
  double lon(lon) ;
  float temp(lat, lon) ;
    temp:long_name = "temperature" ;
    temp:units = "K" ;
    temp:grid_mapping = "crs" ;
```

```
  int crs ;
    crs:grid_mapping_name = "latitude_longitude"
    crs:semi_major_axis = 6371000.0 ;
    crs:inverse_flattening = 0 ;
```

Example 5.9. Latitude and longitude on the WGS 1984 datum

```
dimensions:
  lat = 18 ;
  lon = 36 ;
variables:
  double lat(lat) ;
  double lon(lon) ;
  float temp(lat, lon) ;
    temp:long_name = "temperature" ;
    temp:units = "K" ;
    temp:grid_mapping = "crs" ;
  int crs ;
    crs:grid_mapping_name = "latitude_longitude";
    crs:longitude_of_prime_meridian = 0.0 ;
    crs:semi_major_axis = 6378137.0 ;
    crs:inverse_flattening = 298.257223563 ;
```

Example 5.10. British National Grid

```
dimensions:
    z = 100;
    y = 100000 ;
    x = 100000 ;
  variables:
    double x(x) ;
      x:standard_name = "projection_x_coordinate" ;
      x:long_name = "Easting" ;
      x:units = "m" ;
    double y(y) ;
      y:standard_name = "projection_y_coordinate" ;
      y:long_name = "Northing" ;
      y:units = "m" ;
    double z(z) ;
      z:standard_name = "height_above_reference_ellipsoid" ;
      z:long_name = "height_above_osgb_newlyn_datum_masl" ;
      z:units = "m" ;
    double lat(y, x) ;
      lat:standard_name = "latitude" ;
      lat:units = "degrees_north" ;
    double lon(y, x) ;
      lon:standard_name = "longitude" ;
```

```
    lon:units = "degrees_east" ;
  float temp(z, y, x) ;
    temp:standard_name = "air_temperature" ;
    temp:units = "K" ;
    temp:coordinates = "lat lon" ;
    temp:grid_mapping = "crsOSGB: x y crsWGS84: lat lon" ;
  float pres(z, y, x) ;
    pres:standard_name = "air_pressure" ;
    pres:units = "Pa" ;
    pres:coordinates = "lat lon" ;
    pres:grid_mapping = "crsOSGB: x y crsWGS84: lat lon" ;
  int crsOSGB ;
    crsOSGB:grid_mapping_name = "transverse_mercator";
    crsOSGB:semi_major_axis = 6377563.396 ;
    crsOSGB:inverse_flattening = 299.3249646 ;
    crsOSGB:longitude_of_prime_meridian = 0.0 ;
    crsOSGB:latitude_of_projection_origin = 49.0 ;
    crsOSGB:longitude_of_central_meridian = -2.0 ;
    crsOSGB:scale_factor_at_central_meridian = 0.9996012717 ;
    crsOSGB:false_easting = 400000.0 ;
    crsOSGB:false_northing = -100000.0 ;
    crsOSGB:unit = "metre" ;
  int crsWGS84 ;
    crsWGS84:grid_mapping_name = "latitude_longitude";
    crsWGS84:longitude_of_prime_meridian = 0.0 ;
    crsWGS84:semi_major_axis = 6378137.0 ;
    crsWGS84:inverse_flattening = 298.257223563 ;
```

5.6.1. Use of the CRS Well-known Text Format

An optional grid mapping attribute called `crs_wkt` may be used to specify multiple coordinate system properties in so-called *well-known text* format (usually abbreviated to CRS WKT or OGC WKT). The CRS WKT format is widely recognised and used within the geoscience software community. As such it represents a versatile mechanism for encoding information about a variety of coordinate reference system parameters in a highly compact notational form. The translation of CF coordinate variables to/from OGC Well-Known Text (WKT) format is shown in Examples 5.11 and 5.12 below and described in detail in https://github.com/cf-convention/cf-conventions/wiki/Mapping-from-CF-Grid-Mapping-Attributes-to-CRS-WKT-Elements.

The `crs_wkt` attribute should comprise a text string that conforms to the WKT syntax as specified in reference [OGC_WKT-CRS]. If desired the text string may contain embedded newline characters to aid human readability. However, any such characters are purely cosmetic and do not alter the meaning of the attribute value. It is envisaged that the value of the `crs_wkt` attribute typically will be a single line of text, one intended primarily for machine processing. Other than the requirement to be a valid WKT string, the CF convention does not prescribe the content of the `crs_wkt` attribute since it will necessarily be context-dependent.

Where a `crs_wkt` attribute is added to a `grid_mapping`, the extended syntax for the `grid_mapping`

attribute enables the list of variables containing coordinate values being referenced to be explicitly stated and the CRS WKT Axis order to be explicitly defined. The explicit definition of WKT CRS Axis order is expected by the OGC standards for referencing by coordinates. Software implementing these standards are likely to expect to receive coordinate value tuples, with the correct coordinate value order, along with the coordinate reference system definition that those coordinate values are defined with respect to.

The order of the `<coordinatesVariable>` references within the `grid_mapping` attribute definition defines the order of elements within a derived coordinate value tuple. This enables an application reading the data from a file to construct an array of coordinate value tuples, where each tuple is ordered to match the specification of the coordinate reference system being used whilst the array of tuples is structured according to the netCDF definition. It is the responsibility of the data producer to ensure that the `<coordinatesVariable>` list is consistent with the CRS WKT definition of CS AXIS, with the correct number of entries in the correct order (note: this is not a conformance requirement as CF conformance is not dependent on CRS WKT parsing).

For example, a file has two coordinate variables, lon and lat, and a grid mapping variable `crs` with an associated `crs_wkt` attribute; the WKT definition defines the AXIS order as `["latitude", "longitude"]`. The `grid_mapping` attribute is thus given a value `crs:lat lon` to define that where coordinate pairs are required, these shall be ordered (lat, lon), to be consistent with the provided `crs_wkt` string (and not order inverted). A 2-D array of (lat, lon) tuples can then be explicitly derived from the combination of the lat and lon variables.

The `crs_wkt` attribute is intended to act as a *supplement* to other single-property CF grid mapping attributes (as described in Appendix F); it is not intended to replace those attributes. If data producers omit the single-property grid mapping attributes in favour of the `crs_wkt` attribute, software which cannot interpret `crs_wkt` will be unable to use the `grid_mapping` information. Therefore the CRS should be described as thoroughly as possible with the single-property grid mapping attributes as well as by `crs_wkt`.

In cases where CRS property values can be represented by both a single-property grid mapping attribute and the `crs_wkt` attribute, the grid mapping should be provided, and if both are provided, the onus is on data producers to ensure that their property values are consistent. Therefore information from either one (or both) may be read in by the user without needing to check both. However, if the two values of a given property are different, the CRS information cannot be interpreted accurately and users should inform the provider so the issue can be addressed. For example, if the semi-major axis length of the ellipsoid defined by the grid mapping attribute `semi_major_axis` disagrees with the `crs_wkt` attribute (via the `WKT SPHEROID[…]` element), the value of this attribute cannot be interpreted accurately. Naturally if the two values are equal then no ambiguity arises.

Likewise, in those cases where the value of a CRS WKT element should be used consistently across the CF-netCDF community (names of projections and projection parameters, for example) then, the values shown in https://github.com/cf-convention/cf-conventions/wiki/Mapping-from-CF-Grid-Mapping-Attributes-to-CRS-WKT-Elements should be preferred; these are derived from the OGP/EPSG registry of geodetic parameters, which is considered to represent the definitive authority as regards CRS property names and values.

Examples 5.11 illustrates how the coordinate system properties specified via the `crs` grid mapping

variable in Example 5.9 might be expressed using a crs_wkt attribute. Example 5.12 also illustrates the addition of the crs_wkt attribute, but here the attribute is added to the crs variable of a simplified variant of Example 5.10. For brevity in Example 5.11, only the grid mapping variable and its grid_mapping_name and crs_wkt attributes are included; all other elements are as per the Example 5.9. Names of projection. PARAMETERs follow the spellings used in the EPSG geodetic parameter registry.

Example 5.12 illustrates how certain WKT elements - all of which are optional - can be used to specify CRS properties not covered by existing CF grid mapping attributes, including:

- use of the VERT_DATUM element to specify vertical datum information
- use of additional PARAMETER elements (albeit not essential ones in this example) to define the location of the false origin of the projection
- use of AUTHORITY elements to specify object identifier codes assigned by an external authority, OGP/EPSG in this instance

Example 5.11. Latitude and longitude on the WGS 1984 datum + CRS WKT

```
...
 float data(latitude, longitude) ;
   data:grid_mapping = "crs: latitude, longitude" ;
   ...
 int crs ;
   crs:grid_mapping_name = "latitude_longitude";
   crs:longitude_of_prime_meridian = 0.0 ;
   crs:semi_major_axis = 6378137.0 ;
   crs:inverse_flattening = 298.257223563 ;
   crs:crs_wkt =
    GEODCRS["WGS 84",
    DATUM["World Geodetic System 1984",
      ELLIPSOID["WGS 84",6378137,298.257223563,
        LENGTHUNIT["metre",1.0]]],
    PRIMEM["Greenwich",0],
    CS[ellipsoidal,3],
      AXIS["(lat)",north,ANGLEUNIT["degree",0.0174532925199433]],
      AXIS["(lon)",east,ANGLEUNIT["degree",0.0174532925199433]],
      AXIS["ellipsoidal height (h)",up,LENGTHUNIT["metre",1.0]]]
 ...
```

Note: To enhance readability of these examples, the WKT value has been split across multiple lines and embedded quotation marks (") left unescaped - in real netCDF files such characters would need to be escaped. In CDL, within the CRS WKT definition string, newlines would need to be encoded within the string as \n and double quotes as \". Also for readability, we have dropped the quotation marks which would delimit the entire crs_wkt string. This pseudo CDL will not parse directly.

Example 5.12. British National Grid + Newlyn Datum in CRS WKT format

```
dimensions:
  lat = 648 ;
  lon = 648 ;
  y = 18 ;
  x = 36 ;
variables:
  double x(x) ;
    x:standard_name = "projection_x_coordinate" ;
    x:units = "m" ;
  double y(y) ;
    y:standard_name = "projection_y_coordinate" ;
    y:units = "m" ;
  float temp(y, x) ;
    temp:long_name = "temperature" ;
    temp:units = "K" ;
    temp:coordinates = "lat lon" ;
    temp:grid_mapping = "crs: x y" ;
  int crs ;
    crs:grid_mapping_name = "transverse_mercator" ;
    crs:longitude_of_central_meridian = -2. ;
    crs:false_easting = 400000. ;
    crs:false_northing = -100000. ;
    crs:latitude_of_projection_origin = 49. ;
    crs:scale_factor_at_central_meridian = 0.9996012717 ;
    crs:longitude_of_prime_meridian = 0. ;
    crs:semi_major_axis = 6377563.396 ;
    crs:inverse_flattening = 299.324964600004 ;
    crs:projected_coordinate_system_name = "OSGB 1936 / British National Grid" ;
    crs:geographic_coordinate_system_name = "OSGB 1936" ;
    crs:horizontal_datum_name = "OSGB_1936" ;
    crs:reference_ellipsoid_name = "Airy 1830" ;
    crs:prime_meridian_name = "Greenwich" ;
    crs:towgs84 = 375., -111., 431., 0., 0., 0., 0. ;
    crs:crs_wkt = "COMPOUNDCRS["OSGB 1936 / British National Grid + ODN",
      PROJCRS["OSGB 1936 / British National Grid",
        BASEGEODCRS["OSGB 1936",
          DATUM["OSGB 1936",
            ELLIPSOID["Airy 1830", 6377563.396, 299.3249646,
              LENGTHUNIT["metre",1.0]]
          ],
          PRIMEM ["Greenwich", 0],
          UNIT ["degree", 0.0174532925199433]
        ],
        CONVERSION["OSGB",
          METHOD["Transverse Mercator"],
          PARAMETER["False easting", 400000, LENGTHUNIT["metre",1.0]],
          PARAMETER["False northing", -100000, LENGTHUNIT["metre",1.0]],
          PARAMETER["Longitude of natural origin", -2.0,
```

```
            ANGLEUNIT["degree",0.0174532925199433]],
          PARAMETER["Latitude of natural origin", 49.0,
            ANGLEUNIT["degree",0.0174532925199433]],
          PARAMETER["Longitude of false origin", -7.556,
            ANGLEUNIT["degree",0.0174532925199433]],
          PARAMETER["Latitude of false origin", 49.766,
            ANGLEUNIT["degree",0.0174532925199433]],
          PARAMETER["Scale factor at natural origin", 0.9996012717,
SCALEUNIT["Unity",1.0]]
        ],
        CS[Cartesian, 2],
        AXIS["easting (X)",east],
        AXIS["northing (Y)",north],
        LENGTHUNIT["metre",1.0],
        ID["EPSG",27700]
      ],
      VERTCRS["Newlyn",
        VDATUM["Ordnance Datum Newlyn"],
        CS[vertical,1],
        AXIS["gravity-related height (H)",up],
        LENGTHUNIT["metre",1.0],
        ID["EPSG",5701]
      ]
      ]" ;
  ...
```

Note: There are unescaped double quotes and newlines and the quotation marks which would delimit the entire `crs_wkt` string are missing in this example. This is to enhance readability, but it means that this pseudo CDL will not parse directly.

The preceding two example (5.11 and 5.12) may be combined, if the data provider desires to provide explicit latitude and longitude coordinates as well as projection coordinates and to provide CRS WKT referencing for both sets of coordinates. This is demonstrated in example 5.13.

Example 5.13. British National Grid + Newlyn Datum + referenced WGS84 Geodetic in CRS WKT format

```
...
  double x(x) ;
    x:standard_name = "projection_x_coordinate" ;
    x:units = "m" ;
  double y(y) ;
    y:standard_name = "projection_y_coordinate" ;
    y:units = "m" ;
  double lat(y, x) ;
    lat_standard_name = "latitude" ;
    lat:units = "degrees_north" ;
  double lon(y, x) ;
    lon_standard_name = "longitude" ;
    lon:units = "degrees_east" ;
```

```
  float temp(y, x) ;
    temp:long_name = "temperature" ;
    temp:units = "K" ;
    temp:coordinates = "lat lon" ;
    temp:grid_mapping = "crs_osgb: x y crs_wgs84: latitude longitude" ;
    ...
  int crs_wgs84 ;
    crs_wgs84:grid_mapping_name = "latitude_longitude";
    crs_wgs84:crs_wkt = ...
  int crs_osgb ;
    crs_osgb:grid_mapping_name = "transverse_mercator" ;
    crs_osgb:crs_wkt = ...
  ...
```

Note: There are unescaped double quotes and newlines and the quotation marks which would delimit the entire `crs_wkt` string are missing in this example. This is to enhance readability, but it means that this pseudo CDL will not parse directly.

5.7. Scalar Coordinate Variables

When a variable has an associated coordinate which is single-valued, that coordinate may be represented as a scalar variable (i.e. a data variable which has no netCDF dimensions). Since there is no associated dimension these scalar coordinate variables should be attached to a data variable via the `coordinates` attribute.

The use of scalar coordinate variables is a convenience feature which avoids adding size one dimensions to variables. A numeric scalar coordinate variable has the same information content and can be used in the same contexts as a size one numeric coordinate variable. Similarly, a string-valued scalar coordinate variable has the same meaning and purposes as a size one string-valued auxiliary coordinate variable (Section 6.1, "Labels"). Note however that use of this feature with a latitude, longitude, vertical, or time coordinate will inhibit COARDS conforming applications from recognizing them.

Once a name is used for a scalar coordinate variable it can not be used for a 1D coordinate variable. For this reason we strongly recommend against using a name for a scalar coordinate variable that matches the name of any dimension in the file.

If a data variable has two or more scalar coordinate variables, they are regarded as though they were all independent coordinate variables with dimensions of size one. If two or more single-valued coordinates are not independent, but have related values (this might be the case, for instance, for time and forecast period, or vertical coordinate and model level number, Section 6.2, "Alternative Coordinates"), they should be stored as coordinate or auxiliary coordinate variables of the same size one dimension, not as scalar coordinate variables.

Example 5.14. Multiple forecasts from a single analysis

```
dimensions:
```

```
  lat = 180 ;
  lon = 360 ;
  time = UNLIMITED ;
variables:
  double atime
    atime:standard_name = "forecast_reference_time" ;
    atime:units = "hours since 1999-01-01 00:00" ;
  double time(time);
    time:standard_name = "time" ;
    time:units = "hours since 1999-01-01 00:00" ;
  double lon(lon) ;
    lon:long_name = "station longitude";
    lon:units = "degrees_east";
  double lat(lat) ;
    lat:long_name = "station latitude" ;
    lat:units = "degrees_north" ;
  double p500
    p500:long_name = "pressure" ;
    p500:units = "hPa" ;
    p500:positive = "down" ;
  float height(time,lat,lon);
    height:long_name = "geopotential height" ;
    height:standard_name = "geopotential_height" ;
    height:units = "m" ;
    height:coordinates = "atime p500" ;
data:
  time = 6., 12., 18., 24. ;
  atime = 0. ;
  p500 = 500. ;
```

In this example both the analysis time and the single pressure level are represented using scalar coordinate variables. The analysis time is identified by the standard name `forecast_reference_time` while the valid time of the forecast is identified by the standard name `time`.

5.8. Domain Variables

A domain describes data locations and cell properties. It defines cells that span a collection of dimensions with cell coordinates, cell measures, and coordinate reference systems.

A data variable defines its domain via its own attributes, but a domain variable provides the description of a domain in the absence of any data values. The variable should be a scalar (i.e. it has no dimensions) of arbitrary type, and the value of its single element is immaterial. It acts as a container for the attributes that define the domain. The purpose of a domain variable is to provide domain information to applications that have no need of data values at the domain's locations, thus removing any ambiguity when retrieving a domain from a dataset. Ancillary variables and cell methods are not part of the domain, because they are only defined in relation to data values.

The domain variable supports the same attributes as are allowed on a data variable for describing a

domain, with exactly the same meanings and syntaxes, as described in Appendix A, Attributes. If an attribute is needed by a particular data variable to describe its domain, then that attribute would also be needed by the equivalent domain variable.

The dimensions of the domain must be stored with the `dimensions` attribute, and the presence of a `dimensions` attribute will identify the variable as a domain variable. Therefore the `dimensions` attribute must not be present on any variables that are to be interpreted as data variables. It is necessary to list these dimensions, rather than inferring them from the contents of the other attributes, as it can not be guaranteed that the referenced variables span all of the required dimensions (as could be the case for a discrete axis, for instance). The value of the `dimensions` attribute is a blank separated list of the dimension names. There is no restriction on the order in which the dimensions appear in the `dimensions` attribute string. If a domain has no named dimensions then the value of the `dimensions` attribute must be an empty string, as could be the case if the dimensions of the domain are all defined implicitly by scalar coordinate variables.

The dimensions listed by the `dimensions` attribute constrain the dimensions that may be spanned by variables referenced from any of the other attributes, in the same way that the array dimensions perform that role for a data variable. For instance, all variables named by the `cell_measures` attribute (Section 7.2, "Cell Measures") of a domain variable must span a subset of zero or more of the dimensions given by the `dimensions` attribute.

It is optional for coordinate variables to be listed by a domain variable's `coordinates` attribute. Any coordinate variable that shares its name with a dimension given by the `dimensions` attribute will be considered as part of the domain definition.

It is recommended that a domain variable has a `long_name` attribute to describe its contents.

It is recommended that a domain variable does not have any of the attributes marked in Appendix A, Attributes as applicable to data variables except those which are also marked as applicable to domain variables.

Multiple domain variables may exist in a file with, or without, data variables. Note that the data variable attributes describing its domain can not be replaced by a reference to a domain variable.

Example 5.15. A domain with independent coordinate variables.

```
dimensions:
  lat = 18 ;
  lon = 36 ;
  pres = 15 ;
  time = 4 ;

variables:
  char domain ;
    domain:dimensions = "time pres lat lon" ;
    domain:long_name = "Domain with independent coordinate variables" ;
  float lon(lon) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
  float lat(lat) ;
```

```
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
  float pres(pres) ;
    pres:long_name = "pressure" ;
    pres:units = "hPa" ;
  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1990-1-1 0:0:0" ;
```

In this example the data variable `xwind` from the Independent coordinate variables example has been replaced by the domain variable `domain`.

Example 5.16. A domain with a rotated pole grid and a scalar coordinate variable.

```
dimensions:
  rlon = 128 ;
  rlat = 64 ;
  lev = 18 ;

variables:
  char domain ;
    domain:dimensions = "lev rlat rlon" ;
    domain:coordinates = "lon lat time" ;
    domain:grid_mapping = "rotated_pole" ;
    domain:long_name = "Domain with grid mapping and scalar coordinate" ;
  char rotated_pole ;
    rotated_pole:grid_mapping_name = "rotated_latitude_longitude" ;
    rotated_pole:grid_north_pole_latitude = 32.5 ;
    rotated_pole:grid_north_pole_longitude = 170. ;
  double time
    time:standard_name = "time" ;
    time:units = "days since 2000-12-01 00:00" ;
  float rlon(rlon) ;
    rlon:long_name = "longitude in rotated pole grid" ;
    rlon:units = "degrees" ;
    rlon:standard_name = "grid_longitude" ;
  float rlat(rlat) ;
    rlat:long_name = "latitude in rotated pole grid" ;
    rlat:units = "degrees" ;
    rlat:standard_name = "grid_latitude" ;
  float lev(lev) ;
    lev:long_name = "pressure level" ;
    lev:units = "hPa" ;
  float lon(rlat,rlon) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
  float lat(rlat,rlon) ;
    lat:long_name = "latitude" ;
```

```
    lat:units = "degrees_north" ;
```

Example 5.17. A domain containing cell areas for a spherical geodesic grid.

```
dimensions:
  cell = 2562 ;  // number of grid cells
  time = 12 ;
  nv = 6 ;       // maximum number of cell vertices

variables:
  char domain ;
    domain:dimensions = "time cell" ;
    domain:coordinates = "lon lat" ;
    domain:cell_measures = "area: cell_area" ;
    domain:long_name = "Domain with cell measures" ;
  float lon(cell) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
    lon:bounds = "lon_vertices" ;
  float lat(cell) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
    lat:bounds = "lat_vertices" ;
  float time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1979-01-01" ;
  float cell_area(cell) ;
    cell_area:long_name = "area of grid cell" ;
    cell_area:standard_name = "cell_area" ;
    cell_area:units = "m2"
  float lon_vertices(cell, nv) ;
  float lat_vertices(cell, nv) ;
```

In this example the data variable PS from the Cell areas for a spherical geodesic grid example has been replaced by the domain variable domain.

Example 5.18. A domain with no explicit dimensions.

```
dimensions:

variables:
  char domain ;
    domain:dimensions = "" ;
    domain:coordinates = "t" ;
    domain:long_name = "Domain with no explicit dimensions" ;
  double t ;
    t:standard_name = "time" ;
```

```
    t:units = "days since 2021-01-01" ;
```

Example 5.19. A domain containing a timeseries geometry.

```
dimensions:
  instance = 2 ;
  node = 5 ;
  time = 4 ;

variables:
  char domain ;
    domain:dimensions = "instance time" ;
    domain:coordinates = "lat lon" ;
    domain:grid_mapping = "datum" ;
    domain:geometry = "geometry_container" ;
    domain:long_name = "Domain with a geometry variable" ;
  int time(time) ;
  double lat(instance) ;
    lat:units = "degrees_north" ;
    lat:standard_name = "latitude" ;
    lat:nodes = "y" ;
  double lon(instance) ;
    lon:units = "degrees_east" ;
    lon:standard_name = "longitude" ;
    lon:nodes = "x" ;
  int datum ;
    datum:grid_mapping_name = "latitude_longitude" ;
    datum:longitude_of_prime_meridian = 0.0 ;
    datum:semi_major_axis = 6378137.0 ;
    datum:inverse_flattening = 298.257223563 ;
  int geometry_container ;
    geometry_container:geometry_type = "line" ;
    geometry_container:node_count = "node_count" ;
    geometry_container:node_coordinates = "x y" ;
  int node_count(instance) ;
  double x(node) ;
    x:units = "degrees_east" ;
    x:standard_name = "longitude" ;
    x:axis = "X" ;
  double y(node) ;
    y:units = "degrees_north" ;
    y:standard_name = "latitude" ;
    y:axis = "Y" ;
```

In this example the data variable `someData` from the Timeseries with geometry. example has been replaced by the domain variable `domain`.

Example 5.20. A domain containing a timeseries of station data in the indexed ragged array representation.

```
dimensions:
  station = 23 ;
  obs = UNLIMITED ;
  name_strlen = 23 ;

variables:
  char domain ;
    domain:dimensions = "obs" ;
    domain:coordinates = "time lat lon alt station_name" ;
    domain:long_name = "Domain with a discrete sampling geometry" ;
  float lon(station) ;
    lon:standard_name = "longitude" ;
    lon:long_name = "station longitude" ;
    lon:units = "degrees_east" ;
  float lat(station) ;
    lat:standard_name = "latitude" ;
    lat:long_name = "station latitude" ;
    lat:units = "degrees_north" ;
  float alt(station) ;
    alt:long_name = "vertical distance above the surface" ;
    alt:standard_name = "height" ;
    alt:units = "m" ;
    alt:positive = "up" ;
    alt:axis = "Z" ;
  char station_name(station, name_strlen) ;
    station_name:long_name = "station name" ;
    station_name:cf_role = "timeseries_id" ;
  int station_info(station) ;
    station_info:long_name = "some kind of station info" ;
  int stationIndex(obs) ;
    stationIndex:long_name = "which station this obs is for" ;
    stationIndex:instance_dimension = "station" ;
  double time(obs) ;
    time:standard_name = "time" ;
    time:long_name = "time of measurement" ;
    time:units = "days since 1970-01-01 00:00:00" ;

attributes:
    :featureType = "timeSeries" ;
```

In this example the data variables `humidity` and `temp` from the Timeseries of station data in the indexed ragged array representation. example have been replaced by the domain variable `domain`.

5.9. Mesh Topology Variables

A *mesh topology variable* defines the geospatial topology of cells arranged in two or three dimensions in real space but indexed by a single dimension. It explicitly describes the topological relationships between cells, i.e. spatial relationships which do not depend on the cell locations, via a mesh of connected nodes. A mesh topology variable may provide the topology for one or more domains, defined at the nodes, edges, or faces of the mesh. See the Domain topology construct and Cell connectivity construct descriptions in the CF data model for more details, including on how the mesh relates to the cells of the domain.

The canonical definitions of mesh topology variables and location index set variables are given externally by the UGRID conventions [UGRID], but their standardized attributes, many of which are optional, are listed in Appendix K, Mesh Topology Attributes and Appendix A, Attributes. Some features of the UGRID conventions [UGRID] are not currently recognized by the CF conventions: mesh topology volume cells (that are used to describe fully three-dimensional unstructured mesh topologies); and the "boundary node connectivity" variable (that specifies an index variable identifying the nodes that define where boundary condtions have been provided).

A data or domain variable may use one of a mesh topology variable's domains by referencing the mesh topology variable with the **`mesh`** attribute; along with the identity of required domain provided by the **`location`** attribute (see example A two-dimensional UGRID mesh topology variable).

The variables containing the coordinate values for cells indexed by the mesh topology are defined by the mesh topology variable but are equivalent to one-dimensional auxiliary coordinate variables, and so may also be provided by the data or domain variable's **`coordinates`** attribute. Note that the mesh topology variable allows cell bounds to be provided without any cell coordinate values, via its **`node_coordinates`** attribute.

A *location index set variable* defines a subset of locations of a mesh topology variable, e.g. only special locations like weirs and gates. It is provided as a space saving device to prevent the need to redefine parts of an existing mesh topology variable, and as such is logically equivalent to a mesh topology variable. A data or domain variable references a location index set variable via its **`location_index_set`** attribute.

Example 5.21. A two-dimensional UGRID mesh topology variable

```
dimensions:
  node = 5 ;  // Number of mesh nodes
  edge = 6 ;  // Number of mesh edges
  face = 2 ;  // Number of mesh faces
  two = 2 ;   // Number of nodes per edge
  four = 4 ;  // Maximum number of nodes per face
  time = 12 ;

variables:
  // Mesh topology variable
  integer mesh ;
    mesh:cf_role = "mesh_topology" ;
    mesh:long_name = "Topology of a 2-d unstructured mesh" ;
```

```
    mesh:topology_dimension = 2 ;
    mesh:node_coordinates = "mesh_node_x mesh_node_y" ;
    mesh:edge_node_connectivity = "mesh_edge_nodes" ;
    mesh:face_node_connectivity = "mesh_face_nodes" ;

  // Mesh node coordinates
  double mesh2_node_x(node) ;
    mesh_node_x:standard_name = "longitude" ;
    mesh_node_x:units = "degrees_east" ;
  double mesh2_node_y(node) ;
    mesh_node_y:standard_name = "latitude" ;
    mesh_node_y:units = "degrees_north" ;

  // Mesh connectivity variables
  integer mesh_face_nodes(face, four) ;
    mesh_face_nodes:long_name = "Maps each face to its 3 or 4 corner nodes" ;
  integer mesh_edge_nodes(edge, two) ;
    mesh_edge_nodes:long_name = "Maps each edge to the 2 nodes it connects" ;

  // Coordinate variables
  float time(time) ;
    time:standard_name = "time" ;
    time:units = "days since 2004-06-01" ;

  // Data at mesh faces
  double volume_at_faces(time, face) ;
    volume_at_faces:standard_name = "air_density" ;
    volume_at_faces:units = "kg m-3" ;
    volume_at_faces:mesh = "mesh" ;
    volume_at_faces:location = "face" ;
  // Data at mesh edges
  double flux_at_edges(time, edge) ;
    fluxe_at_edges:standard_name = "northward_wind" ;
    fluxe_at_edges:units = "m s-1" ;
    fluxe_at_edges:mesh = "mesh"
    fluxe_at_edges:location = "edge" ;
  // Data at mesh nodes
  double height_at_nodes(time, node) ;
    height_at_nodes:standard_name = "sea_surface_height_above_geoid" ;
    height_at_nodes:units = "m" ;
    height_at_nodes:mesh = "mesh" ;
    height_at_nodes:location = "node" ;
```

A two-dimensional UGRID mesh topology variable for the mesh depicted in Figure I.5, with data variables defined at face, edge and node elements of the mesh. All optional attributes have been omitted.

Chapter 6. Labels and Alternative Coordinates

6.1. Labels

Character strings can be used to provide a name or label for each element of an axis. This is particularly useful for discrete axes (section 4.5). For instance, if a data variable contains time series of observational data from a number of observing stations, it may be convenient to provide the names of the stations as labels for the elements of the station dimension (Section H.2, "Time Series Data"). There are several other uses for labels in CF. For instance, Northward heat transport in Atlantic Ocean shows the use of labels to indicate geographic regions.

Character strings labelling the elements of an axis are regarded as string-valued auxiliary coordinate variables. The `coordinates` attribute of the data variable names the variable that contains the string array. An application processing the variables listed in the `coordinates` attribute can recognize a string-valued auxiliary coordinate variable because it has a type of `char` or `string`. If the variable has a type of `char`, the inner dimension (last dimension in CDL terms) is the maximum length of each string, and the other dimensions are axis dimensions. If an auxiliary coordinate variable has a type of `string` and has no dimensions, or has a type of `char` and has only one dimension (the maximum length of the string), it is a string-valued scalar coordinate variable (see Section 5.7, "Scalar Coordinate Variables"). As such, it has the same information content and can be used in the same contexts as a string-valued auxiliary coordinate variable of a size one dimension. This is a convenience feature.

6.1.1. Geographic Regions

When data is representative of geographic regions which can be identified by names but which have complex boundaries that cannot practically be specified using longitude and latitude boundary coordinates, a labeled axis should be used to identify the regions. We recommend that the names be chosen from the list of standardized region names whenever possible. To indicate that the label values are standardized the variable that contains the labels must be given the `standard_name` attribute with the value `region`.

Example 6.1. Northward heat transport in Atlantic Ocean

Suppose we have data representing northward heat transport across a set of zonal slices in the Atlantic Ocean. Note that the standard names to describe this quantity do not include location information. That is provided by the latitude coordinate and the labeled axis:

```
dimensions:
  times = 20 ;
  lat = 5
  lbl = 1 ;
variables:
  float n_heat_transport(time,lat,lbl);
    n_heat_transport:units="W";
```

```
    n_heat_transport:coordinates="geo_region";
    n_heat_transport:standard_name="northward_ocean_heat_transport";
  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1990-1-1 0:0:0" ;
  float lat(lat) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
  string geo_region(lbl) ;
    geo_region:standard_name="region"
data:
  geo_region = "atlantic_ocean" ;
  lat = 10., 20., 30., 40., 50. ;
```

6.1.2. Taxon Names and Identifiers

A taxon is a named level within a biological classification, such as a class, genus and species. Quantities dependent on taxa have generic standard names containing the phrase "organisms_in_taxon", and the taxa are identified by auxiliary coordinate variables.

The taxon auxiliary coordinate variables are string-valued. The plain-language name of the taxon must be contained in a variable with `standard_name` of `biological_taxon_name`. A Life Science Identifier (LSID) may be contained in a variable with `standard_name` of `biological_taxon_lsid`. This is a URN with the syntax "urn:lsid:<Authority>:<Namespace>:<ObjectID>[:<Version>]". This includes the reference classification in the <Authority> element and these are restricted by the LSID governance. It is strongly recommended in CF that the authority chosen is World Register of Marine Species (WoRMS) for oceanographic data and Integrated Taxonomic Information System (ITIS) for freshwater and terrestrial data. WoRMS LSIDs are built from the WoRMS AphiaID taxon identifier such as "urn:lsid:marinespecies.org:taxname:104464" for AphiaID 104464. This may be converted to a URL by adding prefixes such as https://www.lsid.info/. ITIS LSIDs are built from the ITIS Taxonomic Serial Number (TSN), such as "urn:lsid:itis.gov:itis_tsn:180543".

The `biological_taxon_name` auxiliary coordinate variable included for human readability is mandatory. The `biological_taxon_lsid` auxliary coordinate variable included for software agent readability is optional, but strongly recommended. If both are present then each `biological_taxon_name` coordinate must exactly match the name resolved from the `biological_taxon_lsid` coordinate. If LSIDs are available for some taxa in a dataset then the `biological_taxon_lsid` auxiliary coordinate variable should be included and missing data given for those taxa that do not have an identifier.

Example 6.1.2. Taxon names and identifiers

A skeleton example for taxonomic abundance time series.

```
dimension:
  time = 100 ;
  string80 = 80 ;
  taxon = 2 ;
```

```
variables:
  float time(time);
    time:standard_name = "time" ;
    time:units = "days since 2019-01-01" ;
  float abundance(time,taxon) ;
    abundance:standard_name =
"number_concentration_of_biological_taxon_in_sea_water" ;
    abundance:coordinates = "taxon_lsid taxon_name" ;
  char taxon_name(taxon,string80) ;
    taxon_name:standard_name = "biological_taxon_name" ;
  char taxon_lsid(taxon,string80) ;
    taxon_lsid:standard_name = "biological_taxon_lsid" ;
data:
  time = // 100 values ;
  abundance = // 200 values ;
  taxon_name = "Calanus finmarchicus", "Calanus helgolandicus" ;
  taxon_lsid = "urn:lsid:marinespecies.org:taxname:104464",
"urn:lsid:marinespecies.org:taxname:104466" ;
```

6.2. Alternative Coordinates

In some situations a dimension may have alternative sets of coordinates values. Since there can only be one coordinate variable for the dimension (the variable with the same name as the dimension), any alternative sets of values have to be stored in auxiliary coordinate variables. For such alternative coordinate variables, there are no mandatory attributes, but they may have any of the attributes allowed for coordinate variables.

Example 6.2. Model level numbers

Levels on a vertical axis may be described by both the physical coordinate and the ordinal model level number.

```
float xwind(sigma,lat);
  xwind:coordinates="model_level";
float sigma(sigma); // physical height coordinate
  sigma:long_name="sigma";
  sigma:positive="down";
int model_level(sigma); // model level number at each height
  model_level:long_name="model level number";
  model_level:positive="up";
```

Chapter 7. Data Representative of Cells

When gridded data does not represent the point values of a field but instead represents some characteristic of the field within cells of non-zero size, a complete description of the variable should include metadata that describes the domain or extent of each cell, and the characteristic of the field that the cell values represent. The commonest cases are one-dimensional cells along spatiotemporal axes, for instance cells along a time axis for consecutive months whose values contain monthly means. The methods presented in Section 7.1, "Cell Boundaries" and Section 7.3, "Cell Methods" describe cases in which each grid point is associated with a cell consisting of a single one-dimensional interval, a single two-dimensional polygonal area, or in general a single *n*-dimensional volume in the *n*-dimensional space described by its coordinate variables.

It is possible for a single data value to be the result of an operation whose domain is a disjoint set of intervals or areas. This is true for many types of climatological statistic; for example, the mean January temperature for the years 1971-2000 is computed from the 30 individual months of January, which are a set of discontiguous time-intervals. Climatological statistics are of such importance that we provide special methods for describing their associated computational domains in Section 7.4, "Climatological Statistics". As an alternative to *n*-dimensional volumes with bounds, we provide Section 7.5, "Geometries", for the case of geospatial applications in which each data value pertains to a single real-world feature, such as a river, watershed or country, represented by one or more points, lines or polygons.

7.1. Cell Boundaries

To delimit the cells, the `bounds` attribute may be added to the appropriate coordinate variable(s). The value of `bounds` is the name of the variable that contains the vertices of the cell boundaries. We refer to this type of variable as a "boundary variable." If cell boundaries are provided, it is recommended that each gridpoint should lie somewhere within or upon the boundaries of its own cell.

If cell boundaries are not provided (using the `bounds` attribute), an application can make no assumption about the location or extent of the cells. Without a boundary variable, it is unknown whether adjacent cells are contiguous, separated by a gap, or overlapping. If the data value pertains to the gridpoint alone, rather than to an interval, area or *n*-dimensional volume of non-zero size, it is recommended to indicate this with a `cell_methods` entry of `point` (Section 7.3, "Cell Methods"). In that case, the cell is irrelevant to the data and the bounds are arbitrary. Nonetheless, the bounds may still be included, for instance because the grid is shared by other data variables that pertain to cells, or to provide some indication of cells to generic applications for graphical purposes. A cell of truly zero size can be indicated by giving it coincident boundaries.

A boundary variable must have one more dimension than its associated coordinate or auxiliary coordinate variable. We refer to the additional dimension as the "vertex dimension". The vertex dimension must be the most rapidly varying dimension (the last dimension in CDL order), and its size is the maximum number of cell vertices.

The vertex dimension must be of size two if the associated variable is one-dimensional (Section 7.1.2, "Bounds for one-dimensional coordinate variables"), and of size greater than two if the associated variable has more than one dimension (Section 7.1.1, "Bounds for horizontal coordinate

variables with four-sided cells"). For grids constructed from cells that do not all have the same number of sides (e.g., a grid with some rectangular cells and some triangular cells), the vertex dimension must be at least as large as the maximum number of cell vertices (Section 7.1.3, "Bounds for coordinate variables with p-sided cells in two spatial dimensions"). For cells with fewer vertices than the size of vertex dimension, the unneeded elements must appear as the last elements in the vertex dimension and must be assigned the `_FillValue`. CF can currently describe boundaries for cells which have one or two spatial dimensions, but does not provide conventions to describe the boundaries of cells with three spatial dimensions. Such conventions are under consideration in [UGRID].

A boundary variable inherits the values of some attributes from its parent coordinate variable. If a coordinate variable has any of the attributes marked "BI" (for "inherit") in the "Use" column of Appendix A, Attributes, they are assumed to apply to its bounds variable as well. It is recommended that BI attributes not be included on a boundary variable. If a BI attribute is included, it must also be present in the parent variable, and it must exactly match the parent attribute's data type and value. A bounds variable may have any of the attributes marked "BO" for ("own") in the "Use" column of Appendix A, Attributes. These attributes take precedence over any corresponding attributes of the parent variable. In these cases, the parent variable's attribute does not apply to the bounds variable, regardless of whether the latter has its own attribute.

7.1.1. Bounds for one-dimensional coordinate variables

For a one-dimensional coordinate variable of size *N*, the boundary variable is an array of shape (*N*,2). The bounds for cell *i* are the elements *B*(*i*,0) and *B*(*i*,1) of the boundary variable *B*. Element *C*(*i*) of the coordinate variable *C* should lie between the boundaries of the cell, or upon one of them i.e. *B*(*i*,0) - *C*(*i*) and *B*(*i*,1) - *C*(*i*) should not have the same sign, though one of them could be zero (Figure 7.1).

If $N > 1$, the bounds of each cell must be ordered consistently with the coordinates i.e. $B(i,0) < B(i,1)$ for all i if $C(i) < C(i + 1)$, and $B(i,0) > B(i,1)$ for all i if $C(i) > C(i + 1)$.

If any two cells are contiguous, their shared boundary must be represented identically in each instance where it occurs in the boundary variable. This means that in the common case of *N* non-overlapping contiguous intervals, *N* - 1 of the boundaries are duplicated, because they are shared by adjacent intervals. This representation has the advantage that it is general enough to handle, without modification, non-contiguous intervals, as well as intervals on an axis using the unlimited dimension.

Example 7.1. Cells on a time axis

```
dimensions:
  time = 60;
  nv = 2;    // number of vertices
variables:
  float time(time);
    time:standard_name = "time";
    time:units = "days since 2024-11-8 09:00:00Z";
    time:bounds = "time_bnds";
```

```
float time_bnds(time,nv);
```

The boundary variable `time_bnds` associates a time point `i` with the time interval whose boundaries are `time_bnds(i,0)` and `time_bnds(i,1)`. The instant `time(i)` should be contained within the interval, or be at one end of it. For instance, with `i=2` we might have `time(2)=10.5`, `time_bnds(2,0)=10.0`, `time_bnds(2,1)=11.0`. If the times are increasing e.g. `time(3)` = `11.5` > `10.5` = `time(2)`, which implies `time(i+1)` > `time(i)` for all `i` because coordinates must be monotonic, the bounds must also be increasing for all `i`, e.g. `timebnd(2,1)` >= `timebnd(2,0)`. If adjacent intervals are contiguous, the shared endpoint must be identical. For example, if the interval `i=3` begins at `11.0` days, when interval `i=2` ends, the values in `timebnd(3,0)` and `timebnd(2,1)` must be *exactly* the same.

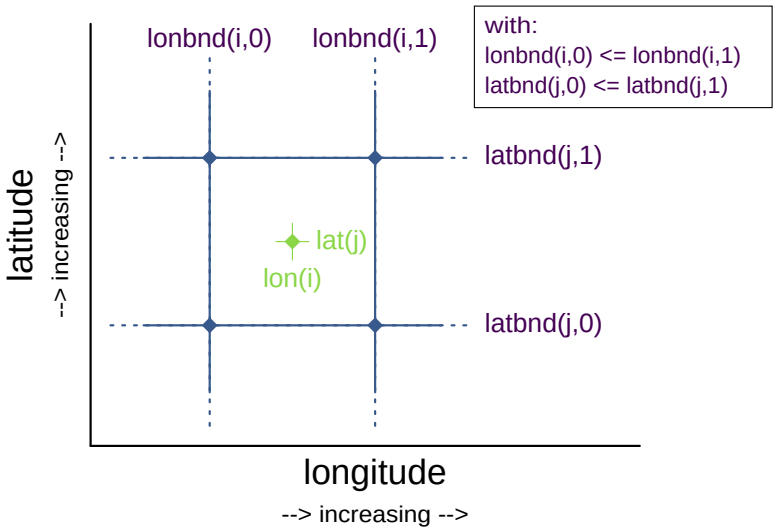


Figure 7.1. Order of `lonbnd(i,0)` and `lonbnd(i,1)` as well as of `latbnd(i,0)` and `latbnd(i,1)` in the case of one-dimensional horizontal coordinate axes. Tuples `(lon(i),lat(j))` represent grid cell centers. The four grid cell vertices are given by `(lonbnd(i,0),latbnd(j,0))`, `(lonbnd(i,1),latbnd(j,0))`, `(lonbnd(i,1),latbnd(j,1))` and `(lonbnd(i,0),latbnd(j,1))`.

7.1.2. Bounds for horizontal coordinate variables with four-sided cells

There is a common case of a rectangular horizontal grid, with four-sided cells, whose two axes are not latitude and longitude (e.g. it uses a map projection from Section 5.6, "Horizontal Coordinate Reference Systems, Grid Mappings, and Projections" or a curvilinear grid, such as the tripolar ocean grid). In that case, two-dimensional auxiliary coordinate variables in latitude `lat(n,m)` and longitude `lon(n,m)` may be provided as well. Since the sides of the cells do not generally have constant latitude or longitude, all four vertices must be specified individually. Therefore the boundary variables for the two-dimensional auxiliary coordinate variables are given in the form `latbnd(n,m,4)` and `lonbnd(n,m,4)`, where the trailing index runs over the four vertices of the cells.

Example 7.2. Cells in a non-latitude-longitude horizontal grid

```
dimensions:
  imax = 128;
  jmax = 64;
  nv = 4;
```

```
variables:
  float lat(jmax,imax);
    lat:long_name = "latitude";
    lat:units = "degrees_north";
    lat:bounds = "lat_bnds";
  float lon(jmax,imax);
    lon:long_name = "longitude";
    lon:units = "degrees_east";
    lon:bounds = "lon_bnds";
  float lat_bnds(jmax,imax,nv);
  float lon_bnds(jmax,imax,nv);
```

The boundary variables `lat_bnds` and `lon_bnds` associate a gridpoint `(j,i)` with the cell determined by the vertices `(lat_bnds(j,i,n),lon_bnds(j,i,n))`, `n=0,..,3`. The gridpoint location, `(lat(j,i),lon(j,i))`, should be contained within this region.

The vertices must be ordered such that, when visiting the vertices in order, the four-sided perimeter of the cell is traversed anticlockwise on the lon-lat surface as seen from above. If i-j-upward is a right-handed coordinate system (like lon-lat-upward), this can be arranged as in Figure 7.2. Let us call the side of cell `(j,i)` facing cell `(j,i-1)` the "`i-1`" side, the side facing cell `(j,i+1)` the "`i+1`" side, and similarly for "`j-1`" and "`j+1`". Then we can refer to the vertex formed by sides `i-1` and `j-1` as `(j-1,i-1)`. With this notation, the four vertices are indexed as follows: `0=(j-1,i-1)`, `1=(j-1,i+1)`, `2=(j+1,i+1)`, `3=(j+1,i-1)`.

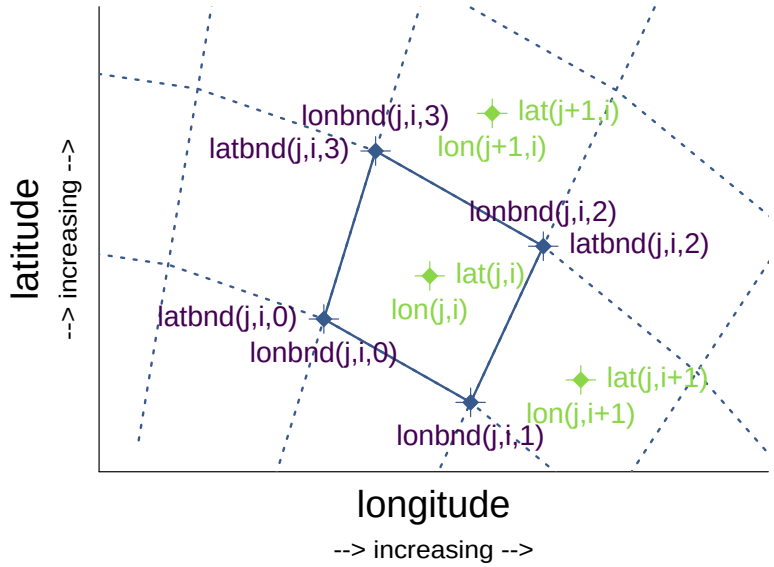


Figure 7.2. Order of `lonbnd(j,i,0)` to `lonbnd(j,i,3)` and of `latbnd(j,i,0)` and `latbnd(j,i,3)` in the case of two-dimensional horizontal coordinate axes. Tuples `(lon(j,i),lat(j,i))` represent grid cell centers and tuples `(lonbnd(j,i,n),latbnd(j,i,n))` represent the grid cell vertices.

The bounds can be used to decide whether cells are contiguous via the following relationships. In these equations the variable `bnd` is used generically to represent either the latitude or longitude boundary variable.

```
For 0 < j < n and 0 < i < m,
```

```
    If cells (j,i) and (j,i+1) are contiguous, then
        bnd(j,i,1)=bnd(j,i+1,0)
        bnd(j,i,2)=bnd(j,i+1,3)
    If cells (j,i) and (j+1,i) are contiguous, then
        bnd(j,i,3)=bnd(j+1,i,0) and bnd(j,i,2)=bnd(j+1,i,1)
```

7.1.3. Bounds for coordinate variables with p-sided cells in two spatial dimensions

In the general case of a grid composed of polygonal cells in two spatial dimensions with `p` sides and vertices, or a mixture of polygons where `p` is the maximum number of sides and vertices, the grid could have one, two or more dimensions, depending on how it is organised logically (e.g. as a 1-D list or a 2-D rectangular arrangement). The boundary variables for the auxiliary coordinate variables are dimensioned `(⋯,m,p)`, giving coordinates for the `p` vertices of each cell, where `(⋯,m)` are the dimensions of the auxiliary coordinate variables. If the cells are in a horizontal plane, the vertices must be traversed anticlockwise in the lon-lat plane as viewed from above. The starting vertex is not specified.

The case of a 2-D horizontal coordinate variables with 4-sided cells (Section 7.1.1, "Bounds for horizontal coordinate variables with four-sided cells") is a particular case, with `p=4` for boundary variables dimensioned `(n,m,p)`, where `n` and `m` are horizontal dimensions. See also Section 7.5, "Geometries" for conventions describing horizontal cells with more complicated geometry and topology.

7.1.4. Boundaries and Formula Terms

If a parametric coordinate variable with a `formula_terms` attribute (section 4.3.2) also has a `bounds` attribute, its boundary variable must have a `formula_terms` attribute too. In this case the same terms would appear in both (as specified in Appendix D), since the transformation from the parametric coordinate values to physical space is realized through the same formula. For any term that depends on the vertical dimension, however, the variable names appearing in the formula terms would differ from those found in the `formula_terms` attribute of the coordinate variable itself because the boundary variables for formula terms are two-dimensional while the formula terms themselves are one-dimensional.

Whenever a `formula_terms` attribute is attached to a boundary variable, the formula terms may additionally be identified using a second method: variables appearing in the vertical coordinates' `formula_terms` may be declared to be coordinate, scalar coordinate or auxiliary coordinate variables, and those coordinates may have `bounds` attributes that identify their boundary variables. In that case, the `bounds` attribute of a formula terms variable must be consistent with the `formula_terms` attribute of the boundary variable. Software digesting legacy datasets (constructed prior to version 1.7 of this standard) may have to rely in some cases on the first method of identifying the formula term variables and in other cases, on the second. Starting from version 1.7, however, the first method will be sufficient.

Example 7.3. Specifying `formula_terms` when a parametric coordinate variable has bounds.

```
float eta(eta) ;
   eta:long_name = "eta at full levels" ;
   eta:positive = "down" ;
   eta:standard_name = " atmosphere_hybrid_sigma_pressure_coordinate" ;
   eta:formula_terms = "a: A b: B ps: PS p0: P0" ;
   eta:bounds="eta_bnds" ;
 float eta_bnds(eta, 2) ;
   eta_bnds:formula_terms = "a: A_bnds b: B_bnds ps: PS p0: P0" ; // This
attribute is mandatory
 float A(eta) ;
   A:long_name = "'a' coefficient for vertical coordinate at full levels" ;
   A:units = "Pa" ;
   A:bounds = "A_bnds" ; // This attribute is included for the optional second
method
 float B(eta) ;
   B:long_name = "'b' coefficient for vertical coordinate at full levels" ;
   B:units = "1" ;
   B:bounds = "B_bnds" ; // This attribute is included for the optional second
method
 float A_bnds(eta, 2) ;
 float B_bnds(eta, 2) ;
 float PS(lat, lon) ;
   PS:units = "Pa" ;
 float P0 ;
   P0:units = "Pa" ;
 float temp(eta, lat, lon) ;
   temp:standard_name = "air_temperature" ;
   temp:units = "K";
   temp:coordinates = "A B" ; // This attribute is included for the optional
second method
```

7.2. Cell Measures

For some calculations, information is needed about the size, shape or location of the cells that cannot be deduced from the coordinates and bounds without special knowledge that a generic application cannot be expected to have. For instance, in computing the mean of several cell values, it is often appropriate to "weight" the values by area. When computing an area-mean each grid cell value is multiplied by the grid-cell area before summing, and then the sum is divided by the sum of the grid-cell areas. Area weights may also be needed to map data from one grid to another in such a way as to preserve the area mean of the field. The preservation of area-mean values while regridding may be essential, for example, when calculating surface heat fluxes in an atmospheric model with a grid that differs from the ocean model grid to which it is coupled.

In many cases the areas can be calculated from the cell bounds, but there are exceptions. Consider, for example, a spherical geodesic grid composed of contiguous, roughly hexagonal cells. The

vertices of the cells can be stored in the variable identified by the `bounds` attribute, but the cell perimeter is not uniquely defined by its vertices (because the vertices could, for example, be connected by straight lines, or, on a sphere, by lines following a great circle, or, in general, in some other way). Thus, given the cell vertices alone, it is generally impossible to calculate the area of a grid cell. This is why it may be necessary to store the grid-cell areas in addition to the cell vertices.

In other cases, the grid cell-volume might be needed and might not be easily calculated from the coordinate information. In ocean models, for example, it is not uncommon to find "partial" grid cells at the bottom of the ocean. In this case, rather than (or in addition to) indicating grid cell area, it may be necessary to indicate volume.

To indicate extra information about the spatial properties of a variable's grid cells, a `cell_measures` attribute may be defined for a variable. This is a string attribute comprising a list of blank-separated pairs of words of the form "`measure: name`". For the moment, "`area`" and "`volume`" are the only defined measures, but others may be supported in future. The "name" is the name of the variable containing the measure values, which we refer to as a "measure variable". The dimensions of a measure variable must be the same as or a subset of the dimensions of the variable to which it is related, but their order is not restricted, and with one exception: If a cell measure variable of a data variable that has been compressed by gathering (Section 8.2, "Lossless Compression by Gathering") does not span the compressed dimension, then its dimensions may be any subset of the data variable's uncompressed dimensions, i.e. any of the dimensions of the data variable except the compressed dimension, and any of the dimensions listed by the `compress` attribute of the compressed coordinate variable. In the case of area, for example, the field itself might be a function of longitude, latitude, and time, but the variable containing the area values would only include longitude and latitude dimensions (and the dimension order could be reversed, although this is not recommended). The variable must have a `units` attribute and may have other attributes such as a `standard_name`.

For rectangular longitude-latitude grids, the area of grid cells can be calculated from the bounds: the area of a cell is proportional to the product of the difference in the longitude bounds of the cell and the difference between the sine of each latitude bound of the cell. In this case supplying grid-cell areas via the `cell_measures` attribute is unnecessary because it may be assumed that applications can perform this calculation, using their own value for the radius of the Earth.

A variable referenced by `cell_measures` is not required to be present in the file containing the data variable. If the `cell_measures` variable is located in another file (an "external file"), rather than in the file where it is referenced, it must be listed in the `external_variables` attribute of the referencing file (Section 2.6.3).

Example 7.4. Cell areas for a spherical geodesic grid

```
dimensions:
  cell = 2562 ;  // number of grid cells
  time = 12 ;
  nv = 6 ;       // maximum number of cell vertices
variables:
  float PS(time,cell) ;
    PS:units = "Pa" ;
    PS:coordinates = "lon lat" ;
```

```
    PS:cell_measures = "area: cell_area" ;
  float lon(cell) ;
    lon:long_name = "longitude" ;
    lon:units = "degrees_east" ;
    lon:bounds="lon_vertices" ;
  float lat(cell) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;
    lat:bounds="lat_vertices" ;
  float time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1979-01-01 0:0:0" ;
  float cell_area(cell) ;
    cell_area:long_name = "area of grid cell" ;
    cell_area:standard_name="cell_area";
    cell_area:units = "m2"
  float lon_vertices(cell,nv) ;
  float lat_vertices(cell,nv) ;
```

7.3. Cell Methods

To describe the characteristic of a field that is represented by cell values, we define the `cell_methods` attribute of the variable. This is a string attribute comprising a list of blank-separated words of the form "*name: method*". Each "*name: method*" pair indicates that for an axis identified by *name*, the cell values representing the field have been determined or derived by the specified *method*. For example, if data values have been generated by computing time means, then this could be indicated with `cell_methods="t: mean"`, assuming here that the name of the time dimension variable is "t".

In the specification of this attribute, *name* can be a dimension of the variable, a scalar coordinate variable, a valid standard name, or the word "`area`". (See Section 7.3.4, "Cell methods when there are no coordinates" concerning the use of standard names in cell_methods.) The values of *method* should be selected from the list in Appendix E, Cell Methods, which includes `point`, `sum`, `mean`, among others. Case is not significant in the method name. Some methods (e.g., `variance`) imply a change of units of the variable, as is indicated in Appendix E, Cell Methods.

It must be remembered that the method applies only to the axis designated in `cell_methods` by *name*, and different methods may apply to other axes. If, for instance, a precipitation value in a longitude-latitude cell is given the method `maximum` for these axes, it means that it is the maximum within these spatial cells, and does not imply that it is also the maximum in time. Furthermore, it should be noted that if any *method* other than "`point`" is specified for a given axis, then `bounds` should also be provided for that axis (except for the relatively rare exceptions described in Section 7.3.4, "Cell methods when there are no coordinates").

The default interpretation for variables that do not have the `cell_methods` attribute specified depends on whether the quantity is extensive (which depends on the size of the cell) or intensive (which does not). Suppose, for example, the quantities "accumulated precipitation" and "precipitation rate" each have a time axis. A variable representing accumulated precipitation is extensive in time because it depends on the length of the time interval over which it is

accumulated. For correct interpretation, it therefore requires a time interval to be completely specified via a boundary variable (i.e., via a `bounds` attribute for the time axis). In this case the default interpretation is that the cell method is a sum over the specified time interval. This can be (optionally) indicated explicitly by setting the cell method to `sum`. A precipitation rate on the other hand is intensive in time and could equally well represent either an instantaneous value or a mean value over the time interval specified by the cell. In this case the default interpretation for the quantity would be "instantaneous" (which, optionally, can be indicated explicitly by setting the cell method to `point`). More often, however, cell values for intensive quantities are means, and this should be indicated explicitly by setting the cell method to `mean` and specifying the cell bounds.

Because the default interpretation for an intensive quantity differs from that of an extensive quantity and because this distinction may not be understood by some users of the data, it is recommended that every data variable include for each of its dimensions and each of its scalar coordinate variables the `cell_methods` information of interest (unless this information would not be meaningful). It is especially recommended that `cell_methods` be explicitly specified for each spatio-temporal dimension and each spatio-temporal scalar coordinate variable.

Example 7.5. Methods applied to a timeseries

Consider 12-hourly timeseries of pressure, temperature and precipitation from a number of stations, where pressure is measured instantaneously, maximum temperature for the preceding 12 hours is recorded, and precipitation is accumulated in a rain gauge. For a period of 48 hours from 6 a.m. on 19 April 1998, the data is structured as follows:

```
dimensions:
  time = UNLIMITED; // (5 currently)
  station = 10;
  nv = 2;
variables:
  float pressure(time,station);
    pressure:long_name = "pressure";
    pressure:units = "kPa";
    pressure:cell_methods = "time: point";
  float maxtemp(time,station);
    maxtemp:long_name = "temperature";
    maxtemp:units = "K";
    maxtemp:cell_methods = "time: maximum";
  float ppn(time,station);
    ppn:long_name = "depth of water-equivalent precipitation";
    ppn:units = "mm";
    ppn:cell_methods = "time: sum";
  double time(time);
    time:long_name = "time";
    time:units = "h since 1998-4-19 6:0:0 Z";
    time:bounds = "time_bnds";
  double time_bnds(time,nv);
data:
  time = 0., 12., 24., 36., 48.;
  time_bnds = -12.,0., 0.,12., 12.,24., 24.,36., 36.,48.;
```

Note that in this example the time axis values coincide with the end of each interval. It is sometimes desirable, however, to use the midpoint of intervals as coordinate values for variables that are representative of an interval. An application may simply obtain the midpoint values by making use of the boundary data in `time_bnds`.

7.3.1. Statistics for more than one axis

If more than one cell method is to be indicated, they should be arranged in the order they were applied. The left-most operation is assumed to have been applied first. Suppose, for example, that within each grid cell a quantity varies in both longitude and time and that these dimensions are named "lon" and "time", respectively. Then values representing the time-average of the zonal maximum are labeled `cell_methods="lon: maximum time: mean"` (i.e. find the largest value at each instant of time over all longitudes, then average these maxima over time); values of the zonal maximum of time-averages are labeled `cell_methods="time: mean lon: maximum"`. If the methods could have been applied in any order without affecting the outcome, they may be put in any order in the `cell_methods` attribute.

If a data value is representative of variation over a combination of axes, a single method should be prefixed by the names of all the dimensions involved (listed in any order, since in this case the order must be immaterial). Dimensions should be grouped in this way only if there is an essential difference from treating the dimensions individually. For instance, the standard deviation of topographic height within a longitude-latitude gridbox could have `cell_methods="lat: lon: standard_deviation"`. (Note also, that in accordance with the recommendation of the following paragraph, this could be equivalently and preferably indicated by `cell_methods="area: standard_deviation"`.) This is not the same as `cell_methods="lon: standard_deviation lat: standard_deviation"`, which would mean finding the standard deviation along each parallel of latitude within the zonal extent of the gridbox, and then the standard deviation of these values over latitude.

To indicate variation over horizontal area, it is recommended that instead of specifying the combination of horizontal dimensions, the special string "`area`" be used. The common case of an area-mean can thus be indicated by `cell_methods="area: mean"` (rather than, for example, "`lon: lat: mean`"). The horizontal coordinate variables to which "`area`" refers are in this case not explicitly indicated in `cell_methods` but can be identified, if necessary, from attributes attached to the coordinate variables, scalar coordinate variables, or auxiliary coordinate variables, as described in Chapter 4, Coordinate Types.

7.3.2. Recording the spacing of the original data and other information

To indicate more precisely how the cell method was applied, extra information may be included in parentheses `( )` at the end of the word list describing the method, after the operation and any `where`, `over` and `within` phrases. This information includes standardized and non-standardized parts. Currently the only standardized information is to provide the typical interval between the original data values to which the method was applied, in the situation where the present data values are statistically representative of original data values which had a finer spacing. The syntax is (`interval`: *value unit*), where *value* is a numerical value and *unit* is a string that can be recognized by UNIDATA's UDUNITS package [UDUNITS]. The *unit* will usually be dimensionally equivalent to the unit of the corresponding dimension, but this is not required (which allows, for example, the

interval for a standard deviation calculated from points evenly spaced in distance along a parallel to be reported in units of length even if the zonal coordinate of the cells is given in degrees). Recording the original interval is particularly important for standard deviations. For example, the standard deviation of daily values could be indicated by `cell_methods="time: standard_deviation (interval: 1 day)"` and of annual values by `cell_methods="time: standard_deviation (interval: 1 year)"`.

If the cell method applies to a combination of axes, they may have a common original interval e.g. `cell_methods="lat: lon: standard_deviation (interval: 10 km)"`. Alternatively, they may have separate intervals, which are matched to the names of axes by position e.g. `cell_methods="lat: lon: standard_deviation (interval: 0.1 degree_N interval: 0.2 degree_E)"`, in which 0.1 degree applies to latitude and 0.2 degree to longitude.

If there is both standardized and non-standardized information, the non-standardized follows the standardized information and the keyword `comment:`. If there is no standardized information, the keyword `comment:` should be omitted. For instance, an area-weighted mean over latitude could be indicated as `lat: mean (area-weighted)` or `lat: mean (interval: 1 degree_north comment: area-weighted)`.

A dimension of size one may be the result of "collapsing" an axis by some statistical operation, for instance by calculating a variance from time series data. We strongly recommend that dimensions of size one be retained (or scalar coordinate variables be defined) to enable documentation of the method (through the `cell_methods` attribute) and its domain (through the `bounds` attribute).

Example 7.6. Surface air temperature variance

The variance of the diurnal cycle on 1 January 1990 has been calculated from hourly instantaneous surface air temperature measurements. The time dimension of size one has been retained.

```
dimensions:
  lat=90;
  lon=180;
  time=1;
  nv=2;
variables:
  float TS_var(time,lat,lon);
    TS_var:long_name="surface air temperature variance"
    TS_var:units="K2";
    TS_var:cell_methods="time: variance (interval: 1 hr comment: sampled
instantaneously)";
  float time(time);
    time:units="days since 1990-01-01 00:00:00 Z";
    time:bounds="time_bnds";
  float time_bnds(time,nv);
data:
  time=.5;
  time_bnds=0.,1.;
```

Notice that a parenthesized comment in the `cell_methods` attribute provides the nature of the samples used to calculate the variance.

7.3.3. Statistics applying to portions of cells

By default, the statistical method indicated by `cell_methods` is assumed to have been evaluated over the entire horizontal area of the cell. Sometimes, however, it is useful to limit consideration to only a portion of a cell (e.g. a mean over the sea-ice area). Cell portions are referred to by means of standardised `area_type` strings, maintained in the area-type table, using one of two conventions.

The first convention is a method that can be used for the common case of a single area-type. In this case, the `cell_methods` attribute may include a string of the form "*name: method* `where` *type*". Here *name* could, for example, be `area` and *type* may be any of the standardised `area_type` strings. As an example, if the method were `mean` and the `area_type` were `sea_ice`, then the data would represent a mean over only the sea ice portion of the grid cell. If the data writer expects *type* to be interpreted as one of the standard `area_type` strings, then none of the variables in the netCDF file should be given a name identical to that of the string (because the second convention, described in the next paragraph, takes precedence).

The second convention is the more general. In this case, the `cell_methods` entry is of the form "*name: method* `where` *typevar*". Here *typevar* is a string-valued auxiliary coordinate variable or string-valued scalar coordinate variable (see Section 6.1, "Labels") with a `standard_name` of `area_type`. The variable *typevar* contains the name(s) of the selected portion(s) of the grid cell to which the *method* is applied. These name(s) must be a subset of the standardised `area_type` strings. This convention can accommodate cases in which a method is applied to more than one area type and the result is stored in a single data variable (with a dimension which ranges across the various area types). It provides a convenient way to store output from land surface models, for example, since they deal with many area types within each surface gridbox (e.g., `vegetation`, `bare_ground`, `snow`, etc.).

Example 7.7. Mean surface temperature over land and sensible heat flux averaged separately over land and sea.

```
dimensions:
  lat=73;
  lon=96;
  maxlen=20;
  ls=2;
variables:
  float surface_temperature(lat,lon);
    surface_temperature:cell_methods="area: mean where land";
  float surface_upward_sensible_heat_flux(ls,lat,lon);
    surface_upward_sensible_heat_flux:coordinates="land_sea";
    surface_upward_sensible_heat_flux:cell_methods="area: mean where land_sea";
  char land_sea(ls,maxlen);
    land_sea:standard_name="area_type";
data:
  land_sea="land","sea";
```

If the *method* is `mean`, various ways of calculating the mean can be distinguished in the `cell_methods` attribute with a string of the form "mean where *type1* [over *type2*]". Here, *type1* can be any of the possibilities allowed for *typevar* or *type* (as specified in the two paragraphs preceding above Example). The same options apply to *type2*, except it is not allowed to be the name of an auxiliary coordinate variable with a dimension greater than one (ignoring the possible dimension accommodating the maximum string length). A `cell_methods` attribute with a string of the form "mean where *type1* over *type2*" indicates the mean is calculated by summing over the *type1* portion of the cell and dividing by the area of the *type2* portion. In particular, a `cell_methods` string of the form "mean where all_area_types over *type2*" indicates the mean is calculated by summing over all types of area within the cell and dividing by the area of the *type2* portion. (Note that `all_area_types` is one of the valid strings permitted for a variable with the `standard_name` `area_type`.) If "over *type2*" is omitted, the mean is calculated by summing over the *type1* portion of the cell and dividing by the area of this portion.

Example 7.8. Thickness of sea-ice and snow on sea-ice averaged over sea area.

```
variables:
  float sea_ice_thickness(lat,lon);
    sea_ice_thickness:cell_methods="area: mean where sea_ice over sea";
    sea_ice_thickness:standard_name="sea_ice_thickness";
    sea_ice_thickness:units="m";
  float snow_thickness(lat,lon);
    snow_thickness:cell_methods="area: mean where sea_ice over sea";
   snow_thickness:standard_name="lwe_thickness_of_surface_snow_amount";
    snow_thickness:units="m";
```

In the case of sea-ice thickness, the phrase “where sea_ice” could be replaced by “where all_area_types” without changing the meaning since the integral of sea-ice thickness over all area types is obviously the same as the integral over the sea-ice area only. In the case of snow thickness, “where sea_ice” differs from “where all_area_types” because “where sea_ice” excludes snow on land from the average.

7.3.4. Cell methods when there are no coordinates

To provide an indication that a particular cell method is relevant to the data without having to provide a precise description of the corresponding cell, the "*name*" that appears in a "*name*: *method*" pair may be an appropriate `standard_name` (which identifies the dimension) or the string, "*area*" (rather than the name of a scalar coordinate variable or a dimension with a coordinate variable). This convention cannot be used, however, if the name of a dimension or scalar coordinate variable is identical to *name*. There are two situations where this convention is useful.

First, it allows one to provide some indication of the method when the cell coordinate range cannot be precisely defined. For example, a climatological mean might be based on any data that exists, and, in general, the data might not be available over the same time periods everywhere. In this case, the time range would not be well defined (because it would vary, depending on location), and it could not be precisely specified through a time dimension’s bounds. Nevertheless, useful

information can be conveyed by a `cell_methods` entry of "`time: mean`" (where `time`, it should be noted, is a valid `standard_name`). (As required by this convention, it is assumed here that for the data referred to by this `cell_methods` attribute, "time" is not a dimension or coordinate variable.)

Second, for a few special dimensions, this convention allows one to indicate (without explicitly defining the coordinates) that the method applies to the domain covering the entire permitted range of those dimensions. This is allowed only for longitude, latitude, and area (indicating a combination of horizontal coordinates). For longitude, the domain is indicated according to this provision by the string "longitude" (rather than the name of a longitude coordinate variable), and this implies that the method applies to all possible longitudes (i.e., from 0E to 360E). For latitude, the string "latitude" is used and implies the method applies to all possible latitudes (i.e., from 90S to 90N). For area, the string "area" is used and implies the method applies to the whole world.

In the second case if, in addition, the data variable has a dimension with a corresponding labeled axis that specifies a geographic region (Section 6.1.1, "Geographic Regions"), the implied range of longitude and latitude is the valid range for each specified region, or in the case of `area` the domain is the geographic region. For example, there could be a `cell_methods` entry of "`longitude: mean`", where `longitude` is *not* the name of a dimension or coordinate variable (but is one of the special cases given above). That would indicate a mean over all longitudes. Note, however, that if in addition the data variable had a scalar coordinate variable with a `standard_name` of `region` and a value of `atlantic_ocean`, it would indicate a mean over longitudes that lie within the Atlantic Ocean, not all longitudes.

We recommend that whenever possible, cell bounds should be supplied by giving the variable a dimension of size one and attaching bounds to the associated coordinate variable.

7.4. Climatological Statistics

Climatological statistics may be derived from corresponding portions of the annual cycle in a set of years, e.g., the average January temperatures in the climatology of 1961-1990, where the values are derived by averaging the 30 Januarys from the separate years. Portions of the climatological cycle are specified by references to dates within the calendar year. However, a calendar year is not a well-defined unit of time, because it differs between leap years and other years, and among calendars. Nonetheless for practical purposes we wish to compare statistics for months or seasons from different calendars, and to make climatologies from a mixture of leap years and other years. Hence we provide special conventions for indicating dates within the climatological year. Climatological statistics may also be derived from corresponding portions of a range of days, for instance the average temperature for each hour of the average day in April 1997. In addition the two concepts may be used at once, for instance to indicate not April 1997, but the average April of the five years 1995-1999.

Climatological variables have a climatological time axis. Like an ordinary time axis, a climatological time axis may have a dimension of unity (for example, a variable containing the January average temperatures for 1961-1990), but often it will have several elements (for example, a climatological time axis with a dimension of 12 for the climatological average temperatures in each month for 1961-1990, a dimension of 3 for the January mean temperatures for the three decades 1961-1970, 1971-1980, 1981-1990, or a dimension of 24 for the hours of an average day). Intervals of climatological time are conceptually different from ordinary time intervals; a given interval of

climatological time represents a set of subintervals which are not necessarily contiguous. To indicate this difference, a climatological time coordinate variable does not have a `bounds` attribute, instead it has a `climatology` attribute which names the climatological boundary variable. The climatological boundary variable must have dimensions (n,2), n being the dimension of the climatological time axis. The rules and recommendations for attributes of the climatological boundary variable are the same as those for boundary variables in general, as described in Section 7.1, "Cell Boundaries". Using the units and calendar of the time coordinate variable, element (i,0) of the climatology boundary variable specifies the beginning of the first subinterval and element (i,1) the end of the last subinterval used to evaluate the climatological statistics with index i in the time dimension. The time coordinates should be values that are representative of the climatological time intervals, such that an application which does not recognise climatological time will nonetheless be able to make a reasonable interpretation.

For compatibility with the COARDS standard, a climatological time coordinate in the default `standard` and `julian` calendars may be indicated by setting the datetime reference string in the time coordinate's `units` attribute to midnight at 0 `degrees_east` on 1 January in year 0 (i.e., `since 0-1-1`). This convention is deprecated because it does not provide any information about the intervals used to compute the climatology, and there may be inconsistencies among software packages in the interpretation of the time coordinates with a reference time of year 0. Use of year 0 for this purpose is impossible in all other calendars, because year 0 is a valid year.

A climatological axis may use different statistical methods to represent variation among years, within years and within days. For example, the average January temperature in a climatology is obtained by averaging both within years and over years. This is different from the average January-maximum temperature and the maximum January-average temperature. For the former, we first calculate the maximum temperature in each January, then average these maxima; for the latter, we first calculate the average temperature in each January, then find the largest one. As usual, the statistical operations are recorded in the `cell_methods` attribute, which may have two or three entries for the climatological time dimension.

Valid values of the `cell_methods` attribute must be in one of the forms from the following list. The intervals over which various statistical methods are applied are determined by decomposing the date and time specifications of the climatological time bounds of a cell, as recorded in the variable named by the `climatology` attribute. (The date and time specifications must be calculated from the time coordinates expressed in units of "time interval since reference date and time".) In the descriptions that follow we use the abbreviations *y*, *m*, *d*, *H*, *M*, and *S* for year, month, day, hour, minute, and second respectively. The suffix *0* indicates the earlier bound and *1* the latter.

time: method1 `within years` **time: method2** `over years`

method1 is applied to the time intervals (mdHMS0-mdHMS1) within individual years and *method2* is applied over the range of years (y0-y1).

time: method1 `within days` **time: method2** `over days`

method1 is applied to the time intervals (HMS0-HMS1) within individual days and *method2* is applied over the days in the interval (ymd0-ymd1).

time: method1 `within days` **time: method2** `over days` **time: method3** `over years`

method1 is applied to the time intervals (HMS0-HMS1) within individual days and *method2* is

applied over the days in the interval (md0-md1), and *method3* is applied over the range of years (y0-y1).

The methods which can be specified are those listed in Appendix E, Cell Methods and each entry in the `cell_methods` attribute may also, as usual, contain non-standardised information in parentheses after the method. For instance, a mean over ENSO years might be indicated by "`time: mean over years (ENSO years)`".

When considering intervals within years, if the earlier climatological time bound is later in the year than the later climatological time bound, it implies that the time intervals for the individual years run from each year across January 1 into the next year e.g. DJF intervals run from December 1 0:00 to March 1 0:00. Analogous situations arise for daily intervals running across midnight from one day to the next.

When considering intervals within days, if the earlier time of day is equal to the later time of day, then the method is applied to a full 24 hour day.

We have tried to make the examples in this section easier to understand by translating all time coordinate values to date and time formats. This is not currently valid CDL syntax.

Example 7.9. Climatological seasons

This example shows the metadata for the average seasonal-minimum temperature for the four standard climatological seasons MAM JJA SON DJF, made from data for March 1960 to February 1991.

```
dimensions:
  time=4;
  nv=2;
variables:
  float temperature(time,lat,lon);
    temperature:long_name="surface air temperature";
    temperature:cell_methods="time: minimum within years time: mean over years";
    temperature:units="K";
  double time(time);
    time:climatology="climatology_bounds";
    time:units="days since 1960-1-1";
  double climatology_bounds(time,nv);
data:  // time coordinates translated to datetime format
  time="1960-4-16", "1960-7-16", "1960-10-16", "1961-1-16" ;
  climatology_bounds="1960-3-1",  "1990-6-1",
                     "1960-6-1",  "1990-9-1",
                     "1960-9-1",  "1990-12-1",
                     "1960-12-1", "1991-3-1" ;
```

Example 7.10. Decadal averages for January

Average January precipitation totals are given for each of the decades 1961-1970, 1971-1980,

1981-1990.

```
dimensions:
  time=3;
  nv=2;
variables:
  float precipitation(time,lat,lon);
    precipitation:long_name="precipitation amount";
    precipitation:cell_methods="time: sum within years time: mean over years";
    precipitation:units="kg m-2";
  double time(time);
    time:climatology="climatology_bounds";
    time:units="days since 1901-1-1";
  double climatology_bounds(time,nv);
data:  // time coordinates translated to datetime format
  time="1965-1-15", "1975-1-15", "1985-1-15" ;
  climatology_bounds="1961-1-1", "1970-2-1",
                     "1971-1-1", "1980-2-1",
                     "1981-1-1", "1990-2-1" ;
```

Example 7.11. Temperature for each hour of the average day

Hourly average temperatures are given for April 1997.

```
dimensions:
  time=24;
  nv=2;
variables:
  float temperature(time,lat,lon);
    temperature:long_name="surface air temperature";
    temperature:cell_methods="time: mean within days time: mean over days";
    temperature:units="K";
  double time(time);
    time:climatology="climatology_bounds";
    time:units="hours since 1997-4-1";
  double climatology_bounds(time,nv);
data:  // time coordinates translated to datetime format
  time="1997-4-1 0:30", "1997-4-1 1:30", ... "1997-4-1 23:30" ;
  climatology_bounds="1997-4-1 0:00",  "1997-4-30 1:00",
                     "1997-4-1 1:00",  "1997-4-30 2:00",
                      ...
                      "1997-4-1 23:00", "1997-5-1 0:00" ;
```

Example 7.12. Extreme statistics and spell-lengths

Number of frost days during NH winter 2007-2008, and maximum length of spells of consecutive frost days. A "frost day" is defined as one during which the minimum temperature

falls below freezing point (0 degC). This is described as a climatological statistic, in which the minimum temperature is first calculated within each day, and then the number of days or spell lengths meeting the specified condition are evaluated. In this operation, the standard name is also changed; the original data are `air_temperature`.

```
variables:
  float n1(lat,lon);
    n1:standard_name="number_of_days_with_air_temperature_below_threshold";
    n1:coordinates="threshold time";
    n1:cell_methods="time: minimum within days time: sum over days";
  float n2(lat,lon);
    n2:standard_name="spell_length_of_days_with_air_temperature_below_threshold";
    n2:coordinates="threshold time";
    n2:cell_methods="time: minimum within days time: maximum over days";
  float threshold;
    threshold:standard_name="air_temperature";
    threshold:units="degC";
  double time;
    time:climatology="climatology_bounds";
    time:units="days since 2000-6-1";
  double climatology_bounds(time,nv);
data: // time coordinates translated to datetime format
  time="2008-1-16 6:00";
  climatology_bounds="2007-12-1 6:00", "2008-3-1 6:00";
  threshold=0.;
```

Example 7.13. Temperature for each hour of the typical climatological day

This is a modified version of the previous example, "Temperature for each hour of the average day". It now applies to April from a 1961-1990 climatology.

```
variables:
  float temperature(time,lat,lon);
    temperature:long_name="surface air temperature";
    temperature:cell_methods="time: mean within days time: mean over days time:
mean over years";
    temperature:units="K";
  double time(time);
    time:climatology="climatology_bounds";
    time:units="days since 1961-1-1";
  double climatology_bounds(time,nv);
data:  // time coordinates translated to datetime format
  time="1961-4-1 0:30", "1961-4-1 1:30", ..., "1961-4-1 23:30" ;
  climatology_bounds="1961-4-1 0:00", "1990-4-30 1:00",
                     "1961-4-1 1:00", "1990-4-30 2:00",
                     ...
                     "1961-4-1 23:00", "1990-5-1 0:00" ;
```

Example 7.14. Monthly-maximum daily precipitation totals

Maximum of daily precipitation amounts for each of the three months June, July and August 2000 are given. The first daily total applies to 6 a.m. on 1 June to 6 a.m. on 2 June, the 30th from 6 a.m. on 30 June to 6 a.m. on 1 July. The maximum of these 30 values is stored under time index 0 in the precipitation array.

```
dimensions:
  time=3;
  nv=2;
variables:
  float precipitation(time,lat,lon);
    precipitation:long_name="Accumulated precipitation";
    precipitation:cell_methods="time: sum within days time: maximum over days";
    precipitation:units="kg";
  double time(time);
    time:climatology="climatology_bounds";
    time:units="days since 2000-6-1";
  double climatology_bounds(time,nv);
data:  // time coordinates translated to datetime format
  time="2000-6-16", "2000-7-16", "2000-8-16" ;
  climatology_bounds="2000-6-1 6:00:00", "2000-7-1 6:00:00",
                     "2000-7-1 6:00:00", "2000-8-1 6:00:00",
                     "2000-8-1 6:00:00", "2000-9-1 6:00:00" ;
```

7.5. Geometries

For many geospatial applications, data values are associated with a geometry, which is a spatial representation of a real-world feature, for instance a time-series of areal average precipitation over a watershed. Polygonal cells with an arbitrary number of vertices can be described using Section 7.1, "Cell Boundaries", but in that case every cell must have the same number of vertices and must be a single polygon ring. In contrast, each geometry may have a different number of nodes, the geometries may be lines (as alternatives to points and polygons), and they may be *multipart*, i.e., include several disjoint parts. While line and point geometries don't describe an interval along a dimension as the traditional cell bounds described above do, they do describe the extent of a geometry or real-world feature so are included in this section. The approach described here specifies how to encode such geometries following the pattern in **9.3.3 Contiguous ragged array representation** and attach them to variables in a way that is consistent with the cell bounds approach.

All geometries are made up of one or more nodes. The geometry type specifies the set of topological assumptions to be applied to relate the nodes (see Table 7.1). For example, multipoint and line geometries are nearly the same except nodes are interpreted as being connected for lines. Lines and polygons are also nearly the same except that the first and last nodes are assumed to be connected for polygons. Note that CF does not require the first and last node to be identical but allows them to be coincident if desired. Polygons that have holes, such as waterbodies in a land unit, are encoded as a collection of polygon ring parts, each identified as *exterior* or *interior*

polygons. Multipart geometries, such as multiple lines representing the same river or multiple islands representing the same jurisdiction, are encoded as collections of unconnected points, lines, or polygons that are logically grouped into a single geometry.

Any data variable can be given a `geometry` attribute that indicates the geometry for the quantity held in the variable. One of the dimensions of the data variable must be the number of geometries to which the data applies. As shown in Example 7.15, if the data variable has a discrete sampling geometry, the number of geometries is the length of the instance dimension (Section 9.2).

Example 7.15. Timeseries with geometry.

```
dimensions:
  instance = 2 ;
  node = 5 ;
  time = 4 ;
variables:
  int time(time) ;
    time:units = "days since 2000-01-01" ;
  double lat(instance) ;
    lat:units = "degrees_north" ;
    lat:standard_name = "latitude" ;
    lat:nodes = "y" ;
  double lon(instance) ;
    lon:units = "degrees_east" ;
    lon:standard_name = "longitude" ;
    lon:nodes = "x" ;
  int datum ;
    datum:grid_mapping_name = "latitude_longitude" ;
    datum:longitude_of_prime_meridian = 0.0 ;
    datum:semi_major_axis = 6378137.0 ;
    datum:inverse_flattening = 298.257223563 ;
  int geometry_container ;
    geometry_container:geometry_type = "line" ;
    geometry_container:node_count = "node_count" ;
    geometry_container:node_coordinates = "x y" ;
  int node_count(instance) ;
  double x(node) ;
    x:units = "degrees_east" ;
    x:standard_name = "longitude" ;
    x:axis = "X" ;
  double y(node) ;
    y:units = "degrees_north" ;
    y:standard_name = "latitude" ;
    y:axis = "Y" ;
  double someData(instance, time) ;
    someData:coordinates = "time lat lon" ;
    someData:grid_mapping = "datum" ;
    someData:geometry = "geometry_container" ;
// global attributes:
  :featureType = "timeSeries" ;
```

```
data:
  time = 1, 2, 3, 4 ;
  lat = 30, 50 ;
  lon = 10, 60 ;
  someData =
    1, 2, 3, 4,
    1, 2, 3, 4 ;
  node_count = 3, 2 ;
  x = 30, 10, 40, 50, 50 ;
  y = 10, 30, 40, 60, 50 ;
```

The time series variable, someData, is associated with line geometries via the geometry attribute. The first line geometry is comprised of three nodes, while the second has two nodes. Client applications unaware of CF geometries can fall back to the lat and lon variables to locate feature instances in space. In this example, lat and lon coordinates are identical to the first node in each line geometry, though any representative point could be used.

A *geometry container* variable acts as a container for attributes that describe a set of geometries. The `geometry` attribute of the data variable contains the name of a geometry container variable. The geometry container variable must hold `geometry_type` and `node_coordinates` attributes. The `grid_mapping` and `coordinates` attributes can be carried by the geometry container variable provided they are also carried by the data variables associated with the container.

The `geometry_type` attribute indicates the type of geometry present. Its allowable values are: *point*, *line*, *polygon*. Multipart geometries are allowed for all three geometry types. For example, polygon geometries could include single part geometries like the State of Colorado and multipart geometries like the State of Hawaii.

The `node_coordinates` attribute contains the blank-separated names of the variables that contain geometry node coordinates (one variable for each spatial dimension). The geometry node coordinate variables must each have an `axis` attribute whose allowable values are *X*, *Y*, and *Z*.

If a `coordinates` attribute is carried by the geometry container variable or its parent data variable, then those coordinate variables that have a meaningful correspondence with node coordinates are indicated as such by a `nodes` attribute that names the corresponding node coordinates, but only if the `grid_mapping` associated with the geometry node variables is the same as that of the coordinate variables. If a different grid mapping is used, then the provided coordinates must not have the `nodes` attribute.

Whether linked to normal CF space-time coordinates with a `nodes` attribute or not, inclusion of such coordinates is recommended to maintain backward compatibility with software that has not implemented geometry capabilities.

The geometry node coordinate variables must all have the same single dimension, which is the total number of nodes in all the geometries. The nodes must be stored consecutively for each geometry and in the order of the geometries, and within each multipart geometry the nodes must be stored consecutively for each part and in the order of the parts. Polygon exterior rings must be stored before any interior rings they may contain. Nodes for polygon exterior rings must be ordered using

the right-hand rule, e.g., anticlockwise in the lon-lat plane as viewed from above. Polygon interior rings must be in clockwise order. They are put in opposite orders to facilitate calculation of area and consistency with the typical implementation pattern.

When more than one geometry instance is present, the geometry container variable must have a `node_count` attribute that contains the name of a variable indicating the count of nodes per geometry. The node count is the total number of nodes in all the parts. The exception is when all geometries are single part point geometries, in which case a node count is not needed since each geometry contains a single node. However in that case, the dimension of the node coordinate variables must be one of the dimensions of the data variable (because it serves also as the instance dimension for geometries).

For multipart *lines*, multipart *polygons*, and *polygons* with holes, the geometry container variable must have a `part_node_count` attribute that indicates a variable of the count of nodes per geometry part. Note that because multipoint geometries always have a single node per part, the `part_node_count` is not required for *point* geometry types. The single dimension of the part node count variable must equal the total number of parts in all the geometries.

For *polygon* geometries with holes, the geometry container variable must have an `interior_ring` attribute that contains the name of a variable that indicates if the polygon parts are interior rings (i.e., holes) or not. This interior ring variable must contain the value 0 to indicate an exterior ring polygon and 1 to indicate an interior ring polygon. The single dimension of the interior ring variable must be the same dimension as that of the part node count variable. The geometry types included in this convention are listed in Table 7.1.

Table 7.1. Dimensionality, description, and additional required attributes for geometry_types.

geometry_type	**Dimensionality**	**Description of Geometry Instance**	**Additional required attributes on geometry container variable**
point	0	A collection of one or more points, where a point is a single location in space	node_count (if multipart geometries are present)
line	1	A collection of one or more lines, where a line is an ordered set of data points connected by linearly interpolating between points	node_count, part_node_count (if multipart geometries are present)

geometry_type	Dimensionality	Description of Geometry Instance	Additional required attributes on geometry container variable
polygon	2	A collection of one or more polygons, where a polygon is a planar surface comprised of an exterior ring and zero or more interior rings (i.e., holes), where a ring is a closed line (i.e., the last point in the line is assumed to be connected to the first point)	node_count, part_node_count (if holes or multipart geometries are present), interior_ring (if holes are present)

Example 7.16. Polygons with holes

This example demonstrates all potential attributes and variables for encoding geometries.

```
dimensions:
  node = 12 ;
  instance = 2 ;
  part = 4 ;
  time = 4 ;
variables:
  int time(time) ;
    time:units = "days since 2000-01-01" ;
  double x(node) ;
    x:units = "degrees_east" ;
    x:standard_name = "longitude" ;
    x:axis = "X" ;
  double y(node) ;
    y:units = "degrees_north" ;
    y:standard_name = "latitude" ;
    y:axis = "Y" ;
  double lat(instance) ;
    lat:units = "degrees_north" ;
    lat:standard_name = "latitude" ;
    lat:nodes = "y" ;
  double lon(instance) ;
    lon:units = "degrees_east" ;
    lon:standard_name = "longitude" ;
    lon:nodes = "x" ;
  float geometry_container ;
    geometry_container:geometry_type = "polygon" ;
    geometry_container:node_count = "node_count" ;
    geometry_container:node_coordinates = "x y" ;
```

```
    geometry_container:grid_mapping = "datum" ;
    geometry_container:coordinates = "lat lon" ;
    geometry_container:part_node_count = "part_node_count" ;
    geometry_container:interior_ring = "interior_ring" ;
  int node_count(instance) ;
  int part_node_count(part) ;
  int interior_ring(part) ;
  float datum ;
    datum:grid_mapping_name = "latitude_longitude" ;
    datum:semi_major_axis = 6378137. ;
    datum:inverse_flattening = 298.257223563 ;
    datum:longitude_of_prime_meridian = 0. ;
  double someData(instance, time) ;
    someData:coordinates = "time lat lon" ;
    someData:grid_mapping = "datum" ;
    someData:geometry = "geometry_container" ;
// global attributes:
  :featureType = "timeSeries" ;
data:
 time = 1, 2, 3, 4 ;
 x = 20, 10, 0, 5, 10, 15, 20, 10, 0, 50, 40, 30 ;
 y = 0, 15, 0, 5, 10, 5, 20, 35, 20, 0, 15, 0 ;
 lat = 25, 7 ;
 lon = 10, 40 ;
 node_count = 9, 3 ;
 part_node_count = 3, 3, 3, 3 ;
 interior_ring = 0, 1, 0, 0 ;
 someData =
   1, 2, 3, 4,
   1, 2, 3, 4 ;
```

Chapter 8. Reduction of Dataset Size

There are three methods for reducing dataset size: packing, lossless compression, and lossy compression. By packing we mean altering the data in a way that reduces its precision (but has no other effect on accuracy). By lossless compression we mean techniques that store the data more efficiently and result in no loss of precision or accuracy. By lossy compression we mean techniques that either store the data more efficiently and retain its precision but result in some loss in accuracy, or techniques that intentionally reduce data precision to improve the efficiency of subsequent lossless compression.

Lossless compression only works in certain circumstances, e.g., when a variable contains a significant amount of missing or repeated data values. In this case it is possible to make use of standard utilities, e.g., UNIX `compress` or GNU `gzip`, to compress the entire file after it has been written. In this section we offer an alternative compression method that is applied on a variable by variable basis. This has the advantage that only one variable need be uncompressed at a given time. The disadvantage is that generic utilities that don't recognize the CF conventions will not be able to operate on compressed variables.

8.1. Packed Data

At the current time the netCDF interface does not provide for packing data. However a simple packing may be achieved through the use of the optional [NUG] defined attributes `scale_factor` and `add_offset`. After the data values of a variable have been read, they are to be multiplied by the `scale_factor`, and have `add_offset` added to them. If both attributes are present, the data are scaled before the offset is added. When scaled data are written, the application should first subtract the offset and then divide by the scale factor. The units of a variable should be representative of the unpacked data.

This standard is more restrictive than the [NUG] with respect to the use of the `scale_factor` and `add_offset` attributes; ambiguities and precision problems related to data type conversions are resolved by these restrictions.

When packed data is written, the `scale_factor` and `add_offset` attributes must be of the same type as the unpacked data, which must be either `float` or `double`. Data of type `float` must be packed into one of these types: `byte`, `unsigned byte`, `short`, `unsigned short`. Data of type `double` must be packed into one of these types: `byte`, `unsigned byte`, `short`, `unsigned short`, `int`, `unsigned int`.

When packed data is read, it should be unpacked to the type of the `scale_factor` and `add_offset` attributes, which must have the same type if both are present. For guidance only, we suggest that packed data which does not conform to the rules of this section regarding the types of the data variable and attributes should be unpacked to `double` type, in order to minimise the risk of loss of precision.

When data to be packed contains missing values the attributes that indicate missing values (`_FillValue`, `valid_min`, `valid_max`, `valid_range`) must be of the same data type as the packed data. See Section 2.5.1, "Missing data, valid and actual range of data" for a discussion of how applications should treat variables that have attributes indicating both missing values and transformations defined by a scale and/or offset.

8.2. Lossless Compression by Gathering

To save space in the netCDF file, it may be desirable to eliminate points from data arrays that are invariably missing. Such a compression can operate over one or more adjacent axes, and is accomplished with reference to a list of the points to be stored. The list is constructed by considering a mask array that only includes the axes to be compressed, and then mapping this array onto one dimension without reordering. The list is the set of indices in this one-dimensional mask of the required points. In the compressed array, the axes to be compressed are all replaced by a single axis, whose dimension is the number of wanted points. The wanted points appear along this dimension in the same order they appear in the uncompressed array, with the unwanted points skipped over. Compression and uncompression are executed by looping over the list.

The list is stored as the coordinate variable for the compressed axis of the data variable. Thus, the list variable and its dimension have the same name. If any auxiliary coordinate variable has all the dimensions to be compressed, adjacent and in the same order as in the data variable, and if the auxiliary coordinate variable has missing data at all the points which are to be eliminated from the data variable, then the affected dimensions can optionally be replaced by the list dimension for the auxiliary coordinate variable just as for the data variable. The list variable has a string attribute `compress`, *containing a blank-separated list of the dimensions which were affected by the compression in the order of the CDL declaration of the uncompressed array*. The presence of this attribute identifies the list variable as such. The list, the original dimensions and coordinate variables (including boundary variables), and the compressed variables with all the attributes of the uncompressed variables are written to the netCDF file. The uncompressed variables can be reconstituted exactly as they were using this information. The list variable must not have an associated boundary variable.

Example 8.1. Horizontal compression of a three-dimensional array

We eliminate sea points at all depths in a longitude-latitude-depth array of soil temperatures. In this case, only the longitude and latitude axes would be affected by the compression. We construct a list `landpoint(landpoint)` containing the indices of land points.

```
dimensions:
  lat=73;
  lon=96;
  landpoint=2381;
  depth=4;
variables:
  int landpoint(landpoint);
    landpoint:compress="lat lon";
  float landsoilt(depth,landpoint);
    landsoilt:long_name="soil temperature";
    landsoilt:units="K";
  float depth(depth);
  float lat(lat);
  float lon(lon);
data:
  landpoint=363, 364, 365, ...;
```

Since `landpoint(0)=363`, for instance, we know that `landsoilt(*,0)` maps on to point 363 of the original data with dimensions `(lat,lon)`. This corresponds to indices `(3,75)`, i.e., `363 = 3*96 + 75`.

Example 8.2. Compression of a three-dimensional field

We compress a longitude-latitude-depth field of ocean salinity by eliminating points below the sea-floor. In this case, all three dimensions are affected by the compression, since there are successively fewer active ocean points at increasing depths.

```
variables:
  float salinity(time,oceanpoint);
  int oceanpoint(oceanpoint);
    oceanpoint:compress="depth lat lon";
  float depth(depth);
  float lat(lat);
  float lon(lon);
  double time(time);
```

This information implies that the salinity field should be uncompressed to an array with dimensions `(depth,lat,lon)`.

In A single timeseries with time-varying deviations from a nominal point spatial location, two auxiliary coordinate variables are compressed as described in this section, although their data variable is not.

8.3. Lossy Compression by Coordinate Subsampling

For some applications the coordinates of a data variable can require considerably more storage than the data itself. Space may be saved in the netCDF file by storing a subsample of the coordinates that describe the data. The uncompressed coordinate and auxiliary coordinate variables can be reconstituted by interpolation, from the subsampled coordinate values to the domain of the data (i.e. the target domain). This process will likely result in a loss in accuracy (as opposed to precision) in the uncompressed variables, due to rounding and approximation errors in the interpolation calculations, but it is assumed that these errors will be small enough to not be of concern to users of the uncompressed dataset. The creator of the compressed dataset can control the accuracy of the reconstituted coordinates through the degree of subsampling and the choice of interpolation method, see Appendix J, Coordinate Interpolation Methods.

The subsampled coordinates are called *tie points* and are stored in *tie point coordinate variables*.

In addition to the tie point coordinate variables themselves, metadata defining the coordinate interpolation method is stored in attributes of the data variable and of the associated *interpolation variable*. The partitioning of metadata between the data variable and the interpolation variable has been designed to minimise redundancy and maximise the reusability of the interpolation variable within a dataset.

The metadata that define the interpolation formula and its inputs are complete, so that the results of the coordinate reconstitution process are well defined and of a predictable accuracy.

8.3.1. Tie Points and Interpolation Subareas

Reconstitution of the uncompressed coordinate and auxiliary coordinate variables is based on interpolation. To accomplish this, the target domain is segmented into smaller *interpolation subareas*, for each of which the interpolation method is applied independently. For one-dimensional interpolation, an interpolation subarea is defined by two tie points, one at each end of the interpolation subarea; for two-dimensional interpolation, an interpolation subarea is defined by four tie points, one at each corner of a rectangular area aligned with the domain axes; etc. For the reconstitution of the uncompressed coordinate and auxiliary coordinate variables within an interpolation subarea, the interpolation method is permitted to access its defining tie points, and no others.

As an interpolation method relies on the regularity and continuity of the coordinate values within each interpolation subarea, special attention must be given to the case when uncompressed coordinates contain discontinuities. A discontinuity could be an overlap or a gap in the coordinates' coverage, or a change in cell size or cell alignment. As an example, such discontinuities are common in remote sensing data and may be caused by combinations of the instrument scan motion, the motion of the sensor platform and changes in the instrument scan mode. When discontinuities are present, the domain is first divided into multiple *continuous areas*, each of which is free of discontinuities. When no discontinuities are present, the whole domain is a single continuous area. Following this step, each continuous area is segmented into interpolation subareas. The processes of generating interpolation subareas for a domain without discontinuities and for a domain with discontinuities is illustrated in Figure 8.1, and described in more detail in Appendix J, Coordinate Interpolation Methods.

For each *interpolated dimension*, i.e. a target domain dimension for which coordinate interpolation is required, the locations of the tie point coordinates are defined by a corresponding *tie point index variable*, which also indicates the locations of the continuous areas (Section 8.3.7, "Tie Point Index Mapping").

The interpolation subareas within a continuous area do not overlap, ensuring that each coordinate of an interpolated dimension is computed from a unique interpolation subarea. These interpolation subareas, however, share the tie point coordinates that define their common boundaries. Such a shared tie point coordinate can only be located in one of a pair of adjacent interpolation subareas, which is always the first of the pair in index space. For instance, in Figure 8.1, the interpolation subarea labelled `(0,0)` contains all four of its tie point coordinates, and the interpolation subarea `(0,1)` only contains two of them. When applied for a given interpolation subarea, interpolation methods (such as those described in Appendix J, Coordinate Interpolation Methods) must ensure that reconstituted coordinate points are only generated inside the interpolation subarea being processed, even if some of the tie point coordinates lie outside of that interpolation subarea.

Adjacent interpolation subareas that are in different continuous areas never share tie point coordinates, as consequence of the grid discontinuity between them. This results in a different number of tie point coordinates in the two cases shown in Figure 8.1.

For each interpolated dimension, the number of interpolation subareas is equal to the number of

tie points minus the number of continuous areas.

Tie point coordinate variables for both coordinate and auxiliary coordinate variables must be defined as numeric data types and are not allowed to have missing values.

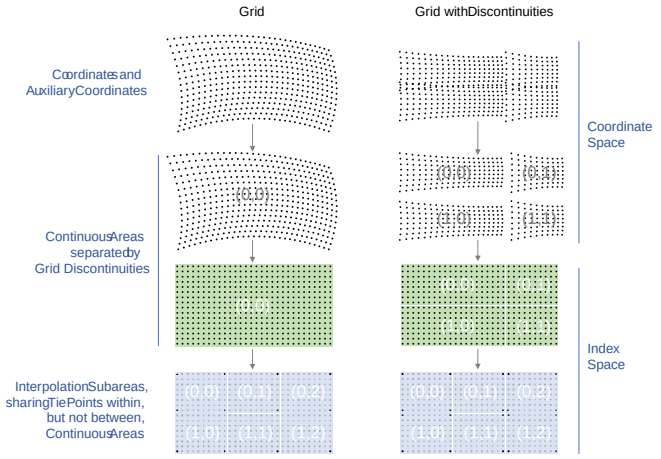


Figure 8.1. Process for generating the interpolation subareas for a grid without discontinuities and for a grid with discontinuities.

8.3.2. Coordinate Interpolation Attribute

To indicate that coordinate interpolation is required, a `coordinate_interpolation` attribute must be defined for a data variable. This is a string attribute that both identifies the tie point coordinate variables, and maps non-overlapping subsets of them to their corresponding interpolation variables. It is a blank-separated list of words of the form "*tie_point_coordinate_variable: [tie_point_coordinate_variable: ...] interpolation_variable [tie_point_coordinate_variable: [tie_point_coordinate_variable: ...] interpolation_variable ...]*". For example, to specify that the tie point coordinate variables `lat` and `lon` are to be interpolated according to the interpolation variable `bi_linear` could be indicated with `lat: lon: bi_linear`.

8.3.3. Interpolation Variable

The method used to uncompress the tie point coordinate variables is described by an interpolation variable that acts as a container for the attributes that define the interpolation technique and the parameters that should be used. The variable should be a scalar (i.e. it has no dimensions) of arbitrary type, and the value of its single element is immaterial.

The interpolation method must be identified in one of two ways. Either by the `interpolation_name` attribute, which takes a string value that contains the method's name, or else by the `interpolation_description` attribute, which takes a string value that contains a non-standardized description of the method. These attributes must not be both set.

The valid values of `interpolation_name` are given in Appendix J, Coordinate Interpolation Methods. This appendix describes the interpolation technique for each method, and optional interpolation variable attributes for configuring the interpolation process.

If a standardized interpolation name is not given, the interpolation variable must have an `interpolation_description` attribute defined instead, containing a description of the non-

standardised interpolation (in a similar manner to a long name being used instead of a standard name). This description is free text that can take any form (including fully qualified URLs, for example). Whilst it is recommended that a standardised interpolation is provided, the alternative is provided to promote interoperability in cases where a well defined user community needs to use sophisticated interpolation techniques that may also be under development.

The definition of the interpolation method, however it is specified, may include instructions to treat groups of physically related coordinates simultaneously, if such tie points are present. For example, there are cases where longitudes cannot be interpolated without considering the corresponding latitudes. It is up to the interpolation description to describe how such coordinates are to be identified (e.g. it may be that such tie point coordinate variables require particular units or standard names).

Note that the interpolation method is always applied on a per interpolation subarea basis, for which the construction of the uncompressed coordinates may only access those tie points that define the extent of the of the interpolation subarea.

In addition to the `interpolation_name` and `interpolation_description` attributes described in this section, further attributes of the interpolation variable are described in Section 8.3.5, "Tie Point Mapping Attribute" and Section 8.3.8, "Interpolation Parameters", Section 8.3.9, "Interpolation of Cell Boundaries" and Section 8.3.10, "Interpolation Method Implementation".

8.3.4. Subsampled, Interpolated and Non-Interpolated Dimensions

For each interpolation variable identified in the `coordinate_interpolation` attribute, all of the associated tie point coordinate variables must share the same set of one or more dimensions. This set of dimensions must correspond to the set of dimensions of the uncompressed coordinate or auxiliary coordinate variables, such that each of these dimensions must be either the uncompressed dimension itself, or a dimension that is to be interpolated to the uncompressed dimension.

Dimensions of the tie point coordinate variable which are to be interpolated are called *subsampled dimensions*, and the corresponding data variable dimensions are called *interpolated dimensions*, while those for which no interpolation is required, being the same in the data variable and the tie point coordinate variable, are called *non-interpolated dimensions*. The dimensions of a tie point coordinate variable must contain at least one subsampled dimension, for each of which the corresponding interpolated dimension cannot be included.

The size of a subsampled dimension will be less than the size of the corresponding interpolated dimension. For example, if the interpolated dimensions are `xc = 30` and `yc = 10`, interpolation could be applied in both of these dimensions, based on tie point variables of the dimensions `tp_xc = 4` and `tp_yc = 2`. Here, `tp_xc` is the subsampled dimension related to the interpolated dimension `xc`, and `tp_yc` is the subsampled dimension related to the interpolated dimension `yc`.

The presence of non-interpolated dimensions in the tie point coordinate variable impacts the interpolation process in that there must be a separate application of the interpolation method for each combination of indices of the non-interpolated dimensions. For example, if `xc = 30` is an interpolated dimension and `yc = 10` is a non-interpolated dimension, interpolation could be applied in the `xc` dimension only, based on tie point variables that have the subsampled dimension `tp_xc =`

4 and the non-interpolated dimension `yc = 10`. The interpolation in the `xc` dimension would then be repeated for each of the 10 indices of the `yc` non-interpolated dimension.

8.3.5. Tie Point Mapping Attribute

The `tie_point_mapping` attribute provides mapping at two levels. It associates interpolated dimensions with the corresponding subsampled dimensions, and for each of these sets of corresponding dimensions, it associates index values of the interpolated dimension with index values of the subsampled dimension, thereby uniquely associating the tie points with their corresponding location in the target domain.

The mappings are stored in the interpolation variable's `tie_point_mapping` attribute that contains a blank-separated list of words of the form *"interpolated_dimension: tie_point_index_variable subsampled_dimension [interpolation_subarea_dimension] [interpolated_dimension: ...]"*, the details of which are described in the following two sections.

8.3.6. Tie Point Dimension Mapping

The `tie_point_mapping` attribute defined above associates each interpolated dimension with its corresponding subsampled dimension and, if required, its corresponding *interpolation subarea dimension* that defines the number of interpolation subareas which partition the interpolated dimension. It is only required to associate an interpolated dimension to an interpolation subarea dimension in the case that the interpolation subarea dimension is spanned by an interpolation parameter variable, as described in Section 8.3.8, "Interpolation Parameters". If an interpolation subarea dimension is provided, then it must be the second of the two named dimensions following the tie point index variable.

Note that the size of an interpolation subarea dimension is, by definition, the size of the corresponding subsampled dimension minus the number of continuous areas.

An overview of the different dimensions for coordinate interpolation is shown in Figure 8.2.

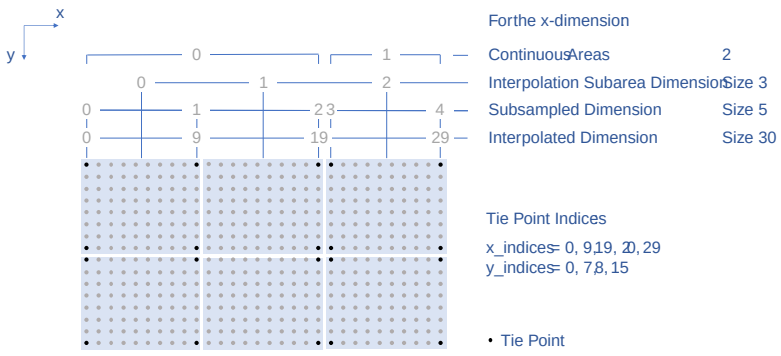


Figure 8.2. Overview of the different dimensions for coordinate interpolation.

8.3.7. Tie Point Index Mapping

The `tie_point_mapping` attribute defined in Section 8.3.5, "Tie Point Mapping Attribute" identifies for each subsampled dimension a tie point index variable. The tie point index variable defines the relationship between the indices of the subsampled dimension and the indices of its corresponding interpolated dimension.

A tie point index variable is a one-dimensional integer variable that must span the subsampled dimension. Each tie point index variable value is a zero-based index of the related interpolated dimension which maps an element of that interpolated dimension to the corresponding location in the subsampled dimension.

The tie point index values must be strictly monotonically increasing. The location in index space of a continuous area boundary that relates to a grid discontinuity (Section 8.3.1, "Tie Points and Interpolation Subareas") is indicated by a pair of adjacent tie point index values differing by one. In this case, each tie point index of the pair defines a boundary of a different continuous area. As a consequence, any pair of tie point index values that defines an extent of an interpolation subarea must differ by two or more, i.e. in general, an interpolation subarea spans at least two points in each of its interpolated dimensions. Interpolation subareas that are the first in index space of a continuous area, in one or more of the subsampled dimensions are, however, special. These interpolation subareas contain tie points at both of the subarea boundaries with respect to those subsampled dimensions and so must span at least three points in the corresponding interpolated dimensions (see Figure 8.1).

For instance, in example Two-dimensional tie point interpolation the tie point coordinate variables represent a subset of the target domain and the tie point index variable `int x_indices(tp_xc)` contains the indices `x_indices = 0, 9, 19, 29` that identify the location in the interpolated dimension `xc` of size 30. The corresponding **`tie_point_mapping`** attribute of the interpolation variable is `xc: x_indices tp_xc yc: y_indices tp_yc`.

Example 8.3. Two-dimensional tie point interpolation

```
dimensions:
  xc = 30;
  yc = 10;
  tp_xc = 4 ;
  tp_yc = 2 ;

variables:
  // Data variable
  float Temperature(yc, xc) ;
    Temperature:standard_name = "air_temperature" ;
    Temperature:units = "K" ;
    Temperature:coordinate_interpolation = "lat: lon: bl_interpolation" ;

  // Interpolation variable
  char bl_interpolation ;
    bl_interpolation:interpolation_name = "bi_linear" ;
    bl_interpolation:tie_point_mapping = "xc: x_indices tp_xc  yc: y_indices
tp_yc"  ;
    bl_interpolation:computational_precision = "64" ;

  // tie point coordinate variables
  double lat(tp_yc, tp_xc) ;
    lat:units = "degrees_north" ;
    lat:standard_name = "latitude" ;
```

```
  double lon(tp_yc, tp_xc) ;
    lon:units = "degrees_east" ;
    lon:standard_name = "longitude" ;

  // Tie point index variables
  int y_indices(tp_yc) ;
  int x_indices(tp_xc) ;

data:
  x_indices = 0, 9, 19, 29 ;
  y_indices = 0, 9 ;
  ...
```

Example 8.4. One-dimensional tie point interpolation of two-dimensional domain.

```
dimensions:
  xc = 30;
  yc = 10;
  tp_xc = 4 ;

variables:
  // Data variable
  float Temperature(yc, xc) ;
    Temperature:standard_name = "air_temperature" ;
    Temperature:units = "K" ;
    Temperature:coordinate_interpolation = "lat: lon: l_interpolation" ;

  // Interpolation variables
  char l_interpolation ;
    l_interpolation:interpolation_name = "linear" ;
    l_interpolation:tie_point_mapping = "xc: x_indices tp_xc"  ;
    l_interpolation:computational_precision = "64" ;

  // tie point coordinate variables
  double lat(yc, tp_xc) ;
    lat:units = "degrees_north" ;
    lat:standard_name = "latitude" ;
  double lon(yc, tp_xc) ;
    lon:units = "degrees_east" ;
    lon:standard_name = "longitude" ;

  // Tie point index variables
  int x_indices(tp_xc) ;

data:
  x_indices = 0, 9, 19, 29 ;
  ...
```

8.3.8. Interpolation Parameters

The interpolation variable attribute `interpolation_parameters` may be used to provide extra information to the interpolation process. This attribute names *interpolation parameter variables* that provide values for coefficient terms in the interpolation equation, or for any other terms that configure the interpolation process. The `interpolation_parameters` attribute takes a string value, the string comprising blank-separated elements of the form `"term: variable"`, where `term` is a case-insensitive keyword that defines one of the terms in the interpolation method's definition given in Appendix J, Coordinate Interpolation Methods, and `variable` is the name of the interpolation parameter variable that contains the values for that term. The order of elements is not significant. Any numerical term that is specified as optional in Appendix J, Coordinate Interpolation Methods and is omitted from the `interpolation_parameters` attribute should be assumed to be zero.

The `interpolation_parameters` attribute may only be provided if allowed by the definition of the interpolation method. Interpolation parameters may always be provided to non-standardized interpolation methods.

The interpolation parameters are not permitted to contain absolute coordinate information, such as additional tie points, but may contain relative coordinate information, for example an offset with respect to a tie point or with respect to a combination of tie points. This is to ensure that interpolation methods are equally applicable to both coordinate and bounds interpolation.

The interpolation parameter variable dimensions must include, for all of the interpolated dimensions, either the associated subsampled dimension or the associated interpolation subarea dimension. Additionally, any subset of zero or more of the non-interpolated dimensions of the tie point coordinate variable are permitted as interpolation parameter variable dimensions.

The application of an interpolation parameter variable is independent of its non-interpolated dimensions, but depends on its set of subsampled dimensions and interpolation subarea dimensions:

- If the set only contains subsampled dimensions, then the variable provides values for every tie point and therefore equally applicable to the interpolation subareas that share that tie point, see example a) in Figure 8.3;
- If the set only contains interpolation subarea dimensions, then the variable provides values for every interpolation subarea and therefore only applicable to that interpolation subarea, see example b) in Figure 8.3;
- If the set contains both subsampled dimensions and interpolation subarea dimensions, then the variable's values are to be shared by the interpolation subareas that are adjacent along each of the specified subsampled dimensions. This case is akin to the values being defined at the interpolation subarea boundaries, and therefore equally applicable to the interpolation subareas that share that boundary, see example c) and d) in Figure 8.3;

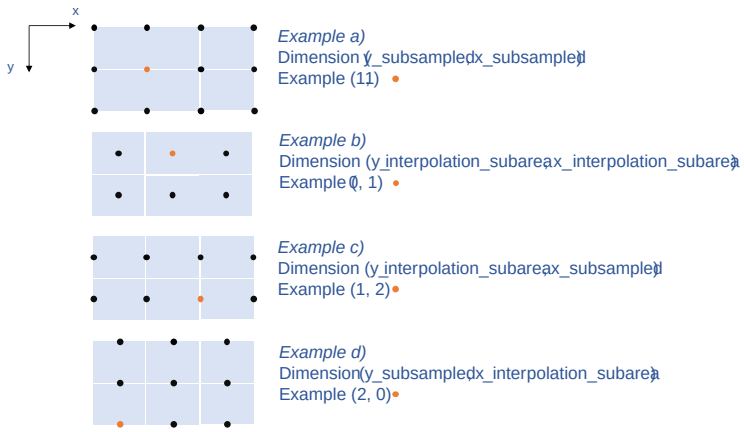


Figure 8.3. Through combination of dimensions, interpolation parameter variables may provide values for a) interpolation subareas sharing a tie point, b) each interpolation subarea, c) and d) interpolation subareas sharing a boundary.

Example 8.5. Multiple interpolation variables with interpolation parameter attributes.

```
dimensions :
  // VIIRS I-Band (375 m resolution imaging)
  track = 1536 ;
  scan = 6400 ;
  // Tie points and interpolation subareas
  tp_track = 96 ;  // 48 VIIRS scans
  tp_scan = 205 ;
  subarea_track = 48 ;   // track interpolation subarea
  subarea_scan= 200 ;    // scan interpolation subarea
  // Time, stored at scan-start and scan-end of each scan
  tp_time_scan = 2;

variables:
  // VIIRS I-Band Channel 04
  float I04_radiance(track, scan) ;
    I04_radiance:coordinate_interpolation = "lat: lon: tp_interpolation  t:
time_interpolation" ;
    I04_radiance:standard_name = "toa_outgoing_radiance_per_unit_wavelength" ;
    I04_radiance:units = "W m-2 sr-1 m-1" ;
  float I04_brightness_temperature(track, scan) ;
    I04_brightness_temperature:coordinate_interpolation = "lat: lon:
tp_interpolation  t: time_interpolation" ;
    I04_brightness_temperature:standard_name = "brightness_temperature" ;
    I04_brightness_temperature:units = "K" ;

  // Interpolation variable
  char tp_interpolation ;
    tp_interpolation:interpolation_name = "bi_quadratic_latitude_longitude" ;
    tp_interpolation:tie_point_mapping = "track: track_indices tp_track
subarea_track
                                        scan: scan_indices tp_scan subarea_scan"
;
    tp_interpolation:interpolation_parameters =
```

```
         "ce1: ce1  ca2: ca2  ce3: ce3 interpolation_subarea_flags:
interpolation_subarea_flags" ;
    tp_interpolation:computational_precision = "32" ;

  // Interpolation parameters
  short ce1(tp_track , subarea_scan) ;
  short ca2(subarea_track , tp_scan) ;
  short ce3(subarea_track, subarea_scan) ;
  byte interpolation_subarea_flags(subarea_track , subarea_scan) ;
    interpolation_subarea_flags:valid_range = 1b, 7b ;
    interpolation_subarea_flags:flag_masks = 1b, 2b, 4b ;
    interpolation_subarea_flags:flag_meanings =
         "location_use_3d_cartesian
          sensor_direction_use_3d_cartesian
          solar_direction_use_3d_cartesian" ;

  // Tie point index variables
  int track_indices(tp_track) ;   // shared by tp_interpolation and
time_interpolation
  int scan_indices(tp_scan) ;
  int time_scan_indices(tp_time_scan)

  // Tie points
  float lat(tp_track, tp_scan) ;
    lat:standard_name = "latitude" ;
    lat:units = "degrees_north" ;
  float lon(tp_track, tp_scan) ;
    lon:standard_name = "longitude" ;
    lon:units = "degrees_east" ;

  // Time interpolation variable
  char time_interpolation ;
    time_interpolation:interpolation_name = "bi_linear" ;
    time_interpolation:tie_point_mapping = "track: track_indices tp_track scan:
time_scan_indices tp_time_scan"  ;
    time_interpolation:computational_precision = "64" ;

  // Time tie points
  double t(tp_track, tp_time_scan) ;
    t:standard_name = "time" ;
    t:units = "days since 1990-1-1 0:0:0" ;
```

This example demonstrates the use of multiple interpolation variables, the reusability of the interpolation variable between data variables of different dimensions and the use of the interpolation parameter attribute.

Example 8.6. Combining a grid mapping and coordinate interpolation, with time as a non-interpolated dimension.

```
dimensions:
  y = 228;
  x = 306;
  time = 41;

  // Tie point dimensions
  tp_y = 58;
  tp_x = 52;

variables:
  // Data variable
  float Temperature(time, y, x) ;
    Temperature:standard_name = "air_temperature" ;
    Temperature:units = "K" ;
    Temperature:grid_mapping = "lambert_conformal" ;
    Temperature:coordinate_interpolation = "lat: lon: bi_linear x: linear_x y:
linear_y" ;

  int lambert_conformal ;
    lambert_conformal:grid_mapping_name = "lambert_conformal_conic" ;
    lambert_conformal:standard_parallel = 25.0 ;
    lambert_conformal:longitude_of_central_meridian = 265.0 ;
    lambert_conformal:latitude_of_projection_origin = 25.0 ;

  // Interpolation variables
  char bi_linear ;
    bi_linear:interpolation_name = "bi_linear" ;
    bi_linear:tie_point_mapping = "y: y_indices tp_y  x: x_indices tp_x"  ;
    bi_linear:computational_precision = "64" ;

  char linear_x ;
    linear_x:interpolation_name = "linear" ;
    linear_x:tie_point_mapping = "x: x_indices tp_x" ;
    linear_x:computational_precision = "64" ;

  char linear_y ;
    linear_y:interpolation_name = "linear" ;
    linear_y:tie_point_mapping = "y: y_indices tp_y" ;
    linear_y:computational_precision = "64" ;

  // tie point coordinate variables
  double time(time) ;
    time:standard_name = "time" ;
    time:units = "days since 2021-03-01" ;
  double y(time, tp_y) ;
    y:units = "km" ;
    y:standard_name = "projection_y_coordinate" ;
```

```
double x(time, tp_x) ;
  x:units = "km" ;
  x:standard_name = "projection_x_coordinate" ;
double lat(time, tp_y, tp_x) ;
  lat:units = "degrees_north" ;
  lat:standard_name = "latitude" ;
double lon(time, tp_y, tp_x) ;
  lon:units = "degrees_east" ;
  lon:standard_name = "longitude" ;

// Tie point index variables
int y_indices(tp_y) ;
  y_indices:long_name = "Mapping of y dimension to its ",
                        "corresponding tie point dimension" ;
int x_indices(tp_x) ;
  x_indices:long_name = "Mapping of x dimension to its ",
                        "corresponding tie point dimension" ;
```

In this the projection coordinates are two-dimensional, but are only linearly interpolated in one of their dimensions - the one which is given by the `tie_point_mapping` attribute.

8.3.9. Interpolation of Cell Boundaries

Coordinates may have cell bounds. For the case that the reconstituted cells are contiguous and have exactly two cell bounds along each interpolated dimension, cell bounds of interpolated dimensions can be stored as *bounds tie points* and reconstituted through interpolation. In this process, the coordinate tie points are a prerequisite for the bounds tie points and the same interpolation method used for the coordinate interpolation is used for the bounds interpolation.

For the reconstituted coordinates, cell bounds are stored separately for each coordinate point, as shown in the left part of Figure 8.4 for the example of 2D bounds. Since the cell bounds are contiguous, bounds points of adjacent cells will coincide and so the full set of bounds points may be represented as a grid, comparable to the coordinate points grid. In the middle part of Figure 8.4, both the reconstituted bounds points grid and the reconstituted coordinate points grid are shown for a continuous area, where each bounds point may be shared by up to four cells.

Bounds interpolation uses the same tie point index variables and therefore the same tie point cells as coordinate interpolation. One of the vertices of each coordinate tie point cell is chosen as the bounds tie point for the cell. It is selected as the vertex of the tie point cell that is the closest to the boundary of the interpolation subarea with respect to each interpolated dimension. For the example of 2D bounds, the resulting set of bounds tie points are marked in Figure 8.4, where the selected vertices are those closest to the corners of the interpolation subareas.

Note that within a continuous area, there is one more reconstituted bounds point than there are reconstituted coordinate points in each dimension. For this reason, a virtual *interpolated bounds dimension* is introduced for each dimension, having a size equal to the size of the interpolated dimension plus one. This dimension is used for solely descriptive purposes, and is not required in a compressed dataset.

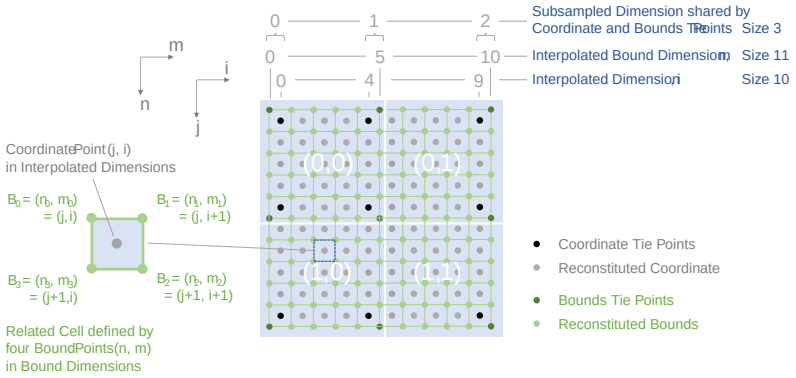


Figure 8.4. Example of 2D bounds interpolation showing the bounds tie points and reconstituted bound points within a continuous area consisting of four interpolation subareas. The dimensions are shown for one of the two axes only. The index relationship between coordinate point indices and the related bound points indices is indicated.

Both the process of compressing bounds and the process of uncompressing bounds requires the steps to be carried out for a full continuous area, however, individual continuous areas can be processed independently. In the following description of these processes, indices relative to the origin of each continuous area are used for the interpolated dimension and the interpolated bounds dimension. Consequently, for both coordinate tie points and bounds tie points, the first point in index space of the continuous area has got index 0 in all the interpolated dimensions and interpolated bounds dimensions, respectively.

Note that the numbering of the bounds `B0`, `B1`, etc, in this section is identical to the numbering in Section 7.1, "Cell Boundaries".

A bounds tie point is located in the same interpolation subarea as its corresponding coordinate tie point. The interpolation subareas do not overlap, ensuring that each bound point is computed from a unique interpolation subarea, see also the description of interpolation subareas in Section 8.3.1, "Tie Points and Interpolation Subareas". That bounds are computed only once ensures that the reconstituted bounds are contiguous.

For the generation of bounds tie points as part of the process of compressing bounds, the indices of the corresponding coordinate tie points are available in the tie point index variables, see Section 8.3.7, "Tie Point Index Mapping".

Compressing one-dimensional coordinate bounds

In the one-dimensional case, a coordinate point at index `i` in the interpolated dimension will be bounded by the two bounds:

```
B0 = (n0) = (i)
B1 = (n1) = (i+1)
```

where `n` is the bound index in the interpolated bound dimension.

For one-dimensional bound interpolation, an interpolation subarea is defined by two bounds tie points. The full set of bounds tie points is formed by appending, for each continuous area of the domain, the bound point `B0` of the first coordinate tie points of the continuous area, followed by the bound points `B1` of all subsequent coordinate tie point of the continuous area.

Compressing two-dimensional coordinate bounds
In the two-dimensional case, a coordinate point at indices `(j, i)` in the interpolated dimension will be bounded by the four bounds:

```
B0 = (n0, m0) = (j, i)
B1 = (n1, m1) = (j, i+1)
B3 = (n3, m3) = (j+1, i)
B2 = (n2, m2) = (j+1, i+1)
```

where `(n, m)` are the bounds point indices in the interpolated bound dimensions.

For two-dimensional bound interpolation, an interpolation subarea is defined by four bounds tie points. The full set of bounds tie points is formed by appending, for each continuous area of the domain, the bound point `B0` of the coordinate tie point at origin the of the continuous area `(0, 0)`, followed by the bound points `B1` of all remaining coordinate tie point of the continuous area with index `j = 0`, followed by the bound points `B3` of all remaining coordinate tie point of the continuous area with index `i = 0`, followed by the bound points `B2` of all remaining coordinate tie point of the continuous area.

Bounds Tie Point Attribute and Bounds Tie Point Variable
A `bounds_tie_points` attribute must be defined for each tie point coordinate variable corresponding to reconstituted coordinates with cell boundaries. Its string value identifies a *bounds tie point variable* which contains a bounds tie point coordinate value for each tie point stored in its tie point coordinate variable, and therefore the bounds tie point variable has the same set of dimensions as its tie point coordinate variable. An example of the usage of the `bounds_tie_points` is shown in Example 8.7. Since a bounds tie point variable is considered to be part of a tie point coordinate variable's metadata, it is not necessary to provide it with attributes such as long_name and units, following the same rules as for the bounds of an uncompressed coordinate variable, see Section 7.1, "Cell Boundaries".

Uncompressing coordinate bounds
The reconstitution of the full set of bounds from the bounds tie point is a two-step process. In a first step, which must be carried out for a full continuous area at a time, each bound point is reconstituted by interpolation between the bounds tie points within each interpolation subarea, using the same interpolation method as defined for the ordinary tie points. This step results in a grid of bound points spanning the interpolated bound dimensions. In a second step the reconstituted bounds vertices are replicated to the boundary variables of the reconstituted coordinates.

Uncompressing one-dimensional coordinate bounds
For one-dimensional coordinate bounds, in the second step of the process, for each index `i` of the interpolated dimension, the two bounds of the boundary variable are set to the value of the interpolated bounds point grid at the indices `B0` and `B1`, respectively, where the indices are defined above under "Compressing one-dimensional coordinate bounds".

Uncompression of two-dimensional coordinate bounds
For two-dimensional coordinate bounds, in the second step of the process, for each index pair `(j, i)` of the interpolated dimension, the four bounds of the boundary variable is set to the value of the

interpolated bounds point grid at index pairs B0, B1, B2 and B3, respectively, where the index pairs are defined above under "Compressing two-dimensional coordinate bounds".

Example 8.7. Interpolation of the 2D cell boundaries corresponding to Figure 8.4

```
dimensions:
  ic = 10;
  itp = 3;

  jc = 10;
  jtp = 3;

variables:
  // Data variable
  float Temperature(jc, ic) ;
    Temperature:standard_name = "air_temperature" ;
    Temperature:units = "K" ;
    Temperature:coordinate_interpolation = "lat: lon: bl_interpolation" ;

  // Interpolation variable
  char bl_interpolation ;
    bl_interpolation:interpolation_name = "bi_linear" ;
    bl_interpolation:tie_point_mapping = "ic: i_indices itp  jc: j_indices jtp"  ;
    bl_interpolation:computational_precision = "64" ;

  // Tie point index variables
  int i_indices(itp) ;
  int j_indices(jtp) ;

  // Tie point coordinate variables
  double lat(jtp, itp) ;
    lat:units = "degrees_north" ;
    lat:standard_name = "latitude" ;
    lat:bounds_tie_points = "lat_bounds" ;

  double lon(jtp, itp) ;
    lon:units = "degrees_east" ;
    lon:standard_name = "longitude" ;
    lon:bounds_tie_points = "lon_bounds" ;

  // Bounds tie point variables
  double lat_bounds(jtp, itp) ;
  double lon_bounds(jtp, itp) ;
```

8.3.10. Interpolation Method Implementation

The accuracy of the reconstituted coordinates depends mainly on the degree of subsampling and the choice of interpolation method, both of which are set by the creator of the dataset. The accuracy

and reproducibility will also depend, however, on how the interpolation method is implemented and on the computer platform carrying out the computations. To facilitate a good level of reproducibility of the processes of compressing and uncompressing coordinates, requirements are placed on the specification of interpolation methods and on stating the computational precision.

Interpolation Method Specification
The interpolation method specifications provided in Appendix J, Coordinate Interpolation Methods are complete in their description of steps and formulas required for compressing and uncompressing coordinate data. Formulas are structured in a way that encourages an efficient implementation of the interpolation method in a high-level programming language. For instance, expressions that are constant within a computational loop should be externalised from that loop.

Computational Precision Attribute
The data creator shall specify the floating-point arithmetic precision used during the preparation and validation of the compressed coordinates by setting the interpolation variable's `computational_precision` attribute to one of the following values:

Value	Description
"32"	32-bit floating-point arithmetic, comparable to the IEEE binary32 standard [IEEE_754]
"64"	64-bit floating-point arithmetic, comparable to the IEEE binary64 standard [IEEE_754]

Using the given computational precision in the interpolation computations is a necessary, but not sufficient, condition for the data user to be able to reconstitute the coordinates to an accuracy comparable to that intended by the data creator. For instance, a `computational_precision` value of `"64"` would specify that, using the same implementation and hardware as the creator of the compressed dataset, sufficient accuracy could not be reached when using a floating-point precision lower than 64-bit floating-point arithmetic in the interpolation computations required to reconstitute the coordinates.

8.4. Lossy Compression via Quantization

Geoscientific models and measurements generate false floating-point precision (scientifically meaningless data bits) that wastes storage space. False precision can mislead (by implying noise is signal) and is scientifically pointless. Quantization algorithms can eliminate false precision, usually by rounding the least significant bits of [IEEE_754] floating-point mantissas to zeros. (Quantization of integer types, although theoretically allowed, is not covered by this convention.) The quantized results are valid [IEEE_754] values---no special software or decoder is necessary to read them. Importantly, the quantized bits compress more efficiently than random bits. Thus quantization is sometimes referred to as a form of lossy compression although, strictly speaking, quantization only pre-conditions data for more efficient compression by a subsequent compressor.

The CF conventions of this section define a metadata framework to record quantization properties alongside quantized floating-point data variables. The goals are twofold. First, to inform interested users how, and to what degree, the quantized data differ from the original unquantized data, which are not stored in the dataset and may no longer exist. Second, to provide the necessary provenance

metadata for users to reproduce the data transformations on the same or other raw data. These conventions also allow users to better understand the precision that data producers expect from source models or measurements.

These conventions must not be used with data variables of integer type. They must not be used with any variable, even if it is also a data variable, that serves as a coordinate variable, or is named by a `coordinates`, `formula_terms` or `cell_measures` attribute of any other variable. This is because variables that provide metadata or are used in computation of domain metrics are often known to the highest precision possible, and degrading the precision of metadata properties may have unintended side effects on the accuracy of subsequent operations such as regridding, interpolation, and conservation checks. These variables can include spatial and temporal coordinate variables (e.g., `latitude`, `longitude`, `level`, `time`), properties derived from these coordinates (e.g., `area`, `volume`), and variables referenced by the `formula_terms` attribute of a coordinate variable.

8.4.1. Quantization variables

A quantization variable describes a quantization algorithm via a collection of attached attributes. It is of arbitrary type since it contains no data. Its purpose is to act as a container for the generic attributes of a quantization algorithm. Quantization variables are required to have at least two attributes: `algorithm` and `implementation`.

The `algorithm` attribute names a specific quantization algorithm. Four quantization algorithms are currently recognized: BitRound, BitGroom, DigitRound, and Granular BitRound. The controlled vocabulary for these algorithms thus consists of `bitround`, `bitgroom`, `digitround`, and `granular_bitround`. See Section 8.4.3, "Description of Quantization Algorithms" for a brief summary of these algorithms.

The second attribute required in a quantization variable is `implementation`. This attribute contains unstandardized text that concisely conveys the algorithm provenance including the name of the library or client that performed the quantization, the software version, and any other information required to disambiguate the source of the algorithm employed. The text must take the form "*software-name* version *version-string* [(*optional-information*)]" such as `libnetcdf version 4.9.2` in Quantization performed by BitRound algorithm in libnetcdf.

8.4.2. Per-variable quantization attributes

Each data variable that has been quantized must include at least two attributes to describe the quantization. First, all such data variables must have a `quantization` attribute containing the name of the quantization variable describing the algorithm. Second, all such variables must record the specific parameter value used in the quantization algorithm. The input parameter for all quantization algorithms determines the precision preserved by the algorithm.

BitRound retains the specified number of significant bits (NSB) in the IEEE mantissa, and quantizes the trailing bits. All data variables quantized by BitRound must record the NSB in the `quantization_nsb` attribute. Note that BitRound *counts only explicitly represented mantissa bits*. It does not include the most-significant-bit with value 1 that implicitly begins all [IEEE_754] mantissas. Thus `quantization_nsb` is an integer type attribute with `1 <= NSB <= 23` for data type `float` or `real`, and `1 <= NSB <= 52` for data type `double`.

The BitGroom, Granular BitRound, and DigitRound algorithms guarantee preservation of a specified number of significant digits (NSD) in base 10 representation. The actual number of mantissa bits quantized depends on the algorithm. Thus all data variables quantized by BitGroom, Granular BitRound, or DigitRound must have a corresponding attribute `quantization_nsd`. The value of `quantization_nsd` is an integer with `1 <= NSD <= 7` for data type `float` or `real`, and `1 <= NSD <= 15` for data type `double`.

Example 8.8. Quantization performed by BitRound algorithm in libnetcdf

```
variables:
  char quantization_info ;
    quantization_info:algorithm = "bitround" ;
    quantization_info:implementation = "libnetcdf version 4.9.2" ;

  float ps(time,lat,lon) ;
    ps:_QuantizeBitRoundNumberOfSignificantBits = 9 ;
    ps:quantization = "quantization_info" ;
    ps:quantization_nsb = 9 ;
    ps:standard_name = "surface_air_pressure" ;
    ps:units = "Pa" ;
```

Note how the same NSB is reported in two attributes of the data variable `ps`. The quantization variable (`quantization_info`) `implementation` attribute reveals that the netCDF library applied the BitRound algorithm. The netCDF library wrote the system-defined `_QuantizeBitRoundNumberOfSignificantBits` attribute [NetCDF] which contains the same parameter value as the CF `quantization_nsb` attribute (see the main text for further details).

Example 8.9. Quantization performed by Granular BitRound algorithm in NCO

Quantization of different variables to different levels often makes good scientific sense. Here the pressure variable `ps` has four significant digits of precision while the temperature variable `ts` retains only three significant digits.

```
variables:
  char quantization_info ;
    quantization_info:algorithm = "granular_bitround" ;
    quantization_info:implementation = "NCO version 5.2.7" ;

  float ps(time,lat,lon) ;
    ps:standard_name = "surface_air_pressure" ;
    ps:units = "Pa" ;
    ps:quantization = "quantization_info" ;
    ps:quantization_nsd = 4 ;

  float ts(time) ;
    ts:standard_name = "surface_temperature" ;
    ts:units = "K" ;
```

```
        ts:quantization = "quantization_info" ;
        ts:quantization_nsd = 3 ;
```

Both variables were quantized by the same algorithm and so utilize the same quantization variable. `quantization_info` reveals that the Granular BitRound algorithm in NCO performed the quantization. Since the netCDF library did not perform the quantization, there is no system-defined underscored quantization attribute.

8.4.3. Description of quantization algorithms

This section briefly describes and contrasts each recognized `quantize` algorithm and points to further documentation. BitRound is also called the "round-to-nearest" method [KRD21] and the "half-to-even" method [Kou21]. This is the default [IEEE_754] rounding method and is bias-free and conservative for random distributions of numbers. BitRound is preferred when the number of significant bits (NSB) to retain is known.

The other `quantize` algorithms guarantee to preserve a given number of significant (base-10 representation) digits (NSD). Their quantization errors never exceed half of the unit value at the NSD decimal place [Zen16]. BitGroom [Zen16] appeared first, though is now known to be suboptimal in accuracy [Kou21] and in compressibility compared to later methods. DigitRound [DCG19] has superior compressibility for a given NSD compared to BitGroom. Granular BitRound combines the DigitRound approach for compressibility with the BitRound approach for quantization. Granular BitRound and DigitRound are both good choices when the NSD to retain is known.

The netCDF C and Fortran libraries can directly invoke BitRound, BitGroom, and Granular BitRound [[NetCDF]]. The netCDF library attaches a long, system-defined attribute to every data variable that it quantizes, such as `_QuantizeBitRoundNumberOfSignificantBits = 9` in Quantization performed by BitRound algorithm in libnetcdf. The leading underscore indicates that the netCDF library wrote this attribute [NUG]. Any variable that has the library-defined attribute must, in addition, contain the corresponding CF metadata. Example 8.9 shows how the CF metadata might appear for other (non-netCDF library) implementations of `quantize` algorithms.

Chapter 9. Discrete Sampling Geometries

This chapter provides representations for **discrete sampling geometries**, such as time series, vertical profiles and trajectories. Discrete sampling geometry datasets are characterized by a dimensionality that is lower than that of the space-time region that is sampled; discrete sampling geometries are typically “paths” through space-time.

9.1. Features and feature types

Each type of discrete sampling geometry (point, time series, profile or trajectory) is defined by the relationships among its spatiotemporal coordinates. We refer to the type of discrete sampling geometry as its **featureType**. The term “ **feature** ” refers herein to a single instance of the **discrete sampling geometry** (such as a single time series). The representation of such features in a CF dataset was supported previous to the introduction of this chapter using a particular convention, which is still supported (that described by section 9.3.1). This chapter describes further conventions which offer advantages of efficiency and clarity for storing a collection of features in a single file. When using these new conventions, *the features contained within a collection must always be of the same type; and all the collections in a CF file must be of the same feature type*. (Future versions of CF may allow mixing of multiple feature types within a file.) Table 9.1 presents the feature types covered by this chapter. Details and examples of storage of each of these feature types are provided in Appendix H, as indicated in the table.

Table 9.1. Logical structure and mandatory coordinates for discrete sampling geometry featureTypes

featureType	**Description of a single feature with this discrete sampling geometry**		**Link**
	Form of a data variable containing values defined on a collection of these features	**Mandatory space-time coordinates for a collection of these features**	
point	a single data point (having no implied coordinate relationship to other points)		
	data(i)	x(i) y(i) t(i)	Section H.1, "Point Data"
timeSeries	a series of data points at the same spatial location with time values in strict monotonically increasing order		
	data(i,o)	x(i) y(i) t(i,o)	Section H.2, "Time Series Data"
trajectory	a series of data points along a path through space with time values in strict monotonically increasing order		
	data(i,o)	x(i,o) y(i,o) t(i,o)	Section H.4, "Trajectory Data"

featureType	Description of a single feature with this discrete sampling geometry		Link
profile	an ordered set of data points along a vertical line at a fixed horizontal position and fixed time		
	data(i,o)	x(i) y(i) z(i,o) t(i)	Section H.3, "Profile Data"
timeSeriesProfile	a series of profile features at the same horizontal position with time values in strict monotonically increasing order		
	data(i,p,o)	x(i) y(i) z(i,p,o) t(i,p)	Section H.5, "Time Series of Profiles"
trajectoryProfile	a series of profile features located at points ordered along a trajectory		
	data(i,p,o)	x(i,p) y(i,p) z(i,p,o) t(i,p)	Section H.6, "Trajectory of Profiles"

In Table 9.1 the spatial coordinates x and y typically refer to longitude and latitude but other horizontal coordinates could also be used (see sections 4 and 5.6). The spatial coordinate z refers to vertical position. The time coordinate is indicated as t. The space-time coordinates that are indicated for each feature are mandatory. However a featureType may also include other space-time coordinates which are not mandatory (notably the z coordinate, and for instance a `forecast_reference_time` coordinate in addition to a mandatory time coordinate). The array subscripts that are shown illustrate only the *logical* structure of the data. The subscripts found in actual CF files are determined by the specific type of representations (see section 9.3).

The designation of dimensions as mandatory precludes the encoding of data variables where geo-positioning cannot be described as a discrete point location. Problematic examples include:

- time series that refer to a geographical region (e.g. the northern hemisphere), a volume (e.g. the troposphere), or a geophysical quantity in which geolocation information is inherent (e.g. the Southern Oscillation Index (SOI) is the difference between values at two point locations);
- vertical profiles that similarly represent geographically area-averaged values; and
- paths in space that indicate a geographically located feature, but lack a suitable time coordinate (e.g. a meteorological front).

Future versions of CF will generalize the concepts of geolocation to encompass these cases. As of CF version 1.6 such data can be stored using the representations that are documented here by two means: 1) by utilizing the orthogonal multidimensional array representation and omitting the featureType attribute; or 2) by assigning arbitrary coordinates to the mandatory dimensions. For example a globally-averaged latitude position (90s to 90n) could be represented arbitrarily (and poorly) as a latitude position at the equator.

9.2. Collections, instances and elements

In Table 9.1 the dimension with subscript i identifies a particular feature within a collection of

features. It is called the **instance dimension**. One-dimensional variables in a Discrete Geometry CF file, which have *only* this dimension (such as x(i), y(i) and z(i) for a timeseries), are **instance variables**. Instance variables provide the metadata that differentiates individual features.

The subscripts o and p distinguish the data elements that compose a single feature. For example in a collection of **timeSeries** features, each time series instance, i, has data values at various times, o. In a collection of **profile** features, the subscript, o, provides the index position along the vertical axis of each profile instance. We refer to data values in a feature as its **elements**, and to the dimensions of o and p as **element dimensions**. Each feature can have its own set of element subscripts o and p. For instance, in a collection of timeSeries features, each individual timeSeries can have its own set of times. The notation t(i,o) means there is a set of times with subscripts o for the elements of each feature i. Feature instances within a collection need not have the same numbers of elements. If the features do all have the same number of elements, and the sequence of element coordinates is identical for all features, savings in simplicity and space are achievable by storing only one copy of these coordinates. This is the essence of the orthogonal multidimensional representation (see section 9.3.1).

If there is only a single feature to be stored in a data variable, there is no need for an instance dimension and it is permitted to omit it. The data will then be one-dimensional, which is a special (degenerate) case of the multidimensional array representation. The instance variables will be scalar coordinate variables; the data variable and other auxiliary coordinate variables will have only an element dimension and not have an instance dimension, e.g. data(o) and t(o) for a single timeSeries.

9.3. Representations of collections of features in data variables

The individual features within a collection need not necessarily contain the same number of elements. For instance observed in situ time series will commonly contain unique numbers of time points, reflecting different deployment dates of the instruments. Other data sources, such as the output of numerical models, may commonly generate features of identical size. CF offers multiple representations to allow the storage to be optimized for the character of the data. Four types of representation are utilized in this chapter:

- two **multidimensional array representations**, in which each feature instance is allocated the identical amount of storage space. In these representations the instance dimension and the element dimension(s) are distinct CF coordinate axes (typical of coordinate axes discussed in chapter 4); and
- two **ragged array representations**, in which each feature is provided with the minimum amount of space that it requires. In these representations the instances of the individual features are stacked sequentially along the same array dimension as the elements of the features; we refer to this combined dimension as the **sample dimension**.

In the multidimensional array representations, data variables have both an instance dimension and an element dimension. The dimensions may be given in any order. If there is a need for either the instance or an element dimension to be the netCDF unlimited dimension (so that more features or more elements can be appended), then that dimension must be the outer dimension of the data

variable i.e. the leading dimension in CDL.

In the ragged array representations, the instance dimension (i), which sequences the individual features within the collection, and the element dimension, which sequences the data elements of each feature (o and p), both occupy the same dimension (the sample dimension). If the sample dimension is the netCDF unlimited dimension, new data can be appended to the file.

In all representations, the instance dimension (which is also the sample dimension in ragged representations) may be set initially to a size that is arbitrarily larger than what is required for the features which are available at the time that the file is created. Allocating unused array space in this way (pre-filled with missing values — see also section 9.6, *Missing data*), can be useful as a means to reserve space that will be available to add features at a later time.

9.3.1. Orthogonal multidimensional array representation

The **orthogonal multidimensional array representation**, the simplest representation, can be used if each feature instance in the collection has identical coordinates along the element axis of the features. For example, for a collection of the timeSeries that share a common set of times, or a collection of profiles that share a common set of vertical levels, this is likely to be the natural representation to use. In both examples, there will be longitude and latitude coordinate variables, x(i), y(i), that are one-dimensional and defined along the instance dimension.

Table 9.2 illustrates the storage of a data variable using the orthogonal multidimensional array representation. The data variable holds a collection of 4 features. The individual features, distinguished by color, are sequenced along the horizontal axis by the instance dimension indices, i1, i2, i3, i4. Each instance contains three elements, sequenced along the vertical with element dimension indices, o1, o2, o3. The i and o subscripts would be interchanged (i.e. Table 9.2 would be transposed) if the element dimension were the netCDF unlimited dimension.

Table 9.2. The storage of a data variable using the orthogonal multidimensional array representation (subscripts in CDL order)

(i1, o1)	**(i2, o1)**	**(i3, o1)**	**(i4, o1)**
(i1, o2)	(i2, o2)	(i3, o2)	(i4, o2)
(i1, o3)	(i2, o3)	(i3, o3)	(i4, o3)

The instance variables of a dataset corresponding to Table 9.2 will be one-dimensional with size 4 (for example, the latitude locations of timeSeries),

lat(i1)	lat(i2)	lat(i3)	lat(i4)

and the element coordinate axis will be one-dimensional with size 3 (for example, the time

time(o1)
time(o2)
time(o3)

coordinates that are shared by all of the timeSeries). This representation is consistent with the

multidimensional fields described in chapter 5; the characteristic that makes it atypical from chapter 5 (though not incompatible) is that the instance dimension is a discrete axis (see section 4.5).

9.3.2. Incomplete multidimensional array representation

The **incomplete multidimensional array representation** can used if the features within a collection do not all have the same number of elements, but sufficient storage space is available to allocate the number of elements required by the longest feature to all features. That is, features that are shorter than the longest feature must be padded with missing values to bring all instances to the same storage size. This representation sacrifices storage space to achieve simplicity for reading and writing.

Table 9.3 illustrates the storage of a data variable using the orthogonal multidimensional array representation. The data variable holds a collection of 4 features. The individual features, distinguished by color, are sequenced by the instance dimension indices, i1, i2, i3, i4. The instances contain respectively 2, 4, 3 and 6 elements, sequenced by the element dimension index with values of o1, o2, o3, … . The i and o subscripts would be interchanged (i.e. Table 9.3 would be transposed) if the element dimension were the netCDF unlimited dimension.

Table 9.3. The storage of data using the incomplete multidimensional array representation (subscripts in CDL order)

(i1, o1)	(i2, o1)	(i3, o1)	(i4, o1)
(i1, o2)	(i2, o2)	(i3, o2)	(i4, o2)
	(i2, o3)	(i3, o3)	(i4, o3)
	(i2, o4)		(i4, o4)
			(i4, o5)
			(i4, o6)

9.3.3. Contiguous ragged array representation

The **contiguous ragged array representation** can be used only if the size of each feature is known at the time that it is created. In this representation the data for each feature will be contiguous on disk, as shown in Table 9.4.

Table 9.4. The storage of data using the contiguous ragged representation (subscripts in CDL order)

(i1, o1)
(i1, o2)
(i2, o1)
(i2, o2)
(i2, o3)

(i2, o4)
(i3, o1)
(i3, o2)
(i3, o3)
(i4, o1)
(i4, o2)
(i4, o3)
(i4, o4)
(i4, o5)
(i4, o6)

In this representation, the file contains a **count variable**, which must be an integer type and

count(i1)	count(i2)	count(i3)	count(i4)
2	4	3	6

must have the instance dimension as its sole dimension. The count variable contains the number of elements that each feature has. This representation and its count variable are identifiable by the presence of an attribute, `sample_dimension`, found on the count variable, which names the sample dimension being counted. For indices that correspond to features, whose data have not yet been written, the count variable should have a value of zero or a missing value.

9.3.4. Indexed ragged array representation

The **indexed ragged array representation** stores the features interleaved along the sample dimension in the data variable as shown in Table 9.5. The canonical use case for this representation is the storage of real-time data streams that contain reports from many sources; the data can be written as it arrives.

Table 9.5. The storage of data using the indexed ragged representation (subscripts in CDL order)

Data variable indices		Index variable values
(i1, o1)		0
(i2, o1)		1
(i3, o1)		2
(i4, o1)		3
(i4, o2)		3
(i2, o2)		1
(i4, o3)		3
(i4, o4)		3

Data variable indices		Index variable values
(i1, o2)		0
(i2, o3)		1
(i3, o2)		2
(i4, o5)		3
(i3, o3)		2
(i2, o4)		1
(i4, o6)		3

In this representation, the file contains an **index variable**, which must be an integer type, and must have the sample dimension as its single dimension. The index variable contains the zero-based index of the feature to which each element belongs. This representation is identifiable by the presence of an attribute, `instance_dimension`, on the index variable, which names the dimension of the instance variables. For those indices of the sample dimension, into which data have not yet been written, the index variable should be pre-filled with missing values.

9.4. The featureType attribute

A global attribute, **featureType**, is required for all Discrete Geometry representations except the orthogonal multidimensional array representation, for which it is highly recommended. The exception is allowed for backwards compatibility, as discussed in 9.3.1. A Discrete Geometry file may include arbitrary numbers of data variables, but (as of CF v1.6) all of the data variables contained in a single file must be of the single feature type indicated by the global `featureType` attribute, if it is present.1 The value assigned to the `featureType` attribute is case-insensitive; it must be one of the string values listed in the left column of Table 9.1.

9.5. Coordinates and metadata

Every feature within a Discrete Geometry CF file must be unambiguously associated with an extensible collection of instance variables that identify the feature and provide other metadata as needed to describe it. Every element of every feature must be unambiguously associated with its space and time coordinates and with the feature that contains it. The `coordinates` attribute must be attached to every data variable to indicate the spatiotemporal coordinate variables that are needed to geo-locate the data.

Where feasible, one of the coordinate or auxiliary coordinate variables of a discrete sampling geometry should have an attribute named `cf_role`, whose only permitted values for this purposes are `timeseries_id`, `profile_id`, and `trajectory_id`. (Despite its general-sounding name, this attribute only one other function, namely in Section 5.9, "Mesh Topology Variables".) The variable carrying the `cf_role` attribute may have any data type. When a variable is assigned this attribute, it must provide a unique identifier for each feature instance. CF files that contain timeSeries, profile or trajectory featureTypes, should include only a single occurrence of a `cf_role` attribute; CF files that contain timeSeriesProfile or trajectoryProfile may contain two occurrences, corresponding to the two levels of structure in these feature types.

It is not uncommon for observational data to have two sets of coordinates for particular coordinate axes of a feature: a nominal point location and a more precise location that varies with the elements in the feature. For example, although an idealized vertical profile is measured at a fixed horizontal position and time, a realistic representation might include the time variations and horizontal drift that occur during the duration of the sampling. Similarly, although an idealized time series exists at a fixed lat-long position, a realistic representation of a moored ocean time series might include the “watch cycle” excursions of horizontal position that occur as a result of tidal currents.

CF Discrete Geometries provides a mechanism to encode both the nominal and the precise positions, while retaining the semantics of the idealized feature type. Only the set of coordinates which are regarded as the nominal (default or preferred) positions should be indicated by the attribute `axis`, which should be assigned string values to indicate the orientations of the axes (`X`, `Y`, `Z`, or `T`). See A single timeseries with time-varying deviations from a nominal point spatial location (a single timeseries with time-varying deviations from a nominal point spatial location): Auxiliary coordinate variables containing the nominal and the precise positions should be listed in the relevant `coordinates` attributes of data variables. In orthogonal representations the nominal positions could be coordinate variables, which do not need to be listed in the `coordinates` attribute, rather than auxiliary coordinate variables.

Coordinate bounds may optionally be associated with coordinate variables and auxiliary coordinate variables using the bounds attribute, following the conventions described in section 7.1. Coordinate bounds are especially important for accurate representations of model output data using discrete geometry representations; they record the boundaries of the model grid cells.

If there is a vertical coordinate variable or auxiliary coordinate variable, it must be identified by the means specified in section 4.3. The use of the attribute `axis=Z` is recommended for clarity. A `standard_name` attribute (see section 3.3) that identifies the vertical coordinate is recommended, e.g. "altitude", "height", etc. (See the CF Standard Name Table).

9.6. Missing Data

In data for discrete sampling geometries written according to the rules of this section, wherever there are unused elements in data storage, the data variable and all its auxiliary coordinate variables (spatial and time) must contain missing values. This situation may arise for the incomplete multidimensional array representation, and in any representation if the instance dimension is set to a larger size than the number of features currently stored. Data variables should (as usual) also contain missing values to indicate when there is no valid data available for the element, although the coordinates are valid.

Similarly, for indices where the instance variable identified by `cf_role` contains a missing value indicator, all other instance variables should also contain missing values.

Appendix A: Attributes

All CF attributes are listed here except for those that are used to describe grid mappings and mesh topologies. See Appendix F, Grid Mappings for the grid mapping attributes, and Appendix K, Mesh Topology Attributes for mesh topology attributes.

The "Type" values are **S** for string, **N** for numeric, and **D** for the type of the data variable. Each attribute may be used in any of the ways shown in its "Use" entry. **G** indicates it can appear as a global attribute, and **Gr** as a group attribute.

For variable attributes, the possible values of "Use" are:

- **C** for variables containing coordinate data,
- **D** for data variables,
- **M** for geometry container variables,
- **Q** for quantization container variables,
- **Do** for domain variables,
- **BI** and **BO** for boundary variables (see Section 7.1, "Cell Boundaries" for the distinction between **BI** and **BO**),
- **A** for aggregation variables (see Section 2.8, "Aggregation Variables"),
- **-** for variables with some other purpose.

CF does not prohibit any of these attributes from being attached to variables of different kinds from those listed as their "Use" in this table, but their meanings are not defined by CF if they are used in these other ways. "Links" indicates the location of the attribute's original definition (first link) and sections where the attribute is discussed in this document (additional links as necessary).

Table A.1. Attributes

Attribute	Type	Use	Links	Description
`actual_range`	N	C, D, BO	Section 2.5.1, "Missing data, valid and actual range of data"	The smallest and the largest valid non-missing values occurring in the variable.
`add_offset`	N	C, D, BO	NUG Appendix A, "Attribute Conventions", and Section 8.1, "Packed Data"	If present for a variable, this number is to be added to the data after it is read by an application. If both `scale_factor` and `add_offset` attributes are present, the data are first scaled before the offset is added. In cases where there is a strong constraint on dataset size, it is allowed to pack the coordinate variables (using add_offset and/or scale_factor), but this is not recommended in general.

Attribute	Type	Use	Links	Description
aggregated_data	S	A	Section 2.8, "Aggregation Variables"	Records the instructions that define how to create the aggregated data of an aggregation variable.
aggregated_dimensions	S	A	Section 2.8, "Aggregation Variables"	Identifies the dimensions of the aggregated data of an aggregation variable.
algorithm	S	Q	Section 8.4.1, "Quantization Variables", and Section 8.4.3, "Description of Quantization Algorithms"	Name of the quantization algorithm employed.
ancillary_variables	S	D	Section 3.4, "Ancillary Data"	Identifies a variable that contains closely associated data, e.g., the measurement uncertainties of instrument data.
axis	S	C, BI	Chapter 4, Coordinate Types	Identifies latitude, longitude, vertical, or time axes.
bounds	S	C	Section 7.1, "Cell Boundaries"	Identifies a boundary variable.
calendar	S	C, BI	Section 4.4.2, "Calendar"	Calendar used for encoding time axes.
cell_measures	S	D, Do	Section 7.2, "Cell Measures"	Identifies variables that contain cell areas or volumes.
cell_methods	S	D	Section 7.3, "Cell Methods", Section 7.4, "Climatological Statistics"	Records the method used to derive data that represents cell values.
cf_role	S	C, BI	Section 9.5, "Coordinates and metadata"	Identifies the roles of variables that identify features in discrete sampling geometries. Identifies the roles of mesh topology and location index set variables (see Appendix K, Mesh Topology Attributes).
climatology	S	C	Section 7.4, "Climatological Statistics"	Identifies a climatology boundary variable.
comment	S	G, C, D	Section 2.6.2, "Description of file contents"	Miscellaneous information about the data or methods used to produce it.
compress	S	C	Section 8.2, "Lossless Compression by Gathering", Section 5.3, "Reduced Horizontal Grid"	Records dimensions which have been compressed by gathering.

Attribute	Type	Use	Links	Description
`computed_standard_name`	S	C, BI	Section 4.3.3, "Parametric Vertical Coordinate"	Indicates the standard name, from the standard name table, of the computed vertical coordinate values, computed according to the formula in the definition.
`Conventions`	S	G	NUG Appendix A, "Attribute Conventions"	Name of the conventions followed by the dataset.
`coordinate_interpolation`	S	D, Do	Section 8.3, "Lossy Compression by Coordinate Subsampling"	Indicates that coordinates have been compressed by sampling and identifies the tie point coordinate variables and their associated interpolation variables.
`coordinates`	S	D, M, Do	Chapter 5, Coordinate Systems and Domain, Section 6.1, "Labels", Section 6.2, "Alternative Coordinates"	Identifies auxiliary coordinate variables, label variables, and alternate coordinate variables.
`dimensions`	S	Do	Section 5.8, "Domain Variables"	Identifies the dimensions that define a domain variable.
`external_variables`	S	G	Section 2.6.3, "External variables", Section 7.2, "Cell Measures"	Identifies variables which are named by `cell_measures` attributes in the file but which are not present in the file.
`_FillValue`	D	C, D, BO	NUG Appendix A, "Attribute Conventions", and Section 2.5.1, "Missing data, valid and actual range of data", and Section 9.6, "Missing Data".	A value used to represent missing or undefined data. Allowed for auxiliary coordinate variables but not allowed for coordinate variables.
`featureType`	S	G	Section 9.4, "The featureType attribute"	Specifies the type of discrete sampling geometry to which the data in the scope of this attribute belongs, and implies that all data variables in the scope of this attribute contain collections of features of that type.
`flag_masks`	D	D	Section 3.5, "Flags"	Provides a list of bit fields expressing Boolean or enumerated flags.

Attribute	Type	Use	Links	Description
flag_meanings	S	D	Section 3.5, "Flags"	Use in conjunction with flag_values to provide descriptive words or phrases for each flag value. If multi-word phrases are used to describe the flag values, then the words within a phrase should be connected with underscores.
flag_values	D	D	Section 3.5, "Flags"	Provides a list of the flag values. Use in conjunction with flag_meanings.
formula_terms	S	C, BO	Section 4.3.3, "Parametric Vertical Coordinate"	Identifies variables that correspond to the terms in a formula.
geometry	S	C, D, Do	Section 7.5, "Geometries"	Identifies a variable that defines geometry.
geometry_type	S	M	Section 7.5, "Geometries"	Indicates the type of geometry present.
grid_mapping	S	D, M, Do	Section 5.6, "Horizontal Coordinate Reference Systems, Grid Mappings, and Projections"	Identifies a variable that defines a grid mapping.
history	S	G, Gr	NUG Appendix A, "Attribute Conventions"	List of the applications that have modified the original data.
implementation	S	Q	Section 8.4.1, "Quantization Variables", and Section 8.4.3, "Description of Quantization Algorithms"	The name and version of the library or client software that performed the quantization with algorithm.
instance_dimension	S	-	Section 9.3, "Representations of collections of features in data variables"	An attribute which identifies an index variable and names the instance dimension to which it applies. The index variable indicates that the indexed ragged array representation is being used for a collection of features.
institution	S	G, D	Section 2.6.2, "Description of file contents"	Where the original data was produced.
interior_ring	S	M	Section 7.5, "Geometries"	Identifies a variable that indicates if polygon parts are interior rings (i.e., holes) or not.
leap_month	N	C, BI	Section 4.4.2, "Calendar"	Specifies which month is lengthened by a day in leap years for a user defined calendar.

Attribute	Type	Use	Links	Description
leap_year	N	C, BI	Section 4.4.2, "Calendar"	Provides an example of a leap year for a user defined calendar. It is assumed that all years that differ from this year by a multiple of four are also leap years.
location	S	D, Do	Section 5.9, "Mesh Topology Variables", and Appendix K, Mesh Topology Attributes	Specifies the location type within the mesh topology at which the variable is defined.
location_index_set	S	D, Do	Section 5.9, "Mesh Topology Variables", and Appendix K, Mesh Topology Attributes	Specifies a variable that defines the subset of locations of a mesh topology at which the variable is defined.
long_name	S	C, D, Do, BI	NUG Appendix A, "Attribute Conventions", and Section 3.2, "Long Name"	A descriptive name that indicates a variable's content. This name is not standardized.
mesh	S	D, Do	Section 5.9, "Mesh Topology Variables", and Appendix K, Mesh Topology Attributes	Specifies a variable that defines a mesh topology.
missing_value	D	C, D, BO	Section 2.5.1, "Missing data, valid and actual range of data", and Section 9.6, "Missing Data"	A value or values used to represent missing or undefined data. Allowed for auxiliary coordinate variables but not allowed for coordinate variables.
month_lengths	N	C, BI	Section 4.4.2, "Calendar"	Specifies the length of each month in a non-leap year for a user defined calendar.
node_coordinates	S	M	Section 7.5, "Geometries"	Identifies variables that contain geometry node coordinates.
node_count	S	M	Section 7.5, "Geometries"	Identifies a variable indicating the count of nodes per geometry.
nodes	S	C	Section 7.5, "Geometries"	Identifies a coordinate node variable.
part_node_count	S	M	Section 7.5, "Geometries"	Identifies a variable providing the count of nodes per geometry part.
positive	S	C, BI	[COARDS]	Direction of increasing vertical coordinate value.
quantization	S	D	Section 8.4.1, "Quantization Variables"	Identifies a variable that defines a quantization algorithm and its provenance.

Attribute	Type	Use	Links	Description
quantization_nsb	N	D	Section 8.4.2, "Per-variable Quantization Attributes", and Section 8.4.3, "Description of Quantization Algorithms"	Specifies the number of significant bits retained in the IEEE mantissa of data quantized with the BitRound algorithm. Use in conjunction with quantization.
quantization_nsd	N	D	Section 8.4.2, "Per-variable Quantization Attributes", and Section 8.4.3, "Description of Quantization Algorithms"	Specifies the number of significant base-10 digits retained in the IEEE mantissa of data quantized with base-10 quantization algorithms. Use in conjunction with quantization.
references	S	G, D	Section 2.6.2, "Description of file contents"	References that describe the data or methods used to produce it.
sample_dimension	S	-	Section 9.3, "Representations of collections of features in data variables"	An attribute which identifies a count variable and names the sample dimension to which it applies. The count variable indicates that the contiguous ragged array representation is being used for a collection of features.
scale_factor	N	C, D, BO	NUG Appendix A, "Attribute Conventions", and Section 8.1, "Packed Data"	If present for a variable, the data are to be multiplied by this factor after the data are read by an application. See also the add_offset attribute. In cases where there is a strong constraint on dataset size, it is allowed to pack the coordinate variables (using add_offset and/or scale_factor), but this is not recommended in general.
source	S	G, D	Section 2.6.2, "Description of file contents"	Method of production of the original data.
standard_error_multiplier	N	D	Appendix C, Standard Name Modifiers	If a data variable with a standard_name modifier of standard_error has this attribute, it indicates that the values are the stated multiple of one standard error.
standard_name	S	C, D, BI	Section 3.3, "Standard Name"	A standard name that references a description of a variable"s content in the standard name table.
title	S	G, Gr	NUG Appendix A, "Attribute Conventions"	Short description of the file contents.

Attribute	Type	Use	Links	Description
units	S	C, D, BI	NUG Appendix A, "Attribute Conventions", and Section 3.1, "Units"	Units of a variable's content.
units_metadata	S	C, D, BI	Section 3.1, "Units", and Section 4.4, "Time Coordinate"	Specifies the interpretation of a unit of measure appearing in the units attribute.
valid_max	N	C, D, BO	NUG Appendix A, "Attribute Conventions"	Largest valid value of a variable.
valid_min	N	C, D, BO	NUG Appendix A, "Attribute Conventions"	Smallest valid value of a variable.
valid_range	N	C, D, BO	NUG Appendix A, "Attribute Conventions"	Smallest and largest valid values of a variable.

Appendix B: Standard Name Table Format

The CF standard name table is an XML document (i.e., its format adheres to the XML 1.0 [XML] recommendation). The XML suite of protocols provides a reasonable balance between human and machine readability. It also provides extensive support for internationalization. See the W3C [W3C] home page for more information.

The document begins with a header that identifies it as an XML file:

```
<?xml version="1.0"?>
```

Next is the `standard_name_table` itself, which is bracketed by the tags `<standard_name_table>` and `</standard_name_table>`. The opening tag has two attributes: `xmlns:xsi` that provides a link to the standard XML namespace, and `xsi:noNamespaceSchemaLocation` that provides a link to the file that provides the XML schema rules for the content of the Standard Name Table XML file.

```
<standard_name_table
 xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="https://cfconventions.org/Data/schema-files/cf-
standard-name-table-2.0.xsd">
```

The content (delimited by the `<standard_name_table>` tags) consists of, in order,

```
  <version_number>Version number here ... </version_number>
  <conventions>Conventions string here ... </conventions>
  <first_published>Datetime of first publication to the website ... </first_published>
  <last_modified>Datetime stamp here ... </last_modified>
  <institution>Name of institution here ... </institution>
  <contact>E-mail address of contact person ... </contact>
```

where the "Conventions string" is composed of the fixed string `CF-StandardNameTable-` immediately followed by the version number.

Next follows a sequence of `entry` elements which may optionally be followed by a sequence of `alias` elements. The `entry` and `alias` elements take the following forms:

```
  <entry id="an_id">
      Define the variable whose standard_name attribute has the value "an_id".
  </entry>
  <alias id="another_id">
      Provide alias for a variable whose standard_name attribute has the value
"another_id".
  </alias>
```

The value of the `id` attribute appearing in the `entry` and `alias` tags is a case sensitive string,

containing no whitespace, which uniquely identifies the entry relative to the table. *This is the value used for a variable's* `standard_name` *attribute.*

The purpose of the **entry** elements are to provide definitions for the **id** strings. Each **entry** element contains the following elements:

```
<entry id="an_id">
  <canonical_units>Representative units for the variable ... </canonical_units>
  <description>Description of the variable ... </description>
</entry>
```

The **alias** elements do not contain definitions. Rather they contain the value of the **id** attribute of an **entry** element that contains the sought after definition. The purpose of the **alias** elements are to provide a means for maintaining the table in a backwards compatible fashion. For example, if more than one **id** string was found to correspond to identical definitions, then the redundant definitions can be converted into aliases. It is not intended that the **alias** elements be used to accommodate the use of local naming conventions in the **standard_name** attribute strings. Each **alias** element contains a single element:

```
<alias id="an_id">
  <entry_id>Identifier of the defining entry ... </entry_id>
</alias>
```

In exceptional cases the **alias** element may contain two elements:

```
<alias id="an_id">
  <entry_id>Identifier of a defining entry ... </entry_id>
  <entry_id>Identifier of another defining entry ... </entry_id>
</alias>
```

Example B.1. A name table containing three entries

```
  <?xml version="1.0"?>
  <standard_name_table xmlns:xsi="https://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation=
        "http://cf.conventions.org/Data/schema-files/cf-standard-name-table-
2.0.xsd">
    <version_number>83</version_number>
    <conventions>CF-StandardNameTable-83</conventions>
    <first_published>2023-10-17T15:09:35Z</first_published>
    <last_modified>2023-10-17T15:09:35Z</last_modified>
    <institution>Program for Climate Model Diagnosis and
Intercomparison</institution>
    <contact>support@pcmdi.llnl.gov</contact>
    <entry id="surface_air_pressure">
      <canonical_units>Pa</canonical_units>
```

```
    <description>
        The surface called "surface" means the lower boundary of the atmosphere.
    </description>
  </entry>
  <entry id="air_pressure_at_sea_level">
    <canonical_units>Pa</canonical_units>
    <description>
        Air pressure at sea level is the quantity often abbreviated
        as MSLP or PMSL. sea_level means mean sea level, which is close
        to the geoid in sea areas.
    </description>
  </entry>
  <alias id="mean_sea_level_pressure">
    <entry_id>air_pressure_at_sea_level</entry_id>
  </alias>
</standard_name_table>
```

The definition of a variable with the `standard_name` attribute `surface_air_pressure` is found directly since the element with `id="surface_air_pressure"` is an `entry` element which contains the definition.

The definition of a variable with the `standard_name` attribute mean_sea_level_pressure is found indirectly by first finding the element with the `id="mean_sea_level_pressure"`, and then, since this is an alias element, by searching for the element with `id="air_pressure_at_sea_level"` as indicated by the value of the `entry_id` tag.

It is possible that new tags may be added in the future. Any applications that parse the standard table should be written so that unrecognized tags are gracefully ignored.

Appendix C: Standard Name Modifiers

In the *Units* column, *u* indicates units dimensionally equivalent to those for the unmodified standard name.

Table C.1. Standard Name Modifiers

Modifier	Units	Description
`detection_minimum`	*u*	The smallest data value which is regarded as a detectable signal.
`number_of_observations`	1	The number of discrete observations or measurements from which a data value has been derived. The use of this modifier is deprecated and the standard_name number_of_observations is preferred to describe this type of metadata variable.
`standard_error`	*u*	*The uncertainty of the data value. The standard error includes both systematic and statistical uncertainty. By default it is assumed that the values supplied are for one standard error. If the values supplied are for some multiple of the standard error, the `standard_error` ancillary variable should have an attribute **`standard_error_multiplier`** stating the multiplication factor.
`status_flag`		Flag values indicating the quality or other status of the data values. The variable should have **`flag_values`** or **`flag_masks`** (or both) and **`flag_meanings`** attributes to show how it should be interpreted (Section 3.5, "Flags"). The use of this modifier is deprecated and the standard_name status_flag is preferred to describe this type of metadata variable.

*The definition of this modifier implies that if *u* is a either unit of temperature, or a unit of temperature multiplied by some other unit, the temperature in *u* must be interpreted as a temperature difference. Therefore the **`units_metadata`** attribute, if present, must have the value `temperature: difference`, even if the corresponding data variable without the modifier would have `units_metadata="temperature: on_scale"`. See Section 3.1.2, "Temperature units" for explanation.

Appendix D: Parametric Vertical Coordinates

The definitions given here allow an application to compute dimensional coordinate values from the parametric vertical coordinate values (usually dimensionless) and associated variables. The formulas are expressed for a gridpoint `(n,k,j,i)` where `i` and `j` are the horizontal indices, `k` is the vertical index and `n` is the time index. A coordinate variable is associated with its definition by the value of the `standard_name` attribute. The terms in the definition are associated with file variables by the `formula_terms` attribute. The `formula_terms` attribute takes a string value, the string being comprised of blank-separated elements of the form "`term: variable`", where `term` is a case-insensitive keyword that represents one of the terms in the definition, and `variable` is the name of the variable in a netCDF file that contains the values for that term. The order of elements is not significant.

The gridpoint indices are not formally part of the definitions, but are included to illustrate the indices that *might* be present in the file variables. For example, a vertical coordinate whose definition contains a time index is not necessarily time dependent in all netCDF files. Also, the definitions are given in general forms that may be simplified by omitting certain terms. A term that is omitted from the `formula_terms` attribute should be assumed to be zero.

The variables containing the terms may optionally have `standard_name` attributes, with values as indicated in this Appendix. The `standard_name` of the dimensional coordinate values which are computed by the formula may optionally be specified by the `computed_standard_name` attribute of the vertical coordinate variable, as indicated in this Appendix. A `computed_standard_name` is uniquely implied by the formula in some cases, while in others it depends on the `standard_name` of one or more of the terms, with which it must be consistent.

Atmosphere natural log pressure coordinate

```
standard_name = "atmosphere_ln_pressure_coordinate"
```

Definition

```
p(k) = p0 * exp(-lev(k))
```

where `p(k)` is the pressure at gridpoint `(k)`, `p0` is a reference pressure, `lev(k)` is the dimensionless coordinate at vertical gridpoint `(k)`.

The format for the `formula_terms` attribute is:

```
formula_terms = "p0: var1 lev: var2"
```

The standard_name of `p0` is `reference_air_pressure_for_atmosphere_vertical_coordinate`. The `computed_standard_name` is `air_pressure`.

Atmosphere sigma coordinate

```
standard_name = "atmosphere_sigma_coordinate"
```

Definition

```
p(n,k,j,i) = ptop + sigma(k)*(ps(n,j,i)-ptop)
```

where `p(n,k,j,i)` is the pressure at gridpoint `(n,k,j,i)`, `ptop` is the pressure at the top of the model, `sigma(k)` is the dimensionless coordinate at vertical gridpoint `(k)`, and `ps(n,j,i)` is the surface pressure at horizontal gridpoint `(j,i)`and time `(n)`.

The format for the formula_terms attribute is

```
formula_terms = "sigma: var1 ps: var2 ptop: var3"
```

The standard_name of `ptop` is `air_pressure_at_top_of_atmosphere_model`, and of `ps` is `surface_air_pressure`. The `computed_standard_name` is `air_pressure`.

Atmosphere hybrid sigma pressure coordinate

```
standard_name = "atmosphere_hybrid_sigma_pressure_coordinate"
```

Definition

```
p(n,k,j,i) = a(k)*p0 + b(k)*ps(n,j,i)
```

or

```
p(n,k,j,i) = ap(k) + b(k)*ps(n,j,i)
```

where `p(n,k,j,i)` is the pressure at gridpoint `(n,k,j,i)`, `a(k)` or `ap(k)` and `b(k)` are components of the hybrid coordinate at level `k`, `p0` is a reference pressure, and `ps(n,j,i)` is the surface pressure at horizontal gridpoint `(j,i)` and time `(n)`. The choice of whether `a(k)` or `ap(k)` is used depends on model formulation; the former is a dimensionless fraction, the latter a pressure value. In both formulations, `b(k)` is a dimensionless fraction.

The format for the `formula_terms` attribute is

```
formula_terms = "a: var1 b: var2 ps: var3 p0: var4"
```

where `a` is replaced by `ap` if appropriate.

The hybrid sigma-pressure coordinate for level `k` is defined as `a(k)+b(k)` or `ap(k)/p0+b(k)`, as appropriate.

The `standard_name` of `p0` is `reference_air_pressure_for_atmosphere_vertical_coordinate`, and of `ps` is `surface_air_pressure`. The `computed_standard_name` is `air_pressure`. No `standard_name` has been defined for `a`, `b` or `ap`.

Atmosphere hybrid height coordinate

```
standard_name = "atmosphere_hybrid_height_coordinate"
```

Definition

```
z(n,k,j,i) = a(k) + b(k)*orog(n,j,i)
```

where `z(n,k,j,i)` is the height above the datum (e.g. the geoid, which is approximately mean sea level) at gridpoint `(k,j,i)` and `time (n)`, `orog(n,j,i)` is the height of the surface above the datum at `(j,i)` and `time (n)`, and `a(k)` and `b(k)` are the coordinates which define hybrid height level `k`. `a(k)` has the dimensions of height and `b(i)` is dimensionless.

The format for the `formula_terms` attribute is

```
formula_terms = "a: var1 b: var2 orog: var3"
```

The `standard_name` of `orog` may be `surface_altitude` (i.e. above the geoid) or `surface_height_above_geopotential_datum`. The `computed_standard_name` is `altitude` or `height_above_geopotential_datum` in these cases respectively. No `standard_name` has been defined for `b`. There is no dimensionless coordinate because `a`, which has the `standard_name` of `atmosphere_hybrid_height_coordinate`, is the best choice for a level-dependent but geographically constant coordinate.

Atmosphere smooth level vertical (SLEVE) coordinate

```
standard_name = "atmosphere_sleve_coordinate"
```

Definition

```
z(n,k,j,i) = a(k)*ztop + b1(k)*zsurf1(n,j,i) + b2(k)*zsurf2(n,j,i)
```

where `z(n,k,j,i)` is the height above the datum (e.g. the geoid, which is approximately mean sea level) at gridpoint `(k,j,i)` and time `(n)`, `ztop` is the height of the top of the model above the datum, and `a(k)`, `b1(k)`, and `b2(k)` are the dimensionless coordinates which define hybrid level `k`. `zsurf1(n,j,i)` and `zsurf2(n,j,i)` are respectively the large-scale and small-scale components of the topography, and `a`, `b1` and `b2` are all functions of the dimensionless SLEVE coordinate. See Shaer et al

[SCH02] for details.

The format for the formula_terms attribute is

```
formula_terms = "a: var1 b1: var2 b2: var3 ztop: var4 zsurf1: var5
                 zsurf2: var6"
```

The standard_name of ztop may be altitude_at_top_of_atmosphere_model (i.e. above the geoid) or height_above_geopotential_datum_at_top_of_atmosphere_model.

The computed_standard_name is altitude or height_above_geopotential_datum in these cases respectively. No standard_name has been defined for b1, zsurf1, b2 or zsurf2.

Ocean sigma coordinate

```
standard_name = "ocean_sigma_coordinate"
```

Definition

```
z(n,k,j,i) = eta(n,j,i) + sigma(k)*(depth(j,i)+eta(n,j,i))
```

where z(n,k,j,i) is height (positive upwards) relative to the datum (e.g. mean sea level) at gridpoint (n,k,j,i), eta(n,j,i) is the height of the sea surface (positive upwards) relative to the datum at gridpoint (n,j,i), sigma(k) is the dimensionless coordinate at vertical gridpoint (k), and depth(j,i) is the distance (a positive value) from the datum to the sea floor at horizontal gridpoint (j,i).

The format for the formula_terms attribute is

```
formula_terms = "sigma: var1 eta: var2 depth: var3"
```

The standard_names for eta and depth and the computed_standard_name must be one of the consistent sets shown in Table D.1.

Ocean s-coordinate

```
standard_name = "ocean_s_coordinate"
```

Definition

```
z(n,k,j,i) = eta(n,j,i)*(1+s(k)) + depth_c*s(k) +
             (depth(j,i)-depth_c)*C(k)
```

where

```
C(k) = (1-b)*sinh(a*s(k))/sinh(a) +
       b*[tanh(a*(s(k)+0.5))/(2*tanh(0.5*a)) - 0.5]
```

where `z(n,k,j,i)` is height (positive upwards) relative to the datum (e.g. mean sea level) at gridpoint `(n,k,j,i)`, `eta(n,j,i)` is the height of the sea surface (positive upwards) relative to the datum at gridpoint `(n,j,i)`, `s(k)` is the dimensionless coordinate at vertical gridpoint `(k)`, and `depth(j,i)` is the distance (a positive value) from the datum to the sea floor at horizontal gridpoint `(j,i)`. The constants `a`, `b`, and `depth_c` control the stretching. The constants `a` and `b` are dimensionless, and `depth_c` must have units of length.

The format for the `formula_terms` attribute is

```
formula_terms = "s: var1 eta: var2 depth: var3 a: var4 b: var5 depth_c: var6"
```

The `standard_name`s for `eta` and `depth` and the `computed_standard_name` must be one of the consistent sets shown in Table D.1. No `standard_name` has been defined for `a`, `b` or `depth_c`.

Ocean s-coordinate, generic form 1

```
standard_name = "ocean_s_coordinate_g1"
```

Definition

```
z(n,k,j,i) = S(k,j,i) + eta(n,j,i) * (1 + S(k,j,i) / depth(j,i))
```

where

```
S(k,j,i) = depth_c * s(k) + (depth(j,i) - depth_c) * C(k)
```

where `z(n,k,j,i)` is height, positive upwards, relative to ocean datum (e.g. mean sea level) at gridpoint `(n,k,j,i)`, `eta(n,j,i)` is the height of the ocean surface, positive upwards, relative to ocean datum at gridpoint `(n,j,i)`, `s(k)` is the dimensionless coordinate at vertical gridpoint `(k)` with a range of `-1 <= s(k) <= 0`, `s(0)` corresponds to `eta(n,j,i)` whereas `s(-1)` corresponds to `depth(j,i)`; `C(k)` is the dimensionless vertical coordinate stretching function at gridpoint `(k)` with a range of `-1 <= C(k) <= 0`, `C(0)` corresponds to `eta(n,j,i)` whereas `C(-1)` corresponds to `depth(j,i)`; the constant `depth_c`, (positive value), is a critical depth controlling the stretching and `depth(j,i)` is the distance from ocean datum to sea floor (positive value) at horizontal gridpoint `(j,i)`.

The format for the `formula_terms` attribute is

```
formula_terms = "s: var1 C: var2 eta: var3 depth: var4 depth_c: var5"
```

The standard_names for eta and depth and the computed_standard_name must be one of the consistent sets shown in Table D.1. No standard_name has been defined for C or depth_c.

Ocean s-coordinate, generic form 2

```
standard_name = "ocean_s_coordinate_g2"
```

Definition

```
z(n,k,j,i) = eta(n,j,i) + (eta(n,j,i) + depth(j,i)) * S(k,j,i)
```

where

```
S(k,j,i) = (depth_c * s(k) + depth(j,i) * C(k)) / (depth_c + depth(j,i))
```

where z(n,k,j,i) is height, positive upwards, relative to ocean datum (e.g. mean sea level) at gridpoint (n,k,j,i), eta(n,j,i) is the height of the ocean surface, positive upwards, relative to ocean datum at gridpoint (n,j,i), s(k) is the dimensionless coordinate at vertical gridpoint (k) with a range of -1 <= s(k) <= 0, S(0) corresponds to eta(n,j,i) whereas s(-1) corresponds to depth(j,i); C(k) is the dimensionless vertical coordinate stretching function at gridpoint (k) with a range of -1 <= C(k) <= 0, C(0) corresponds to eta(n,j,i) whereas C(-1) corresponds to depth(j,i); the constant depth_c, (positive value), is a critical depth controlling the stretching and depth(j,i) is the distance from ocean datum to sea floor (positive value) at horizontal gridpoint (j,i).

The format for the formula_terms attribute is

```
formula_terms = "s: var1 C: var2 eta: var3 depth: var4 depth_c: var5"
```

The standard_names for eta and depth and the computed_standard_name must be one of the consistent sets shown in Table D.1. No standard_name has been defined for C or depth_c.

Ocean sigma over z coordinate

The description of this type of parametric vertical coordinate is defective in version 1.8 and earlier versions of the standard, in that it does not state what values the vertical coordinate variable should contain. Therefore, in accordance with the rules, all versions of the standard before 1.9 are deprecated for datasets that use the "ocean sigma over z" coordinate.

```
standard_name = "ocean_sigma_z_coordinate"
```

Definition

for levels k where sigma(k) has a defined value and zlev(k) is not defined:

```
z(n,k,j,i) = eta(n,j,i) + sigma(k)*(min(depth_c,depth(j,i))+eta(n,j,i))
```

for levels k where zlev(k) has a defined value and sigma(k) is not defined:

```
z(n,k,j,i) = zlev(k)
```

where `z(n,k,j,i)` is height, positive upwards, relative to ocean datum (e.g. mean sea level) at gridpoint `(n,k,j,i)`, `eta(n,j,i)` is the height of the ocean surface, positive upwards, relative to ocean datum at gridpoint `(n,j,i)`, and `depth(j,i)` is the distance from ocean datum to sea floor (positive value) at horizontal gridpoint `(j,i)`.

The parameter `sigma(k)` is defined only for the `nsigma` layers nearest the ocean surface, while `zlev(k)` is defined for the `nlayer - nsigma` deeper layers, where `0 <= nsigma <= nlayer` and `nlayer` is the size of the dimension of the vertical coordinate variable. Both `sigma` and `zlev` must have this dimension. For any `k`, whichever of `sigma(k)` or `zlev(k)` is undefined must contain missing data, while the other must not.

The format for the `formula_terms` attribute is

```
formula_terms = "sigma: var1 eta: var2 depth: var3 depth_c: var4 nsigma: var5
                 zlev: var6"
```

The `standard_name`s for `eta`, `depth`, `zlev` and the `computed_standard_name` must be one of the consistent sets shown in Table D.1. The `standard_name` for `sigma` is `ocean_sigma_coordinate`. No `standard_name` has been defined for `depth_c` or `nsigma`. The `nsigma` parameter is deprecated and optional in `formula_terms`; if supplied, it must equal the number of elements of `zlev` which contain missing data.

The `standard_name` for the vertical coordinate variable is `ocean_sigma_z_coordinate`. This variable should contain `sigma(k)*depth_c` for the layers where `sigma` is defined and `zlev(k)` for the other layers, with units of length. The layers must be arranged so that the vertical coordinate variable contains a strictly monotonic set of indicative values for the heights of the levels relative to the datum, either increasing or decreasing, and the direction must be indicated by the `positive` attribute, in the usual way for a vertical coordinate variable.

Ocean double sigma coordinate

```
standard_name = "ocean_double_sigma_coordinate"
```

Definition

```
for k <= k_c:

  z(k,j,i)= sigma(k)*f(j,i)

for k > k_c:
```

```
  z(k,j,i)= f(j,i) + (sigma(k)-1)*(depth(j,i)-f(j,i))

f(j,i)= 0.5*(z1+ z2) + 0.5*(z1-z2)* tanh(2*a/(z1-z2)*(depth(j,i)-href))
```

where `z(k,j,i)` is height (positive upwards) relative to the datum (e.g. mean sea level) at gridpoint `(k,j,i)`, `sigma(k)` is the dimensionless coordinate at vertical gridpoint `(k)` for `k <= k_c`, and `depth(j,i)` is the distance (a positive value) from the datum to sea floor at horizontal gridpoint `(j,i)`. `z1`, `z2`, `a`, and `href` are constants with units of length.

The format for the `formula_terms` attribute is

```
formula_terms = "sigma: var1 depth: var2 z1: var3 z2: var4 a: var5 href: var6
                 k_c: var7"
```

The `standard_name` for `depth` and the `computed_standard_name` must be one of the consistent sets shown in Table D.1. No `standard_name` has been defined for `z1`, `z2`, `a`, `href` or `k_c`.

Table D.1. Consistent sets of values for the standard_names of formula terms and the computed_standard_name

option	standard_name of computed dimensional coordinate	formula term name	standard_name of formula term
1	altitude	zlev	altitude
		eta	sea_surface_height_above_geoid
		depth	sea_floor_depth_below_geoid
2	height_above_geopotential_ datum	zlev	height_above_geopotential_datu m
		eta	sea_surface_height_above_ geopotential_datum
		depth	sea_floor_depth_below_ geopotential_datum
3	height_above_reference_ ellipsoid	zlev	height_above_reference_ellipsoi d
		eta	sea_surface_height_above_ reference_ellipsoid
		depth	sea_floor_depth_below_ reference_ellipsoid
4	height_above_mean_sea_ level	zlev	height_above_mean_sea_level
		eta	sea_surface_height_above_mea n_ sea_level
		depth	sea_floor_depth_below_mean_ sea_level

Appendix E: Cell Methods

In the *Units* column, *u* indicates the units of the physical quantity before the method is applied.

Table E.1. Cell Methods

cell_methods	Units	Description
point	*u*	The data values are representative of points in space or time (instantaneous). This is the default method for a quantity that is intensive with respect to the specified dimension.
sum	*u*	The data values are representative of a sum or accumulation over the cell. This is the default method for a quantity that is extensive with respect to the specified dimension.
maximum	*u*	Maximum
maximum_absolute_value	*u*	Maximum absolute value
median	*u*	Median
mid_range	*u*	Average of maximum and minimum
minimum	*u*	Minimum
minimum_absolute_value	*u*	Minimum absolute value
mean	*u*	Mean (average value)
mean_absolute_value	*u*	Mean absolute value
mean_of_upper_decile	*u*	Mean of the upper group of data values defined by the upper tenth of their distribution
mode	*u*	Mode (most common value)
range	*u*	*Absolute difference between maximum and minimum
root_mean_square	*u*	Root mean square (RMS)
standard_deviation	*u*	*Standard deviation
sum_of_squares	u^2	Sum of squares
variance	u^2	*Variance

*The definition of this method implies that if *u* is a either a unit of temperature, or a unit of temperature multiplied by some other unit, the temperature in *u* must be interpreted as a temperature difference. Therefore the **units_metadata** attribute, if present, must have the value `temperature: difference`. See Section 3.1.2, "Temperature units" for explanation.

Appendix F: Grid Mappings

Each recognized grid mapping is described in one of the sections below. Each section contains: the valid name that is used with the `grid_mapping_name` attribute; a list of the specific attributes that may be used to assign values to the mapping's parameters; the standard names used to identify the coordinate variables that contain the mapping's independent variables; and references to the mapping's definition or other information that may help in using the mapping. Since the attributes used to set a mapping's parameters may be shared among several mappings, their definitions are contained in a table in the final section. The attributes which describe the ellipsoid and prime meridian may be included, when applicable, with any grid mapping. These are:

- `earth_radius`
- `inverse_flattening`
- `longitude_of_prime_meridian`
- `prime_meridian_name`
- `reference_ellipsoid_name`
- `semi_major_axis`
- `semi_minor_axis`

In general we have used the FGDC "Content Standard for Digital Geospatial Metadata" [FGDC] as a guide in choosing the values for `grid_mapping_name` and the attribute names for the parameters describing map projections.

Albers Equal Area

```
grid_mapping_name = albers_conical_equal_area
```

Map parameters:

- `standard_parallel` - There may be 1 or 2 values.
- `longitude_of_central_meridian`
- `latitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at https://proj.org/operations/projections/aea.html and http://geotiff.maptools.org/proj_list/albers_equal_area_conic.html.

Azimuthal equidistant

```
grid_mapping_name = azimuthal_equidistant
```

Map parameters:

- `longitude_of_projection_origin`
- `latitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at http://geotiff.maptools.org/proj_list/azimuthal_equidistant.html and https://proj.org/operations/projections/aeqd.html.

Geostationary projection

```
grid_mapping_name = geostationary
```

Map parameters:

- `latitude_of_projection_origin`
- `longitude_of_projection_origin`
- `perspective_point_height`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)
- `sweep_angle_axis`
- `fixed_angle_axis`

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_angular_coordinate` and `projection_y_angular_coordinate`, respectively. It is assumed that the y-axis is aligned to the Earth's N/S axis, whereas the x-axis aligns with the E/W axis. CF specified the standard names `projection_x_coordinate` and `projection_y_coordinate` for these coordinates prior to version 1.9, but that use is deprecated. In the case of this projection, the projection coordinates are the scanning angle of the satellite instrument.

Notes:

The geostationary projection assumes a hypothetical view of the Earth from a perspective above the Earth where the azimuth and elevation viewing angles are described using a hypothetical gimbal model. This model is independent of the physical scan principles of any observing instrument. The model consists conceptually of a set of two rotating circles with a colocated centre, whose axes of rotation are perpendicular to each other. The axis of the outer circle is stationary, while the axis of the inner circle moves about the stationary axis. This means that a given viewing angle described using this model is the result of matrix multiplications, which is not commutative, so that order of operations is essential in achieving accurate results. The two axes are conventionally called the sweep-angle and fixed-angle axes; we adhere to this terminology, although some find these terms confusing, for the sake of interoperability with existing implementations.

The algorithm for computing the mapping may be found at https://www.cgms-info.org/documents/pdf_cgms_03.pdf. This document assumes the point of observation is directly over the equator, and that the `sweep_angle_axis` is y.

Explanatory diagrams for the projection may be found on the PROJ website, as well as notes on using the PROJ software for computing the mapping.

The `perspective_point_height` is the distance to the surface of the ellipsoid.

The `sweep_angle_axis` attribute indicates the axis on which the view sweeps. It corresponds to the outer-gimbal (stable) axis of the gimbal view model. For example, the value = "y" corresponds to the Meteosat satellites, the value = "x" to the GOES satellites.

The `fixed_angle_axis` attribute indicates the axis on which the view is fixed. It corresponds to the inner-gimbal axis of the gimbal view model, whose axis of rotation moves about the outer-gimbal axis. If `fixed_angle_axis` is "x", `sweep_angle_axis` is "y", and vice versa. Only one of those the attributes `fixed_angle_axis` or `sweep_angle_axis` is mandatory, as they can be used to infer each other. Note also that the values "x" and "y" are not case-sensitive.

The use of `projection_x_coordinate` and `projection_y_coordinate` was deprecated in version 1.9 of the CF Conventions. The initial definition of this projection used these standard names to identify the projection coordinates even though their canonical units (meters) do not match those required for this projection (radians).

HEALPix

```
grid_mapping_name = healpix
```

Map parameters:

- `indexing_scheme` - The HEALPix indexing scheme: One of `nested`, `ring`, `nuniq`, or `zuniq`
- `refinement_level` - The HEALPix refinement level: Mandatory for indexing scheme `nested` or `ring`, but must be omitted for indexing scheme `nuniq` or `zuniq`

Map coordinates:

The coordinate system is defined by discrete cells on a reference ellipsoid (or sphere), arranged on a HEALPix grid. Each cell is represented by its cell index, which, together with the `indexing_scheme` and possible `refinement_level`, determines center coordinates (latitude and longitude) as well as boundaries. See [GHB05] for the calculation of center coordinates and cell boundary coordinates.

The integer-valued HEALPix index coordinates are non-negative and identified by the **`standard_name`** attribute value **`healpix_index`**. The HEALPix index coordinates may be stored in a coordinate, auxiliary, or scalar coordinate variable. Indexing schemes are optimized to allow applications efficient data access when the HEALPix indices are ordered strictly monotonically. Therefore it is recommended to store strictly monotonic HEALPix indices in a coordinate variable, which allows an application to assess their monotonicity without expensive checking.

Notes:

Cells are indexed using the scheme identified by the **`indexing_scheme`** parameter.

For both the **`nested`** and **`ring`** schemes, the grid resolution is controlled by the refinement level parameter **`refinement_level`** [GHB05]. The total number of cells on the surface of the sphere (or reference ellipsoid) is `12 * (4 ^ refinement_level)`. The lowest allowed refinement level is `0`, giving 12 cells globally. The algorithm that assigns cell locations on a sphere to **`nested`** or **`ring`** indices at a given refinement level is defined in [GHB05]. In this algorithm, a cell with index `0`, at any refinement level, always has its southernmost vertex on the meridian at 45 degrees east. If the **`healpix`** grid mapping is used on an ellipsoid, the latitudes returned by the [GHB05] algorithm must be interpreted as authalic latitude, instead of geodetic latitude (for a sphere, the two latitudes are the same). This is necessary to retain the equal area property. For instance, on the WGS84 ellipsoid this results in the center of the HEALPix cell at refinement level `0` with index `0` being at geodetic latitude 41.938 degrees north, instead of at 41.810 degrees north in the case of a sphere, whilst the longitude is at 45 degrees east in both cases.

The **`ring`** scheme sorts pixels along iso-latitude rings from north to south; and **`nested`** indexing sorts pixels along the Z-order curve within each of the 12 global cells at refinement level `0`, resulting in spatial proximity for consecutive pixels.

The **`nuniq`** and **`zuniq`** indexing schemes allow pixels from multiple refinement levels to be combined: **`nuniq`** effectively concatenates **`nested`** indices from lower to the higher refinement levels, resulting in spatial proximity for consecutive pixels within a refinement level but not across different refinement levels; and **`zuniq`** indexing sorts pixels of any refinement level along the Z-order curve, resulting in spatial proximity for consecutive pixels, regardless of their refinement level.

An index according to the **`nuniq`** indexing scheme (`index_nuniq`) can be computed from a refinement level and an index according to the **`nested`** indexing scheme (`index_nested`) with the formula `index_nuniq = index_nested + 4 ^ (1 + refinement_level)` [RH15].

An index according to the **`zuniq`** indexing scheme (`index_zuniq`) can be computed from a refinement level and an index according to the **`nested`** indexing scheme (`index_nested`) with the formula `index_zuniq = (2 * index_nested + 1) * 4 ^ (29 - refinement_level)` [K24].

The `refinement_level` parameter must be omitted for the `nuniq` and `zuniq` schemes, for which the resolution of a cell is implied by the index alone.

A refinement level, either given by the `refinement_level` parameter or implied by `nuniq` and `zuniq` indices, must be no greater than `29`, which is the highest refinement level for which all indices of any indexing scheme can be represented by 64-bit integers.

Lambert azimuthal equal area

```
grid_mapping_name = lambert_azimuthal_equal_area
```

Map parameters:

- `longitude_of_projection_origin`
- `latitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at https://proj.org/operations/projections/laea.html and http://geotiff.maptools.org/proj_list/lambert_azimuthal_equal_area.html

Lambert conformal

```
grid_mapping_name = lambert_conformal_conic
```

Map parameters:

- `standard_parallel` - There may be 1 or 2 values.
- `longitude_of_central_meridian`
- `latitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at

https://proj.org/operations/projections/lcc.html. and http://geotiff.maptools.org/proj_list/lambert_conic_conformal_1sp.html ("Lambert Conic Conformal (1SP)" or EPSG 9801) or http://geotiff.maptools.org/proj_list/lambert_conic_conformal_2sp.html ("Lambert Conic Conformal (2SP)" or EPSG 9802). For the 1SP variant, latitude_of_projection_origin=standard_parallel and the PROJ scale factor is 1.

Lambert Cylindrical Equal Area

```
grid_mapping_name = lambert_cylindrical_equal_area
```

Map parameters:

- `longitude_of_central_meridian`
- Either `standard_parallel` or `scale_factor_at_projection_origin` (deprecated)
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software packages for computing the mapping may be found at https://proj.org/operations/projections/cea.html and http://geotiff.maptools.org/proj_list/cylindrical_equal_area.html ("Lambert Cylindrical Equal Area" or EPSG 9834 or EPSG 9835). Detailed formulas can be found in [Snyder] pages 76-85.

Latitude-Longitude

```
grid_mapping_name = latitude_longitude
```

This grid mapping defines the canonical 2D geographical coordinate system based upon latitude and longitude coordinates. It is included so that the figure of the Earth can be described.

Map parameters:

None.

Map coordinates:

The rectangular coordinates are longitude and latitude identified by the usual conventions (Section 4.1, "Latitude Coordinate" and Section 4.2, "Longitude Coordinate").

Mercator

```
grid_mapping_name = mercator
```

Map parameters:

- `longitude_of_projection_origin`
- Either `standard_parallel` (EPSG 9805) or `scale_factor_at_projection_origin` (EPSG 9804)
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software packages for computing the mapping may be found at https://proj.org/operations/projections/merc.html and http://geotiff.maptools.org/proj_list/mercator_1sp.html ("Mercator (1SP)" or EPSG 9804) or http://geotiff.maptools.org/proj_list/mercator_2sp.html ("Mercator (2SP)" or EPSG 9805).

More information on formulas available in [OGP-EPSG_GN7_2].

Oblique Mercator

```
grid_mapping_name = oblique_mercator
```

Map parameters:

- `azimuth_of_central_line`
- `latitude_of_projection_origin`
- `longitude_of_projection_origin`
- `scale_factor_at_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at https://proj.org/operations/projections/omerc.html and http://geotiff.maptools.org/proj_list/oblique_mercator.html.

Orthographic

```
grid_mapping_name = orthographic
```

Map parameters:

- `longitude_of_projection_origin`
- `latitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software packages for computing the mapping may be found at https://proj.org/operations/projections/ortho.html and http://geotiff.maptools.org/proj_list/orthographic.html ("Orthographic" or EPSG 9840).

More information on formulas available in [OGP-EPSG_GN7_2].

Polar stereographic

```
grid_mapping_name = polar_stereographic
```

Map parameters:

- `longitude_of_projection_origin` or `straight_vertical_longitude_from_pole` (deprecated)
- `latitude_of_projection_origin` - Either +90. or -90.
- Either `standard_parallel` (EPSG 9829) or `scale_factor_at_projection_origin` (EPSG 9810)
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at https://proj.org/operations/projections/stere.html and http://geotiff.maptools.org/proj_list/polar_stereographic.html.

The standard_parallel variant corresponds to EPSG Polar Stereographic (Variant B) (EPSG dataset coordinate operation method code 9829), while the scale_factor_at_projection_origin variant

corresponds to EPSG Polar Stereographic (Variant A) (EPSG dataset coordinate operation method code 9810). As PROJ requires the standard parallel, [Snyder] formula 21-7 can be used to compute it from the scale factor if needed.

Rotated pole

```
grid_mapping_name = rotated_latitude_longitude
```

Map parameters:

- `grid_north_pole_latitude`
- `grid_north_pole_longitude`
- `north_pole_grid_longitude` - This parameter is optional (default is 0).

Map coordinates:

The rotated latitude and longitude coordinates are identified by the `standard_name` attribute values `grid_latitude` and `grid_longitude` respectively.

Sinusoidal

```
grid_mapping_name = sinusoidal
```

Map parameters:

- `longitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Notes on using the `PROJ` software package for computing the mapping may be found at https://proj.org/operations/projections/sinu.html and http://geotiff.maptools.org/proj_list/sinusoidal.html. Detailed formulas can be found in [Snyder], pages 243-248.

Stereographic

```
grid_mapping_name = stereographic
```

Map parameters:

- `longitude_of_projection_origin`
- `latitude_of_projection_origin`

- `scale_factor_at_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Formulas for the mapping and its inverse along with notes on using the `PROJ` software package for doing the calcuations may be found at https://proj.org/operations/projections/stere.html and http://geotiff.maptools.org/proj_list/stereographic.html. See the section "Polar stereographic" for the special case when the projection origin is one of the poles.

Transverse Mercator

```
grid_mapping_name = transverse_mercator
```

Map parameters:

- `scale_factor_at_central_meridian`
- `longitude_of_central_meridian`
- `latitude_of_projection_origin`
- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute values `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

Formulas for the mapping and its inverse along with notes on using the `PROJ` software package for doing the calcuations may be found at https://proj.org/operations/projections/tmerc.html and http://geotiff.maptools.org/proj_list/transverse_mercator.html.

Vertical perspective

```
grid_mapping_name = vertical_perspective
```

Map parameters:

- `latitude_of_projection_origin`
- `longitude_of_projection_origin`
- `perspective_point_height`

- `false_easting` - This parameter is optional (default is 0)
- `false_northing` - This parameter is optional (default is 0)

Map coordinates:

The x (abscissa) and y (ordinate) rectangular coordinates are identified by the `standard_name` attribute value `projection_x_coordinate` and `projection_y_coordinate` respectively.

Notes:

A general description of vertical perspective projection is given in [Snyder], pages 169-181.

The corresponding projection in PROJ is nsper. This should not be confused with the PROJ geos projection.

In the following table the "Type" values are **S** for string and **N** for numeric.

Table F.1. Grid Mapping Attributes

Attribute	Type	Description
`azimuth_of_central_line`	N	Specifies a horizontal angle measured in degrees clockwise from North. Used by certain projections (e.g., Oblique Mercator) to define the orientation of the map projection relative to a reference direction.
`crs_wkt`	S	This optional attribute may be used to specify multiple coordinate system properties in well-known text (WKT) format. The syntax must conform to the WKT format as specified in reference [OGC_WKT-CRS]. Use of the `crs_wkt` attribute is described in section 5.6.1.
`earth_radius`	N	Used to specify the radius, in metres, of the spherical figure used to approximate the shape of the Earth. This attribute should be specified for those projected coordinate reference systems in which the X-Y cartesian coordinates have been derived using a spherical Earth approximation. If the cartesian coordinates were derived using an ellipsoid, this attribute should not be defined. Example: "6371007", which is the radius of the GRS 1980 Authalic Sphere.
`false_easting`	N	Applied to all abscissa values in the rectangular coordinates for a map projection in order to eliminate negative numbers. Expressed in the unit of the coordinate variable identified by the standard name `projection_x_coordinate`. If `false_easting` is not provided it is assumed to be 0. The formula to convert from the coordinate value as written in the `projection_x_coordinate` (xf) to a value (x0) used in a transformation without `false_easting`, i.e. `false_easting`= 0, is: x0 = xf -`false_easting`

Attribute	Type	Description
false_northing	N	Applied to all ordinate values in the rectangular coordinates for a map projection in order to eliminate negative numbers. Expressed in the unit of the coordinate variable identified by the standard name projection_y_coordinate. If false_northing is not provided it is assumed to be 0. The formula to convert from the coordinate value as written in the projection_y_coordinate (yf) to a value (y0) used in a transformation without false_northing, i.e. false_northing= 0, is: y0 = yf -false_northing
fixed_angle_axis	S	Indicates the axis on which the view is fixed in a hypothetical gimbal view model of the Earth, as used in the geostationary grid mapping. It corresponds to the inner-gimbal axis of the gimbal view model, whose axis of rotation moves about the outer-gimbal axis. This value can adopt two values, "x" or "y", corresponding with the Earth's E-W or N-S axis, respectively. The counterpart to this attribute is sweep_angle_axis. If set to "x", sweep_angle_axis is "y", and vice versa. If one of the attributes fixed_angle_axis or sweep_angle_axis is provided, the other is not mandatory, as they can be used to infer each other.
geographic_crs_name	S	The name of the geographic coordinate reference system. Corresponds to a OGC WKT GEOGCS node name.
geoid_name	S	The name of the estimate or model of the geoid being used as a datum, e.g. GEOID12B. Corresponds to an OGC WKT VERT_DATUM name. The geoid is the surface of constant geopotential that the ocean would follow if it were at rest. This attribute and geopotential_datum_name cannot both be specified.
geopotential_datum_name	S	The name of an estimated surface of constant geopotential being used as a datum, e.g. NAVD88. Such a surface is often called an equipotential surface in geodesy. Corresponds to an OGC WKT VERT_DATUM name. This attribute and geoid_name cannot both be specified.
grid_mapping_name	S	The name used to identify the grid mapping.
grid_north_pole_latitude	N	True latitude (degrees_north) of the north pole of the rotated grid.
grid_north_pole_longitude	N	True longitude (degrees_east) of the north pole of the rotated grid.

Attribute	Type	Description
horizontal_datum_name	S	The name of the geodetic (horizontal) datum, which corresponds to the procedure used to measure positions on the surface of the Earth. Valid datum names and their associated parameters are given in horiz_datum.csv (OGC_DATUM_NAME column), following Table 2 in [CF-WKT]. The valid names are obtained by transforming the EPSG name using the following rules (used by OGR and Cadcorp): convert all non alphanumeric characters (including +) to underscores, then strip any leading, trailing or repeating underscores. This is to ensure that named datums can be correctly identified for precise datum transformations (see [CF-WKT] for more details). Corresponds to a OGC WKT DATUM node name.
indexing_scheme	S	The HEALPix cell indexing scheme, either nested, ring, nuniq, or zuniq. See HEALPix for more detailed definitions.
inverse_flattening	N	Used to specify the *inverse* flattening (*1/f*) of the ellipsoidal figure associated with the geodetic datum and used to approximate the shape of the Earth. The flattening (*f*) of the ellipsoid is related to the semi-major and semi-minor axes by the formula *f = (a-b)/a*. In the case of a spherical Earth this attribute should be omitted or set to zero. Example: 298.257222101 for the GRS 1980 ellipsoid. (Note: By convention the dimensions of an ellipsoid are specified using either the semi-major and semi-minor axis lengths, or the semi-major axis length and the inverse flattening. If all three attributes are specified then the supplied values must be consistent with the aforementioned formula.)
latitude_of_projection_origin	N	The latitude (degrees_north) chosen as the origin of rectangular coordinates for a map projection. Domain: `-90.0 <= latitude_of_projection_origin <= 90.0`
refinement_level	N	The refinement level of the grid within the HEALPix hierarchy, starting at 0 for the base tessellation with 12 cells globally, 1 for the next level with 48 cells globally, etc.
longitude_of_central_meridian	N	The line of longitude (degrees_east) at the center of a map projection generally used as the basis for constructing the projection. Domain: `-180.0 <= longitude_of_central_meridian < 180.0`
longitude_of_prime_meridian	N	Specifies the longitude, with respect to Greenwich, of the prime meridian associated with the geodetic datum. The prime meridian defines the origin from which longitude values are determined. Not to be confused with the projection origin longitude (cf. longitude_of_projection_origin, a.k.a. central meridian) which defines the longitude of the map projection origin. Domain: + `-180.0 <= longitude_of_prime_meridian < 180.0` decimal degrees. Default = `0.0`

Attribute	Type	Description
longitude_of_projection_origin	N	The longitude (degrees_east) chosen as the origin of rectangular coordinates for a map projection. Domain: + -180.0 <= longitude_of_projection_origin < 180.0
north_pole_grid_longitude	N	Longitude (degrees) of the true north pole in the rotated grid.
perspective_point_height	N	Records the height, *in metres*, of the map projection perspective point above the ellipsoid (or sphere). Used by perspective-type map projections, for example the Vertical Perspective Projection, which may be used to simulate the view from a Meteosat satellite.
prime_meridian_name	S	The name of the prime meridian associated with the geodetic datum. Valid names are given in prime_meridian.csv, following Table 2 in [CF-WKT]. Corresponds to a OGC WKT PRIMEM node name.
projected_crs_name	S	The name of the projected coordinate reference system. Corresponds to a OGC WKT PROJCS node name.
reference_ellipsoid_name	S	The name of the reference ellipsoid. Valid names are given in ellipsoid.csv, following Table 2 in [CF-WKT]. Corresponds to a OGC WKT SPHEROID node name.
scale_factor_at_central_meridian	N	A multiplier for reducing a distance obtained from a map by computation or scaling to the actual distance along the central meridian. Domain: scale_factor_at_central_meridian > 0.0
scale_factor_at_projection_origin	N	A multiplier for reducing a distance obtained from a map by computation or scaling to the actual distance at the projection origin. Domain: scale_factor_at_projection_origin > 0.0
semi_major_axis	N	Specifies the length, *in metres*, of the semi-major axis of the ellipsoidal figure associated with the geodetic datum and used to approximate the shape of the Earth. Commonly denoted using the symbol *a*. In the case of a spherical Earth approximation this attribute defines the radius of the Earth. See also the inverse_flattening attribute.
semi_minor_axis	N	Specifies the length, *in metres*, of the semi-minor axis of the ellipsoidal figure associated with the geodetic datum and used to approximate the shape of the Earth. Commonly denoted using the symbol *b*. In the case of a spherical Earth approximation this attribute should be omitted (the preferred option) or else set equal to the value of the semi_major_axis attribute. See also the inverse_flattening attribute.

Attribute	Type	Description
standard_parallel	N	Specifies the line, or lines, of latitude at which the developable map projection surface (plane, cone, or cylinder) touches the reference sphere or ellipsoid used to represent the Earth. Since there is zero scale distortion along a standard parallel it is also referred to as a "latitude of true scale". In the situation where a conical developable surface intersects the reference ellipsoid there are two standard parallels, in which case this attribute can be used as a vector to record both latitude values, with the additional convention that the standard parallel nearest the pole (N or S) is provided first. Domain: `-90.0 <= standard_parallel <= 90.0`
straight_vertical_longitude_from_pole	N	*Deprecated*. Has the same meaning as longitude_of_projection_origin, which must be used instead. Domain: `-180.0 <= straight_vertical_longitude_from_pole < 180.0`
sweep_angle_axis	S	Indicates the axis on which the view sweeps in a hypothetical gimbal view model of the Earth, as used in the geostationary grid mapping. It corresponds to the outer-gimbal axis of the gimbal view model, about whose axis of rotation the gimbal-gimbal axis moves. This value can adopt two values, "x" or "y", corresponding with the Earth's E-W or N-S axis, respectively. The counterpart to this attribute is fixed_angle_axis. If set to "x", fixed_angle_axis is "y", and vice versa. If one of the attributes fixed_angle_axis or sweep_angle_axis is provided, the other is not mandatory, as they can be used to infer each other.
towgs84	N	This indicates a list of up to 7 Bursa Wolf transformation parameters., which can be used to approximate a transformation from the horizontal datum to the WGS84 datum. More precise datum transformations can be done with datum shift grids. Represented as a double-precision array, with 3, 6 or 7 values (if there are less than 7 values the remaining are considered to be zero). Corresponds to a OGC WKT TOWGS84 node.

Notes:

1. The various *_name attributes are optional but recommended when known as they allow for a better description and interoperability with WKT definitions.
2. reference_ellipsoid_name, prime_meridian_name, horizontal_datum_name and geographic_crs_name must be all defined if any one is defined, and if projected_crs_name is defined then geographic_crs_name must be also.

Appendix G: Revision History

The content in this appendix has moved to Revision History.

Appendix H: Annotated Examples of Discrete Geometries

H.1. Point Data

To represent data at scattered locations and times with no implied relationship among of coordinate positions, both data and coordinates must share the same (sample) instance dimension. Because each feature contains only a single data element, there is no need for a separate element dimension. The representation of point features is a special, degenerate case of the standard four representations. The `coordinates` attribute is used on the data variables to unambiguously identify the relevant space and time auxiliary coordinate variables.

Example H.1. Point data.

```
dimensions:
   obs = 1234 ;

variables:
   double time(obs) ;
       time:standard_name = "time";
       time:long_name = "time of measurement" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float lon(obs) ;
       lon:standard_name = "longitude";
       lon:long_name = "longitude of the observation";
       lon:units = "degrees_east";
   float lat(obs) ;
       lat:standard_name = "latitude";
       lat:long_name = "latitude of the observation" ;
       lat:units = "degrees_north" ;
   float alt(obs) ;
       alt:long_name = "vertical distance above the surface" ;
       alt:standard_name = "height" ;
       alt:units = "m";
       alt:positive = "up";
       alt:axis = "Z";

   float humidity(obs) ;
       humidity:standard_name = "specific_humidity" ;
       humidity:coordinates = "time lat lon alt" ;
   float temp(obs) ;
       temp:standard_name = "air_temperature" ;
       temp:units = "Celsius" ;
       temp:coordinates = "time lat lon alt" ;

attributes:
   :featureType = "point";
```

In this example, the `humidity(i)` and `temp(i)` data are associated with the coordinate values `time(i)`, `lat(i)`, `lon(i)`, and `alt(i)`. The `obs` dimension may optionally be the netCDF unlimited dimension of the netCDF file.

H.2. Time Series Data

Data may be taken over periods of time at a set of discrete point, spatial locations called stations (see also discussion in 9.1). The set of elements at a particular station is referred to as a timeSeries feature and a data variable may contain a collection of such features. The instance dimension in the case of timeSeries specifies the number of time series in the collection and is also referred to as the station dimension. The instance variables, which have just this dimension, including latitude and longitude for example, are also referred to as station variables and are considered to contain information describing the stations. The station variables may contain missing values, allowing one to reserve space for additional stations that may be added at a later time, as discussed in section 9.6. In addition,

- It is strongly recommended that there should be a station variable (which may be of any type) with the attribute `cf_role=timeseries_id`, whose values uniquely identify the stations.
- It is recommended that there should be station variables with standard_name attributes "`platform_name`", "`surface_altitude`" and `platform_id` when applicable.

All the representations described in section 9.3 can be used for time series. The global attribute `featureType=timeSeries` (case-insensitive) must be included.

H.2.1. Orthogonal multidimensional array representation of time series

If the time series instances have the same number of elements and the time values are identical for all instances, you may use the orthogonal multidimensional array representation. This has either a one-dimensional coordinate variable, time(time), provided the time values are in strict monotonically increasing order, or a one-dimensional auxiliary coordinate variable, time(o), where o is the element dimension. In the former case, listing the time variable in the **`coordinates`** attributes of the data variables is optional.

Example H.2. Timeseries with common element times in a time coordinate variable using the orthogonal multidimensional array representation.

```
dimensions:
  station = 10 ;  // measurement locations
  time = UNLIMITED ;

variables:
  float humidity(station,time) ;
    humidity:standard_name = "specific humidity" ;
    humidity:coordinates = "lat lon alt station_name" ;
    humidity:_FillValue = -999.9f;
  double time(time) ;
    time:standard_name = "time";
```

```
      time:long_name = "time of measurement" ;
      time:units = "days since 1970-01-01 00:00:00" ;
    float lon(station) ;
      lon:standard_name = "longitude";
      lon:long_name = "station longitude";
      lon:units = "degrees_east";
    float lat(station) ;
      lat:standard_name = "latitude";
      lat:long_name = "station latitude" ;
      lat:units = "degrees_north" ;
    float alt(station) ;
      alt:long_name = "vertical distance above the surface" ;
      alt:standard_name = "height" ;
      alt:units = "m";
      alt:positive = "up";
      alt:axis = "Z";
    string station_name(station) ;
      station_name:long_name = "station name" ;
      station_name:cf_role = "timeseries_id";
  attributes:
      :featureType = "timeSeries";
```

In this example, `humidity(i,o)` is element `o` of time series `i`, and associated with the coordinate values `time(o)`, `lat(i)`, and `lon(i)`. Either the instance (`station`) or the element (`time`) dimension may optionally be the netCDF unlimited dimension.

H.2.2. Incomplete multidimensional array representation of time series

Much of the simplicity of the orthogonal multidimensional representation can be preserved even in cases where individual time series have different time coordinate values. All time series must be allocated the amount of storage needed by the longest, so the use of this representation will trade off simplicity against storage space in some cases.

Example H.3. Timeseries of station data in the incomplete multidimensional array representation.

```
dimensions:
   station = UNLIMITED ;
   obs = 13 ;
   name_strlen = 23 ;

variables:
   float lon(station) ;
       lon:standard_name = "longitude";
       lon:long_name = "station longitude";
       lon:units = "degrees_east";
   float lat(station) ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
```

```
            lat:units = "degrees_north" ;
        float alt(station) ;
            alt:long_name = "vertical distance above the surface" ;
            alt:standard_name = "height" ;
            alt:units = "m";
            alt:positive = "up";
            alt:axis = "Z";
        char station_name(station, name_strlen) ;
            station_name:long_name = "station name" ;
            station_name:cf_role = "timeseries_id";
        int station_info(station) ;
            station_info:long_name = "any kind of station info" ;
        float station_elevation(station) ;
            station_elevationalt:long_name = "height above the geoid" ;
            station_elevationalt:standard_name = "surface_altitude" ;
            station_elevationalt:units = "m";

        double time(station, obs) ;
            time:standard_name = "time";
            time:long_name = "time of measurement" ;
            time:units = "days since 1970-01-01 00:00:00" ;
            time:missing_value = -999.9;
        float humidity(station, obs) ;
            humidity:standard_name = "specific_humidity" ;
            humidity:coordinates = "time lat lon alt station_name" ;
            humidity:_FillValue = -999.9f;
        float temp(station, obs) ;
            temp:standard_name = "air_temperature" ;
            temp:units = "Celsius" ;
            temp:coordinates = "time lat lon alt station_name" ;
            temp:_FillValue = -999.9f;

    attributes:
            :featureType = "timeSeries";
```

In this example, the `humidity(i,o)` and `temp(i,o)` data for element `o` of time series `i` are associated with the coordinate values `time(i,o)`, `lat(i)`, `lon(i)` and `alt(i)`. Either the instance (`station`) dimension or the element (`obs`) dimension could be the unlimited dimension of a netCDF file. Any unused elements of the data and auxiliary coordinate variables must contain the missing data flag value(section 9.6).

H.2.3. Single time series, including deviations from a nominal fixed spatial location

When the intention of a data variable is to contain only a single time series, the preferred encoding is a special case of the multidimensional array representation.

Example H.4. A single timeseries.

```
dimensions:
   time = 100233 ;

variables:
   float lon ;
       lon:standard_name = "longitude";
       lon:long_name = "station longitude";
       lon:units = "degrees_east";
   float lat ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;
   float alt ;
       alt:long_name = "vertical distance above the surface" ;
       alt:standard_name = "height" ;
       alt:units = "m";
       alt:positive = "up";
       alt:axis = "Z";
   string station_name ;
       station_name:long_name = "station name" ;
       station_name:cf_role = "timeseries_id";

   double time(time) ;
       time:standard_name = "time";
       time:long_name = "time of measurement" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float humidity(time) ;
       humidity:standard_name = specific_humidity ;
       humidity:coordinates = "time lat lon alt station_name" ;
       humidity:_FillValue = -999.9f;
   float temp(time) ;
       temp:standard_name = air_temperature ;
       temp:units = "Celsius" ;
       temp:coordinates = "time lat lon alt station_name" ;
       temp:_FillValue = -999.9f;

attributes:
       :featureType = "timeSeries";
```

While an idealized time series is defined at a single, stable point location, there are examples of time series, such as cabled ocean surface mooring measurements, in which the precise position of the observations varies slightly from a nominal fixed point. It is quite common that the deployment position of a station changes after maintenance or repositioning after it drifts. In the following example we show how the spatial positions of such a time series should be encoded in CF. In addition, this example shows how lossless compression by gathering Section 8.2, "Lossless Compression by Gathering" has been applied to the deployment coordinate variables, which

otherwise would contain a lot of missing or repetitive data. Note that although this example shows only a single time series, the technique is applicable to all of the representations.

Example H.5. A single timeseries with time-varying deviations from a nominal point spatial location

```
dimensions:
   time = 100233 ;
   name_strlen = 23 ;
   deployment = 5 ;

variables:
   float lon ;
       lon:standard_name = "longitude";
       lon:long_name = "station longitude";
       lon:units = "degrees_east";
       lon:axis = "X";
   float lat ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;
       lat: axis = "Y" ;
   float precise_lon (time);
       precise_lon:standard_name = "longitude";
       precise_lon:long_name = "station longitude";
       precise_lon:units = "degrees_east";
   float precise_lat (time);
       precise_lat:standard_name = "latitude";
       precise_lat:long_name = "station latitude" ;
       precise_lat:units = "degrees_north" ;
   float deploy_lon (deployment);
       deploy_lon:standard_name = "deployment_longitude";
       deploy_lon:long_name = station longitude";
       deploy_lon:units = "degrees_east";
   float deploy_lat (deployment);
       deploy_lat:standard_name = "deployment_latitude";
       deploy_lat:long_name = station latitude";
       deploy_lat:units = "degrees_north";
   int deployment (deployment) ;
       deployment:long_name = "index of the first time after (re)deployment" ;
       deployment:compress="time";
   float alt ;
       alt:long_name = "vertical distance above the surface" ;
       alt:standard_name = "height" ;
       alt:units = "m";
       alt:positive = "up";
       alt:axis = "Z";
   char station_name(name_strlen) ;
       station_name:long_name = "station name" ;
       station_name:cf_role = "timeseries_id";
   double time(time) ;
```

```
          time:standard_name = "time";
          time:long_name = "time of measurement" ;
          time:units = "days since 1970-01-01 00:00:00" ;
      float humidity(time) ;
          humidity:standard_name = "specific_humidity" ;
          humidity:coordinates = "time lat lon alt precise_lon precise_lat
deploy_lon deploy_lat station_name" ;
          humidity:_FillValue = -999.9f;
      float temp(time) ;
          temp:standard_name = "air_temperature" ;
          temp:units = "Celsius" ;
          temp:coordinates = "time lat lon alt precise_lon precise_lat deploy_lon
deploy_lat station_name" ;
          temp:_FillValue = -999.9f;

   attributes:
          :featureType = "timeSeries";
```

H.2.4. Contiguous ragged array representation of time series

When the time series have different lengths and the data values for entire time series are available to be written in a single operation, the contiguous ragged array representation is efficient.

Example H.6. Timeseries of station data in the contiguous ragged array representation.

```
   dimensions:
      station = 23 ;
      obs = 1234 ;

   variables:
      float lon(station) ;
          lon:standard_name = "longitude";
          lon:long_name = "station longitude";
          lon:units = "degrees_east";
      float lat(station) ;
          lat:standard_name = "latitude";
          lat:long_name = "station latitude" ;
          lat:units = "degrees_north" ;
      float alt(station) ;
          alt:long_name = "vertical distance above the surface" ;
          alt:standard_name = "height" ;
          alt:units = "m";
          alt:positive = "up";
          alt:axis = "Z";
      string station_name(station) ;
          station_name:long_name = "station name" ;
          station_name:cf_role = "timeseries_id";
      int station_info(station) ;
          station_info:long_name = "some kind of station info" ;
```

```
      int row_size(station) ;
          row_size:long_name = "number of observations for this station " ;
          row_size:sample_dimension = "obs" ;

      double time(obs) ;
          time:standard_name = "time";
          time:long_name = "time of measurement" ;
          time:units = "days since 1970-01-01 00:00:00" ;
      float humidity(obs) ;
          humidity:standard_name = specific_humidity ;
          humidity:coordinates = "time lat lon alt station_name" ;
          humidity:_FillValue = -999.9f;
      float temp(obs) ;
          temp:standard_name = air_temperature ;
          temp:units = "Celsius" ;
          temp:coordinates = "time lat lon alt station_name" ;
          temp:_FillValue = -999.9f;

   attributes:
          :featureType = "timeSeries";
```

The data `humidity(o)` and `temp(o)` are associated with the coordinate values `time(o)`, `lat(i)`, `lon(i)`, and `alt(i)`, where `i` indicates which time series. Time series `i` comprises the data elements from

```
   rowStart(i) to rowStart(i) + row_size(i) - 1
```

where

```
      rowStart(i) = 0 if i = 0
      rowStart(i) = rowStart(i-1) + row_size(i-1) if i > 0
```

The variable, `row_size`, is the count variable containing the length of each time series feature. It is identified by having an attribute with name `sample_dimension` whose value is name of the sample dimension (`obs` in this example). The sample dimension could optionally be the netCDF unlimited dimension. The variable bearing the `sample_dimension` attribute must have the instance dimension (`station` in this example) as its single dimension, and must have an integer type. This variable implicitly partitions into individual instances all variables that have the sample dimension. The auxiliary coordinate variables `lat`, `lon`, `alt` and `station_name` are station variables.

H.2.5. Indexed ragged array representation of time series

When time series with different lengths are written incrementally, the indexed ragged array representation is efficient.

Example H.7. Timeseries of station data in the indexed ragged array representation.

```
dimensions:
   station = 23 ;
   obs = UNLIMITED ;
   name_strlen = 23 ;

variables:
   float lon(station) ;
       lon:standard_name = "longitude";
       lon:long_name = "station longitude";
       lon:units = "degrees_east";
   float lat(station) ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;
   float alt(station) ;
       alt:long_name = "vertical distance above the surface" ;
       alt:standard_name = "height" ;
       alt:units = "m";
       alt:positive = "up";
       alt:axis = "Z";
   char station_name(station, name_strlen) ;
       station_name:long_name = "station name" ;
       station_name:cf_role = "timeseries_id";
   int station_info(station) ;
       station_info:long_name = "some kind of station info" ;

   int stationIndex(obs) ;
       stationIndex:long_name = "which station this obs is for" ;
       stationIndex:instance_dimension= "station" ;
   double time(obs) ;
       time:standard_name = "time";
       time:long_name = "time of measurement" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float humidity(obs) ;
       humidity:standard_name = specific_humidity ;
       humidity:coordinates = "time lat lon alt station_name" ;
       humidity:_FillValue = -999.9f;
   float temp(obs) ;
       temp:standard_name = air_temperature ;
       temp:units = "Celsius" ;
       temp:coordinates = "time lat lon alt station_name" ;
       temp:_FillValue = -999.9f;

attributes:
       :featureType = "timeSeries";
```

The humidity(o) and temp(o) data are associated with the coordinate values time(o), lat(i),

lon(i), and alt(i), where i = stationIndex(o) is a zero-based index indicating which time series. Thus, time(0), humidity(0) and temp(0) belong to the element of the station dimension that is indicated by stationIndex(0) ; time(1), humidity(1) and temp(1) belong to element stationIndex(1) of the station dimension, etc.

The variable, stationIndex, is identified as the index variable by having an attribute with name of instance_dimension whose value is the instance dimension (station in this example). The variable bearing the instance_dimension attribute must have the sample dimension (obs in this example) as its single dimension, and must have an integer type. This variable implicitly assigns the station to each value of any variable having the sample dimension. The sample dimension need not be the netCDF unlimited dimension, though it commonly is.

H.3. Profile Data

A series of connected observations along a vertical line, like an atmospheric or ocean sounding, is called a profile. For each profile, there is a single time, lat and lon. A data variable may contain a collection of profile features. The instance dimension in the case of profiles specifies the number of profiles in the collection and is also referred to as the **profile dimension**. The instance variables, which have just this dimension, including latitude and longitude for example, are also referred to as **profile variables** and are considered to be information about the profiles. It is strongly recommended that there always be a profile variable (of any data type) with cf_role attribute " profile_id ", whose values uniquely identify the profiles. The profile variables may contain missing values. This allows one to reserve space for additional profiles that may be added at a later time, as discussed in section 9.6. All the representations described in section 9.1.3 can be used for profiles. The global attribute featureType="profile" (case-insensitive) should be included if all data variables in the file contain profiles.

H.3.1. Orthogonal multidimensional array representation of profiles

If the profile instances have the same number of elements and the vertical coordinate values are identical for all instances, you may use the orthogonal multidimensional array representation. This has either a one-dimensional coordinate variable, z(z), provided the vertical coordinate values are in strict monotonic order, or a one-dimensional auxiliary coordinate variable, alt(o), where o is the element dimension. In the former case, listing the vertical coordinate variable in the **coordinates** attributes of the data variables is optional.

Example H.8. Atmospheric sounding profiles for a common set of vertical coordinates stored in the orthogonal multidimensional array representation.

```
dimensions:
   z = 42 ;
   profile = 142 ;

variables:
   int profile(profile) ;
         profile:cf_role = "profile_id";
   double time(profile);
```

```
        time:standard_name = "time";
        time:long_name = "time" ;
        time:units = "days since 1970-01-01 00:00:00" ;
    float lon(profile);
        lon:standard_name = "longitude";
        lon:long_name = "longitude" ;
        lon:units = "degrees_east" ;
    float lat(profile);
        lat:standard_name = "latitude";
        lat:long_name = "latitude" ;
        lat:units = "degrees_north" ;

    float z(z) ;
        z:standard_name = "altitude";
        z:long_name = "height above mean sea level" ;
        z:units = "km" ;
        z:positive = "up" ;
        z:axis = "Z" ;

    float pressure(profile, z) ;
        pressure:standard_name = "air_pressure" ;
        pressure:long_name = "pressure level" ;
        pressure:units = "hPa" ;
        pressure:coordinates = "time lon lat z" ;

    float temperature(profile, z) ;
        temperature:standard_name = "surface_temperature" ;
        temperature:long_name = "skin temperature" ;
        temperature:units = "Celsius" ;
        temperature:coordinates = "time lon lat z" ;

    float humidity(profile, z) ;
        humidity:standard_name = "relative_humidity" ;
        humidity:long_name = "relative humidity" ;
        humidity:units = "%" ;
        humidity:coordinates = "time lon lat z" ;

attributes:
    :featureType = "profile";
```

The `pressure(i,o)`, `temperature(i,o)`, and `humidity(i,o)` data for element `o` of profile `i` are associated with the coordinate values `time(i)`, `lat(i)`, and `lon(i)`. The vertical coordinate for element `o` in each profile is altitude `z(o)`. Either the instance (`profile`) or the element (`z`) dimension could be the netCDF unlimited dimension.

H.3.2. Incomplete multidimensional array representation of profiles

If there are the same number of levels in each profile, but they do not have the same set of vertical coordinates, one can use the incomplete multidimensional array representation, which the vertical

coordinate variable is two-dimensional e.g. replacing `z(z)` in "Atmospheric sounding profiles for a common set of vertical coordinates stored in the orthogonal multidimensional array representation." with `alt(profile,z)`. This representation also allows one to have a variable number of elements in different profiles, at the cost of some wasted space. In that case, any unused elements of the data and auxiliary coordinate variables must contain missing data values (section 9.6).

H.3.3. Single profile

When a single profile is stored in a file, there is no need for the profile dimension; the data arrays are one-dimensional. This is a special case of the orthogonal multidimensional array representation (9.3.1).

Example H.9. Data from a single atmospheric sounding profile.

```
dimensions:
   z = 42 ;

variables:
   int profile ;
       profile:cf_role = "profile_id";

   double time;
       time:standard_name = "time";
       time:long_name = "time" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float lon;
       lon:standard_name = "longitude";
       lon:long_name = "longitude" ;
       lon:units = "degrees_east" ;
   float lat;
       lat:standard_name = "latitude";
       lat:long_name = "latitude" ;
       lat:units = "degrees_north" ;

   float z(z) ;
       z:standard_name = "altitude";
       z:long_name = "height above mean sea level" ;
       z:units = "km" ;
       z:positive = "up" ;
       z:axis = "Z" ;

   float pressure(z) ;
       pressure:standard_name = "air_pressure" ;
       pressure:long_name = "pressure level" ;
       pressure:units = "hPa" ;
       pressure:coordinates = "time lon lat z" ;

   float temperature(z) ;
       temperature:standard_name = "air_temperature" ;
```

```
        temperature:units = "degree_celsius" ;
        temperature:coordinates = "time lon lat z" ;

    float humidity(z) ;
        humidity:standard_name = "relative_humidity" ;
        humidity:long_name = "relative humidity" ;
        humidity:units = "%" ;
        humidity:coordinates = "time lon lat z" ;

attributes:
    :featureType = "profile";
```

The `pressure(o)`, `temperature(o)`, and `humidity(o)` data is associated with the coordinate values `time`, `z(o)`, `lat`, and `lon`. The profile variables `time`, `lat` and `lon`, shown here as scalar, could alternatively be one-dimensional `time(profile)`, `lat(profile)`, `lon(profile)` if a size-one profile dimension were retained in the file.

H.3.4. Contiguous ragged array representation of profiles

When the number of vertical levels for each profile varies, and one can control the order of writing, one can use the contiguous ragged array representation. The canonical use case for this is when rewriting raw data, and you expect that the common read pattern will be to read all the data from each profile.

Example H.10. Atmospheric sounding profiles for a common set of vertical coordinates stored in the contiguous ragged array representation.

```
dimensions:
    obs = UNLIMITED ;
    profile = 142 ;

variables:
    int profile(profile) ;
        profile:cf_role = "profile_id";
    double time(profile);
        time:standard_name = "time";
        time:long_name = "time" ;
        time:units = "days since 1970-01-01 00:00:00" ;
    float lon(profile);
        lon:standard_name = "longitude";
        lon:long_name = "longitude" ;
        lon:units = "degrees_east" ;
    float lat(profile);
        lat:standard_name = "latitude";
        lat:long_name = "latitude" ;
        lat:units = "degrees_north" ;
     int rowSize(profile) ;
        rowSize:long_name = "number of obs for this profile " ;
```

```
          rowSize:sample_dimension = "obs" ;

       float z(obs) ;
           z:standard_name = altitude;
           z:long_name = "height above mean sea level" ;
           z:units = "km" ;
           z:positive = "up" ;
           z:axis = "Z" ;

       float pressure(obs) ;
           pressure:standard_name = "air_pressure" ;
           pressure:long_name = "pressure level" ;
           pressure:units = "hPa" ;
           pressure:coordinates = "time lon lat z" ;

       float temperature(obs) ;
           temperature:standard_name = "air_temperature" ;
           temperature:units = "degree_celsius" ;
           temperature:coordinates = "time lon lat z" ;

       float humidity(obs) ;
           humidity:standard_name = "relative_humidity" ;
           humidity:long_name = "relative humidity" ;
           humidity:units = "%" ;
           humidity:coordinates = "time lon lat z" ;

   attributes:
      :featureType = "profile";
```

The `pressure(o)`, `temperature(o)`, and `humidity(o)` data is associated with the coordinate values `time(i)`, `z(o)`, `lat(i)`, and `lon(i)`, where `i` indicates which profile. All elements for one profile are contiguous along the sample dimension. The sample dimension (`obs`) may be the unlimited dimension or not. All variables that have the instance dimension (`profile`) as their single dimension are considered to be information about the profiles.

The count variable (`row_size`) contains the number of elements for each profile, and is identified by having an attribute with name `sample_dimension` whose value is the sample dimension being counted. It must have the profile dimension as its single dimension, and must have an integer type. The elements are associated with the profile using the same algorithm as in H.2.4.

H.3.5. Indexed ragged array representation of profiles

When the number of vertical levels for each profile varies, and one cannot write them contiguously, one can use the indexed ragged array representation. The canonical use case is when writing real-time data streams that contain reports from many profiles, arriving randomly. If the sample dimension is the unlimited dimension, this allows data to be appended to the file.

Example H.11. Atmospheric sounding profiles for a common set of vertical coordinates stored in the indexed ragged array representation.

```
dimensions:
   obs = UNLIMITED ;
   profile = 142 ;

variables:
   int profile(profile) ;
       profile:cf_role = "profile_id";
   double time(profile);
       time:standard_name = "time";
       time:long_name = "time" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float lon(profile);
       lon:standard_name = "longitude";
       lon:long_name = "longitude" ;
       lon:units = "degrees_east" ;
   float lat(profile);
       lat:standard_name = "latitude";
       lat:long_name = "latitude" ;
       lat:units = "degrees_north" ;

   int parentIndex(obs) ;
       parentIndex:long_name = "index of profile " ;
       parentIndex:instance_dimension= "profile" ;

    float z(obs) ;
       z:standard_name = "altitude";
       z:long_name = "height above mean sea level" ;
       z:units = "km" ;
       z:positive = "up" ;
       z:axis = "Z" ;

   float pressure(obs) ;
       pressure:standard_name = "air_pressure" ;
       pressure:long_name = "pressure level" ;
       pressure:units = "hPa" ;
       pressure:coordinates = "time lon lat z" ;

   float temperature(obs) ;
       temperature:standard_name = "air_temperature" ;
       temperature:units = "degree_celsius" ;
       temperature:coordinates = "time lon lat z" ;

   float humidity(obs) ;
       humidity:standard_name = "relative_humidity" ;
       humidity:long_name = "relative humidity" ;
       humidity:units = "%" ;
       humidity:coordinates = "time lon lat z" ;
```

```
attributes:
   :featureType = "profile";
```

The `pressure(o)`, `temperature(o)`, and `humidity(o)` data are associated with the coordinate values `time(i)`, `z(o)`, `lat(i)`, and `lon(i)`, where `i` indicates which profile. The sample dimension (`obs`) may be the unlimited dimension or not. The profile index variable (`parentIndex`) is identified by having an attribute with name of `instance_dimension` whose value is the profile dimension name. It must have the sample dimension as its single dimension, and must have an integer type. Each value in the profile index variable is the zero-based profile index that the element belongs to. The elements are associated with the profiles using the same algorithm as in H.2.5.

H.4. Trajectory Data

Data may be taken along discrete paths through space, each path constituting a connected set of points called a trajectory, for example along a flight path, a ship path or the path of a parcel in a Lagrangian calculation. A data variable may contain a collection of trajectory features. The instance dimension in the case of trajectories specifies the number of trajectories in the collection and is also referred to as the **trajectory dimension**. The instance variables, which have just this dimension, are also referred to as **trajectory variables** and are considered to be information about the trajectories. It is strongly recommended that there always be a trajectory variable (of any data type) with the attribute `cf_role=trajectory_id` attribute, whose values uniquely identify the trajectories. The trajectory variables may contain missing values. This allows one to reserve space for additional trajectories that may be added at a later time, as discussed in section 9.6. All the representations described in section 9.3 can be used for trajectories. The global attribute `featureType=trajectory` (case-insensitive) should be included if all data variables in the file contain trajectories.

H.4.1. Multidimensional array representation of trajectories

When storing multiple trajectories in the same file, and the number of elements in each trajectory is the same, one can use the multidimensional array representation. This representation also allows one to have a variable number of elements in different trajectories, at the cost of some wasted space. In that case, any unused elements of the data and auxiliary coordinate variables must contain missing data values (section 9.6).

Example H.12. Trajectories recording atmospheric composition in the incomplete multidimensional array representation.

```
dimensions:
   obs = 1000 ;
   trajectory = 77 ;

variables:
   string trajectory(trajectory) ;
       trajectory:cf_role = "trajectory_id";
       trajectory:long_name = "trajectory name" ;
```

```
      int trajectory_info(trajectory) ;
          trajectory_info:long_name = "some kind of trajectory info"

      double time(trajectory, obs) ;
          time:standard_name = "time";
          time:long_name = "time" ;
          time:units = "days since 1970-01-01 00:00:00" ;
      float lon(trajectory, obs) ;
          lon:standard_name = "longitude";
          lon:long_name = "longitude" ;
          lon:units = "degrees_east" ;
      float lat(trajectory, obs) ;
          lat:standard_name = "latitude";
          lat:long_name = "latitude" ;
          lat:units = "degrees_north" ;

      float z(trajectory, obs) ;
          z:standard_name = "altitude";
          z:long_name = "height above mean sea level" ;
          z:units = "km" ;
          z:positive = "up" ;
          z:axis = "Z" ;

      float O3(trajectory, obs) ;
          O3:standard_name = "mass_fraction_of_ozone_in_air";
          O3:long_name = "ozone concentration" ;
          O3:units = "1e-9" ;
          O3:coordinates = "time lon lat z" ;

      float NO3(trajectory, obs) ;
          NO3:standard_name = "mass_fraction_of_nitrate_radical_in_air";
          NO3:long_name = "NO3 concentration" ;
          NO3:units = "1e-9" ;
          NO3:coordinates = "time lon lat z" ;

   attributes:
      :featureType = "trajectory";
```

The `NO3(i,o)` and `O3(i,o)` data for element `o` of trajectory `i` are associated with the coordinate values `time(i,o)`, `lat(i,o)`, `lon(i,o)`, and `z(i,o)`. Either the instance (trajectory) or the element (`obs`) dimension could be the netCDF unlimited dimension. All variables that have trajectory as their only dimension are considered to be information about that trajectory.

If the trajectories all have the same set of times, the time auxiliary coordinate variable could be one-dimensional `time(obs)`, or replaced by a one-dimensional coordinate variable `time(time)`, where the size of the time dimension is now equal to the number of elements of each trajectory. In the latter case, listing the time coordinate variable in the coordinates attribute is optional.

H.4.2. Single trajectory

When a single trajectory is stored in the data variable, there is no need for the trajectory dimension and the arrays are one-dimensional. This is a special case of the multidimensional array representation.

Example H.13. A single trajectory recording atmospheric composition.

```
dimensions:
   time = 42;
   name_strlen = 23 ;

variables:
   char trajectory(name_strlen) ;
       trajectory:cf_role = "trajectory_id";

   double time(time) ;
       time:standard_name = "time";
       time:long_name = "time" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float lon(time) ;
       lon:standard_name = "longitude";
       lon:long_name = "longitude" ;
       lon:units = "degrees_east" ;
   float lat(time) ;
       lat:standard_name = "latitude";
       lat:long_name = "latitude" ;
       lat:units = "degrees_north" ;
   float z(time) ;
       z:standard_name = "altitude";
       z:long_name = "height above mean sea level" ;
       z:units = "km" ;
       z:positive = "up" ;
       z:axis = "Z" ;

   float O3(time) ;
       O3:standard_name = "mass_fraction_of_ozone_in_air";
       O3:long_name = "ozone concentration" ;
       O3:units = "1e-9" ;
       O3:coordinates = "time lon lat z" ;

   float NO3(time) ;
       NO3:standard_name = "mass_fraction_of_nitrate_radical_in_air";
       NO3:long_name = "NO3 concentration" ;
       NO3:units = "1e-9" ;
       NO3:coordinates = "time lon lat z" ;

attributes:
   :featureType = "trajectory";
```

The NO3(o) and O3(o) data are associated with the coordinate values time(o), z(o), lat(o), and lon(o). In this example, the time coordinate is ordered, so time values are contained in a coordinate variable i.e. time(time) and time is the element dimension. The time dimension may be unlimited or not.

Note that structurally this looks like unconnected point data as in example 9.5. The presence of the featureType = "trajectory" global attribute indicates that in fact the points are connected along a trajectory.

H.4.3. Contiguous ragged array representation of trajectories

When the number of elements for each trajectory varies, and one can control the order of writing, one can use the contiguous ragged array representation. The canonical use case for this is when rewriting raw data, and you expect that the common read pattern will be to read all the data from each trajectory.

Example H.14. Trajectories recording atmospheric composition in the contiguous ragged array representation.

```
dimensions:
   obs = 3443;
   trajectory = 77 ;

variables:
   string trajectory(trajectory) ;
         trajectory:cf_role = "trajectory_id";
   int rowSize(trajectory) ;
       rowSize:long_name = "number of obs for this trajectory " ;
       rowSize:sample_dimension = "obs" ;

   double time(obs) ;
       time:standard_name = "time";
       time:long_name = "time" ;
       time:units = "days since 1970-01-01 00:00:00" ;
   float lon(obs) ;
       lon:standard_name = "longitude";
       lon:long_name = "longitude" ;
       lon:units = "degrees_east" ;
   float lat(obs) ;
       lat:standard_name = "latitude";
       lat:long_name = "latitude" ;
       lat:units = "degrees_north" ;
   float z(obs) ;
       z:standard_name = "altitude";
       z:long_name = "height above mean sea level" ;
       z:units = "km" ;
       z:positive = "up" ;
       z:axis = "Z" ;
```

```
      float O3(obs) ;
          O3:standard_name = "mass_fraction_of_ozone_in_air";
          O3:long_name = "ozone concentration" ;
          O3:units = "1e-9" ;
          O3:coordinates = "time lon lat z" ;

      float NO3(obs) ;
          NO3:standard_name = "mass_fraction_of_nitrate_radical_in_air";
          NO3:long_name = "NO3 concentration" ;
          NO3:units = "1e-9" ;
          NO3:coordinates = "time lon lat z" ;

   attributes:
      :featureType = "trajectory";
```

The O3(o) and NO3(o) data are associated with the coordinate values time(o), lat(o), lon(o), and alt(o). All elements for one trajectory are contiguous along the sample dimension. The sample dimension (obs) may be the unlimited dimension or not. All variables that have the instance dimension (trajectory) as their single dimension are considered to be information about that trajectory.

The count variable (row_size) contains the number of elements for each trajectory, and is identified by having an attribute with name sample_dimension whose value is the sample dimension being counted. It must have the trajectory dimension as its single dimension, and must have an integer type. The elements are associated with the trajectories using the same algorithm as in H.2.4.

H.4.4. Indexed ragged array representation of trajectories

When the number of elements at each trajectory vary, and the elements cannot be written in order, one can use the indexed ragged array representation. The canonical use case is when writing real-time data streams that contain reports from many trajectories. The data can be written as it arrives; if the flatsample dimension is the unlimited dimension, this allows data to be appended to the file.

Example H.15. Trajectories recording atmospheric composition in the indexed ragged array representation.

```
   dimensions:
      obs = UNLIMITED ;
      trajectory = 77 ;
      name_strlen = 23 ;

   variables:
      char trajectory(trajectory, name_strlen) ;
          trajectory:cf_role = "trajectory_id";

      int trajectory_index(obs) ;
          trajectory_index:long_name = "index of trajectory this obs belongs to "
;
```

```
            trajectory_index:instance_dimension= "trajectory" ;
        double time(obs) ;
            time:standard_name = "time";
            time:long_name = "time" ;
            time:units = "days since 1970-01-01 00:00:00" ;
        float lon(obs) ;
            lon:standard_name = "longitude";
            lon:long_name = "longitude" ;
            lon:units = "degrees_east" ;
        float lat(obs) ;
            lat:standard_name = "latitude";
            lat:long_name = "latitude" ;
            lat:units = "degrees_north" ;
        float z(obs) ;
            z:standard_name = "altitude";
            z:long_name = "height above mean sea level" ;
            z:units = "km" ;
            z:positive = "up" ;
            z:axis = "Z" ;

        float O3(obs) ;
            O3:standard_name = "mass_fraction_of_ozone_in_air";
            O3:long_name = "ozone concentration" ;
            O3:units = "1e-9" ;
            O3:coordinates = "time lon lat z" ;

        float NO3(obs) ;
            NO3:standard_name = "mass_fraction_of_nitrate_radical_in_air";
            NO3:long_name = "NO3 concentration" ;
            NO3:units = "1e-9" ;
            NO3:coordinates = "time lon lat z" ;

    attributes:
        :featureType = "trajectory";
```

The `O3(o)` and `NO3(o)` data are associated with the coordinate values `time(o)`, `lat(o)`, `lon(o)`, and `alt(o)`. All elements for one trajectory will have the same trajectory index value. The sample dimension (`obs`) may be the unlimited dimension or not.

The index variable (`trajectory_index`) is identified by having an attribute with name of `instance_dimension` whose value is the trajectory dimension name. It must have the sample dimension as its single dimension, and must have an integer type. Each value in the `trajectory_index` variable is the zero-based trajectory index that the element belongs to. The elements are associated with the trajectories using the same algorithm as in H.2.5.

H.5. Time Series of Profiles

When profiles are taken repeatedly at a station, one gets a time series of profiles (see also section H.2 for discussion of stations and time series). The resulting collection of profiles is called a

timeSeriesProfile. A data variable may contain a collection of such timeSeriesProfile features, one feature per station. The instance dimension in the case of a timeSeriesProfile is also referred to as the **station dimension**. The instance variables, which have just this dimension, including latitude and longitude for example, are also referred to as **station variables** and are considered to contain information describing the stations. The station variables may contain missing values. This allows one to reserve space for additional stations that may be added at a later time, as discussed in section 9.6. In addition,

- It is strongly recommended that there should be a station variable (which may be of any type) with `cf_role` attribute `timeseries_id`, whose values uniquely identify the stations.
- It is recommended that there should be station variables with standard_name attributes `platform_name`, `surface_altitude` and `platform_id` when applicable.

TimeSeriesProfiles are more complicated than timeSeries because there are two element dimensions (profile and vertical). Each time series has a number of profiles from different times as its elements, and each profile has a number of data from various levels as its elements. It is strongly recommended that there always be a variable (of any data type) with the profile dimension and the `cf_role` attribute `profile_id`, whose values uniquely identify the profiles.

H.5.1. Multidimensional array representations of time series profiles

When storing time series of profiles at multiple stations in the same data variable, if there are the same number of time points for all timeSeries, and the same number of vertical levels for every profile, one can use the multidimensional array representation:

Example H.16. Time series of atmospheric sounding profiles from a set of locations stored in a multidimensional array representation.

```
dimensions:
   station = 22 ;
   profile = 3002 ;
   z = 42 ;

variables:
   float lon(station) ;
       lon:standard_name = "longitude";
       lon:long_name = "station longitude";
       lon:units = "degrees_east";
   float lat(station) ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;
   string station_name(station) ;
       station_name:cf_role = "timeseries_id" ;
       station_name:long_name = "station name" ;
   int station_info(station) ;
       station_info:long_name = "some kind of station info" ;

   float alt(station, profile , z) ;
```

```
          alt:standard_name = "altitude";
          alt:long_name = "height above mean sea level" ;
          alt:units = "km" ;
          alt:positive = "up" ;
          alt:axis = "Z" ;

       double time(station, profile ) ;
          time:standard_name = "time";
          time:long_name = "time of measurement" ;
          time:units = "days since 1970-01-01 00:00:00" ;
          time:missing_value = -999.9;

       float pressure(station, profile , z) ;
          pressure:standard_name = "air_pressure" ;
          pressure:long_name = "pressure level" ;
          pressure:units = "hPa" ;
          pressure:coordinates = "time lon lat alt station_name" ;

       float temperature(station, profile , z) ;
          temperature:standard_name = "surface_temperature" ;
          temperature:long_name = "skin temperature" ;
          temperature:units = "Celsius" ;
          temperature:coordinates = "time lon lat alt station_name" ;

       float humidity(station, profile , z) ;
          humidity:standard_name = "relative_humidity" ;
          humidity:long_name = "relative humidity" ;
          humidity:units = "%" ;
          humidity:coordinates = "time lon lat alt station_name" ;

    attributes:
     :featureType = "timeSeriesProfile";
```

The `pressure(i,p,o)`, `temperature(i,p,o)`, and `humidity(i,p,o)` data for element `o` of profile `p` at station `i` are associated with the coordinate values `time(i,p)`, `z(i,p,o)`, `lat(i)`, and `lon(i)`. Any of the three dimensions could be the netCDF unlimited dimension, if it might be useful to be able enlarge it.

If all of the profiles at any given station have the same set of vertical coordinates values, the vertical auxiliary coordinate variable could be dimensioned `alt(station, z)`. If all the profiles have the same set of vertical coordinates, the vertical auxiliary coordinate variable could be one-dimensional `alt(z)`, or replaced by a one-dimensional coordinate variable `z(z)`, provided the values are in strict monotonic order. In the latter case, listing the vertical coordinate variable in the coordinates attribute is optional.

If the profiles are taken at all stations at the same set of times, the time auxiliary coordinate variable could be one-dimensional `time(profile)`, or replaced by a one-dimensional coordinate variable `time(time)`, where the size of the `time` dimension is now equal to the number of profiles at each station. In the latter case, listing the time coordinate variable in the coordinates attribute is optional.

If there is only a single set of levels and a single set of times, the multidimensional array representation is formally orthogonal:

Example H.17. Time series of atmospheric sounding profiles from a set of locations stored in an orthogonal multidimensional array representation.

```
dimensions:
  station = 10 ;  // measurement locations
  pressure = 11 ; // pressure levels
  time = UNLIMITED ;
variables:
  float humidity(time,pressure,station) ;
    humidity:standard_name = specific_humidity ;
    humidity:coordinates = "lat lon" ;
  double time(time) ;
    time:standard_name = "time";
    time:long_name = "time of measurement" ;
    time:units = "days since 1970-01-01 00:00:00" ;
  float lon(station) ;
    lon:long_name = "station longitude";
    lon:units = "degrees_east";
  float lat(station) ;
    lat:long_name = "station latitude" ;
    lat:units = "degrees_north" ;
  float pressure(pressure) ;
    pressure:standard_name = "air_pressure" ;
    pressure:long_name = "pressure" ;
    pressure:units = "hPa" ;
    pressure:axis = "Z" ;
```

`humidity(p,o,i)` is associated with the coordinate values `time(p)`, `pressure(o)`, `lat(i)`, and `lon(i)`. The number of profiles equals the number of times.

At the cost of some wasted space, the multidimensional array representation also allows one to have a variable number of profiles for different stations, and varying numbers of levels for different profiles. In these cases, any unused elements of the data and auxiliary coordinate variables must contain missing data values (section 9.6).

H.5.2. Time series of profiles at a single station

If there is only one station in the data variable, there is no need for the station dimension:

Example H.18. Time series of atmospheric sounding profiles from a single location stored in a multidimensional array representation.

```
dimensions:
   profile = 30 ;
```

```
    z = 42 ;
    name_strlen = 23 ;

 variables:
    float lon ;
        lon:standard_name = "longitude";
        lon:long_name = "station longitude";
        lon:units = "degrees_east";
    float lat ;
        lat:standard_name = "latitude";
        lat:long_name = "station latitude" ;
        lat:units = "degrees_north" ;
    char station_name(name_strlen) ;
        station_name:cf_role = "timeseries_id" ;
        station_name:long_name = "station name" ;
    int station_info;
        station_info:long_name = "some kind of station info" ;

    float alt(profile , z) ;
        alt:standard_name = "altitude";
        alt:long_name = "height above mean sea level" ;
        alt:units = "km" ;
        alt:axis = "Z" ;
        alt:positive = "up" ;

    double time(profile ) ;
        time:standard_name = "time";
        time:long_name = "time of measurement" ;
        time:units = "days since 1970-01-01 00:00:00" ;
        time:missing_value = -999.9;

    float pressure(profile , z) ;
        pressure:standard_name = "air_pressure" ;
        pressure:long_name = "pressure level" ;
        pressure:units = "hPa" ;
        pressure:coordinates = "time lon lat alt station_name" ;

    float temperature(profile , z) ;
        temperature:standard_name = "surface_temperature" ;
        temperature:long_name = "skin temperature" ;
        temperature:units = "Celsius" ;
        temperature:coordinates = "time lon lat alt station_name" ;

    float humidity(profile , z) ;
        humidity:standard_name = "relative_humidity" ;
        humidity:long_name = "relative humidity" ;
        humidity:units = "%" ;
        humidity:coordinates = "time lon lat alt station_name" ;

 attributes:
```

```
    :featureType = "timeSeriesProfile";
```

The `pressure(p,o)`, `temperature(p,o)`, and `humidity(p,o)` data for element `o` of profile `p` are associated with the coordinate values `time(p)`, `alt(p,o)`, `lat`, and `lon`. If all the profiles have the same set of vertical coordinates, the vertical auxiliary coordinate variable could be one-dimensional `alt(z)`, or replaced by a one-dimensional coordinate variable `z(z)`, provided the values are in strict monotonic order. In the latter case, listing the vertical coordinate variable in the coordinates attribute is optional.

H.5.3. Ragged array representation of time series profiles

When the number of profiles and levels for each station varies, one can use a ragged array representation. Each of the two element dimensions (time and vertical) could in principle be stored either contiguous or indexed, but this convention supports only one of the four possible choices. This uses the contiguous ragged array representation for each profile (9.5.43.3), and the indexed ragged array representation to organise the profiles into time series (9.3.54). The canonical use case is when writing real-time data streams that contain profiles from many stations, arriving randomly, with the data for each entire profile written all at once.

Example H.19. Time series of atmospheric sounding profiles from a set of locations stored in a ragged array representation.

```
dimensions:
   obs = UNLIMITED ;
   profiles = 1420 ;
   stations = 42;

variables:
   float lon(station) ;
       lon:standard_name = "longitude";
       lon:long_name = "station longitude";
       lon:units = "degrees_east";
   float lat(station) ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;
   float alt(station) ;
       alt:long_name = "altitude above MSL" ;
       alt:units = "m" ;
   string station_name(station) ;
       station_name:long_name = "station name" ;
       station_name:cf_role = "timeseries_id";
   int station_info(station) ;
       station_info:long_name = "some kind of station info" ;

   int profile(profile) ;
       profile:cf_role = "profile_id";
   double time(profile);
```

```
        time:standard_name = "time";
        time:long_name = "time" ;
        time:units = "days since 1970-01-01 00:00:00" ;
    int station_index(profile) ;
        station_index:long_name = "which station this profile is for" ;
        station_index:instance_dimension = "station" ;
    int row_size(profile) ;
        row_size:long_name = "number of obs for this profile " ;
        row_size:sample_dimension = "obs" ;

    float z(obs) ;
        z:standard_name = "altitude";
        z:long_name = "height above mean sea level" ;
        z:units = "km" ;
        z:axis = "Z" ;
        z:positive = "up" ;

    float pressure(obs) ;
        pressure:standard_name = "air_pressure" ;
        pressure:long_name = "pressure level" ;
        pressure:units = "hPa" ;
        pressure:coordinates = "time lon lat z station_name" ;

    float temperature(obs) ;
        temperature:standard_name = "surface_temperature" ;
        temperature:long_name = "skin temperature" ;
        temperature:units = "Celsius" ;
        temperature:coordinates = "time lon lat z station_name" ;

    float humidity(obs) ;
        humidity:standard_name = "relative_humidity" ;
        humidity:long_name = "relative humidity" ;
        humidity:units = "%" ;
        humidity:coordinates = "time lon lat z station_name" ;

attributes:
    :featureType = "timeSeriesProfile";
```

The `pressure(o)`, `temperature(o)`, and `humidity(o)` data for element `o` of profile `p` at station `i` are associated with the coordinate values `time(p)`, `z(o)`, `lat(i)`, and `lon(i)`.

The index variable (`station_index`) is identified by having an attribute with name of instance_dimension whose value is the instance dimension name (`station` in this example). The index variable must have the profile dimension as its sole dimension, and must have an integer type. Each value in the index variable is the zero-based station index that the profile belongs to i.e. profile `p` belongs to `station i=station_index(p)`, as in section H.2.5.

The count variable (`row_size`) contains the number of elements for each profile, which must be written contiguously. The count variable is identified by having an attribute with name `sample_dimension` whose value is the sample dimension (`obs` in this example) being counted. It

must have the profile dimension as its sole dimension, and must have an integer type. The number of elements in profile `p` is recorded in `row_size(p)`, as in section H.2.4. The sample dimension need not be the netCDF unlimited dimension, though it commonly is.

H.6. Trajectory of Profiles

When profiles are taken along a trajectory, one gets a collection of profiles called a trajectoryProfile. A data variable may contain a collection of such trajectoryProfile features, one feature per trajectory. The instance dimension in the case of a trajectoryProfile is also referred to as the **trajectory dimension**. The instance variables, which have just this dimension, are also referred to as **trajectory variables** and are considered to contain information describing the trajectories. The trajectory variables may contain missing values. This allows one to reserve space for additional trajectories that may be added at a later time, as discussed in section 9.6. TrajectoryProfiles are more complicated than trajectories because there are two element dimensions. Each trajectory has a number of profiles as its elements, and each profile has a number of data from various levels as its elements. It is strongly recommended that there always be a variable (of any data type) with the profile dimension and the `cf_role` attribute `profile_id`, whose values uniquely identify the profiles.

H.6.1. Multidimensional array representation of trajectory profiles

If there are the same number of profiles for all trajectories, and the same number of vertical levels for every profile, one can use the multidimensional representation:

Example H.20. Time series of atmospheric sounding profiles along a set of trajectories stored in a multidimensional array representation.

```
dimensions:
   trajectory = 22 ;
   profile = 33;
   z = 42 ;

variables:
   int trajectory (trajectory ) ;
       trajectory:cf_role = "trajectory_id" ;

   float lon(trajectory, profile) ;
       lon:standard_name = "longitude";
       lon:units = "degrees_east";
   float lat(trajectory, profile) ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;

   float alt(trajectory, profile , z) ;
       alt:standard_name = "altitude";
       alt:long_name = "height above mean sea level" ;
       alt:units = "km" ;
       alt:positive = "up" ;
```

```
          alt:axis = "Z" ;

      double time(trajectory, profile ) ;
          time:standard_name = "time";
          time:long_name = "time of measurement" ;
          time:units = "days since 1970-01-01 00:00:00" ;
          time:missing_value = -999.9;

      float pressure(trajectory, profile , z) ;
          pressure:standard_name = "air_pressure" ;
          pressure:long_name = "pressure level" ;
          pressure:units = "hPa" ;
          pressure:coordinates = "time lon lat alt" ;

      float temperature(trajectory, profile , z) ;
          temperature:standard_name = "surface_temperature" ;
          temperature:long_name = "skin temperature" ;
          temperature:units = "Celsius" ;
          temperature:coordinates = "time lon lat alt" ;

      float humidity(trajectory, profile , z) ;
          humidity:standard_name = "relative_humidity" ;
          humidity:long_name = "relative humidity" ;
          humidity:units = "%" ;
          humidity:coordinates = "time lon lat alt" ;

   attributes:
    :featureType = "trajectoryProfile";
```

The `pressure(i,p,o)`, `temperature(i,p,o)`, and `humidity(i,p,o)` data for element `o` of profile `p` along trajectory `i` are associated with the coordinate values `time(i,p)`, `alt(i,p,o)`, `lat(i,p)`, and `lon(i,p)`. Any of the three dimensions could be the netCDF unlimited dimension, if it might be useful to be able enlarge it.

If all of the profiles along any given trajectory have the same set of vertical coordinates values, the vertical auxiliary coordinate variable could be dimensioned `alt(trajectory, z)`. If all the profiles have the same set of vertical coordinates, the vertical auxiliary coordinate variable could be one-dimensional `alt(z)`, or replaced by a one-dimensional coordinate variable `z(z)`, provided the values are in strict monotonic order. In the latter case, listing the vertical coordinate variable in the coordinates attribute is optional.

If the profiles are taken along all the trajectories at the same set of times, the time auxiliary coordinate variable could be one-dimensional `time(profile)`, or replaced by a one-dimensional coordinate variable `time(time)`, where the size of the time dimension is now equal to the number of profiles along each trajectory. In the latter case, listing the time coordinate variable in the coordinates attribute is optional.

At the cost of some wasted space, the multidimensional array representation also allows one to have a variable number of profiles for different trajectories, and varying numbers of levels for

different profiles. In these cases, any unused elements of the data and auxiliary coordinate variables must contain missing data values (section 9.6).

H.6.2. Profiles along a single trajectory

If there is only one trajectory in the data variable, there is no need for the trajectory dimension:

Example H.21. Time series of atmospheric sounding profiles along a trajectory stored in a multidimensional array representation.

```
dimensions:
   profile = 33;
   z = 42 ;

variables:
   int trajectory;
       trajectory:cf_role = "trajectory_id" ;

   float lon(profile) ;
       lon:standard_name = "longitude";
       lon:units = "degrees_east";
   float lat(profile) ;
       lat:standard_name = "latitude";
       lat:long_name = "station latitude" ;
       lat:units = "degrees_north" ;

   float alt(profile, z) ;
       alt:standard_name = "altitude";
       alt:long_name = "height above mean sea level" ;
       alt:units = "km" ;
       alt:positive = "up" ;
       alt:axis = "Z" ;

   double time(profile ) ;
       time:standard_name = "time";
       time:long_name = "time of measurement" ;
       time:units = "days since 1970-01-01 00:00:00" ;
       time:missing_value = -999.9;

   float pressure(profile, z) ;
       pressure:standard_name = "air_pressure" ;
       pressure:long_name = "pressure level" ;
       pressure:units = "hPa" ;
       pressure:coordinates = "time lon lat alt" ;

   float temperature(profile, z) ;
       temperature:standard_name = "surface_temperature" ;
       temperature:long_name = "skin temperature" ;
       temperature:units = "Celsius" ;
       temperature:coordinates = "time lon lat alt" ;
```

```
      float humidity(profile, z) ;
          humidity:standard_name = "relative_humidity" ;
          humidity:long_name = "relative humidity" ;
          humidity:units = "%" ;
          humidity:coordinates = "time lon lat alt" ;

   attributes:
    :featureType = "trajectoryProfile";
```

The `pressure(p,o)`, `temperature(p,o)`, and `humidity(p,o)` data for element `o` of profile `p` are associated with the coordinate values `time(p)`, `alt(p,o)`, `lat(p)`, and `lon(p)`. If all the profiles have the same set of vertical coordinates, the vertical auxiliary coordinate variable could be one-dimensional `alt(z)`, or replaced by a one-dimensional coordinate variable `z(z)`, provided the values are in strict monotonic order. In the latter case, listing the vertical coordinate variable in the coordinates attribute is optional.

H.6.3. Ragged array representation of trajectory profiles

When the number of profiles and levels for each trajectory varies, one can use a ragged array representation. Each of the two element dimensions (along a trajectory, within a profile) could in principle be stored either contiguous or indexed, but this convention supports only one of the four possible choices. This uses the contiguous ragged array representation for each profile (9.3.3), and the indexed ragged array representation to organise the profiles into time series (9.3.4). The canonical use case is when writing real-time data streams that contain profiles from many trajectories, arriving randomly, with the data for each entire profile written all at once.

Example H.22. Time series of atmospheric sounding profiles along a set of trajectories stored in a ragged array representation.

```
   dimensions:
      obs = UNLIMITED ;
      trajectory = 22 ;
      profile = 142 ;

   variables:
      int trajectory(trajectory) ;
          cf_role = "trajectory_id" ;

      double time(profile);
          time:standard_name = "time";
          time:long_name = "time" ;
          time:units = "days since 1970-01-01 00:00:00" ;
      float lon(profile);
          lon:standard_name = "longitude";
          lon:long_name = "longitude" ;
          lon:units = "degrees_east" ;
      float lat(profile);
```

```
          lat:standard_name = "latitude";
          lat:long_name = "latitude" ;
          lat:units = "degrees_north" ;
       int row_size(profile) ;
          row_size:long_name = "number of obs for this profile " ;
          row_size:sample_dimension = "obs" ;
       int trajectory_index(profile) ;
          trajectory_index:long_name = "which trajectory this profile is for" ;
          trajectory_index:instance_dimension= "trajectory" ;

        float z(obs) ;
          z:standard_name = "altitude";
          z:long_name = "height above mean sea level" ;
          z:units = "km" ;
          z:positive = "up" ;
          z:axis = "Z" ;

       float pressure(obs) ;
          pressure:standard_name = "air_pressure" ;
          pressure:long_name = "pressure level" ;
          pressure:units = "hPa" ;
          pressure:coordinates = "time lon lat z" ;

       float temperature(obs) ;
          temperature:standard_name = "surface_temperature" ;
          temperature:long_name = "skin temperature" ;
          temperature:units = "Celsius" ;
          temperature:coordinates = "time lon lat z" ;

       float humidity(obs) ;
          humidity:standard_name = "relative_humidity" ;
          humidity:long_name = "relative humidity" ;
          humidity:units = "%" ;
          humidity:coordinates = "time lon lat z" ;

    attributes:
       :featureType = "trajectoryProfile";
```

The `pressure(o)`, `temperature(o)`, and `humidity(o)` data for element `o` of profile `p` along trajectory `i` are associated with the coordinate values `time(p)`, `z(o)`, `lat(p)`, and `lon(p)`.

The index variable (`trajectory_index`) is identified by having an attribute with name of `instance_dimension` whose value is the instance dimension name (trajectory in this example). The index variable must have the profile dimension as its sole dimension, and must have an integer type. Each value in the index variable is the zero-based trajectory index that the profile belongs to i.e. profile `p` belongs to trajectory `i=trajectory_index(p)`, as in section H.2.5.

The count variable (`row_size`) contains the number of elements for each profile, which must be written contiguously. The count variable is identified by having an attribute with name `sample_dimension` whose value is the sample dimension (`obs` in this example) being counted. It

must have the profile dimension as its sole dimension, and must have an integer type. The number of elements in profile `p` is recorded in `row_size(p)`, as in section H.2.4. The sample dimension need not be the netCDF unlimited dimension, though it commonly is.

Appendix I: The CF data model

The CF conventions are designed to promote the creation, processing, and sharing of climate and forecasting data using Network Common Data Form (netCDF) files and libraries. This appendix contains the explicit data model for CF to provide an interpretation of the conceptual structure of CF which is consistent, comprehensive, and as far as possible independent of the netCDF encoding. An explicit comprehensive data model promotes the CF conventions being better understood, provides guidance during the development of future extensions to the CF conventions, and helps software developers to design CF-compliant data-processing applications and to build interfaces to other explicit data models.

Introduction

A data model is an abstract interpretation of the data, that identifies the elements of the dataset and their scientific intent, and describes how they are related to one another and to the real or model world from which the data were derived. A data model is necessary because it imposes the rules, constraints, and relationships connecting metadata to the data that are needed to imagine how the quantities included in the dataset should be combined and processed scientifically.

The CF data model was first created for CF version 1.6 and published externally in journal Geoscientific Model Development (GMD) [CFDM], and that version also includes further discussions on the background and motivation, as well as on the relationships between the CF data model and other data models. The data model was transcribed from the GMD paper into the CF conventions at version 1.9, also incorporating the modifications required to represent new features introduced at versions 1.7, 1.8 and 1.9.

Design criteria of the CF data model

The primary requirement of the data model is that it should be able to describe all existing and conceivable CF-compliant datasets.

The data model should comprise a minimal set of elements that are sufficient for accommodating all aspects of the CF conventions. The elements of the data model are restricted to those that are explicitly mentioned in CF, but there do not have to be as many elements in the data model as there are elements described by CF, because a single data model element can incorporate more than one CF entity. For example, in CF, coordinates and coordinate bounds are distinct elements, but coordinate bounds cannot exist without coordinates. Therefore, it makes sense in the data model to group them into a single element.

Similarly, while it is possible to introduce additional elements not presently needed or used by CF, this would not be desirable because it would increase the likelihood of the data model becoming outdated or inconsistent with future versions of CF.

The CF data model should also be independent of the encoding. This means that it should not be constrained by the parts of the CF conventions which describe explicitly how to store (i.e. encode) metadata in a netCDF file. The virtue of this is that should netCDF ever fail to meet the community needs, the groundwork for applying CF to other file formats will already exist.

Elements of CF-netCDF

The CF-netCDF elements are listed in Table I.1 and shown (in blue) with their interrelationships in Figure I.1. The CF data model has been derived from these CF-netCDF elements and relationships with the aims of removing aspects specific to the netCDF encoding, and reducing the number of elements, whilst retaining the ability to describe the CF conventions fully, in order to meet the design criteria.

Table I.1. The elements of the CF-netCDF conventions

CF-netCDF element	Description
Domain variable	Discrete locations in multi-dimensional space
Data variable	Scientific data discretised within a domain
Dimension	Independent axis of the domain
Coordinate variable	Unique coordinates for a single axis
Auxiliary coordinate variable	Additional or alternative coordinates for any axes
Scalar coordinate variable	Coordinate for an implied size one axis
Grid mapping variable	Horizontal coordinate system
Boundary variable	Cell vertices
Cell measure variable	Cell areas or volumes
Ancillary data variable	Metadata that depends on the domain
Mesh topology variable	One or more related domains with cell connectivity
Location index set variable	A domain with cell connectivity
Formula terms attribute	Vertical coordinate system
Feature type attribute	Characteristics of discrete sampling geometry
Cell methods attribute	Description of variation within cells

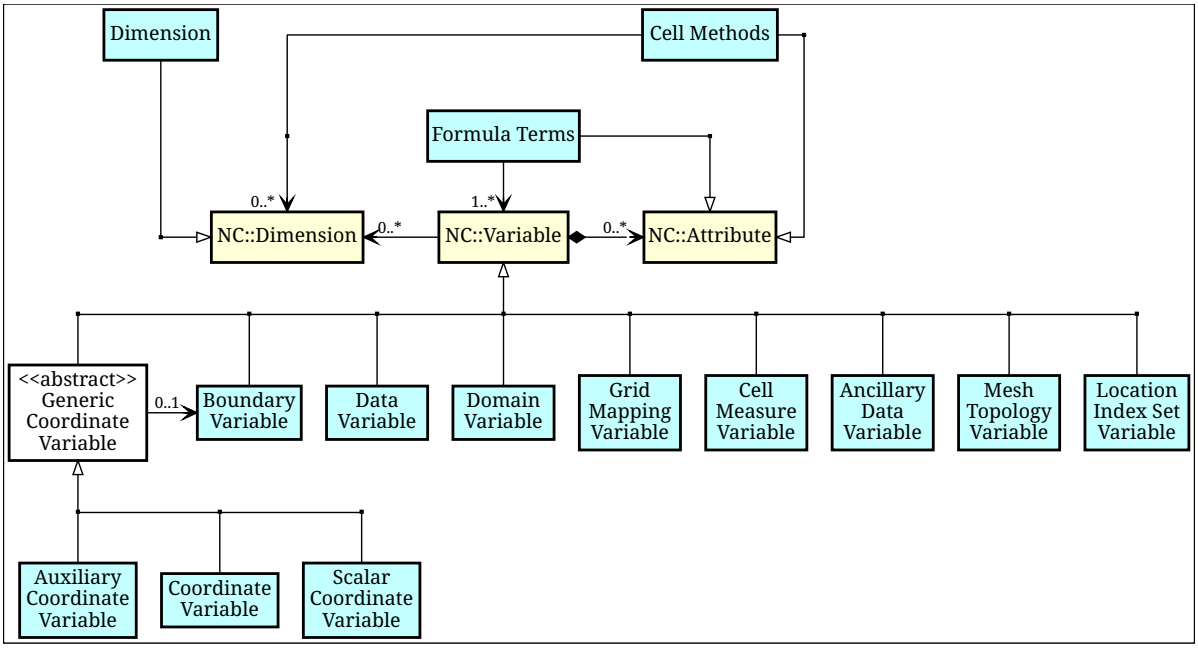


Figure I.1. The relationships between CF-netCDF elements and their corresponding netCDF variables, dimensions and attributes (identified here with the "NC" prefix). It is useful to define an abstract generic coordinate variable that can be used to refer to coordinates when the their type (coordinate, auxiliary or scalar coordinate variable) is not an issue.

The CF data model

The elements of the CF data model (Figure I.2, Figure I.3 and Figure I.4) are called "constructs", a term chosen to differentiate from the CF-netCDF elements previously defined and to be programming language-neutral (i.e. as opposed to "object" or "structure"). The constructs, listed in Table I.2, are related to CF-netCDF elements (Figure I.1), which in turn relate to the components of netCDF file.

Table I.2. The constructs of the CF data model

CF construct	Description
Domain	Discrete locations in multi-dimensional space
Field	Scientific data discretised within a domain
Domain axis	Independent axes of the domain
Dimension coordinate	Cell locations
Auxiliary coordinate	Cell locations
Coordinate reference	Domain coordinate systems
Domain ancillary	Cell locations in alternative coordinate systems
Cell measure	Cell size or shape
Domain topology	Geospatial topology of domain cells
Cell connectivity	Connectivity of domain cells
Field ancillary	Ancillary metadata which varies within the domain

CF construct	Description
Cell method	Describes how data represents variation within cells

The field construct and domain construct are central to the CF data model in that all the other constructs are included in one or other of them (Figure I.2). The constructs contained by the field and domain constructs cannot exist independently, with the exception of the domain construct itself that may be part of a field construct or exist on its own, as is indicated by the nature of the class associations shown in Figure I.2. All CF-netCDF elements are mapped to field constructs, domain constructs or their components; and the field and domain constructs completely contain all the data and metadata which can be extracted from the file using the CF conventions.

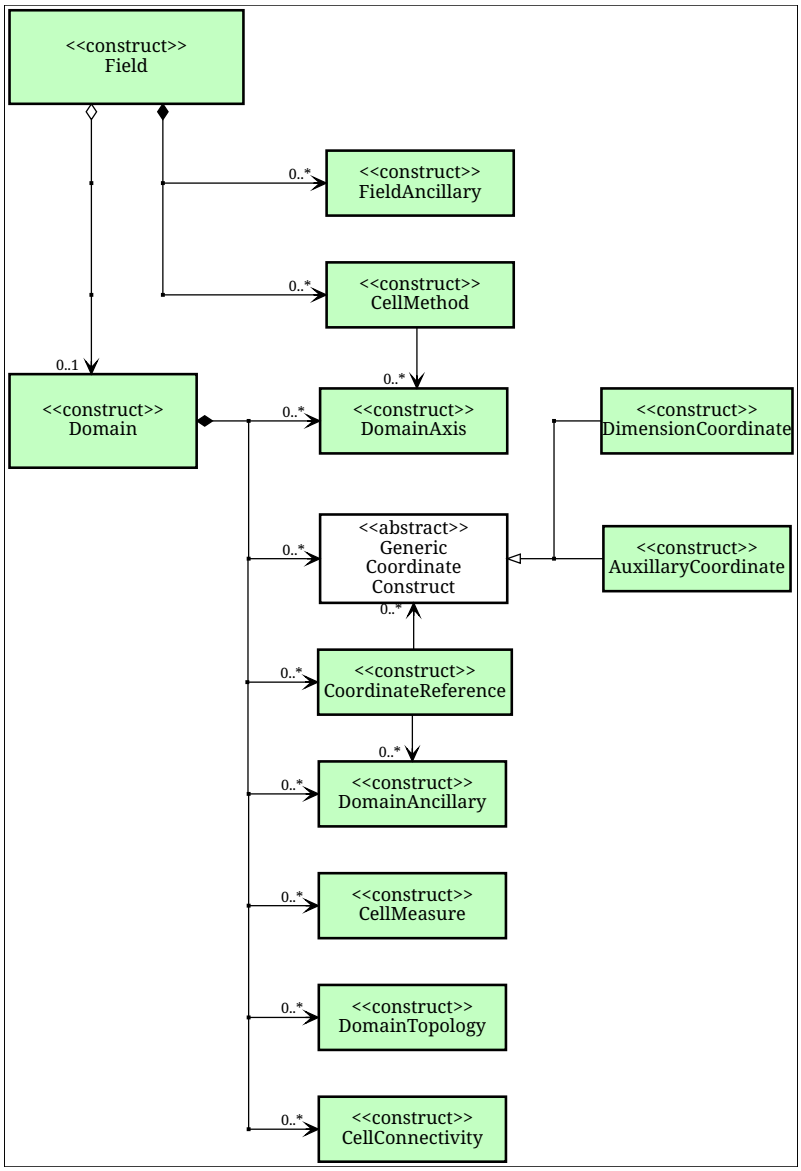


Figure I.2. The constructs of the CF data model. The field and domain constructs correspond to CF-netCDF data and domain variables respectively (defined in Figure I.1 and identified here with the "CN" prefix). Relationships between the other constructs and CF-netCDF are given in Figure I.3 and Figure I.4. It is useful to define an abstract generic coordinate construct that can be used to refer to coordinates when the their type (dimension or auxiliary coordinate construct) is not an issue.

Field construct

A field construct (Figure I.2) corresponds to a CF-netCDF data variable with all of its metadata. The field construct consists of

- A data array.
- A domain construct containing metadata defining the domain that provides measurement locations and cell properties for the data.
- Field ancillary constructs containing ancillary metadata defined over the same domain.
- Cell method constructs containing metadata to describe how the cell values represent the variation of the physical quantity within the cells of the domain.
- Properties to describe aspects of the data that are independent of the domain.

All of the constructs contained by the field construct are optional (as indicated by "0.." in Figure I.2). The only component of the field which is mandatory is the data array.

The properties of the field construct correspond to some netCDF attributes of variables (e.g. `units`, `long_name`, and `standard_name`); and some netCDF group attributes, which include global attributes in the root group, such as `history` and `institution`. The term "property" is used, rather than "attribute", because not all CF-netCDF attributes are properties in this sense—some CF-netCDF attributes are used to point to (i.e. reference) other netCDF variables and so only describe the data indirectly (e.g. the coordinates attribute), and others have structural functions in the CF-netCDF file (e.g. the Conventions attribute).

In the data model, netCDF group attributes apply to every data variable in the file, except where they are overridden by netCDF data variable attributes with the same name. This interpretation of group attributes is not stated in the CF conventions, but for the data model it is necessary because there is no notion of a group. Hence, metadata stored in attributes of the group as a whole have to be transferred to the field construct. If present, the global file attribute (i.e. root group attribute) `featureType` applies to every data variable in the file with a discrete sampling geometry. Hence, the feature type is regarded as a property of the field construct.

The `standard_name` property constrains the `units` property (i.e. only certain units are consistent with each standard name) and in some cases also the dimensions that a data variable must have. These constraints, however, do not supply any further information—they are just for self consistency. Similarly the `featureType` property imposes some requirements on the axes the domain must have. Following the aim of constructing a minimal data model, the standard name and feature type are not regarded as separate constructs within the field, because they do not depend on any other construct for their interpretation.

Domain construct

The domain construct (Figure I.2) describes a domain comprising measurement locations and cell properties. The domain construct is the only metadata construct that may also exist independently of a field construct. The domain construct contains properties to describe the domain (in the same sense as for the field construct) and relates the following metadata constructs

- Domain axis constructs.

- Dimension coordinate and auxiliary coordinate constructs.
- Coordinate reference constructs.
- Domain ancillary constructs.
- Cell measure constructs.
- Domain topology constructs.
- Cell connectivity constructs.

All of the constructs contained by the domain construct are optional (as indicated by "0.." in Figure I.2).

In CF-netCDF, domain information is stored either implicitly via data variable attributes (such as `coordinates`), or explicitly in a domain variable. In the latter case, the domain exists without reference to a data array.

- implicitly via data variable attributes (such as `coordinates`);
- explicitly in a domain variable;
- explicitly in a mesh topology variable, in which case the domain may be one of multiple domains defined by the same variable (for instance, a single mesh topology variable could contain different domains defined respectively at node, edge, and face mesh elements);
- explicitly in a location index set variable that references a subset of another domain defined by a mesh topology variable.

For the explicit cases, the domain exists without reference to a data array.

Domain axis construct and the data array

A domain axis construct (Figure I.3) comprises a positive integer which specifies the number of cells lying along an independent axis of the domain. In CF-netCDF, it is usually defined either by a netCDF dimension or by a scalar coordinate variable, which implies a domain axis of size one. The field construct's data array spans the domain axis constructs of the domain, except that the size-one axes may optionally be omitted, because their presence makes no difference to the order of the elements. Hence, the data array may be zero-dimensional (i.e. scalar) if there are no domain axis constructs of size greater than one.

When a collection of discrete sampling geometry (DSG) features has been combined in a data variable using the incomplete orthogonal or ragged representations to save space, the axis size has to be inferred, but this is an aspect of unpacking the data, rather than its conceptual description. In practice, the unpacked data array may be dominated by missing values (as could occur, for example, if all features in a collection of time series had no common time coordinates), in which case it may be preferable to view the collection as if each DSG feature were a separate variable, each one corresponding to a different field construct.

Coordinates: dimension coordinate and auxiliary constructs

Coordinate constructs (Figure I.3) provide information which locate the cells of the domain and which depend on a subset of the domain axis constructs. A coordinate construct consists of an

optional data array of the coordinate values spanning the subset of the domain axis constructs, properties to describe the coordinates (in the same sense as for the field construct), an optional data array of cell bounds recording the extents of each cell, and any extra arrays needed to interpret the cell bounds values. The data array of the coordinate values is required, execpt for the special cases described below.

There are two distinct types of coordinate construct: dimension coordinate constructs unambiguously describe cell locations for a single domain axis, thus providing independent variables on which the field construct's data depend; and auxiliary coordinate constructs provide any type of coordinate information for one or more of the domain axes.

A dimension coordinate construct contains numeric coordinates for a single domain axis that are non-missing and strictly monotonically increasing or decreasing. CF-netCDF coordinate variables and numeric scalar coordinate variables correspond to dimension coordinate constructs.

Auxiliary coordinate constructs have to be used, instead of dimension coordinate constructs, when a single domain axis requires more than one set of coordinate values, when coordinate values are not numeric, strictly monotonic, or contain missing values, or when they vary along more than one domain axis construct simultaneously. CF-netCDF auxiliary coordinate variables and non-numeric scalar coordinate variables correspond to auxiliary coordinate constructs.

When cell bounds are provided, each cell comprises one or more parts, and each part is either a collection of points, a line defined by a connected series of points, or a polygonal area (i.e. the region enclosed by a connected series of points, where the first and last points are connected as well). All parts of all the cells must be of the same one of these three kinds, which are called "geometry types". The bounds array spans the domain axis constructs of the coordinate construct, with the addition of two trailing ragged dimensions. The first extra dimension indexes the parts of each cell and the second indexes the points that describe each part.

If cell bounds are provided for a dimension coordinate construct then each cell must have exactly two vertices forming a line geometry. For climatological time coordinates the actual cell extent comprises multiple time segments equivalent to multiple line geometry parts, but the bounds require just two points to define each cell, namely the earliest and latest times of the sequence. The cell method constructs indicate how the multiple time segments should be inferred from these climatological bounds.

If a polygonal cell is composed of multiple parts it may have holes, i.e. polygon regions that are to be omitted from, as opposed to included in, the cell extent. When such holes are present an "interior ring" array is required that records whether each polygon is to be included or excluded from the cell, and is supplied by an interior ring variable in CF-netCDF. The interior ring array spans the domain axis constructs of the coordinate construct, with the addition of an extra ragged dimension that indexes the parts for each cell. For example, a cell describing the land area surrounding a lake would require two polygon parts: one defines the outer boundary of the land area; the other, recorded as an interior ring, is the lake boundary, defining the inner boundary of the land area.

If a domain axis construct does not correspond to a continuous physical quantity, then it is not necessary for it to be associated with a dimension coordinate construct. For example, this is the case for an axis that runs over ocean basins or area types, or for a domain axis that indexes a time

series at scattered points. These axes are discrete axes in CF-netCDF. In such cases cells may be described with one-dimensional auxiliary coordinate constructs for which, provided that there is a cell bounds array to describe the cell extents, the coordinate array is optional, since coordinates are not always well defined for such cells. A CF-netCDF geometry container variable or mesh topology variable is used to store cell bounds without coordinates for a discrete axis.

In CF-netCDF, when a geometry container variable is present it explicitly describes the geometry type and identifies the node coordinate variables that contain the cell vertices. The geometry container variable also identifies a node count variable that contains the number of nodes per cell when more than one cell is present, and a part node count variable that contains the number of nodes per cell part when cells are composed of multipart lines, multipart polygons, or polygons with holes. When a geometry container variable is not present then the bounds contain exactly one part and their geometry type is implied by convention: for multidimensional auxiliary coordinates each cell is a single polygon, and for all other types of coordinate each cell is a single line segment defined by two points. In the case of climatological time coordinates, the two points of the cell bounds, in conjunction with the cell methods, imply the existence of multiple line parts, different subsets of which are associated with the different cell methods required to define the climatology. For example, when the field construct's data are multiannual averages of monthly minima, the implied cell parts define the individual months over which the original data was minimised; and all of the implied parts taken together define the exact temporal extent of the average of the monthly minima.

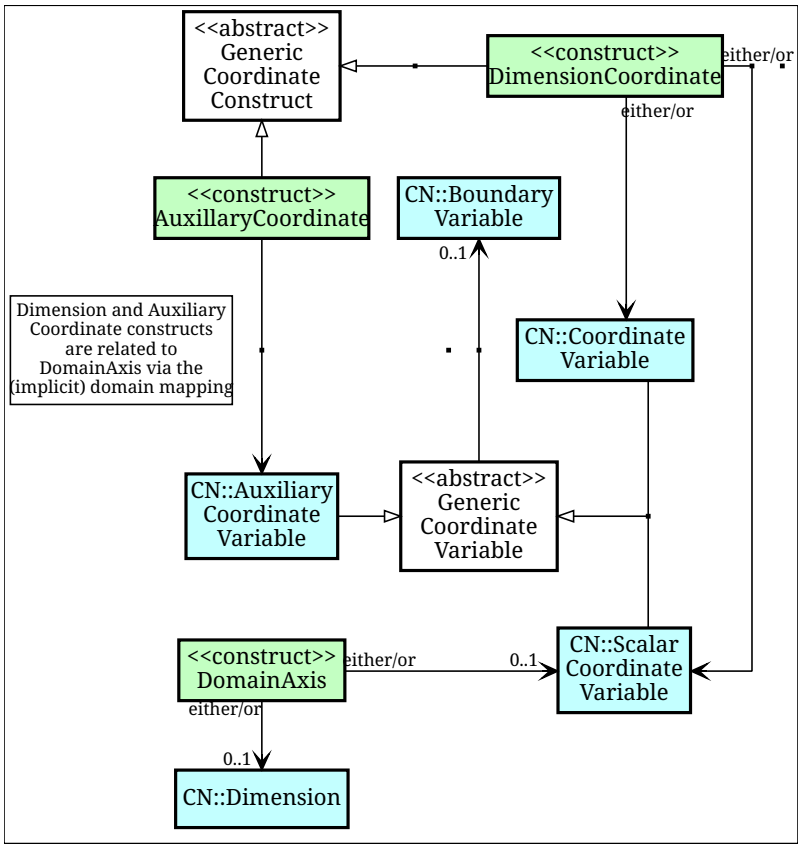


Figure I.3. The relationship between domain axis, dimension coordinate and auxiliary coordinate constructs and CF-netCDF (defined in Figure I.1 and identified here with the "CN" prefix). A dimension or auxiliary coordinate construct is defined by a CF-netCDF coordinate, scalar coordinate or auxiliary coordinate variable, and the associated CF-netCDF boundary variable if it exists. A generic coordinate construct spans one or more domain axis constructs, but the mapping of which ones is only held by the parent field construct.

Coordinate reference construct

The domain may contain various coordinate systems, each of which is constructed from a subset of the dimension and auxiliary coordinate constructs. For example, the domain of a four-dimensional field construct may contain horizontal (*y*-*x*), vertical (*z*), and temporal (*t*) coordinate systems. There may be more than one of each of these, if there is more than one coordinate construct applying to a particular spatiotemporal dimension (for example, there could be both latitude-longitude and *y*-*x* projection coordinate systems).

A coordinate system may be constructed *implicitly* from any subset of the coordinate constructs, yet a coordinate construct does not need to be explicitly or exclusively associated with any coordinate system. A coordinate system of the field construct can be *explicitly* defined by a coordinate reference construct (Figure I.4) which relates the coordinate values of the coordinate system to locations in a planetary reference frame and consists of the following:

- The dimension coordinate and auxiliary coordinate constructs that define the coordinate system to which the coordinate reference construct applies. Note that the coordinate values are not relevant to the coordinate reference construct, only their properties.
- A definition of a datum specifying the zeroes of the dimension and auxiliary coordinate constructs which define the coordinate system. The datum may be explicitly indicated via properties, or it may be implied by the metadata of the contained dimension and auxiliary coordinate constructs. For example, in a two-dimensional geographical latitude-longitude coordinate system based upon a spherical Earth, the datum is assumed to be 0^{o}N, 0^{o}E. Note that the datum may contain the definition of a geophysical surface which corresponds to the zero of a vertical coordinate construct, and this may be required for both horizontal and vertical coordinate systems.
- A coordinate conversion, which defines a formula for converting coordinate values taken from the dimension or auxiliary coordinate constructs to a different coordinate system. A term of the conversion formula can be a scalar or vector parameter which does not depend on any domain axis constructs, may have units (such as a reference pressure value), or may be a descriptive string (such as the projection name "mercator"), or it can be a domain ancillary construct (such as one containing spatially varying orography data).

For *y*-*x* coordinates, the coordinate conversion is either a map projection, which converts between Cartesian coordinates and spherical or ellipsoidal coordinates on the vertical datum, or a conversion between different spherical coordinate systems (as in the case of rotated pole coordinates). In the case of *z* coordinates, the conversion is between a coordinate construct with parameterised values (such as ocean sigma coordinates) and a coordinate construct with dimensional values (such as depths), again with respect to the vertical datum. The coordinate conversion is not required if no other coordinate systems are described.

Some parts of the coordinate reference construct may not be relevant to a given coordinate construct which it contains. The relevant parts are determined by an application using the coordinate reference construct. For example, for a coordinate reference construct which contained coordinate constructs for *y*-*x* projection and latitude and longitude coordinates, a datum comprising a reference ellipsoid would apply to all of them, but projection parameters would only apply to the projection coordinates.

In CF-netCDF, coordinate system information that is not found in coordinate or auxiliary coordinate variables is stored in a grid mapping variable or the formula_terms attribute of a coordinate variable, for horizontal or vertical coordinate variables, respectively. Although these two cases are arranged differently in CF-netCDF, each one contains, sometimes implicitly, a datum or a coordinate conversion formula (or both) and is therefore regarded as a coordinate reference construct by the data model. A grid mapping name or the standard name of a parametric vertical coordinate corresponds to a string-valued scalar parameter of a coordinate conversion formula. A grid mapping parameter which has more than one value (as is possible with the "standard parallel" attribute) corresponds to a vector parameter of a coordinate conversion formula. A data variable referenced by a formula_terms attribute corresponds to the term of a coordinate conversion formula—either a domain ancillary construct or, if it is zero-dimensional, a scalar parameter.

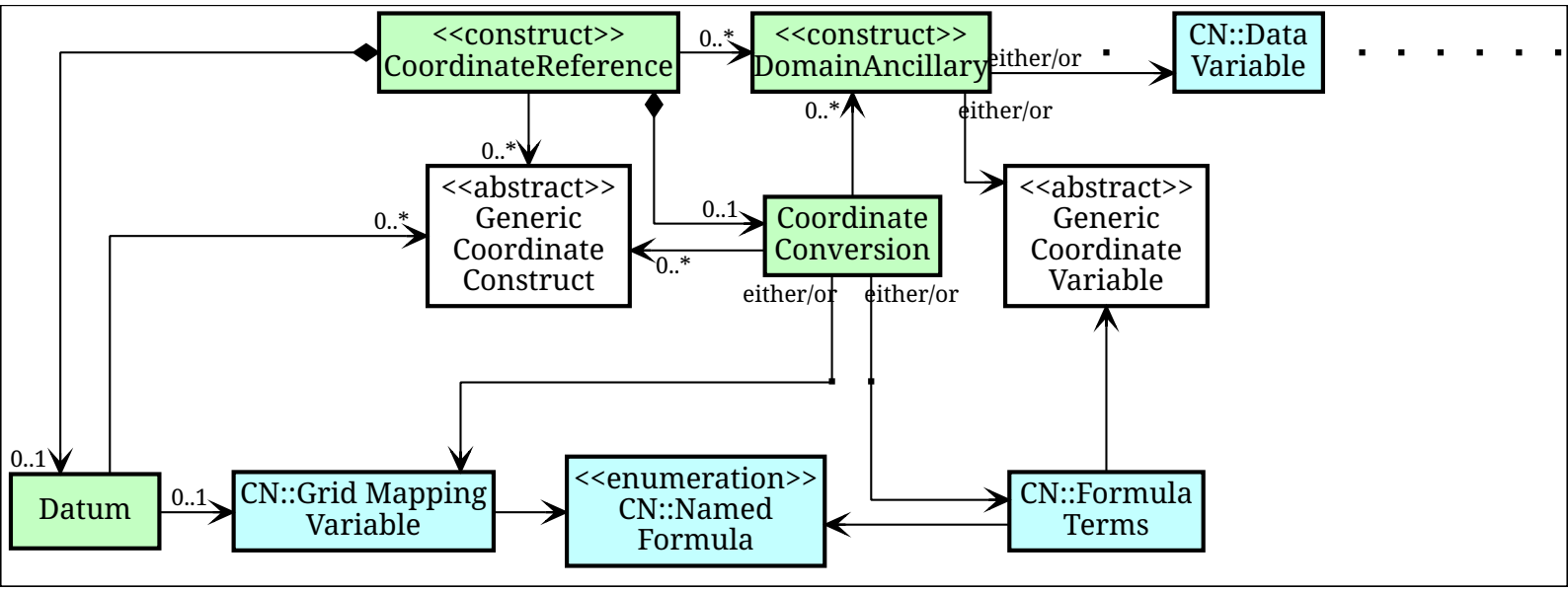


Figure I.4. The relationship between coordinate reference and domain ancillary constructs and CF-netCDF (defined in Figure I.1 and identified here with the "CN" prefix). A coordinate reference construct is defined either by a grid mapping variable, or a `formula_terms` attribute of a CF-netCDF coordinate variable. The coordinate reference construct is composed of generic coordinate constructs, a datum, and a coordinate conversion formula. The coordinate conversion formula is usually defined by a named formula in the CF conventions. A domain ancillary construct term of a coordinate conversion formula is defined by a CF-netCDF data variable or a CF-netCDF generic coordinate variable.

Domain ancillary construct

A domain ancillary construct (Figure I.4) provides information which is needed for computing the location of cells in an alternative coordinate system. It is the value of a term of a coordinate conversion formula that contains a data array which is either scalar or which depends on one, more or all of the domain axis constructs.

It also contains an optional array of cell bounds recording the extents of each cell (only applicable if the array contains coordinate data) and properties to describe the data (in the same sense as for the field construct). An array of cell bounds spans the same domain axes as the data array, with the addition of an extra dimension whose size is that of the number of vertices of each cell.

CF-netCDF variables named by the `formula_terms` attribute of a CF-netCDF coordinate variable correspond to domain ancillary constructs. These CF-netCDF variables may be coordinate, scalar coordinate, or auxiliary coordinate variables, or they may be data variables. For example, in a

coordinate conversion for converting between ocean sigma and height coordinate systems, the value of the "depth" term for horizontally varying distance from ocean datum to sea floor would correspond to a domain ancillary construct. In the case of a named term being a type of coordinate variable, that variable will correspond to an independent domain ancillary construct in addition to the coordinate construct; that is, a single CF-netCDF variable is translated into two constructs (see Example I.1).

Example I.1. A single CF-netCDF variable corresponding to two data model constructs.

```
float eta(eta) ;
  eta:long_name = "eta at full levels" ;
  eta:positive = "down" ;
  eta:standard_name = "atmosphere_hybrid_sigma_pressure_coordinate" ;
  eta:formula_terms = "a: A b: B ps: PS p0: P0" ;
float A(eta) ;
  A:units = "Pa" ;
float B(eta) ;
  B:units = "1" ;
float PS(lat, lon) ;
  PS:units = "Pa" ;
float P0 ;
  P0:units = "Pa" ;
float temp(eta, lat, lon) ;
  temp:standard_name = "air_temperature" ;
  temp:units = "K";
  temp:coordinates = "A B" ;
```

The netCDF variable `A` corresponds to an auxiliary coordinate construct (since it is referenced by the `coordinates` attribute) as well as a domain ancillary construct (since it is referenced by the `formula_terms` attribute). Similarly for the netCDF variable `B`.

Cell measure construct

A cell measure (Figure I.2) construct provides information about the size or shape of the cells and depending on one, more or all of the domain axis constructs. Cell measure constructs have to be used when the size or shape of the cells cannot be deduced from the dimension or auxiliary coordinate constructs without special knowledge that a generic application cannot be expected to have.

The cell measure construct consists of a numeric array of the metric data which span one, more or all of the domain axis constructs, and properties to describe the data (in the same sense as for the field construct). The properties must contain a "measure" property, which indicates which metric of the space it supplies, e.g. cell horizontal areas, and a units property consistent with the measure property, e.g. m2. It is assumed that the metric does not depend on axes of the domain which are not spanned by the array, along which the values are implicitly propagated. CF-netCDF cell measure variables correspond to cell measure constructs.

Domain topology construct

A domain topology construct defines the geospatial topology of cells arranged in two or three dimensions in real space but indexed by a single (discrete) domain axis construct, and at most one domain topology construct may be associated with any such domain axis. The topology describes topological relationships between the cells - spatial relationships which do not depend on the cell locations - and is represented by an undirected graph, i.e. a mesh in which pairs of nodes are connected by links. Each node has a unique arbitrary identity that is independent of its spatial location, and different nodes may be spatially co-located.

The topology may only describe cells that have a common spatial dimensionality, one of:

- Point: A point is zero-dimensional and has no boundary vertices.
- Edge: An edge is one-dimensional and corresponds to a line connecting two boundary vertices.
- Face: A face is two-dimensional and corresponds to a surface enclosed by a set of edges.

Each type of cell implies a restricted topology for which only some kinds of mesh are allowed. For point cells, every node corresponds to exactly one cell; and two cells have a topological relationship if and only if their nodes are connected by a mesh link. For edge and face cells, every node corresponds to a boundary vertex of a cell; the same node can represent vertices in multiple cells; every link in the mesh connects two cell boundary vertices; and two cells have a topological relationship if and only if they share at least one node.

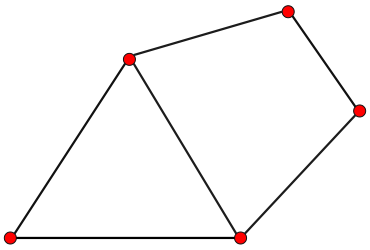


Figure I.5. A topology defined by a mesh with five nodes and six links.

For example, the mesh depicted in Figure I.5 may be used with any of three domain topology constructs for domains comprising two face cells (one triangle and one quadrilateral), six edge cells, and five point cells respectively.

A domain topology construct contains an array defining the mesh, and properties to describe it. There must be a property indicating the spatial dimensionality of the cells. The array values comprise the node identities, and all array elements that refer to the same node must contain the same value, which must differ from any other value in the array. The array spans the domain axis construct and also has a ragged dimension, whose function depends on the spatial dimensionality of the cells.

For each point cell, the first element along the ragged dimension contains the node identity of the cell, and the following elements contain in arbitrary order the identities of all the cells to which it is connected by a mesh link.

For each edge or face cell, the elements along the ragged dimension contain the node identities of the boundary vertices of the cell, in the same order that the boundary vertices are stored by the auxiliary coordinate constructs. Each boundary vertex except the last is connected by a mesh link

to the next vertex along the ragged dimension, and the last vertex is connected to the first.

When a domain topology construct is present it is considered to be definitive and must be used in preference to the topology implied by inspection of any other constructs, which is not guaranteed to be the same.

In CF-netCDF a domain topology construct can only be provided for a UGRID mesh topology variable. The information in the construct array is supplied by the UGRID "edge_nodes_connectivity" variable (for edge cells) or "face_nodes_connectivity" variable (for face cells). The topology for node cells may be provided by any of these three UGRID variables. The integer indices contained in the UGRID variable may be used as the mesh node identities, although the CF data model attaches no significance to the values other than the fact that some values are the same as others. The spatial dimensionality property is provided by the "location" attribute of a variable that references the UGRID mesh topology variable, i.e. a data variable or a UGRID location index set variable.

A single UGRID mesh topology defines multiple domain constructs and defines how they relate to each other. For instance, when "face_node_connectivity" and "edge_node_connectivity" variables are both present there are three implied domain constructs - one each for face, edge and point cells - all of which have the same mesh and so are explicitly linked (e.g. it is known which edge cells define each face cell). The CF data model has no mechanism for explicitly recording such relationships between multiple domain constructs. Whether or not two domains have the same mesh may be reliably deternined by inspection, thereby allowing the creation of netCDF datasets containing UGRID mesh topology variables.

The restrictions on the type of mesh that may be used with a given cell spatial dimensionality excludes some meshes which can be described by an undirected graph, but is consistent with UGRID encoding within CF-netCDF. UGRID also describes meshes for three-dimensional volume cells that correspond to a volume enclosed by a set of faces, but how the nodes relate to volume boundary vertices is undefined and so volume cells are currently omitted from the CF data model.

Cell connectivity construct

A cell connectivity construct defines explicitly how cells arranged in two or three dimensions in real space but indexed by a single domain (discrete) axis are connected. Connectivity can only be provided when the domain axis construct also has a domain topology construct, and two cells can only be connected if they also have a topological relationship. For instance, the connectivity of two-dimensional face cells could be characterised by whether or not they have shared edges, where the edges are defined by connected nodes of the domain topology construct.

The cell connectivity construct consists of an array recording the connectivity, and properties to describe the data. There must be a property indicating the condition by which the connectivity is derived from the domain topology. The array spans the domain axis construct with the addition of a ragged dimension. For each cell, the first element along the ragged dimension contains the unique identity of the cell, and the following elements contain in arbitrary order the identities of all the other cells to which the cell is connected. Note that the connectivity array for point cells is, by definition, equivalent to the array of the domain topology construct.

When cell connectivity constructs are present, they are considered to define the connectivity of the

cells. Exactly the same connectivity information could be derived from the domain topology construct. Connectivity information inferred from inspection of any other constructs is not guaranteed to be the same.

In CF-netCDF a cell topology construct can only be provided by a UGRID mesh topology variable. The construct array is supplied either indirectly by any of the UGRID variables that are used to define a domain topology construct, or directly by the UGRID "face_face_connectivity" variable (for face cells). In the direct case, the integer indices contained in the UGRID variable may be used as the cell identities, although the CF data model attaches no significance to the values other than the fact that some values are the same as others.

Restricting the types of connectivity to those implied by the geospatial topology of the cells precludes connectivity derived from any other sources, but is consistent with UGRID encoding within CF-netCDF.

Field ancillary constructs

The field ancillary construct (Figure I.2) provides metadata which are distributed over the same sampling domain as the field itself. For example, if a data variable holds a variable retrieved from a satellite instrument, a related ancillary data variable might provide the uncertainty estimates for those retrievals (varying over the same spatiotemporal domain).

The field ancillary construct consists of an array of the ancillary data which is either scalar or which depends on one, more or all of the domain axis constructs, and properties to describe the data (in the same sense as for the field construct). It is assumed that the data do not depend on axes of the domain which are not spanned by the array, along which the values are implicitly propagated. CF-netCDF ancillary data variables correspond to field ancillary constructs. Note that a field ancillary construct is constrained by the domain definition of the parent field construct but does not contribute to the domain's definition, unlike, for instance, an auxiliary coordinate construct or domain ancillary construct.

Cell method construct

The cell method constructs (Figure I.2) describe how the cell values represent the variation of the physical quantity within its cells—the structure of the data at a higher resolution. A single cell method construct consists of a set of axes (see below), a "method" property which describes how a value of the field construct's data array describes the variation of the quantity within a cell over those axes (e.g. a value might represent the cell area average), and properties serving to indicate more precisely how the method was applied (e.g. recording the spacing of the original data, or the fact the method was applied only over El Niño years).

The field construct may contain an ordered sequence of cell method constructs describing multiple processes which have been applied to the data, e.g. a temporal maximum of the areal mean has two components—a mean and a maximum, each acting over different sets of axes. It is an ordered sequence because the methods specified are not necessarily commutative. There are properties to indicate climatological time processing, e.g. multiannual means of monthly maxima, in which case multiple cell method constructs need to be considered together to define a special interpretation of boundary coordinate array values. The `cell_methods` attribute of a CF-netCDF data variable corresponds to one or more cell method constructs.

The axes over which a cell method applies are either a subset of the domain axis constructs or a collection of strings which identify axes that are not part of the domain. The latter case is particularly useful when the coordinate range for an axis cannot be precisely defined, making it impossible to define a domain axis construct. For example, a climatological time mean might be based on data which are not available over the same time periods at every horizontal location—the useful information that the data have been temporally averaged can be recorded without specifying the range of times. The strings which identify such axes are well defined in that they must be standard names (e.g. time, longitude) or the special string `area`, indicating a combination of horizontal axes.

Appendix J: Coordinate Interpolation Methods

The general description of the compression by coordinate subsampling is given in section Section 8.3, "Lossy Compression by Coordinate Subsampling". This appendix provides details on the available methods for compression by coordinate subsampling.

The definitions and guidance given here allow an application to compress an existing data set using coordinate subsampling, while letting the creator of the compressed dataset control the accuracy of the reconstituted coordinates through the degree of subsampling, the choice of interpolation method and by setting the computational precision.

Furthermore, the definitions given here allow an application to uncompress coordinate and auxiliary coordinate variables that have been compressed using coordinate subsampling. The key element of this process is the reconstitution of the full resolution coordinates in the domain of the data by interpolation between the subsampled coordinates, the tie points, stored in the compressed dataset.

The appendix is organised in a sections on Section J.1, "Common Definitions and Notation", Section J.2, "Common Conversions and Formulas", Section J.3, "Interpolation Methods" and finally two sections with step procedures Section J.4, "Coordinate Compression Steps" and Section J.5, "Coordinate Uncompression Steps".

Common Definitions and Notation

The coordinate interpolation methods are named to indicate the number of dimensions they interpolate as well as the type of interpolation provided. For example, the interpolation method named `linear` provides linear interpolation in one dimension and the method named `bi_linear` provides linear interpolation in two dimensions. Equivalently, the interpolation method named `quadratic` provides quadratic interpolation in one dimension and the interpolation method named `bi_quadratic` provides quadratic interpolation in two dimensions.

When an interpolation method is referred to as linear or quadratic, it means that the method is linear or quadratic in the indices of the interpolated dimensions.

For convenience, an interpolation argument `s` is introduced, calculated as a function of the index in the target domain of the coordinate value to be reconstituted. In the case of one-dimensional interpolation the variable is computed as:

```
s = s(ia, ib, i)
  = (i - ia)/(ib - ia)
```

where `ia` and `ib` are the indices in the target domain of the tie points `A` and `B` and `i` is the index in the target domain of the coordinate value to be reconstituted.

Note that the value of `s` varies from `0.0` at the tie point `A` to `1.0` at tie point `B`. For example, if `ia =`

`100` and `ib = 110` and the index in the target domain of the coordinate value to be reconstituted is `i = 105`, then `s = (105 - 100)/(110 - 100) = 0.5`.

In the case of two-dimensional interpolation, the interpolation arguments are similarly computed as:

```
s1 = s(ia1, ib1, i1)
   = (i1 - ia1)/(ib1 - ia1)
s2 = s(ia2, ic2, i2)
   = (i2 - ia2)/(ic2 - ia2)
```

where `ia1` and `ib1` are the first dimension indices in the target domain of the tie points `A` and `B` respectively, `ia2` and `ic2` are the second dimension indices in the target domain of the tie points `A` and `C` respectively and the indices `i1` and `i2` are the first and second dimension indices respectively in the target domain of the coordinate value to be reconstituted.

The target domain is segmented into smaller interpolation subareas as described in Section 8.3.1, "Tie Points and Interpolation Subareas".

For one-dimensional interpolation, an interpolation subarea is defined by two tie points, one at each end of the interpolation subarea. However, the tie points may be inside or outside the interpolation subareas as shown in Figure J.1. When interpolation methods are applied for a given interpolation subarea, it must be ensured that reconstituted coordinate points are only generated inside the interpolation subarea being processed, even if some of the tie point coordinates lie outside of that interpolation subarea. See also description in Section 8.3.1, "Tie Points and Interpolation Subareas".

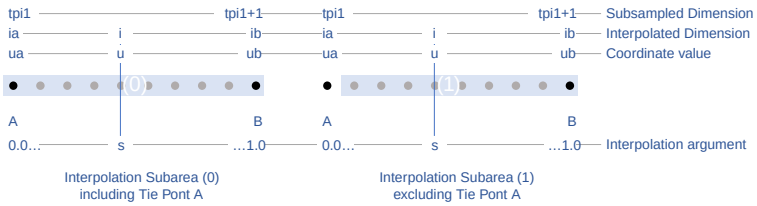


Figure J.1. One-dimensional interpolation subareas, one including and one excluding tie point `A`.

For two-dimensional interpolation, an interpolation subarea is defined by four tie points, one at each corner of a rectangular area aligned with the domain axes, see Figure J.2.

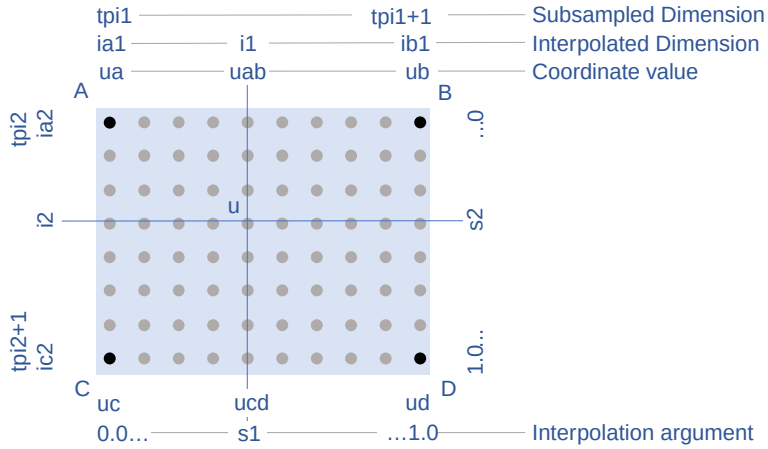


Figure J.2. Two-dimensional interpolation subarea.

For the reconstitution of the uncompressed coordinate and auxiliary coordinate variables the interpolation method can be applied independently for each interpolation subarea, making it possible to parallelize the computational process.

The following notation is used:

A variable staring with `v` denotes a vector and `v.x`, `v.y` and `v.z` refer to the three coordinates of that vector.

A variable staring with `ll` denotes a latitude-longitude coordinate pair and `ll.lat` and `ll.lon` refer to the latitude and longitude coordinates.

For one-dimensional interpolation, `i` is an index in the interpolated dimension, `tpi` is an index in the subsampled dimension and `is` is an index in the interpolation subarea dimensions.

For two-dimensional interpolation, `i1` and `i2` are indices in the interpolated dimensions, `tpi1` and `tpi2` are indices in the subsampled dimensions and `is1` and `is2` are indices in the interpolation subarea dimensions. Dimension 1 is the dimension with index values increasing from tie point `A` to tie point `B`, dimension 2 is the dimension with index values increasing from tie point `A` to tie point `C`.

Note that, for simplicity of notation, the descriptions of the interpolation methods in most places leave out the indices of tie point related variables and refer to these with `a` and `b` in the one-dimensional case and with `a`, `b`, `c` and `d` in the two-dimensional case. In the two-dimensional case, `a = tp(tpi2, tpi1)`, `b = tp(tpi2, tpi1+1)`, `c = tp(tpi2+1, tpi1)` and `d = tp(tpi2+1, tpi1+1)` would reflect the way the tie point data would be stored in the data set, see also Figure J.2.

Common Conversions and Formulas

Table J.1 describes conversions and formulas that are used in the descriptions of the interpolation techniques.

Table J.1. Conversions and formulas used in the definitions of subsampling interpolation methods

	Description	Formula
`sqrt`	Square Root	`s = sqrt(t)`
`atan2`	Inverse Tangent of `y/x`	`a = atan2(y, x)`

	Description	Formula
fll2v	Conversion from geocentric (latitude, longitude) to three-dimensional cartesian vector (x, y, z)	(x, y, z) = fll2v(ll) = (cos(ll.lat)*cos(ll.lon), cos(ll.lat)*sin(ll.lon), sin(ll.lat))
fv2ll	Conversion from three-dimensional cartesian vector (x, y, z) to geocentric (latitude, longitude)	(lat, lon) = fv2ll(v) = (atan2(v.z, sqrt(v.x * v.x + v.y * v.y)), atan2(v.y, v.x))
faz2v	Conversion from (azimuth, zenith) angles to three-dimensional cartesian vector (x, y, z)	(x, y, z) = faz2v(az) = (sin(az.zenith) * sin(az.azimuth), sin(az.zenith) * cos(az.azimuth), cos(az.zenith))
fv2az	Conversion from three-dimensional cartesian vector (x, y, z) to (azimuth, zenith) angles	(azimuth, zenith) = fv2az(v) = (atan2(y, x), atan2(sqrt(x * x + y * y), z))
fplus	Vector Sum	(x, y, z) = fplus(va, vb) = (va.x + vb.x, va.y + vb.y, va.z + vb.z) (x, y, z) = fplus(va, vb, vc) = (va.x + vb.x + vc.x, va.y + vb.y + vc.y, va.z + vb.z + vc.z)
fminus	Vector Difference	(x, y, z) = fminus(va, vb) = (va.x - vb.x, va.y - vb.y, va.z - vb.z)
fmultiply	Vector multiplied by Scalar	(x, y, z) = fmultiply(r, v) = (r * v.x, r * v.y, r * v.z)
fcross	Vector Cross Product	(x, y, z) = fcross(va, vb) = (va.y*vb.z - va.z*vb.y, va.z*vb.x - va.x*vb.z, va.x*vb.y - va.y*vb.x)
fdot	Vector Dot Product	d = fdot(va, vb) = va.x*vb.x + va.y*vb.y + va.z*vb.z

Interpolation Methods

Linear Interpolation

Name
`interpolation_name = "linear"`

Description
General purpose one-dimensional linear interpolation method for one or more coordinates

Interpolation parameter terms
None.

Coordinate compression calculations
None.

Coordinate uncompression calculations
The coordinate value `u(i)` at index `i` between tie points `A` and `B` is calculated from:

```
u(i) = fl(ua, ub, s(i))
     = ua + s*(ub-ua)
```

where `ua` and `ub` are the coordinate values at tie points `A` and `B` respectively.

Bilinear Interpolation

Name
`interpolation_name = "bi_linear"`

Description
General purpose two-dimensional linear interpolation method for one or more coordinates.

Interpolation parameter terms
None.

Coordinate compression calculations
None.

Coordinate uncompression calculations
The interpolation function fl() defined for linear interpolation above is first applied twice in the interpolated dimension 2, once between tie points `A` and `C` and once between tie points `B` and `D`.

It is then applied once in the interpolated dimension 1, between the two resulting coordinate points, yielding the interpolated coordinate value `u(i2, i1)`:

```
uac       = fl(ua, uc, s(ia2, ic2, i2))
ubd       = fl(ub, ud, s(ia2, ic2, i2))
u(i2, i1) = fl(uac, ubd, s(ia1, ib1, i1))
```

Quadratic Interpolation

Name
`interpolation_name = "quadratic"`

Description
General purpose one-dimensional quadratic interpolation method for one coordinate.

Interpolation parameter terms
Optionally the term `w`, specifying a numerical variable spanning the interpolation subarea dimension.

Coordinate compression calculations
The expression

```
w = fw(ua, ub, u(i), s(i))
  = (u - (1 - s) * ua - s * ub)/(4 * (1 - s) * s)
```

enables the creator of the dataset to calculate the coefficient `w` from the coordinate values `ua` and `ub` at tie points `A` and `B` respectively, and the coordinate value `u(i)` at index `i` between the tie points `A` and `B`. If the number of points in the interpolation subarea `(ib - ia + 1)` is odd, then the data point at index `i = (ib + ia)/2` shall be selected for this calculation, otherwise the data point at index `i = (ib + ia - 1)/2` shall be selected.

Coordinate uncompression calculations
The coordinate value `u(i)` at index `i` between tie points `A` and `B` is calculated from:

```
u(i) = fq(ua, ub, w, s(i))
     = ua + s * (ub - ua + 4 * w * (1 - s))
```

where `ua` and `ub` are the coordinate values at tie points `A` and `B` respectively and the coefficient `w` is available as a term in the `interpolation_parameters`, or otherwise defaults to `0.0`.

Quadratic Interpolation of Geographic Coordinates Latitude and Longitude

Name
`interpolation_name = "quadratic_latitude_longitude"`

Description
A one-dimensional quadratic method for interpolation of the geographic coordinates latitude and longitude, typically used for remote sensing products with geographic coordinates on the reference ellipsoid.

Requires a pair of latitude and longitude tie point variables, as defined unambiguously in Section 4.1, "Latitude Coordinate" and Section 4.2, "Longitude Coordinate". For each interpolation subarea, none of the tie points defining the interpolation subarea are permitted to coincide.

By default, interpolation is performed directly in the latitude and longitude coordinates, but may be performed in three-dimensional cartesian coordinates where required for achieving the desired accuracy. This must be indicated by setting the `location_use_3d_cartesian` flag within the interpolation parameter `interpolation_subarea_flags` for each interpolation subarea where interpolation in three-dimensional cartesian coordinates is required.

The quadratic interpolation coefficients `cea = (ce, ca)`, stored as interpolation parameters in the product, describe a point `P` between the tie points `A` and `B`, which is equivalent of an additional tie point in the sense that the method will accurately reconstitute the point `P` in the same way as it accurately reconstitutes the tie points `A` and `B`. See Figure J.3 and Figure J.4.

Although equivalent to a tie point, the coefficients `ce` and `ca` have two advantages over tie points. Firstly, they can often be stored as a lower precision floating point number compared to the tie points, as `ce` and `ca` only describes the position of `P` relative to the midpoint `M` between the tie points `A` and `B`. Secondly, if any of `ce` and `ca` do not contribute significantly to the accuracy of the reconstituted points, it can be left out of the data set and its value will default to zero during uncompression.

The coefficients may be represented in three different ways:

- For storage in the dataset as the non-dimensional coefficients `cea = (ce, ca)`, referred to as the parametric representation. The component `ce` is the offset projected on the line from tie point `B` to tie point `A` and expressed as a fraction of the distance between `A` and `B`. The component `ca` is the offset projected on the line perpendicular to the line from tie point `B` to tie point `A` and perpendicular to the plane spanned by `va` and `vb`, the vector representations of the two tie points, and expressed as a fraction of the length of `A x B`.
- For interpolation in three-dimensional cartesian coordinates as the coefficients `cv = (cv.x, cv.y, cv.z)`, expressing the offset components along the three-dimensional cartesian axes X, Y and Z respectively.
- For interpolation in geographic coordinates latitude and longitude as the coefficients `cll = (cll.lat, cll.lon)`, expressing the offset components along the longitude and latitude directions respectively.

The functions `fq()` and `fw()` referenced in the following are defined in Quadratic Interpolation.

Interpolation parameter terms
Optionally, any subset of terms `ce, ca`, each specifying a numerical variable spanning the interpolation subarea dimension.

The mandatory term `interpolation_subarea_flags`, specifying a flag variable spanning the interpolation subarea dimension and including `location_use_3d_cartesian` in the `flag_meanings` attribute.

Coordinate compression calculations
First calculate the tie point vector representations from the tie point latitude-longitude representations:

```
va = fll2v(lla)
vb = fll2v(llb)
```

Then calculate the three-dimensional cartesian representation of the interpolation coefficients from the tie points `va` and `vb` as well as the point `vp(i)` at index `i` between the tie points `A` and `B`. If the number of points in an interpolation subarea `(ib - ia + 1)` is odd, then the data point at index `i = (ib + ia)/2` shall be selected for this calculation, otherwise the data point at index `i = (ib + ia - 1)/2` shall be selected.

The three-dimensional cartesian interpolation coefficients are found from:

```
cv = fcv(va, vb, vp(i), s(i))
   = (fw(va.x, vb.x, vp(i).x, s(i)),
      fw(va.y, vb.y, vp(i).y, s(i)),
      fw(va.z, vb.z, vp(i).z, s(i)))
```

Finally, for storage in the dataset, convert the coefficients to the parametric representation:

```
cea(is) = (ce(is), ca(is))
```

```
        = fcv2cea(va, vb, cv)
        = (fdot(cv, fminus(va, vb))/gsqr,
           fdot(cv, fcross(va, vb))/(rsqr*gsqr))
```

where

```
vr   = fmultiply(0.5, fplus(va, vb))
rsqr = fdot(vr, vr)
vg   = fminus(va, vb)
gsqr = fdot(vg, vg)
```

The interpolation parameter term `interpolation_subarea_flags(is)` shall have the flag `location_use_3d_cartesian` set if the interpolation subarea intersects the `longitude = 180.0` or if the interpolation subarea extends into `latitude > latitude_limit` or `latitude < -latitude_limit`. The value of `latitude_limit` is set by the data set creator and defines the high latitude areas where interpolation in three-dimensional cartesian coordinates is required for reasons of coordinate reconstitution accuracy. The `latitude_limit` is used solely for setting the flag `location_use_3d_cartesian`, and is not required in a compressed dataset.

Coordinate uncompression calculations
First calculate the tie point vector representations from the tie point latitude-longitude representations:

```
va = fll2v(lla)
vb = fll2v(llb)
```

Then calculate the three-dimensional cartesian representation of the interpolation coefficients from the parametric representation stored in the dataset using:

```
cv = fcea2cv(va, vb, cea(is))
   = fplus(fmultiply(ce, fminus(va, vb)),
           fmultiply(ca, fcross(va, vb)),
           fmultiply(cr, vr))
```

where

```
vr   = fmultiply(0.5, fplus(va, vb))
rsqr = fdot(vr, vr)
cr   = sqrt(1 - ce(is)*ce(is) - ca(is)*ca(is)) - sqrt(rsqr)
```

If the flag `location_use_3d_cartesian` of the interpolation parameter term `interpolation_subarea_flags(is2, is1)` is set, use the following expression to reconstitute any point `llp(i)` between the tie points `A` and `B` using interpolation in three-dimensional cartesian coordinates:

```
vp(i)  = fqv(va, vb, cv, s(i))
       = (fq(va.x, vb.x, cv.x, s(i)),
          fq(va.y, vb.y, cv.y, s(i)),
          fq(va.z, vb.z, cv.z, s(i)))
llp(i) = fv2ll(vp(i))
```

Otherwise, first calculate latitude-longitude representation of the interpolation coefficients:

```
cll = fcll(lla, llb, llab)
    = (fw(lla.lat, llb.lat, llab.lat, 0.5),
       fw(lla.lon, llb.lon, llab.lon, 0.5))
```

where

```
llab = fv2ll(fqv(va, vb, cv, 0.5))
```

Then use the following expression to reconstitute any point `llp(i)` between the tie points `A` and `B` using interpolation in latitude-longitude coordinates:

```
llp(i) = (llp(i).lat, llp(i).lon)
       = fqll(lla, llb, cll, s(i))
       = (fq(lla.lat, llb.lat, cll.lat, s(i)),
          fq(lla.lon, llb.lon, cll.lon, s(i)))
```

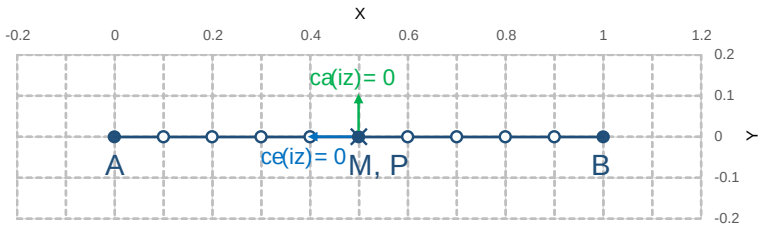


Figure J.3. With the expansion coefficient `ce = 0` and the alignment coefficient `ca = 0`, the method reconstitutes the points at regular intervals along a great circle between tie points `A` and `B`.

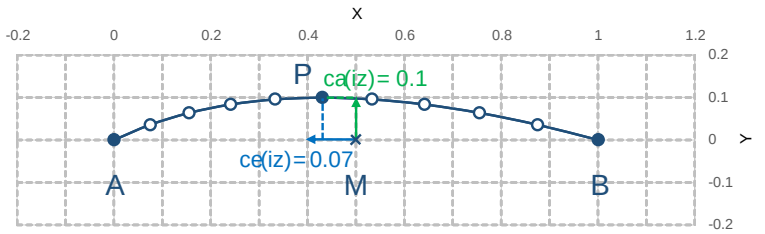


Figure J.4. With the expansion coefficient `ce > 0` and the alignment coefficient `ca > 0`, the method reconstitutes the points at intervals of expanding size (`ce`) along an arc with an alignment offset (`ca`) from the great circle between tie points `A` and `B`.

Biquadratic Interpolation of Geographic Coordinates Latitude and Longitude

Name

`interpolation_name = "bi_quadratic_latitude_longitude"`

Description

A two-dimensional quadratic method for interpolation of the geographic coordinates latitude and longitude, typically used for remote sensing products with geographic coordinates on the reference ellipsoid.

Requires a pair of latitude and longitude tie point variables, as defined unambiguously in Section 4.1, "Latitude Coordinate" and Section 4.2, "Longitude Coordinate". For each interpolation subarea, none of the tie points defining the interpolation subarea are permitted to coincide.

The functions `fcv()`, `fcv2cea()`, `fcea2cv()`, `fcll()`, `fqv()` and `fqll()` referenced in the following are defined in Quadratic Interpolation of Geographic Coordinates Latitude and Longitude. As for that method, interpolation is performed directly in the latitude and longitude coordinates or in three-dimensional cartesian coordinates, where required for achieving the desired accuracy. Similarly, it shares the three different representations of the quadratic interpolation coefficients, the parametric representation `cea = (ce, ca)` for storage in the dataset, `cll = (cll.lat, cll.lon)` for interpolation in geographic coordinates latitude and longitude and `cv = (cv.x, cv.y, cv.z)` for interpolation in three-dimensional cartesian coordinates.

The parametric representation of the interpolation coefficients, stored in the interpolation parameters `ce1`, `ca1`, `ce2`, `ca2`, `ce3` and `ca3`, is equivalent to five additional tie points for the interpolation subarea as shown in Figure J.5, which also shows the orientation and indices of the parameters.

Interpolation parameter terms

Optionally, any subset of terms `ce1` and `ca1`, each specifying a numerical variable spanning the subsampled dimension 2 and the interpolation subarea dimension 1.

Optionally, any subset of terms `ce2` and `ca2`, each specifying a numerical variable spanning the interpolation subarea dimension 2 and the subsampled dimension 1.

Optionally, any subset of terms `ce3` and `ca3`, each specifying a numerical variable spanning the interpolation subarea dimension 2 and the interpolation subarea dimension 1.

The mandatory term `interpolation_subarea_flags`, specifying a flag variable spanning the interpolation subarea dimension 2 and the interpolation subarea dimension 1 and including `location_use_3d_cartesian` in the `flag_meanings` attribute.

Coordinate compression calculations

First calculate the tie point vector representations from the tie point latitude-longitude representations:

```
va = fll2v(lla)
vb = fll2v(llb)
vc = fll2v(llc)
```

```
vd = fll2v(lld)
```

Then calculate the three-dimensional cartesian representation of the interpolation coefficients sets from the tie points as well as a point `vp(i2, i1)` between the tie points. If the number of points in the first dimension of the interpolation subarea `(ib1 - ia1 + 1)` is odd, then the data point at index i1 = `(ib1 + ia1)/2` shall be selected for this calculation, otherwise the data point at index `i1 = (ib1 + ia1 - 1)/2` shall be selected. If the number of points in the second dimension of the interpolation subarea `(ic2 - ia2 + 1)` is odd, then the data point at index `i2 = (ic2 + ica)/2` shall be selected for this calculation, otherwise the data point at index `i2 = (ic2 + ia2 - 1)/2` shall be selected.

Using the selected `(i2, i1)`, the three-dimensional cartesian interpolation coefficients are found from:

```
s1    = s(ia1, ib1, i1)
s2    = s(ia2, ic2, i2)
vac   = fll2v(ll(i2, ia1))
vbd   = fll2v(ll(i2, ib1))
cv_ac = fcv(va, vc, vac, s2)
cv_bd = fcv(vb, vd, vbd, s2)
cv_ab = fcv(va, vb, fll2v(ll(ia2, i1)), s1)
cv_cd = fcv(vc, vd, fll2v(ll(ic2, i1)), s1)
cv_zz = fcv(vac, vbd, fll2v(ll(i2, i1)), s1)
vz    = fqv(vac, vbd, cv_zz, 0.5)
vab   = fqv(va, vb, cv_ab, 0.5)
vcd   = fqv(vc, vd, cv_cd, 0.5)
cv_z  = fcv(vab, vcd, vz, s2)
```

Finally, before storing them in the dataset's interpolation parameters, convert the coefficients to the parametric representation:

```
cea1(tpi2, is1)   = fcv2cea(va, vb, cv_ab)
cea1(tpi2+1, is1) = fcv2cea(vc, vd, cv_cd)
cea2(is2, tpi1)   = fcv2cea(va, vc, cv_ac)
cea2(is2, tpi1+1) = fcv2cea(vb, vd, cv_bd)
cea3(is2, is1)    = fcv2cea(vab, vcd, cv_z)
```

The interpolation parameter term `interpolation_subarea_flags(is2, is1)` shall have the flag `location_use_3d_cartesian` set if the interpolation subarea intersects the `longitude = 180.0` or if the interpolation subarea extends into `latitude > latitude_limit` or `latitude < -latitude_limit`. The value of `latitude_limit` is set by the data set creator and defines the high latitude areas where interpolation in three-dimensional cartesian coordinates is required for reasons of coordinate reconstitution accuracy. The `latitude_limit` is used solely for setting the flag `location_use_3d_cartesian`, and is not required in a compressed dataset.

Coordinate uncompression calculations

First calculate the tie point vector representations from the tie point latitude-longitude representations:

```
va = fll2v(lla)
vb = fll2v(llb)
vc = fll2v(llc)
vd = fll2v(lld)
```

Then calculate the three-dimensional cartesian representation of the interpolation coefficient sets from the parametric representation stored in the dataset:

```
cv_ac = fcea2cv(va, vc, cea2(is2, tpi1))
cv_bd = fcea2cv(vb, vd, cea2(is2, tpi1 + 1))
vab   = fqv(va, vb, fcea2cv(va, vb, cea1(tpi2, is1)), 0.5)
vcd   = fqv(vc, vd, fcea2cv(vc, vd,  cea1(tpi2 + 1, is1)), 0.5)
cv_z  = fcea2cv(vab, vcd, cea3(is2, is1))
```

If the flag `location_use_3d_cartesian` of the interpolation parameter term `interpolation_subarea_flags` is set, use the following expression to reconstitute any point `llp(i2, i1)` between the tie points `A` and `B` using interpolation in three-dimensional cartesian coordinates:

```
llp(i2, i1) = fv2ll(fqv(vac, vbd, cv_zz, s(ia1, ib1, i1)))
```

where

```
s2    = s(ia2, ic2, i2)
vac   = fqv(va, vc, cv_ac, s2)
vbd   = fqv(vb, vd, cv_bd, s2)
vz    = fqv(vab, vcd, cv_z, s2)
cv_zz = fcv(vac, vbd, vz, 0.5)
```

Otherwise, first calculate latitude-longitude representation of the interpolation coefficients:

```
llc_ac = fcll(lla, llc, fv2ll(fqv(va, vc, cv_ac, 0.5)))
llc_bd = fcll(llb, lld, fv2ll(fqv(vb, vd, cv_bd, 0.5)))
llab   = fv2ll(vab)
llcd   = fv2ll(vcd)
llc_z  = fcll(llab, llcd, fv2ll(fqv(vab, vcd, cv_z, 0.5)))
```

Then use the following expression to reconstitute any point `llp(i2, i1)` in the interpolation subarea using interpolation in latitude-longitude coordinates:

```
llp(i2, i1) = fqll(llac, llbd, cl_zz, s(ia1, ib1, i1))
```

where

```
s2    = s(ia2, ic2, i2)
llac  = fqll(lla, llc, llc_ac, s2)
llbd  = fqll(llb, lld, llc_bd, s2)
llz   = fqll(llab, llcd, llc_z, s2)
cl_zz = fcll(llac, llbd, llz)
```

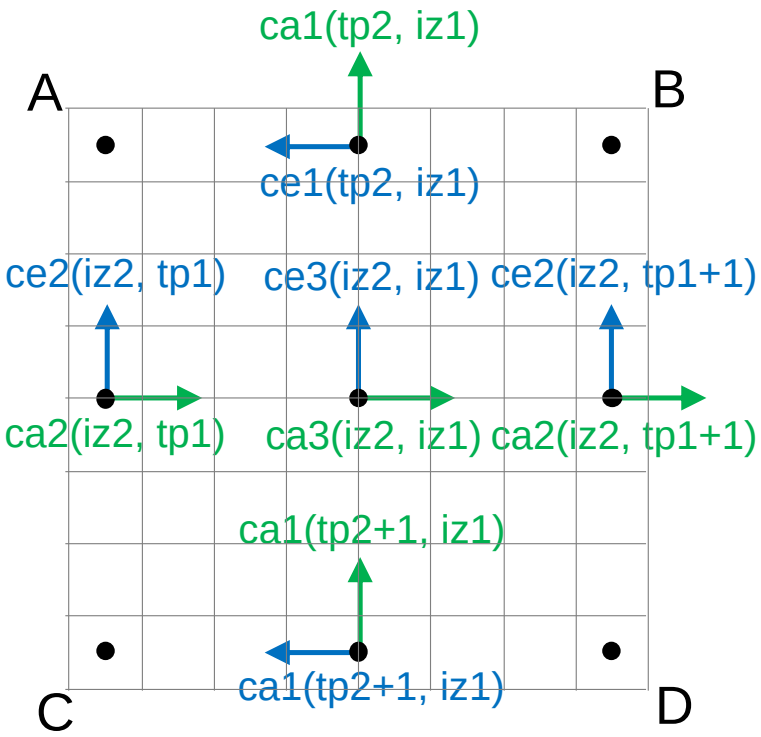


Figure J.5. The parametric representation of the interpolation coefficients `cea = (ce, ca)`, stored in the interpolation parameters `ce1`, `ca1`, `ce2`, `ca2`, `ce3` and `ca3`, is equivalent to five additional tie points for the interpolation subarea. Shown with parameter orientation and indices.

Coordinate Compression Steps

Step	Description	Link
1	Identify the coordinate and auxillary coordinate variables for which tie point and interpolation variables are required.	

Step	Description	Link
2	Identify non-overlapping subsets of the coordinate variables to be interpolated by the same interpolation method. For each coordinate variable subset, create an interpolation variable and specify the selected interpolation method using the `interpolation_name` attribute of the interpolation variable.	Section 8.3.3, "Interpolation Variable"
3	For each coordinate variable subset, add the coordinates variable subset and the corresponding interpolation variable name to the the `coordinate_interpolation` attribute of the data variable.	Section 8.3.2, "Coordinate Interpolation Attribute"
4	For each coordinate variable subset, identify the set of interpolated dimensions and the set of non-interpolated dimensions.	Section 8.3.4, "Subsampled, Interpolated and Non-Interpolated Dimensions"
5	For each set of the interpolated dimensions, identify the continuous areas and select the interpolation subareas and the tie points, taking into account the required coordinate reconstitution accuracy when selecting the density of tie points.	Section 8.3.1, "Tie Points and Interpolation Subareas"
6	For each of the interpolated dimensions, add the interpolated dimension, the corresponding subsampled dimension and, if required by the selected interpolation method, its corresponding interpolation subarea dimension to the `tie_point_mapping` attribute of the interpolation variable.	Section 8.3.5, "Tie Point Mapping Attribute" Section 8.3.6, "Tie Point Dimension Mapping"

Step	Description	Link
7	For each of the interpolated dimensions, record the location of each identified tie point in a tie point index variable. For each interpolated dimension, add the tie point index variable name to the `tie_point_mapping` attribute of the interpolation variable.	Section 8.3.5, "Tie Point Mapping Attribute" Section 8.3.7, "Tie Point Index Mapping"
8	For each of the target coordinate and auxillary coordinate variables, create the corresponding tie point coordinate variable and copy the coordinate values from the target domain coordinate variables to the tie point variables for the target domain indices identified by the tie point index variable. Repeat this step for each combination of indices of the non-interpolated dimensions.	Section 8.3.5, "Tie Point Mapping Attribute" Section 8.3.7, "Tie Point Index Mapping"
9	For each of the target coordinate and auxillary coordinate variable having a `bounds` attribute, add the `bounds_tie_points` attribute to the tie point coordinate variable and create the bounds tie point variable. For each continuous area, copy the selected set of bounds tie points values from the target domain bounds variable to the bounds tie point variable for the target domain indices identified by the tie point index variable. Repeat this step for each combination of indices of the non-interpolated dimensions.	Section 8.3.9, "Interpolation of Cell Boundaries"

Step	Description	Link
10	If required by the selected interpolation method, follow the steps defined for the method in Section J.3, "Interpolation Methods" to create any required interpolation parameter variables. As relevant, create the `interpolation_parameters` attribute and populate it with the interpolation parameter variables.	Section 8.3.3, "Interpolation Variable" Section J.3, "Interpolation Methods"
11	Optionally, check the consistency of the original coordinates and the reconstructed coordinates and add a `comments` attribute to one or more of the tie point coordinate variables reporting key figures like maximum error, mean error, etc.	

Coordinate Uncompression Steps

Step	Description	Link
1	From the `coordinate_interpolation` attribute of the data variable, identify the coordinate and auxillary coordinate variable subsets, for which tie point interpolation is required, and the interpolation variable corresponding to each subset.	Section 8.3.2, "Coordinate Interpolation Attribute"
2	For each coordinate variable subset, identify the interpolation method from the `interpolation_name` attribute of the interpolation variable.	Section 8.3.3, "Interpolation Variable"

Step	Description	Link
3	For each coordinate variable subset, identify the set of interpolated dimensions and the set of non-interpolated dimensions from the `tie_point_mapping` attribute of the interpolation variable.	Section 8.3.5, "Tie Point Mapping Attribute" Section 8.3.6, "Tie Point Dimension Mapping"
4	From the `tie_point_mapping` attribute of the interpolation variable, identify for each of the interpolated dimensions the corresponding subsampled dimension and, if defined, the corresponding interpolation subarea dimension.	Section 8.3.5, "Tie Point Mapping Attribute" Section 8.3.6, "Tie Point Dimension Mapping"
5	From the tie point index variables referenced in the `tie_point_mapping` attribute of the interpolation variable, identify the location of the tie points in the corresponding interpolated dimension.	Section 8.3.5, "Tie Point Mapping Attribute" Section 8.3.7, "Tie Point Index Mapping"
6	For each of the interpolated dimensions, identify pairs of adjacent indices in the tie point index variable with index values differing by more than one, each index pair defining the extend of an interpolation subarea in that dimension. A full interpolation subarea is defined by one such index pair per interpolated dimension, with combinations of one index from each pair defining the interpolation subarea tie points.	Section 8.3.1, "Tie Points and Interpolation Subareas"
7	As required by the selected interpolation method, identify the interpolation parameter variables from the interpolation variable attribute `interpolation_parameters`.	Section 8.3.8, "Interpolation Parameters"

Step	Description	Link
8	For each of the tie point coordinate and auxillary coordinate variables, create the corresponding target coordinate variable. For each interpolation subarea, apply the interpolation method, as described in Section J.3, "Interpolation Methods", to reconstitute the target domain coordinate values and store these in the target domain coordinate variables. Repeat this step for each combination of indices of the non-interpolated dimensions.	Section 8.3.5, "Tie Point Mapping Attribute" Section J.3, "Interpolation Methods"
9	For each of the tie point coordinate and auxillary coordinate variables having a `bounds_tie_points` attribute, add the `bounds` attribute to the target coordinate variable and create the target domain bounds variable. For each interpolation subarea, apply the interpolation method to reconstitute the target domain bound values and store these in the target domain bound variables. Repeat this step for each combination of indices of the non-interpolated dimensions.	Section 8.3.9, "Interpolation of Cell Boundaries"
10	If auxiliary coordinate variables have been reconstituted, then, if not already present, add a `coordinates` attribute to the data variable and add to the attribute the list of the names of the reconstituted auxiliary coordinate variables.	Chapter 5, Coordinate Systems and Domain

Appendix K: Mesh Topologies

The CF attributes listed here may be used to define mesh topologies (Section 5.9, "Mesh Topology Variables"). This list is intended as a summary of the attributes that have been standardized via the UGRID conventions [UGRID], which should be consulted for further details. UGRID attributes that are not currently recognised by the CF convensions are included in the list.

The "Type" values are **S** for string and **I** for integer. The "Use" values are **MT** for mesh topology variables, **LIS** for location index set variables, **D** for data variables, **Do** for domain variables, and **Con** for connectivity index variables.

Table K.1. Mesh topology attributes

Attribute	Type	Use	Description
boundary_node_connectivity	S	MT	Specifies an index variable identifying the nodes that define where boundary condtions have been provided. Not currently recognised by the CF conventions.
cf_role	S	MT, LIS	Specifies the roles of mesh topology or location index set variables.
coordinates	S	LIS	Specifies the auxiliary coordinate variables associated with the characteristic locations of the subset of mesh topology locations.
edge_coordinates	S	MT	Specifies the auxiliary coordinate variables associated with the characteristic location of the edges (commonly the midpoint).
edge_dimension	S	MT	Specifies the dimension used to index the nodes in the edge connectivity variable.
edge_face_connectivity	S	MT	Specifies an index variable identifying all faces that share the same edge, i.e. are neighbours to an edge.
edge_node_connectivity	S	MT	Specifies an index variable identifying for every edge the indices of its begin and end nodes.
face_coordinates	S	MT	Specifies the auxiliary coordinate variables associated with the characteristic location of faces.
face_dimension	S	MT	Specifies the dimension used to index the edges in the face connectivity variable.
face_edge_connectivity	S	MT	Specifies an index variable identifying for every face the indices of its edges.
face_face_connectivity	S	MT	Specifies an index variable identifying all faces that share an edge with each face, i.e. are neighbours.

Attribute	Type	Use	Description
face_node_connectivity	S	MT	Specifies an index variable identifying for every face the indices of its corner nodes.
location	S	D, Do, LIS	Specifies the location within the mesh topology at which the variable is defined.
location_index_set	S	D, Do	Specifies a variable that defines the subset of locations of a mesh topology at which the variable is defined.
mesh	S	D, Do, LIS	Specifies a variable that defines a mesh topology.
node_coordinates	S	MT	Specifies the auxiliary coordinate variables representing the node locations (latitude, longitude, or other spatial coordinates, and optional elevation or other coordinates).
start_index	I	LIS, Con	Indicates whether 0- or 1-based indexing is used to identify connected geometric elements; connectivity indices are 0-based by default.
topology_dimension	I	MT	Indicates the highest dimensionality of the geometric elements.
volume_coordinates	S	MT	Specifies the auxiliary coordinate variables associated with the characteristic location of volumes. Not currently recognised by the CF conventions.
volume_dimension	S	MT	Specifies the dimension used to index the faces in the volume connectivity variable. Not currently recognised by the CF conventions.
volume_edge_connectivity	S	MT	Specifies an index variable identifying for every volume the indices of its edges.
volume_face_connectivity	S	MT	Specifies an index variable identifying for every volume the indices of its faces. Not currently recognised by the CF conventions.
volume_node_connectivity	S	MT	Specifies an index variable identifying for every volume the indices of its corner nodes. Not currently recognised by the CF conventions.
volume_shape_type	S	MT	Specifies a flag variable that specifies for every volume its shape. Not currently recognised by the CF conventions.
volume_volume_connectivity	S	MT	Specifies an index variable identifying all volumes that share a face with each volume, i.e. are neighbours. Not currently recognised by the CF conventions.

Appendix L: Aggregation Variable Examples

This appendix contains examples of aggregation variables. Details of how to encode and decode aggregation variables are found in Section 2.8, "Aggregation Variables".

Example L.1 Aggregation variable with fragment datasets defined by relative-path URI references

```
dimensions:
  time = 12 ;
  level = 1 ;
  latitude = 73 ;
  longitude = 144 ;
  // Array of fragments dimensions
  f_time = 2 ;
  f_level = 1 ;
  f_latitude = 1 ;
  f_longitude = 1 ;
  // Map variable dimensions
  j = 4 ;        // Number of aggregated dimensions
  i = 2 ;        // Largest number of fragments along any aggregated dimension

variables:
  // Data aggregation variable
  double temperature ;
    temperature:standard_name = "air_temperature" ;
    temperature:units = "K" ;
    temperature:cell_methods = "time: mean" ;
    temperature:aggregated_dimensions = "time level latitude longitude" ;
    temperature:aggregated_data = "uris: fragment_uris
                                   identifiers: fragment_identifiers
                                   map: fragment_map" ;
  // Coordinate variables
  double time(time) ;
    time:standard_name = "time" ;
    time:units = "days since 2001-01-01" ;
  double level(level) ;
    level:standard_name = "height_above_mean_sea_level" ;
    level:units = "m" ;
  double latitude(latitude) ;
    latitude:standard_name = "latitude" ;
    latitude:units = "degrees_north" ;
  double longitude(longitude) ;
    longitude:standard_name = "longitude" ;
    longitude:units = "degrees_east" ;
  // Array of fragments feature variables
  string fragment_uris(f_time, f_level, f_latitude, f_longitude) ;
  string fragment_identifiers ;
  int fragment_map(j, i) ;
```

```
data:
  temperature = _ ;
  time = 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 ;
  level = ... ;
  latitude = ... ;
  longitude = ... ;
  fragment_map = 3, 9,
                 1, _,
                 73, _,
                 144, _ ;
  fragment_uris = "January-March.nc", "April-December.nc" ;
  fragment_identifiers = "temperature" ;
```

The `temperature` data variable is an aggregation variable. Its four-dimensional aggregated data with shape `(12, 1, 73, 144)` is constructed from two non-overlapping fragments, with data shapes `(3, 1, 73, 144)` and `(9, 1, 73, 144)`, which span the first 3 and last 9 elements respectively of the `time` aggregated dimension. The fragment dataset names are relative-path URI references, and so in this case are assumed to be in the same directory as the aggregation dataset itself.

The data for the `level`, `latitude` and `longitude` variables are omitted for clarity.

Example L.2 Aggregation variable with fragment datasets defined by absolute URIs

```
dimensions:
  time = 12 ;
  level = 1 ;
  latitude = 73 ;
  longitude = 144 ;
  // Array of fragments dimensions
  f_time = 2 ;
  f_level = 1 ;
  f_latitude = 1 ;
  f_longitude = 1 ;
  // Map variable dimensions
  j = 4 ;        // Number of aggregated dimensions for temperature
  j_time = 1 ;   // Number of aggregated dimensions for time
  i = 2 ;        // Largest number of fragments along any aggregated dimension

variables:
  // Data aggregation variable
  double temperature ;
    temperature:standard_name = "air_temperature" ;
    temperature:units = "K" ;
    temperature:cell_methods = "time: mean" ;
    temperature:aggregated_dimensions = "time level latitude longitude" ;
    temperature:aggregated_data = "uris: fragment_uris
                                   identifiers: fragment_identifiers
                                   map: fragment_map" ;
```

```
  // Coordinate aggregation variable
  double time ;
    time:standard_name = "time" ;
    time:units = "days since 2001-01-01" ;
    time:aggregated_dimensions = "time" ;
    time:aggregated_data = "uris: fragment_uris
                            identifiers: fragment_identifiers_time
                            map: fragment_map_time" ;
  // Coordinate variables
  double level(level) ;
    level:standard_name = "height_above_mean_sea_level" ;
    level:units = "m" ;
  double latitude(latitude) ;
    latitude:standard_name = "latitude" ;
    latitude:units = "degrees_north" ;
  double longitude(longitude) ;
    longitude:standard_name = "longitude" ;
    longitude:units = "degrees_east" ;
  // Array of fragments feature variables
  int fragment_map(j, i) ;
  string fragment_uris(f_time, f_level, f_latitude, f_longitude) ;
  string fragment_identifiers ;
  int fragment_map_time(j_time, i) ;
  string fragment_uris_time(f_time) ;
  string fragment_identifiers_time ;

data:
  temperature = _ ;
  time = _ ;
  level = ... ;
  latitude = ... ;
  longitude = ... ;
  fragment_map = 3, 9,
                 1, _,
                 73, _,
                 144, _ ;
  fragment_uris = "file:///data/January-March.nc",
                  "https:///remote.host/data/April-December.nc" ;
  fragment_identifiers = "temperature" ;
  fragment_map_time = 3, 9 ;
  fragment_uris_time = "file:///data/January-March.nc",
                       "https:///remote.host/data/April-December.nc" ;
  fragment_identifiers_time = "time" ;
```

This example is similar to Example L.1, but now the fragment dataset names are absolute URIs (one local, one remote), and `time` is now also an aggregation coordinate variable, with its aggregated data being derived from the same fragment datasets as `temperature`.

The data for the `level`, `latitude` and `longitude` variables are omitted for clarity.

Example L.3 Aggregation variable with multiple aggregated dimensions

```
dimensions:
  time = 12 ;
  level = 1 ;
  latitude = 73 ;
  longitude = 144 ;
  // Array of fragments dimensions
  f_time = 12 ;
  f_level = 1 ;
  f_latitude = 2 ;
  f_longitude = 4 ;
  // Map variable dimensions
  j = 4 ;        // Number of aggregated dimensions
  i = 12 ;       // Largest number of fragments along any aggregated dimension

variables:
  // Data aggregation variable
  double temperature ;
    temperature:standard_name = "air_temperature" ;
    temperature:units = "K" ;
    temperature:cell_methods = "time: mean" ;
    temperature:aggregated_dimensions = "time level latitude longitude" ;
    temperature:aggregated_data = "uris: fragment_uris
                                   identifiers: fragment_identifiers
                                   map: fragment_map" ;
  double pressure(time, level, latitude, longitude) ;
    temperature:standard_name = "air_pressure" ;
    temperature:units = "hPa" ;
    temperature:cell_methods = "time: mean" ;

  // Coordinate variables
  double time(time) ;
    time:standard_name = "time" ;
    time:units = "days since 2001-01-01" ;
  double level(level) ;
    level:standard_name = "height_above_mean_sea_level" ;
    level:units = "m" ;
  double latitude(latitude) ;
    latitude:standard_name = "latitude" ;
    latitude:units = "degrees_north" ;
  double longitude(longitude) ;
    longitude:standard_name = "longitude" ;
    longitude:units = "degrees_east" ;
  // Array of fragments feature variables
  int fragment_map(j, i) ;
  string fragment_uris(f_time, f_level, f_latitude, f_longitude) ;
  string fragment_identifiers ;

data:
```

```
  temperature = _ ;
  pressure = ...  ;
  time = 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 ;
  level = ... ;
  latitude = ... ;
  longitude = ... ;
  fragment_map = 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
                 1, _, _, _, _, _, _, _, _, _, _, _,
                 37, 36, _, _, _, _, _, _, _, _, _, _,
                 36, 36, 36, 36, _, _, _, _, _, _, _, _ ;
  fragment_uris = ... ;
  fragment_identifiers = "temperature" ;
```

The `temperature` data variable is an aggregation of 96 fragments. The shape of the array of fragments, inferred from the `fragment_map` data, is `(12, 1, 2, 4)`, indicating that three of the four aggregated dimensions are spanned by multiple fragments. The `pressure` data variable is not an aggregation variable.

The data for the `pressure`, `level`, `latitude` and `longitude` variables, and the `fragment_uris` variable, are omitted for clarity.

Example L.4 Aggregation discrete sampling geometry variable

```
dimensions:
  station = 3 ;
  obs = 15000 ;
  // Array of fragments dimensions
  f_station = 3 ;
  // Map variable dimensions
  j = 1 ;        // Number of aggregated dimensions
  i = 3 ;        // Largest number of fragments along any aggregated dimension

variables:
  // Data aggregation variable
  float tas(obs) ;
    tas:standard_name = "air_temperature" ;
    tas:units = "K" ;
    tas:coordinates = "time lat lon station_name" ;
    tas:aggregated_dimensions = "obs" ;
    tas:aggregated_data = "uris: fragment_uris
                           identifiers: fragment_identifiers
                           map: fragment_map" ;
  // DSG count variable
  int row_size(station) ;
    row_size:long_name = "number of observations per station" ;
    row_size:sample_dimension = "obs" ;

  // Auxiliary coordinate aggregation variables
  float time ;
```

```
    time:standard_name = "time" ;
    time:units = "days since 1970-01-01" ;
    time:aggregated_dimensions = "obs" ;
    time:aggregated_data = "uris: fragment_uris
                            identifiers: fragment_identifiers_time
                            map: fragment_map" ;
  float lon(station) ;
    lon:standard_name = "longitude" ;
    lon:long_name = "station longitude" ;
    lon:units = "degrees_east" ;
    lon:aggregated_dimensions = "station" ;
    lon:aggregated_data = "uris: fragment_uris
                           identifiers: fragment_identifiers_lon
                           map: fragment_map_latlon" ;
  float lat(station) ;
    lat:standard_name = "latitude" ;
    lat:long_name = "station latitude" ;
    lat:units = "degrees_north" ;
    lat:aggregated_dimensions = "station" ;
    lat:aggregated_data = "uris: fragment_uris
                           identifiers: fragment_identifiers_lat
                           map: fragment_map_latlon" ;
  // Array of fragments feature variables
  int fragment_map(j, i) ;
  string fragment_uris(f_station) ;
  string fragment_identifiers ;
  int fragment_map_latlon(j, i) ;
  string fragment_identifiers_time(f_station) ;
  string fragment_identifiers_lat ;
  string fragment_identifiers_lon ;

// global attributes:
  :featureType = "timeSeries" ;

data:
  tas = _ ;
  row_size = 5000, 4000, 6000 ;
  time = _ ;
  lat = _ ;
  lon = _ ;
  fragment_map = 5000, 4000, 6000 ;
  fragment_uris = "Harwell.nc", "Abingdon.nc", "Lambourne.nc" ;
  fragment_identifiers = "tas" ;
  fragment_map_latlon = 1, 1, 1 ;
  fragment_identifiers_time = "t1", "t2", "t3" ;
  fragment_identifiers_lat = "lat" ;
  fragment_identifiers_lon = "lon" ;
```

Three fragments are aggregated into a collection of discrete sampling geometry (DSG) timeseries feature types with contiguous ragged array representation. The auxiliary

coordinate variables `time`, `lon`, and `lat` are also aggregation variables. The time variables in the fragment datasets all have different netCDF variable names, which differ from the netCDF name of the `time` aggregation variable. The fragments for all aggregation variables, in this case, come from the same three fragment datasets.

No data have been omitted from the CDL.

Example L.5 Aggregation ancillary variable with unique fragment values

```
dimensions:
  time = 12 ;
  level = 1 ;
  latitude = 73 ;
  longitude = 144 ;
  // Array of fragments dimensions
  f_time = 2 ;
  f_level = 1 ;
  f_latitude = 1 ;
  f_longitude = 1 ;
  // Map variable dimensions
  j = 4 ;        // Number of aggregated dimensions for temperature
  i = 2 ;        // Largest number of fragments along any aggregated dimension
  j_uid = 1 ;    // Number of aggregated dimensions for uid

variables:
  // Data aggregation variable
  double temperature ;
    temperature:standard_name = "air_temperature" ;
    temperature:units = "K" ;
    temperature:cell_methods = "time: mean" ;
    temperature:ancillary_variables = "uid" ;
    temperature:aggregated_dimensions = "time level latitude longitude" ;
    temperature:aggregated_data = "uris: fragment_uris
                                   identifiers: fragment_identifiers
                                   map: fragment_map" ;
  // Ancillary aggregation variable
  string uid ;
    uid:long_name = "Fragment dataset unique identifiers" ;
    uid:missing_value = "" ;
    uid:aggregated_dimensions = "time" ;
    uid:aggregated_data = "unique_values: fragment_unique_values
                           map: fragment_map_uid" ;
  // Coordinate variables
  double time(time) ;
    time:standard_name = "time" ;
    time:units = "days since 2001-01-01" ;
  double level(level) ;
    level:standard_name = "height_above_mean_sea_level" ;
    level:units = "m" ;
```

```
  double latitude(latitude) ;
    latitude:standard_name = "latitude" ;
    latitude:units = "degrees_north" ;
  double longitude(longitude) ;
    longitude:standard_name = "longitude" ;
    longitude:units = "degrees_east" ;
  // Array of fragments feature variables
  int fragment_map(j, i) ;
  string fragment_uris(f_time, f_level, f_latitude, f_longitude) ;
  string fragment_identifiers ;
  int fragment_map_uid(j_uid, i) ;
  string fragment_unique_values(f_time) ;

data:
  temperature = _ ;
  uid = _ ;
  time = 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334 ;
  level = ... ;
  latitude = ... ;
  longitude = ... ;
  fragment_map = 3, 9,
                 1, _,
                 73, _,
                 144, _ ;
  fragment_uris = "January-March.nc", "April-December.nc" ;
  fragment_identifiers = "temperature" ;
  fragment_map_uid = 3, 9 ;
  fragment_unique_values = "04b9-7eb5-4046-97b-0bf8", "05ee0-a183-43b3-a67-1eca" ;
```

This example is similar to Example L.1, but now there is an additional aggregation ancillary variable `uid` which defines its fragments from the unique values stored in the `fragment_unique_values` variable.

The data for the `level`, `latitude` and `longitude` variables are omitted for clarity.

Example L.6 Aggregation variable with a scalar map variable

```
dimensions:

variables:
  // Data aggregation variable
  double temperature ;
    temperature:standard_name = "air_temperature" ;
    temperature:units = "K" ;
    temperature:cell_methods = "time: mean" ;
    temperature:aggregated_dimensions = "" ;
    temperature:aggregated_data = "uris: fragment_uris
                                   identifiers: fragment_identifiers
                                   map: fragment_map" ;
```

```
  // Scalar coordinate variables
  double time ;
    time:standard_name = "time" ;
    time:units = "days since 2001-01-01" ;
  double height ;
    level:standard_name = "height" ;
    level:units = "m" ;
  double latitude ;
    latitude:standard_name = "latitude" ;
    latitude:units = "degrees_north" ;
  double longitude ;
    longitude:standard_name = "longitude" ;
    longitude:units = "degrees_east" ;
  // Array of fragments feature variables
  int fragment_map ;
  string fragment_uris ;
  string fragment_identifiers ;

data:
  temperature = _ ;
  time = 0 ;
  height = 1.5 ;
  latitude = 43.7 ;
  longitude = 7.27 ;
  fragment_map = 1 ;
  fragment_uris = "file.nc" ;
  fragment_identifiers = "tas" ;
```

An aggregation variable with scalar aggregated data, for which the `aggregated_dimensions` attribute is an empty string, and the map variable `fragment_map` is a scalar with the value `1`.

Revision History

Version 1.13-draft

- Issue #433: Convention for HEALPix grid parameters
- Issue #579: Introduce `authors.adoc` as a unified database of author metadata, for consistency within the conventions document and with `zenodo.json` and `CITATION.cff`, which are now generated automatically
- Issue #590: Clarify that grid_mapping can also be used for converting spatial bounds
- Issue #593: Clarify that rules for attributes of boundary variables (including BI and BO) also apply for attributes of climatological boundary variables
- Issue #583: Correct "most rapidly varying dimension" in terminology section.
- Issue #584: Allow `Z` as time-zone offset, with a couple of examples, and allow time-zone offset with date alone, both being consistent with UDUNITS syntax.
- Issue #508: Introduce aggregation variables

Version 1.12 (04 December 2024)

- Issue #513: Include DOI and License information in the conventions document
- Issue #499: Formatting of local links in text
- Issue #566: Fix invalid CRS WKT attribute in example 5.12.
- Issue #527: Clarify the conventions for boundary variables, especially for auxiliary coordinate variables of more than one dimension, state that there is no default for boundaries, add more information about bounds in section 1.
- Issue #550: Include a link to CF area-type table and make explicit the need to use standardized area-type strings in Section 7.3.3.
- Issue #542: Clarify and rearrange text of section 4.4 about time coordinate units and calendars; introduce new text and a diagram explaining leap-seconds in existing calendars; define leap_second keyword of units_metadata attribute; define utc and tai calendars; define "datetime" in section 1.3.
- Issue #166: Clarify that time coordinate variables must have **`units`** containing **`since`** and a reference time; distinguish between canonical units of time with and without **`since`**.
- Issue #141: Clarification that text may be stored in variables and attributes as either vlen strings or char arrays, and must be represented in Unicode Normalization Form C and encoded according to UTF-8.
- Issue #367: Remove the AMIP and GRIB columns from the standard name table format defined by Appendix B.
- Issue #403: Metadata to encode quantization properties
- Issue #530: Define "the most rapidly varying dimension", and use this phrase consistently with the clarification "(the last dimension in CDL order)".

- Issue #163: Provide a convention for boundary variables for grids whose cells do not all have the same number of sides.
- Issue #174: A one-dimensional string-valued variable must not have the same name as its dimension, in order to avoid its being mistaken for a coordinate variable.
- Issue #237: Clarify that the character set given in section 2.3 for variable, dimension, attribute and group names is a recommendation, not a requirement.
- Issue #515: Clarify the recommendation to use the convention of 4.3.3 for parametric vertical coordinates, because the previous wording caused confusion.
- Issue #511: Appendix B: New element in XML file header to record the "first published date"
- Issue #509: In exceptional cases allow a standard name to be aliased into two alternatives
- Issue #501: Clarify that data variables and variables containing coordinate data are highly recommended to have `long_name` or `standard_name` attributes, that `cf_role` is used only for discrete sampling geometries and UGRID mesh topologies, and that CF does not prohibit CF attributes from being used in ways that are not defined by CF but that in such cases their meaning is not defined by CF.
- Issue #500: Appendix B: Added a `conventions` string to the standard name xml file format definition

Version 1.11 (05 December 2023)

- Issue #481: Introduce `units_metadata` attribute and clarify some other aspects of `units`
- Issue #458: Clarify the use of compressed dimensions in related variables
- Issue #486: Fix PDF formatting problems and invalid references
- Issue #490: Simple correction to Example 6.1.2
- Issue #457: Creation date of the draft Conventions document
- Issue #445: Updates concerning the Polar Stereographic Grid Mapping
- Issue #468: Update section 2.3 to clarify recommended character set
- Issue #147: Clarify the use of compressed dimensions in related variables
- Issue #483: Add a missing author
- Issue #463: Convert URLs with HTTP protocol to HTTPS if available, fixed a few dead links
- Issue #383: Link to the CF website and deleted the Preface section
- Issue #472: Fix incorrect formating for some <= symbols
- Issue #458: Fix broken link to Unidata documentation.
- Issue #423: Always use "strictly monotonic" when describing coordinate variables
- Issue #420: Add List of Figures
- Issue #210: Correct errors in examples H9-H11
- Issue #374: Clarify rules for packing and unpacking in Section 8.1
- Issue #449: Typo in end-date in Example 7.12

- Issue #266: Updates to some links in the bibliography
- Issue #286: Some labels of examples contain "Example" so that their label in the list of examples contains "Example" (affects four examples); corrected captions of three tables and five examples
- Issue #418: Add missing examples to TOE (table of examples); corrected captions of three tables and three examples
- Issue #367: Delete obsolete references in section 3.3 for mappings of CF standard names to GRIB and PCMDI tables
- Issue #405: Update ch. 4.4 text on reference time vs. UDUNIT
- Issue #406: Remove duplicate section 8.2 in the conformance document
- Issue #391: Correct link to Snyder and typo in the bibliography
- Issue #437: Correct link to NUG in the bibliography
- Issue #428: Recording deployment positions
- Issue #430: Clarify the function of the `cf_role` attribute
- Pull request #408: Deleted a sentence on "rotated Mercator" under `Oblique Mercator` grid mapping in Appendix F
- Issue #265: Clarification of the requirements on bounds variable attributes
- Issue #260: Clarify use of dimensionless units
- Issue #410: Delete "on a spherical Earth" from the definition of the `latitude_longitude` grid mapping in Appendix F
- Issue #153: Reference UGRID conventions in CF

Version 1.10 (31 August 2022)

- Pull request #378: Fixed missing semicolon in example 7.16
- Issue #366: Clarify the intention of standard names
- Issue #352: Correct errors in description of lossy compression by coordinate subsampling
- Issue #345: Reformat the revision history
- Issue #349: Delete unnecessary Conventions attribute in two examples
- Issue #162: Delete incorrect missing_data attributes of time coordinate variables in two examples
- Issue #129: timeSeries featureType with a forecast/reference time dimension?

Version 1.9 (10 September 2021)

- Issue #327: Lossy compression by coordinate subsampling, including new Appendix J ("Coordinate Subsampling Methods")
- Issue #323: Update data model figures for the Domain
- Issue #319: Restrict "gregorian" label to only dates in the Gregorian calendar

- Issue #298: Interpretation of negative years in the units attribute
- Issue #314: Correction to the definition of "ocean sigma over z coordinate" in Appendix D
- Issue #313: Clarification of the handling of leap seconds
- Issue #304: Clarify formula terms definitions
- Issue #301: Introduce the CF domain variable.
- Issue #288: Remove unnecessary line from table in section 9.3.1
- Issue #284: Fix the mention of example 6.1.2 in the example list
- Issue #273: State the principles for design of the CF conventions
- Issue #295: Correction of figures and their description
- Issue #243: Rewording changes relating to the new integer types
- Issue #222: Allow CRS WKT to represent the CRS without requiring reader to compare with grid mapping parameters
- Issue #193: Figures to clarify the order of the vertices of cell bounds
- Issue #271: Extend the CF data model for geometries
- Issue #272: Remove unnecessary netCDF dimensions from some examples
- Issue #258: Clarification of geostationary projection items
- Issue #216: New text describing usage of ancillary variables as status/quality flags
- Issue #159: Incorporate the CF data model into the conventions in new Appendix I
- Issue #253: Update PROJ links in Appendix F
- Pull request #236: Fixed the link in the COARDS reference
- Issue #243: Add new integer types to CF
- Issue #238: Clarifications to ancillary variables text and examples
- Issue #230: Correct inconsistency in units of geostationary projection

Version 1.8 (11 February 2020)

- Issue #223: Axis Order for CRS-WKT grid mappings
- Issue #212: Inconsistent usage of false_easting and false_northing in grid mappings definitions and in examples
- Issue #218: Taxon Names and Identifiers.
- Issue #203: Clarifications to use of groups.
- Issue #213: Missing `s`s in grid mapping description texts.
- Pull request #202: Fix Section 7 examples numbering in the list of examples
- Issue #198: Clarification of use of standard region names in "region" variables.
- Issue #179: Don’t require longitude and Latitude for projected coordinates.
- Issue #139: Added support for variables of type string.

- Issue #186: Minor corrections to Example 5.10, Section 9.5 & Appendix F
- Issue #136: Missing trajectory dimension in H.22
- Issue #144: Add support for using groups.
- Issue #128: Add definition of 'name_strlen' dimension where missing in Appendix H CDL examples.
- Pull request #142: Fix bad reference to an example in section 6.1 "Labels".
- Issue #155, Issue #156: Allow alternate grid mappings for geometry containers. When node_count attribute is missing, require the dimension of the node coordinate variables to be one of the dimensions of the data variable.
- Pull request #146: Typos (plural dimensions) in section H
- Ticket #164: Add bounds attribute to first geometry CDL example.
- Ticket #164: Replace axis with bounds for coordinate variables related to geometry node variables.
- Ticket #164: Add Tim Whiteaker and Dave Blodgett as authors.
- Ticket #164: Remove geometry attribute from lat/lon variables in examples.
- Ticket #164: If coordinates attribute is carried by geometry container, require coordinate variables which correspond to node coordinate variables to have the corresponding axis attribute.
- Ticket #164: Implement suggestions from trac ticket comments.
- Ticket #164: New Geometries section 7.5.

Version 1.7 (7 August 2017)

- Updated use of WKT-CRS syntax.
- Trivial updates to links for COARDS and UDUNITS in the bibliography.
- Updated the links and references to NUG (The NetCDF User Guide), to refer to the current version.
- A few formatting tweaks.
- Ticket #140: Added 3 paragraphs and an example to Chapter 7, Section 7.1.
- Ticket #100: Clarifications to the preamble of sections 4 and 5.
- Ticket #70: Connecting coordinates to Grid Mapping variables: revisions in Section 5.6 and Examples 5.10 and 5.12
- Ticket #104: Clarify the interpretation of scalar coordinate variables, changes in sections 5.7 and 6.1
- Ticket #102: additional cell_methods, changes in Appendix E and section 7.3
- Ticket #80: added attributes to AppF Table F1, changes in section 5.6 and 5.6.1.
- Ticket #86: Allow coordinate variables to be scaled integers, affects two table rows in Appendix A.

- Ticket #138: Clarification of false_easting / false_northing (Table F.1)
- Ticket #76: More than one name in Conventions attribute (section 2.6.1)
- Ticket #109: resolve inconsistency of positive and standard_name attributes (section 4.3)
- Ticket #75: fix documentation and definitions of 3 grid mapping definitions
- Ticket #143: Supplement the definitions of dimensionless vertical coordinates
- Ticket #85: Added sentence to bottom of first para in Section 9.1 "Features and feature types". Added Links column in Section 9.1. Replaced first para in Section 9.6. "Missing Data". Added verbiage to Section 2.5.1, "Missing data...". Added sentence to Appendix A "Description" "missing_value" and "Fill_Value".
- Ticket #145: Add new sentence to bottom of Section 7.2, Add new Section 2.6.3, "External variables". Add "External variable" attribute to Appendix A.
- Ticket #74: Removed "sea_water_speed" from flag values example and added Note at bottom of Example 3.3 in Chapter 3. Also added a sentence to Appendix C Standard Name Modifiers "number of observations" and and a sentence to "status_flag_modifiers"
- Ticket #103: Corrections to Appendices A and H, finish the ticket with remaining changes to Appendix H.
- Ticket #72: Adding the geostationary projection.
- Ticket #92: Add oblique mercator projection
- Ticket #87: Allow comments in coordinate variables
- Ticket #77: Add sinusoidal projection
- Ticket #149: correction of standard name in example 7.3
- Ticket #148: Added maximum_absolute_value, minimum_absolute_value and mean_absolute_value to cell methods in Appendix E
- Ticket #118: Add geoid_name and geopotential_datum_name to the list of Grid Mapping Attributes.
- Ticket #123: revised section 3.3
- Ticket #73: renamed Appendix G to Revision History
- Ticket #31, add new attribute `actual_range`.
- Ticket #141, update affiliation organisations for Jonathan Gregory and Phil Bentley.
- Ticket #103 updated Type and Use values for some attributes in Appendix A, Attributes and added "special purpose" value. In Appendix H, Annotated Examples of Discrete Geometries, updated coordinate values for the variables in some examples to correct omissions.
- Ticket #71, correction of Vertical perspective projection.
- Ticket #67, remove deprecation of "missing_value" from Appendix A, Attributes.
- Ticket #93: Added two new dimensionless coordinates to Appendix D.
- Ticket #69. Added Section 5.6.1, Use of the CRS Well-known Text Format and related changes.
- Ticket #65: add range entry in Appendix E.
- Ticket #64: section 7.3 editorial correction, replace "cell_bounds" with "bounds".

- Ticket #61: two new cell methods in Appendix E.

Version 1.6 (5 December 2011)

- Ticket #37: Added Chapter 9, Discrete Sampling Geometries, and a related Appendix H, and revised several other chapters.
- In Appendix H (Annotated Examples of Discrete Geometries), updated standard names "station_description" and "station_wmo_id" to "platform_name" and "platform_id".

Version 1.5 (25 October 2010)

- Ticket #47: error in example 7.4
- Ticket #51: syntax consistency for dimensionless vertical coordinate definitions
- Ticket #56: typo in CF conventions doc
- Ticket #57: fix for broken URLs in CF Conventions document
- Ticket #58: remove deprecation of "missing_value" attribute
- Ticket #49: clarification of flag_meanings attribute
- Ticket #33: cell_methods for statistical indices
- Ticket #45: Fixed defect of outdated Conventions attribute.
- Ticket #44: Fixed defect by clarifying that coordinates indicate gridpoint location in Chapter 4, Coordinate Types.
- Fixed defect in Mercator section of Appendix F, Grid Mappings by updating to version 12 of Grid Map Names.
- Ticket #34: Added grid mappings Lambert Cylindrical Equal Area, Mercator, and Orthographic to Appendix F, Grid Mappings.

Version 1.4 (27 February 2009)

- Ticket #17: Changes related to removing ambiguity in Section 7.3, "Cell Methods".
- Ticket #36: Fixed defect related to subsection headings in Appendix D, Parametric Vertical Coordinates.
- Ticket #35: Fixed defect in wording of Chapter 5, Coordinate Systems and Domain.
- Ticket #32: Fixed defect in Chapter 5, Coordinate Systems and Domain.
- Ticket #30: Fixed defect in Example 4.3, “Atmosphere sigma coordinate”.

Version 1.3 (4 May 2008)

- Ticket #26: Section 3.5, "Flags", Appendix A, Attributes, Appendix C, Standard Name Modifiers : Enhanced the Flags definition to support bit field notation using a `flag_masks` attribute.

Version 1.2 (4 May 2008)

- Ticket #25: Table 3.1, "Supported Units" : Corrected Prefix for Factor "1e-2" from "deci" to "centi".
- Ticket #18: Section 5.6, "Horizontal Coordinate Reference Systems, Grid Mappings, and Projections", Appendix F, Grid Mappings : Additions and revisions to CF grid mapping attributes to support the specification of coordinate reference system properties

Version 1.1 (17 January 2008)

- 17 January 2008: Chapter 4, Coordinate Types, Chapter 5, Coordinate Systems and Domain: Made changes regarding use of the axis attribute to identify horizontal coordinate variables.
- 17 January 2008: Changed text to refer to rules of CF governance, and provisional status.
- 21 March 2006: Added the section called "Atmosphere natural log pressure coordinate".
- 21 March 2006: Added the section called "Azimuthal equidistant".
- 25 November 2005: the section called "Atmosphere hybrid height coordinate" : Fixed definition of atmosphere hybrid height coordinate.
- 22 October 2004: Added "Lambert conformal projection".
- 20 September 2004: Section 7.3, "Cell Methods" : Changed several incorrect occurrences of the cell method `"standard deviation"` to `"standard_deviation"`.
- 1 July 2004: "Multiple forecasts from a single analysis" : Added `positive` attribute to the scalar coordinate p500 to make it unambiguous that the pressure is a vertical coordinate value.
- 1 July 2004: Section 5.7, "Scalar Coordinate Variables" : Added note that use of scalar coordinate variables inhibits interoperability with COARDS conforming applications.
- 14 June 2004: the section called "Polar Stereographic" : Added `latitude_of_projection_origin` map parameter.
- 14 June 2004: Added the section called “Lambert azimuthal equal area”.

Version 1.0 (28 October 2003)

Initial release.

Bibliography

References

- [CFDM] A data model of the Climate and Forecast metadata conventions (CF-1.6) with a software implementation (cf-python v2.1). Hassell, D., Gregory, J., Blower, J., Lawrence, B. N., and Taylor, K. E.: *Geosci. Model Dev.*, 10, 4619-4646, 2017.
- [CF-WKT] Mapping from CF Grid Mapping Attributes to CRS WKT Elements.
- [COARDS] Conventions for the standardization of NetCDF Files. Sponsored by the "Cooperative Ocean/Atmosphere Research Data Service," a NOAA/university cooperative for the sharing and distribution of global atmospheric and oceanographic research data sets. May 1995.
- [DCG19] Evaluation of lossless and lossy algorithms for the compression of scientific datasets in netCDF-4 or HDF5 files. Delaunay, X., A. Courtois, and F. Gouillon: *Geosci. Model Dev.*, 12, 4099-4113, 2019.
- [FGDC] Content Standard for Digital Geospatial Metadata. Federal Geographic Data Committee, FGDC-STD-001-1998.
- [GHB05] HEALPix: A Framework for High-Resolution Discretization and Fast Analysis of Data Distributed on the Sphere. K. Gorski, Eric Hivon, A. Banday, B. Wandelt, M. Bartelmann, et al.: The Astrophysical Journal, 2005, 622 (2), pp.759-771.
- [IEEE_754] IEEE Standard for Floating-Point Arithmetic, in *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, 22 July 2019.
- [Kou21] A note on precision-preserving compression of scientific data. Kouznetsov, R.: *Geosci. Model Dev.*, 14, 377-389, 2021.
- [KRD21] Compressing atmospheric data into its real information content. Klöwer, M., Razinger, M., Dominguez, J. J., Düben, P., and Palmer, T. N.: *Nat. Comput. Sci.*, 1, 713-724, 2021.
- [K24] Space-Filling Curves. Kanungo, S.: *arXiv.org*, 2024
- [NetCDF] NetCDF Software Package. UNIDATA Program Center of the University Corporation for Atmospheric Research.
- [NUG] The NetCDF User's Guide.
- [OGC_WKT-CRS] OGC Well-known text representation of coordinate reference systems. OGC document 12-063. 1st May 2015.
- [OGP-EPSG_GN7_2] OGP Surveying and Positioning Guidance Note 7, part 2: Coordinate Conversions and Transformations including Formulas.
- [RH15] Efficient data structures for masks on 2D grids. M Reinecke and E Hivon. 2015. *Astronomy & Astrophysics*. 580, A132
- [SCH02] A new terrain-following vertical coordinate formulation for atmospheric prediction models. C Schaer, D Leuenberger, and O Fuhrer. 2002. *Monthly Weather Review*. 130. 2459-2480.
- [Snyder] Map Projections: A Working Manual. USGS Professional Paper 1395.
- [UDUNITS] UDUNITS Software Package. UNIDATA Program Center of the University Corporation for Atmospheric Research.

- [UGRID] UGRID Conventions for storing unstructured (or flexible mesh) data in netCDF files
- [URI] RFC 3986. Uniform Resource Identifier (URI): Generic Syntax. T. Berners-Lee, R. Fielding, L. Masinter. January 2005.
- [W3C] World Wide Web Consortium (W3C).
- [XML] Extensible Markup Language (XML) 1.0. T. Bray, J. Paoli, and C.M. Sperberg-McQueen. 10 February 1998.
- [Zen16] Bit Grooming: Statistically accurate precision-preserving quantization with compression, evaluated in the netCDF Operators (NCO, v4.4.8+). Zender, C. S.: *Geosci. Model Dev.*, 9, 3199-3211, 2016.