Point Observation Data

Draft 2
09/11/07

This is a convention for writing collections of point observations to a netCDF file. This builds on section 5 of the CF-1.0 document, replacing section 5.4 and 5.5 with a more general convention.

A ***point observation*** is a data measurement at a specific time and location. Each kind of measured data is placed in a *data variable*. The time and location values are placed into *coordinate variables* and *auxiliary coordinate variables.*

The starting idea, as described in section 5, is to use the ***coordinates*** attribute to associate auxiliary coordinate variables with the data variables. For example, consider an unconnected collection of points where ozone has been sampled:

```
dimensions:
  sample = 1000 ;

variables:
  float O3(sample) ;
    O3:long_name = "ozone concentration";
    O3:units = "1e-9" ;
    O3:coordinates = "lon lat z time" ;

  double time(sample) ;
    time:long_name = "time" ;
    time:units = "days since 1970-01-01 00:00:00" ;

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

  float lat(sample) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;

  float z(sample) ;
    z:long_name = "height above mean sea level" ;
    z:units = "km" ;
    z:positive = "up" ;
```

In this example, there are 1000 points in the collection, and we have chosen to name the dimension ***sample*** to clarify the distinction between collection dimensions and coordinates. The coordinates of the ith sample are `time(i), lon(i), lat(i)` and `z(i).`

When the data is time ordered, its natural to use time as the sample dimension:

```
dimensions:
  time = 1000 ;

variables:
  float O3(time) ;
    O3:long_name = "ozone concentration";
    O3:units = "1e-9" ;
    O3:coordinates = "lon lat z time" ;

  double time(time) ;
    time:long_name = "time" ;
    time:units = "days since 1970-01-01 00:00:00" ;

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

  float lat(time) ;
```

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

  float z(time) ;
    z:long_name = "height above mean sea level" ;
    z:units = "km" ;
    z:positive = "up" ;
```

Because *time* is now a coordinate variable, its values should be strictly monotonic (i.e. the data is sorted by time). Formally, you no longer need to include *time* in the *coordinates* attribute, since it is known to be a coordinate. However, a suggested idiom is to list all coordinates in the *coordinates* attribute, for clarity.

Data variables may have other dimensions. The following has a 3D wind vector and a character array:

```
dimensions:
  sample = 1000;
  wind_vector = 3;
  inst_name_strlen = 23;

variables:

  float wind(sample, wind_vector);
    wind:long_name = "3D wind";
    wind:units = "m/s";
    wind:coordinates = "lon lat z time";

  char inst_name(sample, inst_name_strlen);
    inst_name:long_name = "instrument name";
    inst_name:coordinates = "lon lat z time" ;
```

We define ***profile observation data*** as point data that has a vertical dimension in the data, with a constant lat/lon (or x/y) location, for example:

```
dimensions:
  sample = 1000 ;

variables:
  float O3(sample, z) ;
    O3:long_name = "ozone concentration";
    O3:units = "1e-9" ;
    O3:coordinates = "lon lat z time" ;

  double time(sample) ;
    time:long_name = "time" ;
    time:units = "days since 1970-01-01 00:00:00" ;

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

  float lat(sample) ;
    lat:long_name = "latitude" ;
    lat:units = "degrees_north" ;

  float z(sample, z) ;
    z:long_name = "height above mean sea level" ;
    z:units = "km" ;
    z:positive = "up" ;
```

In the above example each sample has the same number of z coordinates, but (possibly) different z coordinate values, creating the 2D *z* coordinate. For the case where all samples have exactly the same z coordinate values, it is more efficient, and better to use:

```
  float z(z) ;
    z:long_name = "height above mean sea level" ;
    z:units = "km" ;
    z:positive = "up" ;
```

There is an important restriction on how an auxiliary coordinate connects to the data variable: ***the dimensions of the auxiliary coordinate must be a subset of the dimensions of any data variable that uses it***. So z(sample, z) and z(z) are ok as an auxiliary coordinate for O3(sample, z), but neither could be an auxiliary coordinate for, say, O3(time).

Time series of station data

Suppose that point data is taken at a set of *named locations* called ***stations***. The set of observations at a particular station, if ordered by time, becomes a time series, and the file is a ***collection of time series of station data***. In this case one could use:

```
dimensions:
  station = 10 ;  // measurement locations
  pressure = 11 ; // pressure levels
  time = UNLIMITED ;

variables:
  float humidity(time, pressure, station) ;
    humidity:long_name = "specific humidity" ;
    humidity:units = "" ;
    humidity:coordinates = "lat lon pressure time" ;

  double time(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:long_name = "pressure" ;
    pressure:units = "hPa" ;
```

There are two problems with this scheme. The first is that each station has the same number of samples (times) allocated to it. This is called a ***rectangular array***. When stations have different numbers of samples, one is forced to allocate the maximum sample size, and use missing data values. In this example, the amount of wasted data is exacerbated by having a vertical (pressure) dimension in the data. Further, if the pressure coordinate variable can vary, one must use:

```
  float pressure(time, pressure, station) ;
    pressure:long_name = "pressure" ;
    pressure:units = "hPa" ;
```

The second problem in this example is that the coordinate values for time are required to be the same for each set of measurements at each station. This can be fixed, however, by using

```
  double time(station, time) ;
    time:long_name = "time of measurement" ;
    time:units = "days since 1970-01-01 00:00:00" ;
```

As we try to represent more complicated arrangements of point observations, this issue of rectangular arrays often appears.

A different way to handle variable number of samples at each station is to remove the station dimension from the data variables, and keep track of the station index for each observation in a separate variable:

```
dimensions:
  station = 10 ;  // measurement locations
  pressure = 11 ; // pressure levels
  profile = UNLIMITED ;

variables:
  float humidity(profile, pressure) ;
    humidity:long_name = "specific humidity" ;
    humidity:coordinates = "lat lon pressure time" ;

  int station_index(profile) ;
    station_index:long_name = "index into station dimension";

  double time(profile) ;
    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" ;
```

If the pressure coordinate is constant, then

```
  float pressure(pressure) ;
    pressure:long_name = "pressure" ;
    pressure:units = "hPa" ;
```

If the pressure coordinate can vary for each profile:

```
  float pressure(profile, pressure) ;
    pressure:long_name = "pressure" ;
    pressure:units = "hPa" ;
```

If its fixed for each station, you'd like to use:

```
  float pressure(station, pressure) ;
    pressure:long_name = "pressure" ;
    pressure:units = "hPa" ;
```

The *station_index* variable associates the ith profile with the station at index *station_index(i)*. But *lat* and *lon* can no longer be considered auxiliary coordinate variables, since they use a dimension that is not present in the data variable. Instead, there is an extra level of indirection represented by the *station_index* variable. So we are really generalizing past previous notions of coordinate variables and auxiliary coordinate variables.

Instead of making coordinate variables more complicated, we are going to generalize the underlying data model, using concepts from relational databases. In addition to the fundamental data type of *multidimensional array*, we add the data type ***table,*** where ***a table is a collection of variables with the same outer dimension***. We then define an ***index join*** as ***connecting two tables using a variable in one table that holds dimension indices into the second table***. Dimension indices are zero based.

Returning to our time series of station data example, we can create a new notation using tables. All variables with the same outer dimension, such as:

```
  float humidity(profile, pressure) ;
    humidity:long_name = "specific humidity" ;
  float temperature(profile, pressure) ;
    temperature:long_name = "air temperature" ;
  float pressure(profile, pressure) ;
    pressure:long_name = "pressure" ;

  int station_index(profile) ;
  double time(profile) ;
```

are rewritten as:

```
  table {
    float humidity(pressure) ;
      humidity:long_name = "specific humidity" ;
    float temperature(pressure) ;
      temperature:long_name = "air temperature" ;
    float pressure(pressure);
      pressure:long_name = "pressure" ;

    int station_index;
    double time;

  } profile (profile);
```

So a “table variable” is created that uses the *profile* (outer) dimension. All the variables that have that outer dimension become part of the table. Similarly for the station table (for clarity, we stop showing the attributes):

```
  table {
    float humidity(pressure) ;
    float temperature(pressure) ;
    float pressure(pressure);
    int station_index;
    double time;
  } profile(profile);

  table {
    float lon;
    float lat;
   } station (station);
```

To specify the index join, if we wanted to write pseudo-SQL, we could say

```
  JOIN profile TO station WITH profile.station_index
```

where *profile* and *station* specify tables with the corresponding dimension, and *station_index* is a variable in the profile table whose values are indices in the station table. In other words:

```
  JOIN <child dimension> TO <parent dimension> WITH <child.variable>
```

Of course, none of this is in the netCDF file, it’s just a short hand notation.

Another compact and useful notation is to consider that the tables are nested, and to ignore the mechanism by which the nesting occurs:

```
  table {
    float lon;
    float lat;

    table {
      double time;

      float humidity(pressure) ;
      float temperature(pressure) ;
      float pressure(pressure);
    } profile (*);
```

```
   } station (station);
```

Here the (*) denotes a variable length dimension. All of the profiles inside of a station table are for that station. Note that because we are using a fixed pressure dimension, all profiles have a fixed number of pressure levels. The values of those pressure levels can vary from profile to profile. If the pressure levels were fixed at each station, you would have:

```
  table {
    float lon;
    float lat;
    float pressure(pressure);

    table {
      double time;

      float humidity(pressure) ;
      float temperature(pressure) ;
    } profile (*);

   } station (station);
```

If the pressure levels were fixed for all profiles:

```
  float pressure(pressure);
  table {
    float lon;
    float lat;

    table {
      double time;

      float humidity(pressure) ;
      float temperature(pressure) ;
    } profile (*);

   } station (station);
```

If the number of pressure levels could vary from profile to profile, we are back in the situation of having to set a maximum, then using missing values. Applying the same principles as before we can create another table, for example (using nested table notation):

```
  table {
    float lon;
    float lat;

    table {
      double time;

      table {
        float humidity;
        float temperature;
        float pressure
      obs(*);

    } profile (*);

   } station (station);
```

OR using table notation:

```
  table {
    float humidity ;
    float temperature;
    float pressure;
    int profile_index;
  } obs (obs);

  table {
    int station_index;
    double time;
  } profile (profile);

  table {
    float lon;
    float lat;
  } station (station);
```

OR using CDL:

```
  float humidity(obs);
  float temperature(obs);
  float pressure(obs);
  int profile_index(obs);

  double time(profile);
  double station_index(profile);

  double lat(station);
  double lon(station);
```

As you can see, there's a mechanical conversion between these 3 notations (CDL, tables, nested tables).

Using the Unlimited Dimension

The use of the unlimited dimension in the netcdf-3 file format warrants attention because it can have a strong effect on performance. Consider the following example:

```
dimensions:
  station = 4021 ;  // measurement locations
  pressure = 30 ; // pressure levels
  time = UNLIMITED ; // currently 117987

variables:

  float humidity(time, pressure) ;
  float temperature(time, pressure) ;
  float pressure(time, pressure) ;
  int time(time) ;
  int station_index(time) ;

  char name(station, name_strlen);
  char desc(station, desc_strlen);
  double lat(station);
  double lon(station);
  double alt(station);
```

All of the variables using the time dimension are called *record variables* because they use the unlimited (record) dimension.

The layout of the netCDF-3 file format is simple: first the header is written, then the non-record variables are each written, then the record variables are written. Non-record variables are written in the order they are defined. The entire space must be allocated for them at define time, which is why their dimension sizes cant change. Record variables are written one record at a time, where record 0 has all the record variable values for index=0, then record 1 with all the record variable values for index=1, etc. The unlimited dimension can thus grow by appending to the file.

Since the file layout is quite different depending whether the unlimited dimension is used, the performance of reading the data can be quite different. In a worse case scenario, for large files, you might see a factor of 100 performance difference, depending on your read access pattern (the actual times are highly dependent on the caching strategy of the underlying file system). So it is sometimes necessary to understand what the common read pattern is and to optimize the file layout for it.

Using the record dimension is often very useful when writing data that arrives sequentially, since the new data can simply be appended to the file, and you don't need to know ahead of time how many records there will be.

The decision to use the record dimension or not must not effect the data type or the semantics of the data – only access efficiency.

Creating Fast Access to Children

Given a row in a child table, one finds the parent using the *parent index* variable. However, one must read the entire parent index variable to find all of the child rows for a given parent row. For efficiency, one can optionally add a way to quickly find all of the child rows for a given parent row, using a *linked list* or a *contiguous list.*

A ***contiguous list*** places all children in contiguous rows, and then adds *firstChild* and *numChildren* variables in the parent table which hold dimension indices into the child table. For the ith parent row, all its children are found at the indices between *firstChild(i)* and *firstChild(i) + numChildren(i).* This method is recommended as the most efficient way to read all the child rows for a parent, since they are stored contiguously.

A ***forward linked list*** adds a *firstChild* in the parent table and *nextChild* variable in the child table, which hold dimension indices into the child table. One reads the *firstChild* row and follows the links in *nextChild* until the dimension index is less than 0, indicating the end of the linked list. This method is recommended when writing data for multiple parents at once, when the total number of children is unknown, so a contiguous list is not possible.

A ***backwards linked list*** adds a *lastChild* in the parent table and *prevChild* variable in the child table, again which hold dimension indices into the child table. One reads the *lastChild* row and follows the links in *prevChild* until the dimension index is less than 0, indicating the end of the linked list. This method is recommended for real-time data arriving serially and unpredictably, since one only has track the last child for each parent in memory and append the new record, then update the *lastChild* array when the data has all been received. With a forward linked list, one must also rewrite the previous record.

Remember that dimension indices are 0 based.

Specifying the type of data

The table data type and technique of connecting tables through dimension index variables is quite general and should be useful for many kinds of data in any domain of science.

Experience has shown that it's important for visualization and analysis tools and for human understanding to classify data into broad categories based on the topology of the collection. We call these ***data types***. We haven't found a systematic or rigorous classification scheme; rather these reflect our experience with observational datasets in the earth sciences, strongly influenced by the type of measuring instruments used.

While one could imagine everything as merely a collection of points, it is usually necessary to take advantage of whatever structure is found in the data. The structure of the data and coordinate systems ideally reflects the *connectedness* (a.k.a. *topology*) of the measurements. This connectedness is not always able to be ascertained by inspecting the structure of the coordinate systems. For example, trajectories and point data have the same structure.

The set of data types we propose to standardize in the convention are:

- **Collection of point data** (unconnected x,y,z,t) Examples: earthquake data.
- **Collection of trajectories** (connected x,y,z,t, ordered t) Examples: aircraft data, drifting buoy.
- **Collection of profiler data** (unconnected x,y,t, connected z) Examples: satellite profiles.
- **Station collection of point** (unconnected x,y,z, connected t) Examples: metars.
- **Station collection of profilers** (unconnected x,y; connected z, connected t) Examples: profilers.

These mostly fit the form (Collection | Station Collection) of (Point | Profile | Trajectory). Others that might be needed:

- **Trajectories of sounding** (connected x,y,z,t, ordered z, ordered t) Examples: ship soundings.

CDL Examples

Collection of point data

```
variables;
  float lon(obs);
  float lat(obs);
  float z(obs);
  double time(obs);

  float humidity(obs);
  float temperature(obs);
  float pressure(obs);
    pressure:coordinates = “lon lat z time”;

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Collection of point data”;
  :CF_datatype = “point”;
  :CF_table = “obs”;
```

Collection of profiler data (rectangular)

```
variables;
  float lon(obs);
  float lat(obs);
  float z(obs, z); // or z(z)
  double time(obs);

  float humidity(obs, z);
  float temperature(obs, z);
  float pressure(obs, z);
    pressure:coordinates = “lon lat z time”;

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Collection of profiler data”;
  :CF_datatype = “profiler”;
  :CF_table = “obs”;
```

Collection of Trajectories

```
variables;
  float lon(obs);
  float lat(obs);
  float z(obs);
  double time(obs);
  float humidity(obs);
  float temperature(obs);
  float pressure(obs);
    pressure:coordinates = “lon lat z time”;

  int trajectory_id(obs); // unneeded if only one trajectory LOOK

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Collection of trajectory data”;
  :CF_datatype = “trajectory”;
  :CF_table = “obs”;
```

Collection of Trajectories of Sounding (rectangular)

```
variables;
  float lon(sounding);
  float lat(sounding);
  double time(sounding);
  float z(sounding, z); // or z(z)

  float humidity(sounding, z);
  float temperature(sounding, z);
  float pressure(sounding, z);
    pressure:coordinates = “lon lat z time”;

  int trajectory_index(sounding); // unneeded if only one trajectory

  char ship_name( trajectory, ship_name_strlen) ;
  char instrument( trajectory, instrument_strlen) ;


attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Collection of trajectory of sounding data”;
  :CF_datatype = “trajectory of sounding”;
  :CF_table = “JOIN sounding TO trajectory WITH trajectory_index”;
```

Collection of Trajectories of Soundings (variable z)

```
Variables:
  float salinity(obs) ;
  float temperature(obs) ;
  float pressure(obs) ;
  double time(obs) ;
  int sounding_index(obs) ;

  float lat(sounding) ;
  float lon(sounding) ;
  int trajectory_index(sounding) ;

  char ship_name( trajectory, ship_name_strlen) ;
  char instrument( trajectory, instrument_strlen) ;

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Collection of trajectory of sounding data”;
  :CF_datatype = “trajectory of sounding”;
  :CF_table = “JOIN sounding TO trajectory WITH trajectory_index AND JOIN obs TO sounding
WITH sounding_index”;
```

Station Collection of Point

```
  float humidity(obs);
  float temperature(obs);
  float pressure(obs);
  double time(obs);
  double station_index(obs);

  double lat(station);
  double lon(station);

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Station Collection of point”;
  :CF_datatype = “Station”;
  :CF_table = “JOIN obs TO station WITH station_index”;
```

Station Collection of Profilers (fixed length)

```
  float humidity(profile, z);
  float temperature(profile, z);
  float pressure(profile, z);

  double time(profile);
  double station_index(profile);

  double lat(station);
  double lon(station);

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Station Profilers”;
  :CF_datatype = “Station Collection of Profiler”;
  :CF_table = “JOIN profile TO station WITH station_index”;
```

Station Collection of Profilers (variable length)

```
  float humidity(obs);
  float temperature(obs);
  float pressure(obs);
  int profile_index(obs);

  double time(profile);
  double station_index(profile);

  double lat(station);
  double lon(station);

attributes:
  :Conventions = “CF-1.1”;
  :CF_datatype = “Station Profilers”;
  :CF_datatype = “Station Collection of Profiler”;
  :CF_table = “JOIN profile TO station WITH station_index AND JOIN obs TO profile WITH
profile_index”;
```

Still To Do:

- Decide on the mechanism by which the join is specified. Do we really want “pseudo-SQL” ?
- Specify the datatypes globally or ??
- What do you put the :coordinate attribute on? All data variables would follow existing CF. Then you have a redundant system somewhat.
- Sorting: when can you count on it being sorted? Eg time series in station data. Required or optional?