General-Purpose Data Containers for Science and Engineering*

OECD/NEA/WPEC Subgroup 38

May 4, 2015

*This work was performed under the auspices

Contents

1 Introduction

In 2012 an international committee [1], herein called SG38, was formed to develop a modern structure to replace the ENDF-6 [2] format for storing evaluated nuclear reaction data on a computer system. This committee divided the project into seven tasks. One of these tasks, the design of **General-Purpose Data Containers** (GPDCs), is described in this article.

What type of data does SG38 need to store and why is the task called **General-Purpose Data Containers**? The most common types of data in an evaluated nuclear reaction database are representations of physical functions in tabulated[1] forms. For example, a cross section $\sigma(E)$ as a function of energy E that is given at a finite set of energies with an interpolation rule to obtain the value between two consecutive points. An example of a tabulated cross section is shown in Table 1. Another example of a tabulated physical function is an average multiplicity $m(E)$ given at a finite set of energies. Both $\sigma(E)$ and $m(E)$ can be stored in the same container type as they represent a 1-dimensional function of the generic form $f(x)$ as tabulated data. There is also a need to store 1-dimensional functions using truncated Legendre or polynomial (or others) expansions. In addition, 2- and 3-dimensional (i.e., $f(x, y)$ and $f(x, y, z)$) tabulated functions are needed as well as containers for matrices. The phrase *General-Purpose* implies that the containers are to be designed to store generic forms of tabulated data (e.g., $f(x)$) rather than one for each physical function (e.g., $\sigma(E)$, $m(E)$). Also, where possible, it would be beneficial to design containers that can store data forms not currently used in evaluated nuclear database or at least be easily extended.

Table 1: A tabulated cross section. That is, a cross section listed at a finite number of points with the cross section between the points obtained via interpolation. The interpolation rule (e.g., 'lin,lin', 'log,lin') must also be defined.

energy (MeV)	cross section (b)
15.5	0.0
15.75	0.25
16.0	0.65
16.5	0.85
20.0	0.95

In addition to containers for storing physical functions as tabulated data, other types of containers are needed. There exists a desire within SG38 to support the storage of documentation at various levels within an evaluated file. Containers for storing non-functional data (e.g., a list of numbers) as well as units and labels for axes are also needed. Herein, containers for storing physical functions are called functional containers.

One of the goals for the general-purpose data containers task is to design containers that will be useful to other scientific and engineering applications. To meet this goal, task members should think outside of the immediate needs of evaluated nuclear data to ensure that the containers are general-purpose rather than simply repackaged versions of existing containers. While the examples in this article may be specific to nuclear reaction data, it is hoped that the end product will be useful for other applications. To this end, some specifications will be left to the end user.

[1] Tabulated data are also called pointwise or piecewise data.

A second goal is to design the structure of the containers and not the representation of the data in a file. In this way, the containers can be implemented in various meta-languages (see Section 3.1).

Another goal for SG38 is to design a structure that is shareable between different nuclear reaction data groups. This led the SG38 committee to choose XML [11] as the primary meta-language for expressing the structure in a computer file. Throughout this article, many examples will be given in XML. However, the structure of the containers defined in this article can be expressed in other meta-languages (e.g., JSON [10], HDF5 [12]) as illustrated in Section A.

In the talks given by David Brown and Morgan White at the Dec. 2013 SG38 meeting in Japan, it was noted that data types (e.g., types of integers and floats) need to be specified for both the general-purpose data containers and the nuclear reaction structure. Since the nuclear reaction structure will inherit from the general-purpose data containers, it would be beneficial if one set of data types can be specified that will work for both. However, if that is not possible, it would still be beneficial for the general-purpose data container types to be a super set of the nuclear reaction structure types.

This article will first give some definitions, then needs for the nuclear data community, the requirements for the general-purpose data containers. This is followed by definitions of various character sets, of basic data types, and finally of general-purpose data containers are specified.

2 Version

This document describes version 1.0 of the General-Purpose Data Containers (GPDC 1.0).

3 Notation and definitions

This section defines some common notation and definitions used in this article.

3.1 Meta-language

In this article, the word **meta-language** means any hierarchical storage system useful for storing the general-purpose data containers. Examples include XML [11], HDF5 [12], Python [15], C [13] and file system hierarchy.

3.2 Data versus meta-data

This article distinguishes between data and meta-data. Data are the values that represent a function, list or table. For example, a pointwise representation of $f(x) = 1/x$ for the domain $1 \leq x \leq 10$ can be represented by the two points (1, 1) and (10, 0.1) or the data "1 1 10 0.1". Meta-data are qualifiers that affect how the data are to be interpreted. Examples of meta-data include interpolation rules, labels and units. In the example above, the data "1 1 10 0.1" exactly represent the function for the domain $1 \leq x \leq 10$ when log-log interpolation is used. One of the requirements for the data containers is that they allow for storage of meta-data and define how meta-data are to be stored.

3.3 Functional expressions in this article

The dimension of a function is defined as the number of independent variable it has. For example, $f(x)$ and $f(x, y, z)$ are 1- and 3-dimensional functions.

In other articles, 1-dimensional functions are often expressed (i.e., written) as $f(x)$ or $y(x)$ while 2-dimensional and 3-dimensional functions may be expressed as $f(x,y)$ and $f(x,y,z)$. In this article a more general expression for an n-dimensional function is used. The 1-dimensional function is expressed as $x_0(x_1)$, while the 2-dimensional and 3-dimensional functions are expressed as $x_0(x_2,x_1)$ and $x_0(x_3,x_2,x_1)$ respectively. An n-dimensional function is expressed as[2] $x_0(x_n,...,x_3,x_2,x_1)$. This representation is used to preserve the meaning of the variables when nesting (n-1)-dimensional functions within an n-dimensional function. For example, consider the angular function $P(E,\mu)$ where the angle θ is given via $\mu=\cos(\theta)$ and E is energy. In tabulated form this function may be given as a list of $P(\mu)$ functions at various energies. A generic expression for the $P(E,\mu)$ function might be $f(x,y)$ where x represents the E variable and and y represents the μ variable. While the generic expression for the $P(\mu)$ function might be $f(x)$ where x now represents the μ variable. That is, the mapping of $P(E,\mu)$ to $f(x,y)$ and $P(\mu)$ to $f(x)$ causes μ to be represent by y in $f(x,y)$ and by x in $f(x)$. Using the generic expressions $x_0(x_2,x_1)$ and $x_0(x_1)$ for $P(E,\mu)$ and $P(\mu)$ respectively, the meaning of the x_1 variable (i.e., $x_1=\mu$) is consistent.

For a 2-dimensional function $x_0(x_2,x_1)$ in which the integral over the x_1 is normalized to 1, this is expressed (i.e., written) as $x_0(x_1|x_2)$. In general, for an n-dimensional function in which the integral over the m fastest independent variables is 1, this is expressed as $x_0(x_m,...,x_1|x_n,...,x_{m+1})$.

When an n-dimensional function is expressed as collated tabulated data (see Sections 4.1) the independent variable x_1 is said to vary the fastest, the independent variable x_2 varies the second fastest and so on to the independent variable x_n which varies the slowest. This verbiage expresses the way data are stored in collated tabulated containers which is similar to the way data are stored in C or Python arrays. For example, for the C array *ABC* declared as *int ABC[3][4][6]* and indexed as *ABC[i3][i2][i1]*, the index *i1* varies the fastest in memory while *i3* varies the slowest.

To understand why the independent variable x_1 is said to vary the fastest (and x_2 the second faster), consider the storage of the tabulated data for the angular distribution function $P(\mu|E)$ as a function of angle θ via $\mu=\cos(\theta)$ and energy E which is a 2-dimensional function. For evaluated nuclear data, these data are often stored as functions of $P(\mu)$ at various E's. Table 2 presents a simple tabulated $P(\mu|E)$ with $P(\mu)$ given at three energies (0, 0.5 and 20.0 eV). Table 3 shows how these data can be stored[3] in a file. As seen in Table 3, the independent variable μ varies faster than the independent variable E.

Table 2: A tabulated $P(\mu|E)$ given as $P(\mu)$ at three energies. $P(\mu)$ between two energies is obtained by interpolation.

$E=0.0$ eV		$E=0.5$ eV		$E=20.0$ eV	
μ	$P(\mu)$	μ	$P(\mu)$	μ	$P(\mu)$
-1	0.5	-1	0.4	-1	0.8
1	0.5	1	0.6	-0.5	0.6
				0	0.0
				0.5	0.6
				1	0.8

[2]It is also reasonable to express the general form as $f(x_n,...,x_3,x_2,x_1)$ where x_0 is replaced with f. Some text will express an n-dimensional function as $f(x_1,x_2,x_3,...,x_n)$ which is similar to the expression used here except for the indexing of the independent variables and the naming of the dependent variable (i.e., f versus x_0).

[3]A better storage representation for this type of data will be presented later.

Table 3: A possible file representation of the tabulated $P(\mu|E)$ from Table2.

```
 0.0   -1     0.5
 0.0    1     0.5
 0.5   -1     0.4
 0.5    1     0.6
20.0   -1     0.8
20.0   -0.5   0.6
20.0    0     0.0
20.0    0.5   0.6
20.0    1     0.8
```

3.4 Number range

When listing a range of number the following notation is used:

- For numbers in the range N_1 to N_2 inclusive, the notation [N_1,N_2] is used.
- For numbers in the range N_1 inclusive to N_2 exclusive the notation [N_1,N_2) is used.
- For numbers in the range N_1 exclusive to N_2 inclusive the notation (N_1,N_2] is used.
- For numbers in the range N_1 exclusive to N_2 exclusive the notation (N_1,N_2) is used.

3.5 Floating point numbers and the e-form numbers.

Two string representations of floating point numbers are allowed: fixed point and e-form. A fixed point string represents a floating point value as an optional sign character (i.e. '+' or '-') followed by a sequence of 0 or more Arabic Numerals (see Section 6.2) followed by an optional period character ('.') which can be followed by 0 or more digits with the requirement that at least one Arabic Numeral be present. Examples of valid strings are: '123.34', '+32', '0.0021' and '-.014'. The **e-form** represents a floating point value using the exponential notation. The exponential notation contains a **fixed point** string followed by either the 'e' or 'E' character and a signed integer (e.g., '1.234e-21', '1.234E-21', '4.331E-03'). The FORTRAN [14] **d-form**[4] is not allowed.

4 Review of nuclear data container needs

This section gives an overview of the various data used within the SG38 communities.

4.1 Tabulated or pointwise data

Tabulated or pointwise data are numerical data that represent a function (e.g., $x_0(x_1)$, $x_0(x_2, x_1)$) by listing the function at a finite number of points. The value of the function at an intermediate point is obtained by interpolating.

[4]For the exponential notation, FORTRAN distinguishes between single and double precision value. Single precision strings use the 'e' or 'E' to signify the exponential value while double precision strings use the 'd' or 'D'.

4.1.1 1-dimensional, 1-d or x0(x1) tabulated data

These data are tabulated representations of the single-valued[5] function $x_0(x_1)$. The tabulation is a list of (x_1, x_0) pairs with the pairs sorted by increasing x_1 values. In addition, an interpolation rule is defined to obtain an x_0 value between consecutive x_1 values. In this article, these data are labeled 'XYs'[6]. These data can be represented by two columns where the first column stores the x_1 values and the second column stores the x_0 values.

4.1.2 2-dimensional, 2-d or x0(x2,x1) tabulated data

These data are tabulated representations of the single-valued function $x_0(x_2, x_1)$. These data can be represented by three columns of data, although not always efficiently (see section 4.2), where the first column stores the x_2 values, the second column stores the x_1 values and the third column stores the x_0 values. There are two types of three-column data that represent the function $x_0(x_2, x_1)$: herein, they are called dispersed and collated data.

Dispersed or scatter $x_0(x_2, x_1)$ data: These data typically have only one x_1 (x_2) value for each x_2 (x_1) value. This type of data can be sorted by either the x_1 or x_2 value. An example is shown in Table 4. This type of data is not currently used in nuclear data and are currently not specified in this document. However, a table (Section 21) or array (Section 13) container can be used as they are an efficient way to store this type of data.

Table 4: An example of scattered $x_0(x_2, x_1)$ data. For this type of data, few if any rows (i.e., points) have the same x_1 and/or x_2 values.

x_2	x_1	x_0
1.0	1.0	0.0
1.1	0.2	0.2
1.2	0.6	0.1
2.1	0.3	1.1
2.2	0.5	3.2
2.4	2.1	2.1
2.9	0.6	1.5
3.3	0.4	0.4

Collated $x_0(x_2, x_1)$ data: These data contain a list of x_2 values and each x_2 value contains an XYs dataset (i.e., a $x_0(x_1)$ function). These data are sorted by increasing x_2 values and include interpolation rules for obtaining the XYs dataset at points between consecutive x_2 values. An example of collated data is shown in Table 5 (interpolation rules not shown). In this example there are three distinct x_2 values (1.0, 1.1 and 1.7) and each has an associated XYs dataset. As example, for $x_2 = 1.1$ the associated

[5] An $x_0(x_1)$ and high dimensional functions are allowed to have a discontinuity when the regions container is used (see Section 17).

[6] Since in this document we prefer to us the generic functional expression $x_0(x_1)$ one should label these data as 'X_0X_1s'. We chose 'XYs' as 'X_0X_1s' is awkward and many would express this function type as $y(x)$ with each point listed as (x_i, y_i))

$x_0(x_1)$ function is the points (0.3, 1.1) and (0.5, 3.2) which will have an interpolation rule for obtaining the x_0 value for a point for x_1 between 0.3 and 0.5.

Table 5: An example of collated $x_0(x_2, x_1)$ data. For this type of data, a given x_2 has an associated XYs (i.e., $x_0(x_1)$) tabulated function. No interpolation meta-data is shown in this example.

x_2	x_1	x_0
1.0	0.0	0.0
1.0	0.2	0.2
1.0	0.6	0.1
1.1	0.3	1.1
1.1	0.5	3.2
1.7	0.3	2.1
1.7	0.6	1.5
1.7	1.1	0.4
1.7	1.4	0.4
1.7	2.4	0.4

4.1.3 3-dimensional, 3-d or x0(x3,x2,x1) tabulated data

These data are tabulated representations of the single valued function $x_0(x_3, x_2, x_1)$. These data can be represented by four-columns of data, although not always efficiently (see section 4.2), where the first column stores the x_3 values, the second column stores the x_2 values, the third column stores the x_1 values and the fourth column stores the x_0 values. Like the $x_0(x_2, x_1)$ data, $x_0(x_3, x_2, x_1)$ tabulated data can be either dispersed or collated.

Dispersed or scatter $x_0(x_3, x_2, x_1)$ *data:* These data typically have only one x_2 and x_1 value for each x_3 value and similarly for any permutation of x_2, x_1 and x_3. These data are currently not specified. However, a table (Section 21) or array (Section 13) contain can be used as they are an efficient way to store this type of data.

Collated $x_0(x_3, x_2, x_1)$ *data:* These data contain a list of x_3 values where each x_3 value contains an associated $x_0(x_2, x_1)$ (or multiD_XYs with dimension = 2) dataset. These data are sorted by increasing x_3 values and include interpolation rules for obtaining the $x_0(x_2, x_1)$ dataset at points between consecutive x_3 values. See Table 8 for an example of collated $x_0(x_3, x_2, x_1)$ data.

4.2 Discussion

One goal of the WPEC SG38 project is to design a set of general-purpose data containers. In this discussion, the general-purpose data containers will be divided into 3 categories: generic, functional and array data containers.

As its name implies, the generic data container could store anything that can be stored in a functional or array data container. Functional and array data containers are added in order to reduce redundancy, storage requirements and access time as well as to clearly delineate sections of the data and to make interfaces simpler (i.e., to satisfy requirements in section 5).

The only generic data container considered here is the table container that has a set of columns and rows. A given column and row defines a cell. As this is a generic data container, there is no restriction on the type of data that can be stored in each cell (e.g., a string, int, float, empty value).

The other two data container categories, functional and array, are designed to store tabulated data representing a mathematical function or transform. As the names imply, the functional data containers are designed to hold single-valued tabulated data representing functions of the form $x_0(x_n, ..., x_1)$ while the array data containers are designed to hold array data (e.g., deterministic transport matrices, rotational and Lorentz transformations).

While any tabulated data stored in a functional data container can be stored in an array, in many cases other storage methods are more efficient than arrays (at least for the types of data often encountered in nuclear evaluations). Consider the following 1-, 2- and 3-dimensional tabulated data examples.

1-d tabulated functional data (i.e., XYs or $x_0(x_1)$) of length N can be stored as a array of size N by 2 (see Section 14). For these data, the array and the function data containers are similar (there may be an implementation difference, for example the array container may have a 'shape=(N,2)' attribute while the functional container only needs to define its length since the number of data points per entity, 2, is known implicitly).

As for collated 2-d tabulated functional data (i.e., $x_0(x_2, x_1)$), the most efficient way to store them is as a list of XYs (i.e., $x_0(x_1)$) data, where each XYs has an associated x_2 value. To understand this, consider the way tabulated angular distribution data $P(\mu|E)$ are stored as a function of projectile energy E in the LLNL ENDL format versus the way they are stored in the ENDF format. (Here, $\mu = \cos(\theta)$ where θ is the outgoing particle's angle in the center-of-mass frame). In ENDL, the data are stored as an array of N by 3 where each entity - each set of E, μ and P (or as generic variables x_2, x_1 and x_0) values - are stored as one row of the array. A snippet from an ENDL file showing data for the last two projectile energies, E, is shown in Table 6.

Table 6: An example of ENDL collated $x_0(x_2, x_1)$ data. For this type of data, a given x_2 has an associated XYs (i.e., $x_0(x_1)$) tabulated function. The example is angular distribution data $P(\mu|E)$. No interpolation meta-data is shown in this example.

E	μ	$P(\mu\|E)$
.		
.		
.		
1.5000000E+01	-1.00000E+00	2.18579E-01
1.5000000E+01	-3.50000E-01	1.74859E-01
1.5000000E+01	0.00000E+00	4.37157E-01
1.5000000E+01	5.00000E-01	6.55736E-01
1.5000000E+01	1.00000E+00	1.31149E+00
2.0000000E+01	-1.00000E+00	2.18579E-01
2.0000000E+01	-3.50000E-01	1.74859E-01
2.0000000E+01	0.00000E+00	4.37157E-01
2.0000000E+01	5.00000E-01	6.55736E-01
2.0000000E+01	1.00000E+00	1.31149E+00

The same data stored in the ENDF format using the ENDF TAB2 and TAB1 records would look

somewhat like that shown in Table 7. That is, attached to each E (e.g., x_2) value is its associated $P(\mu)$ (e.g., $x_0(x_1)$) data. In ENDF, each $P(\mu)$ dataset is stored in a TAB1 record. The TAB1 datasets are then organized inside a TAB2 record along with their associated E values.

Table 7: A schematic example of the way collated $P(\mu|E)$ (i.e., $x_0(x_2, x_1)$) tabulated data are stored in the ENDF format. For this type of data, a given E (i.e., x_2) has an associated $P(\mu)$ (i.e., $x_0(x_1)$) tabulated function. No interpolation meta-data is shown in this example.

```
.
.
.
1.5000000E+01
   5
      -1.00000E+00  2.18579E-01
      -3.50000E-01  1.74859E-01
       0.00000E+00  4.37157E-01
       5.00000E-01  6.55736E-01
       1.00000E+00  1.31149E+00

2.0000000E+01
   5
      -1.00000E+00  2.18579E-01
      -3.50000E-01  1.74859E-01
       0.00000E+00  4.37157E-01
       5.00000E-01  6.55736E-01
       1.00000E+00  1.31149E+00
```


As can be seen from the examples above, the ENDF format does a better job at satisfying requirements in section 5 since

- Requirement 2: The ENDF format re-uses the TAB1 record (the same container used to store $x_0(x_1)$ data) to store the $P(\mu)$ data for each E value.
- Requirement 3 and 4: The ENDL format stores each E value multiple times, which is redundant and requires more memory and parsing time.
- Requirement 5: the ENDF format does a better job of clearly delineating each E value and its associated $P(\mu)$ dataset. In the ENDL format, the code reading the data has to determine where one E dataset starts and ends by checking the E value for each line.

As for collated 3-d tabulated data (i.e., $x_0(x_3, x_2, x_1)$), the most efficient way to store them is as a list of $x_0(x_2, x_1)$ data, where each $x_0(x_2, x_1)$ has an associated x_3 value[7]. To understand this, consider the way outgoing energy distribution versus angle and energy tabulated data, $P(E'|E, \mu)$, are stored in the LLNL ENDL format versus the way they are stored in the ENDF format. A specific example from ENDL2011.0 of $P(E'|E, \mu)$ is shown in Table 8. In this example, the first 97 rows (not all are shown, hence the ...) are for the same E and μ, and the first 398 are for the same E.

[7]Or, to flip it around an x_3 has an associated $x_0(x_2, x_1)$.

Table 8: An example of the way collated $P(E'|E,\mu)$ (i.e., $x_0(x_3,x_2,x_1)$) tabulated data are stored in the ENDL format. No interpolation meta-data is shown in this example.

E	mu	E'	P(E' | E,mu)
1.00000000e-11	-1.00000000e+00	1.00000000e-18	7.32483405e+04
1.00000000e-11	-1.00000000e+00	8.27577349e-18	1.85252842e+05
1.00000000e-11	-1.00000000e+00	1.55515470e-17	2.37626219e+05
1.00000000e-11	-1.00000000e+00	2.28273205e-17	2.74116635e+05
... (89 rows with the same E and mu)			
1.00000000e-11	-1.00000000e+00	1.24996837e-07	1.00001559e+06
1.00000000e-11	-1.00000000e+00	2.49993674e-07	1.00002607e+06
1.00000000e-11	-1.00000000e+00	4.99987347e-07	1.00003348e+06
1.00000000e-11	-1.00000000e+00	9.99974694e-07	1.00003872e+06
1.00000000e-11	-5.00000000e-01	1.00000000e-18	8.11313020e+04
1.00000000e-11	-5.00000000e-01	8.27586555e-18	2.14499782e+05
1.00000000e-11	-5.00000000e-01	1.55517311e-17	2.79645195e+05
1.00000000e-11	-5.00000000e-01	2.28275967e-17	3.25619648e+05
...			
1.00000000e-11	-5.00000000e-01	1.24998418e-07	1.00000847e+06
1.00000000e-11	-5.00000000e-01	2.49996837e-07	1.00001298e+06
1.00000000e-11	-5.00000000e-01	4.99993674e-07	1.00001618e+06
1.00000000e-11	-5.00000000e-01	9.99987347e-07	1.00001844e+06
1.00000000e-11	0.00000000e+00	1.00000000e-18	8.62163443e+04
1.00000000e-11	0.00000000e+00	8.27595761e-18	2.41691941e+05
1.00000000e-11	0.00000000e+00	1.55519152e-17	3.23256255e+05
1.00000000e-11	0.00000000e+00	2.28278728e-17	3.82542594e+05
...			
6.00000000e+01	1.00000000e+00	5.32357998e+01	2.32311309e-09
6.00000000e+01	1.00000000e+00	5.52360863e+01	2.51977485e-09
6.00000000e+01	1.00000000e+00	5.52360883e+01	2.36795140e-07
6.00000000e+01	1.00000000e+00	5.72362140e+01	2.57006565e-07

Here, it is important to note that all E values are stored many times. In addition, each μ value for a given E value is also stored many times. As the number of E', P pairs for a given E and μ increases, the extra storage required to store redundant E and μ values approaches as factor of 2. For one ENDL $P(E'|E,\mu)$ dataset, the ENDL array format requires 16MBs of disk space. In GND, these data are put into a multiD_XYs(dimension = 3) data container (to be defined later) and only requires 6MBs of disk space. In this example, most of the 2.67 savings factor comes from storing the data in a multiD_XYs(dimension = 3) data container instead of a array data container[8]. In addition to saving space, using functional data containers reduces redundancy and speeds up access time.

If these data were scatter data (i.e., E, μ and E' all differing between adjacent rows) then the data could not be stored in a more compact form. However, these data and much of the data in the ENDL

[8] The additional savings result from storing numbers in a more efficient ASCII form (e.g., 1.23456e+03 is stored as 1234.56 and not 1.23456e+03)

and ENDF formats are not scatter data. Instead, the data are divided into sections with common values (called collated in this article), and each section may be further divided into sub-sections. As in the example above for the ENDL $P(E'|E,\mu)$ format, the data are divided into sections by E and within each section they are divided into sub-sections by μ. The first E section contains all data for the lowest value of E, the next section for the next lowest value of E and so on. Within each E-section, the μ values are also organized into μ-sections sorted by increasing μ values and each μ-section contains many E', P pairs sorted by increasing values of E'.

4.2.1 Multi-valued functions

The XYs container stores single-valued, single-interpolation data that represent a function $x_0(x_1)$. Some data in ENDF are multi-valued and/or have regions with different interpolation rules. To store these type of data, XYs containers can be adjoined with a **regions** container. In addition, some $x_0(x_2, x_1)$ or $x_0(x_3, x_2, x_1)$ data are multi-valued and/or have regions with different interpolation rules along x_2 or x_3, respectively. These are stored in a regions container as adjoining $x_0(x_2, x_1)$ or $x_0(x_3, x_2, x_1)$ containers.

4.2.2 Other functional containers

At times, a tabulated n-dimensional function is given at fixed points (i.e., a grid) along each independent dimension (i.e., axis). A gridded tabulated function can be stored more compactly by storing each axis's grid once and storing the functional values (i.e., x_0) in an array. The functional values are stored in the **array** container (see Section 13) which is embedded in a **gridded** container (see Section 18) that also stores axes information.

Two other 1-d functional data containers are defined, one for a Legendre and one for a polynomial representation of 1-d data. Some angular $x_0(x_1)$ data (e.g., the $P(\mu)$ data in $P(\mu|E)$) are stored as Legendre coefficients. For other 1-d functions, it is convenient to store them as a polynomial expansion. Legendre and polynomial data are stored in the **series** container (see Section 15) with the value for the function attribute being **Legendre** and **polynomial**, respectively.

5 Requirements for General-Purpose Data Containers

Before defining the specifications for the general-purpose data containers, it is worth defining some requirements.

requirements:

1. The underlying data types used in general-purpose data containers shall be compatible with commonly used computer languages and libraries, and shall be represented in a form that makes them easy to read and write using standard tools.

2. Data containers should be designed to be consistent with object-oriented programming. This includes the nesting of data containers when it make sense instead of defining a new sub-container.

3. The structures should reduce redundancy.

4. Containers should make efficient use of computer resources. This is a compromise between:

(a) efficient use of memory - volatile (e.g., RAM) and non-volatile storage (e.g., disk drive).

(b) ease of conversion to other forms.

(c) time to convert to other forms.

5. Distinct regions of data with different meta-data shall be clearly delineated.

6. A mechanism for storing units and labels for data should be specified for each container type. Allow units should be defined by each project.

7. Functional containers should support the inclusion of uncertainty data within the functional container.

8. The specification for each data container shall state whether that container is extensible. If the container is extensible, the specification shall define the process for extending the container.

6 Character encoding and various character sets

While many meta-languages can be used for storing the general-purpose data containers, this article focuses on text-based storage which is platform independent and easily shareable. The general-purpose data containers use the UTF-8 character encoding [8] for text [9]. UTF-8 was chosen because it is now the most commonly used character encoding for the World-Wide-Web [8], because of its ability to represent characters from a wide variety of languages and because it is backwards compatible with the ASCII [7] character set.

This section defines various character sets that are subsets of the 128 characters of the ASCII character-encoding scheme. While most of these subsets, if not all, are well known, it is worth repeating them so their definitions are clear. The specifications in this article will in some places restrict the allowed set of characters that can be used to one or more of the following subsets:

6.1 Letter characters subset

In this article, characters from the **letters** subset along with several other character subsets are the only characters allowed for constructing tag and attribute names (see Sections 7.2.1). The GPDCs use the ISO basic Latin alphabet [9] for its letter subset. This consists of 26 uppercase and 26 lowercase letters. The uppercase letters are the following 26 ASCII characters: ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’, ‘G’, ‘H’, ‘I’, ‘J’, ‘K’, ‘L’, ‘M’, ‘N’, ‘O’, ‘P’, ‘Q’, ‘R’, ‘S’, ‘T’, ‘U’, ‘V’, ‘W’, ‘X’, ‘Y’, and ‘Z’. These characters are encoded in the ASCII character set as the base 10 integers 65 to 90, respectively. The lowercase letters are the following 26 ASCII characters: ‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’, ‘g’, ‘h’, ‘i’, ‘j’, ‘k’, ‘l’, ‘m’, ‘n’, ‘o’, ‘p’, ‘q’, ‘r’, ‘s’, ‘t’, ‘u’, ‘v’, ‘w’, ‘x’, ‘y’, and ‘z’. These characters are encoded in the ASCII character set as the base 10 integers 97 to 122, respectively.

6.2 Arabic digits character subset and numbers

A number can be an integer or a real number. This section defines the characters needed for constructing a number.

[9] Encoding for numbers can be meta-language specific. For example, some meta-languages use binary, and not text, encoding. However, when a number is represented in text, it must follow the format given in Sections 6.2 and 7.3.

The main characters in a number are the ten Arabic numerals which consist of the 10 characters: '0', '1', '2', '3', '4', '5', '6', '7', '8', and '9'. These characters are encoded in the ASCII character set as the base 10 integers 48 to 57, respectively.

In addition to the ten Arabic numerals, integers and real numbers need the plus (i.e., '+') and minus (i.e., '−') characters. These characters are encoded in the ASCII character set as the base 10 integers 43 and 45, respectively. For real numbers the period character (i.e., '.') is needed. This character is encoded in the ASCII character set as the base 10 integer 46. The e-form (see Section 3.5) needs the additional characters 'e' and 'E' as defined in the letter subset.

6.3 Underscore

It is also useful to list the underscore character (i.e., '_') which in the ASCII character set is the base 10 integer 95.

7 Basic data types

We divided data types into three classes: text, numbers and others. Also, this discussion will only consider ASCII and UTF-8 [8] representations of the various data types and their representation in XML That is, binary representation of numbers are not defined in this article as we are only concerned with their ASCII representation in a file. Text data types will be described first, then numbers and finally others.

7.1 Text data needs

For the design of an XML general-purpose data container, four text types need to be defined. These types can be understood by noting the basic structure of an XML document. The main component of an XML document is an element. An element contains 1 to 3 parts and they are:

- The name of the element also called the tag, which herein is called the **tagName**. All elements have a **tagName**.
- The attributes of the element, comprised of a list of 0 or more attributes. Each attribute is a key/value pair. Herein, the key is called the **attributeName** and the value is called the **attributeValue**.
- The body (or content) of the element. The body is optional. If present, it can contain text and/or other elements interspersed. The text component of the body is herein called the **bodyText**.

Hence, the four text types for an XML document are **tagName**, **attributeName**, **attributeValue** and **bodyText**. Some simple examples of an element are:

```
<tagName/>
<tagName attributeName="attributeValue"/>
<tagName attributeName="attributeValue"></tagName>
<tagName attributeName="attributeValue">BODY</tagName>
```

All **tagName**, **attributeName** and **attributeValue** text types are case sensitive. For example, the only allowed Boolean true value is '<true/>' (see 7.7.3); neither '<True/>' nor '<TRUE/>' is an allowed value. There is one exception and that is for floating point value (see Section 3.5). In a floating point value the exponent designator can be either an 'e' or 'E' (e.g., both '1e3' and '1E3' are allowed and are equivalent).

XML is very liberal in the characters that are allowed for **tagName** and **attributeName**. To allow for better association between meta-languages' names and programming variable names, it is best to restrict the allowed characters (see the "Best Naming Practices" in ref. [6]). In addition, the allowed character set for **tagName** and **attributeName** should probably be the same. Herein their character set will be generically called **XMLName**.

7.2 Description of text types

This section defines the **XMLName**, **attributeValue** and **bodyText** types. This section defines additional text types that are useful for the **table** and **values** containers. These additional types are **UTF8Text**, **printableText**, **quotedText** and **tdText**.

7.2.1 XMLName

(**tagName** and **attributeName**)

XMLName represents the set and sequence of characters that are allowed for tag and attribute names. Some meta-languages and programming languages (e.g., C and Python) allow a name to start with the underscore character. While other programming languages (e.g., FORTRAN) do not. For maximum compatibility an **XMLName** text shall not start with an underscore character.

Allowed characters: **XMLNames** shall begin with a character from the ISO basic Latin alphabet (see Section 6.1). All other characters shall be from the following: ISO basic Latin alphabet, Arabic numerals, and/or an underscore. There is no limit on the length of a name, except that it shall contain at least 1 character.

7.2.2 attributeValue

This represents the set and sequence of characters that are allowed for attribute values.

Allowed characters: The allowed values for an attribute will depend on the attribute/element and in general, should be specified by the project defining the attribute/element. A project can use any UTF-8 character in a value deemed necessary. However, some general rules apply:

- If a value of an attribute is an Integer32, UInteger32, Float64 or other defined type then it shall follow the specification for an Integer32, UInteger32, Float64 or the other defined type, respectively (see Sections 7.4, 7.5 and 7.6).
- If any part of the value of an attribute is an Integer32, UInteger32, Float64 or other defined type then that part shall follow the specification for an Integer32, UInteger32, Float64 or the other defined type, respectively.

  For example, if an attribute value contains a Float64 with units, as in 'mass="3.2 kg"', the numeric part of the attribute value shall follow the Float64 specification.

7.2.3 bodyText

Allowed characters: In general, any UTF-8 character is allowed. However, the allowed characters for an element's body can be limited by its parent element's specifications.

7.2.4 UTF8Text

This is text composed of any sequence of UTF-8 characters.

Allowed values: Any sequence of 0 or more UTF-8 characters.

7.2.5 printableText

This is text composed of only the printable ascii characters.

Allowed values: Any sequence of the ascii characters between the space character (decimal 32) to the tilde character (' ' or decimal 126) inclusive.

7.2.6 quotedText

This represents a UTF8Text that is contained between two matching quote characters. The allowed quote characters are the ascii double quote character (i.e., " or decimal 34) and the ascii single quote character (i.e., ' or decimal 39).

Allowed values: A UTF8Text contained between matching single or double quote characters.
For example, the quoted string "abc 123 xyz" is expressed as

```
"abc 123 xyz"
```

or

```
'abc 123 xyz'
```

This is not the same as

```
'abc 123 xyz '
```

as leading and trailing spaces are part of the string.

7.2.7 tdText

This represents a UTF8Text that is contained between the XML start and end elements that define a standard html table cell (i.e., "<td>" and "</td>").

Allowed values: The XML element "<td>" and the UTF8Text it contains.
For example, the quoted string "abc 123 xyz" is expressed as

```
<td>abc 123 xyz</td>
```

This is not the same as

```
<td> abc 123 xyz</td>
```

as leading and trailing spaces are part of the string.

7.3 Description of number types

This section defines some common number types relevant for computer programming and storage. The types defined here are the commonly used number types. Additional types can be defined by each project. The following contains some rules that a project should consider when defining number types. For many computer languages it is important to define the form and size allowed for various types of numbers.

In general, an ASCII integer should not start with the '0' character unless its value is zero. Disallowing a '0' character as the first character in a non-zero integer is desirable since some programming languages interpret an integer starting with a 0 as an octal value. For example, in Python 2 the command 'int(077)' returns the base 10 value 63 (note, just to confuse things, the command 'int("077")' returns the base 10 value 77 and 'eval("int(%s)" % "077")' returns the base 10 value 63).[10] Furthermore, it is best that an ASCII representation of an integer not contain a decimal point (i.e., '.') or be stored using the floating-point e-form. For example, the integer value '123456' should not be represented in any of the forms shown in Table 9.

Table 9: Examples of invalid integers.

Invalid representation	Reason
'123456.'	'.' not allowed.
'1.23456e5'	'.' and e-form not allowed.
'12345600e-2'	e-form not allowed.

There are two reasons for not allowing the decimal period (i.e., '.') or the e-form when storing integers in ASCII form. Firstly, programming languages have functions for converting an ASCII string to an integer, such as the **int** function in Python. In general, these functions fail when the string representation of an integer contains a '.' or is an e-form. A few examples are given below for the programming languages Python, C and FORTRAN.

Python programming example:

```
k = int( "120." )       # A Python Exception is raised by the int function.
```

C programming example:

```
int i = -1, j = -1, k = -1;
k = sscanf( "120. 14", "%d %d", &i, &j );   # Only the first value is converted
                                            # (i.e., k = 1, i = 120 and j = -1).
k = sscanf( "120e-1 14", "%d %d", &i, &j ); # Only one value is converted and it
                                 # is incorrect (i.e., k = 1, i = 120 and j = -1).
```

FORTRAN programming example: The FORTRAN programming language does appear to translate integers containing a '.' or an e-form representation correctly provided the integer value can be expressed in the bits of the floating-point representation.

[10] To confuse the issue even more, in Python 3 the command 'int(077)' raises a SyntaxError exception. Octal values in Python 3 must start with '0o', as in 'int(0o77)'.

The second reason to avoid using '.' or an e-form in integer values is that some programmers use floating-point types to manipulate integers in their codes. Often, this yields floating-point values that are not integers. For example, consider the following Python code:

```
a = 5
b = 7
inv_a = 1. / 5
i = 7
c = b * a * i * ( inv_a / i )
print "%.17e" % c, c / b - 1, int( c )
```

which prints,

```
7.00000000000000089e+00 2.22044604925e-16 7
```

while replacing the assignments for 'a' and 'i' with 'a = 3' and 'i = 11', it prints

```
6.99999999999999911e+00 -1.11022302463e-16 6
```


Mathematically both value are exactly 7, but as can be seen from the examples above, not all calculated values are exactly 7, and the latter assignments yield 6 when converted to an integer.

In the ENDF/B-VII.0 [3] release, there are many instances where integer multiplicities are stored as non-integer values (i.e., floating point representation). For example, in the ^{242}Am evaluation the multiplicity for neutrons for reaction MT 16 (n,2n) has 37 energy dependent values for the multiplicity. All the values use a special FORTRAN 'e'-less format which looks like the **e-form** (see 3.5 but does not contain the 'e' (e.g., '1.999963+0' instead of '1.999963e+0). None of these multiplicities can be converted to an integer using the Python command 'int(m)' where m is one of the values even when the 'e' is inserted back into the string. Only four converted to 2 with the Python command 'int(float(m))' - with the 'e' reinserted - while all the others convert to 1.

While it is impossible to restrict coders from working with non-integer types in their codes, the onus of converting an integer from a code's representation to ASCII (the writer) or vice-versa (the reader) should fall on the writer of the number. This makes the proper interpretation of the integer trivial for the many readers.

Furthermore, if a reader wishes to input an integer as a float, the integer value will be properly represented as long as the value has less digits than the number of significant digits in the float. For example, converting the integer '123456' to a Float64 (see below) and then to an Integer32 will produce the correct value, but converting '123456789' to a Float32 and then to an Integer32 will not. FORTRAN converts the string '123456789' to 123456792 without raising a warning[11].

If integer representations are restricted to not containing leading zeros (unless the value is equal to zero) and the floating-point form is not allowed, then each positive integer has two possible representations (since the '+' is optional), while each negative integer has exactly one representation.

[11] All Integer32 values can be represented exactly as Float64 values.

7.4 Integer32:

This represents the allowed set and sequence of characters, and values that are allowed for a 32-bit signed integer.

Allowed values: Any integer in the range [-2^{31} to 2^{31}) - note that the lower limit is inclusive and the upper limit is exclusive. The minimum allowed value (i.e., -2^{31} = -2147483648) is defined to be Integer32_Min and the maximum allowed value (i.e., 2^{31} - 1 = 2147483647) is defined to be Integer32_Max. The Python regular expression (see section B) for an Integer32 shall be

```
'[+-]?([1-9][0-9]*|0+)'
```

with the restriction that the value shall be in the range [Integer32_Min, Integer32_Max].

C programming equivalent: int32_t

7.5 UInteger32:

This represents the allowed set and sequence of characters, and values that are allowed for a 32-bit unsigned integer.

Allowed values: Any integer in the range [0 to 2^{32}). An unsigned integer is represented in the same way as a signed integer, with the exception that a minus sign (e.g., '-') is not allowed). The minimum allowed value (i.e., 0) is defined to be UInteger32_Min and the maximum allowed value (i.e., $2^{32}-1$) is defined to be UInteger32_Max. The Python regular expression (see section B) for an UInteger32 shall be

```
'+?([1-9][0-9]*|0+)'
```

with the restriction that the value shall be in the range [UInteger32_Min, UInteger32_Max].

C programming equivalent: uint32_t

7.6 Float64:

This represents the allowed set and sequence of characters, and values for a 64-bit floating point number.

Allowed values: Any normalized or denormalized IEEE-754 binary64 [5] value. The Python regular expression (see section B) for a Float64 shall be

```
'[+-]?([0-9]+\.?[0-9]*|\.[0-9]+)([eE](+|-)?[0-9]+)?'
```

with the restriction that the value shall only be in the range for a normalized or denormalized IEEE-754 binary64 value. The number of significant digits (base 10) in a normalized IEEE-754 binary64 value is about 16, and is fewer for a denormalized value. The string representation of a Float64 value can have more significant digits than are supported by a normalized or denormalized IEEE-754 binary64 value; however, these additional digits will generally be ignored when the value is read into a Float64 variable.

C programming equivalent: double

7.7 Description of other data types

7.7.1 whiteSpace

This represents the allowed set and sequence of characters for a white space.

Allowed values: Any of the characters shown in Table 10.

Table 10: Valid white spaces and their ASCII characters.

character name	escape sequence	ASCII decimal value
space	' '	32
tab	'\t'	9
linefeed	'\n'	10
vertical tab	'\v'	11
formfeed	'\f'	12
carriage control	'\r'	13

DISCUSSION POINT Or, maybe it should be restricted to the characters shown in Table 11 to comply with JSON (JavaScript Object Notation) [10].

Table 11: Valid JSON white spaces and their ASCII characters.

character name	escape sequence	ASCII decimal value
space	' '	32
tab	'\t'	9
linefeed	'\n'	10
carriage control	'\r'	13

7.7.2 Sep

This represents the allowed set and sequence of characters that separate values in a list inside many of the general-purpose data containers. Several of the containers have a **sep** attribute that defines the separator for each data instance.

Allowed values:

- One or more white spaces (if allowed by the meta-language) or
- any character (e.g., a comma (",")) defined by a project with an arbitrary number of white spaces before and after it.

DISCUSSION POINT The meta-language JSON requires that the comma character shall be used to separate objects in its **array** and **object** types.

DISCUSSION POINT A proposal was made to support more than one type of separator. However, this puts an additional burden on software reading the data. Tools like Python's "split" function, Matlab's "dlmread", and the c++ "std::getline" do not directly support multiple delimiters.

7.7.3 Boolean

This represents the allowed set and sequence of characters that represent the Boolean **true** and **false** values in general-purpose data containers.

Allowed values: For **attributeValues** the allowed strings are 'true' and 'false'. For a table cell, the allowed values are '`<true/>`' and '`<false/>`'.

As a note, the standard XML representation for Boolean values true and false are 'true' and 'false' respectively.

7.7.4 Empty

This is the token that states that the value of the cell in a table is empty.

Allowed values: The ASCII string '`<td/>`'.

7.8 Some additional types to consider

Below is a list of types that are supported by most, if not all, programming languages but are currently are not used by the GPDCs. They are listed so as not to be ignored.

7.8.1 Integer64

This represents the allowed set and sequence of characters, and values for a 64-bit signed integer.

Allowed values: Any integer in the range [-2^{63} to 2^{63}). The minimum allowed value (i.e., -2^{63}) is defined to be Integer64_Min and the maximum allowed value (i.e., $2^{63}-1$) is defined to be Integer64_Max. The Python regular expression (see section B) for an Integer64 shall be

```
'[+-]?([1-9][0-9]*|0+)'
```

with the restriction that the value shall be in the range [Integer64_Min, Integer64_Max].

C programming equivalent: int64_t

7.8.2 Float32

Comments: Since only ASCII representation of data types is being considered here, and since Float32 is a subset of Float64, this is probably not needed. Currently not defined.

DISCUSSION POINT Even though we are only giving ASCII representation here, we could define some flag meaning "can be safely stored as Float32 without loss of precision."

7.8.3 Octal

Comments: Currently not needed so not defined.

7.8.4 Hexadecimal

Comments: Currently not needed so not defined.

8 General-purpose data containers fundamentals

The requirements for the *general-purpose data containers* are listed in Section 5. The *general-purpose data containers* are specified in Sections 9 through 21. This section describes common features found in many of the *general-purpose data containers*.

8.1 Descriptions of specifications

The specifications below are for XML formatted data. The fundamental entity in XML is the element. An element has a tag or name, a list of attributes (i.e., meta-data) and text (also called body) and/or nested elements. For each container, the specifications shall define the tag name, list of allowed attributes, list of allowed nested elements and text (body). For each attribute, the specifications shall define the allowed values and whether the attribute is required or optional and, if relevant, if it has a default value. A required value does not have to be specified if it has a default value.

8.2 Interpolation

The GPDCs employ the ENDF interpolation scheme as it is more general than the interpolation scheme commonly used in science and engineering. In the common or "traditional" interpolation, each axis has an interpolation rule that defines how to interpolate its variable along its axis. Consider, for example, the function $f(x, y)$ with linear ('lin') interpolation along the x-axis, logarithm ('log') interpolation along the y-axis and 'flat'[12] interpolation along the z-axis. Between any two consecutive x values, the interpolation for the z value is 'flat' as it also is for any two consecutive y values. There is, in this scheme, no way to have a difference interpolation rule for z-axis depending on which independent axis is being interpolated (e.g., for the x- and y-axis in the example).

In the ENDF interpolation scheme, interpolation rules are defined for each independent axis and that rule specifies the interpolation rule for that independent axis and the dependent axis while that independent axis is being interpolated. No interpolation is define for the dependent axis as its interpolation is give for each independent axis. This scheme allows for a different interpolation rule for the dependent axis, depending on which independent axis if being interpolated. In this scheme, two interpolation values are typically required for each independent axis: one for the independent axis and one for the dependent axis (e.g., 'lin,lin', 'lin,log'). In the "traditional" interpolation example above, the interpolation rule for the x-axis is 'lin,flat' and for the y-axis it is 'log,flat'. For a dataset representing $f(x, y)$, the ENDF interpolation allows, for example, the x-axis interpolation rule to be 'lin,log' while the y-axis rule is 'lin,lin'. That is, for $x_i \le x \le x_{i+1}$ and for the $y = y_j$, $f(x, y_i)$ is interpolated as

$$f(x, y_i) = f(x_i, y_j) \exp\left(\log\left(\frac{f(x_{i+1}, y_j)}{f(x_i, y_j)}\right)\left(\frac{x - x_i}{x_{i+1} - x_i}\right)\right) \tag{1}$$

while for $y_j \le y \le y_{j+1}$ and for the $x = x_i$, $f(x_i, y)$ is interpolated as

[12] Flat interpolation is also called 'constant' or 'histogram' interpolation.

$$f(x_i, y) = (f(x_i, y_{j+1}) - f(x_i, y_j)) \left(\frac{y - y_i}{y_{j+1} - y_j} \right) - f(x_i, y_j) \quad . \tag{2}$$

Many of the containers have an interpolation attribute. In general, the value for an interpolation attribute can be specified by each format project. However, several predefined values are listed in Table 12.

Table 12: Predefined interpolation strings. The first four strings contain two sub-strings separated by the comma character (e.g., ‘,’). The sub-string to the left of the comma is the interpolation for the independent axis while the sub-string to the right of the comma is the interpolation for the dependent axis.

string	definition
"lin,lin"	The independent and dependent axes are linearly interpolated.
"log,lin"	The independent axis is logarithmically interpolated and the dependent axis is linearly.
"lin,log"	The independent axis is linearly interpolated and the dependent axis is logarithmically.
"log,log"	The independent and dependent axes are logarithmically interpolated.
"flat"	The dependent value between consecutive x values is constant, equal to the dependent value at the lower x value.

In addition to an interpolation attribute, the nuclear data community requires an interpolation qualifier to produce physical results when interpolating a distribution (e.g., $P(\mu, E'|E)$). For this discussion, the details of why the interpolation qualifier is needed is not important nor is how it is used. What is important is that if the interpolation qualifier is added as part of the interpolation attribute, parsing of the interpolation is required to limit the number of interpolation values. For example, an interpolation qualifier required for nuclear data is called 'unitBase' interpolation. As all the interpolations listed in Table 12 can occur with or without 'unitBase' interpolation (e.g., 'lin,lin' and 'unitBase,lin,lin'), defining a new string for each interpolation rule in Table 12 adds 5 more interpolation rules. And, each additional interpolation qualifier adds 5 more interpolation rules. A better way to handle interpolation qualifiers is to add an additional attribute. In GPDCs, containers that require an interpolation qualifier have an attribute called interpolationQualifier.

8.3 Types of containers

The containers can be divided into several types.

basic containers: These containers store either text or numbers. The basic containers are **text** (Section 9), **values** (Section 12) **array** (Section 13) and **table** (Section 21).

axes containers: These containers are used to specify information about an axis representing a variable. For example, for the function $x_0(x_1)$, the variables x_0 and x_1 each require axis information. The axes containers are **axis** (Section 11) and **axes** (Section 10)

functional containers: These containers store data representing single-valued[13] functions of the form $f(x)$, $f(x, y)$, $f(x, y, z)$, etc or in the generic functional expression used in this article and n-dimensional function is $x_0(x_n, ...x_2, x_1)$. These are known as 1-, 2-, 3- and n-dimensional functions (for $n > 3$). The functional containers are **XYs** (Section 14), **series** (Section 15), **multiD_XYs** (Section 16), **regions** (Section 17), and **gridded** (Section 18).

uncertainty containers: These containers allow for the storing of uncertainty data that is associated with a functional container. The uncertainty containers are **uncertainty** (Section 19) and **uncertainties** (Section 20)

8.4 Common attributes

All the container have the following optional attributes:

index This is an integer and should be used to sort like container when embedded in another element. For example, whenever n-dimensional functional containers are embedded in an (n+1)-dimensional functional container, each embedded container shall have an **index** attribute, indexed sequentially starting at 0.

label When a container is embedded in another container, the parent may define the allow values for the embedded containers **label** attribute. Otherwise, projects are allowed to define allowed values.

style When a container is embedded in another container, the parent may define the allow values for the embedded containers **style** attribute. Otherwise, projects are allowed to define allowed values.

While these common attributes are optional, they may be required for a container when it is embedded in other container.

8.5 Functional common attributes

In addition to the common attributes, all functional containers have the following attributes:

value For an n-dimensional function, this is its associated x_{n+1} value.

valueType This is the type of the **value** attribute.

9 text container

This container stores a list of characters. This container can be used to store documentation, for example.

[13] Except for the **regions** container which allows for a discontinuity.

9.1 Specifications

Tag: text

Attributes: The list of allowed attributes are:

index: [Integer32, optional] Defined by the parent element.

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

encoding: [UTF8Text, default is 'ascii'] One of 'ascii', 'utf8'.

markup: [UTF8Text, default is 'none'] One of 'xml', 'xhtml', 'latex' or other markup defined by a project.

length: [Integer32, optional] The number of encoded characters.

Body: The list of characters.

9.2 Examples

Example 1: Text representing the latex markup for $\alpha \times x^{3/2}$.

```
|       <text markup="latex">$\alpha \times x^{3/2}$</text>
```

Example 2: Same as example 1 but with the text wrapped using the special XML CDATA[14] section (i.e., the text between '<![CDATA[' and ']]>').

```
|       <text markup="latex"><![CDATA[$\alpha \times x^{3/2}$]]></text>
```

10 axes container

Many of the data containers represent functions that have independent and dependent axes. The **axes** element provides a way to assign a label and unit to each axis. If an **axes** element is present, each independent and dependent axis must have an **axis** element. The **axis** elements are indexed 0 to n where n is the number of independent axes. For the function $x_0(x_n, ..., x_1)$, index 0 is for dependent axis x_0, 1 is for independent axis x_1, ... and n is for the independent axis x_n.

10.1 Specifications

Two types of axes elements are allowed. One type gives an xlink to another axes element and has the form:

```
<axes xlink:href="/link/to/another/axes/element"/>
```

[14] An XML CDATA section is need whenever a string contains characters that are normally reserved for xml markup, such as '<', '>', '/', and '&'".

The second form is

Tag: axes

Attributes: The list of allowed attributes are:

index: [Integer32, optional] Defined by the parent element.

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

Body: The list of **axis** elements. One required for each independent and dependent axes.

11 axis container

This container stores an *index*, *label*, *style* and *unit* for an axis. Depending on the *style*, a grid for the axis can also be stored.

11.1 Specifications

Tag: axis

Attributes: The list of allowed attributes are:

index: [Integer32, required] An integer that indicates which independent/dependent axis this **axis** element belongs to as defined in Section 10.

label: [UTF8Text, required] The label for the axis.

style: [UTF8Text, default='none'] A string denoting the type of grid associated with this axis. Allowed values are 'none', 'points', 'boundaries' and 'parameters'.

unit: [UTF8Text, optional] The unit for the axis.

interpolation: [UTF8Text, contingent] Defines the interpolation to be used between consecutive domain points along this axis. Contingency: required when **axis**' style is 'points' or 'boundaries' and interpolation is other than 'lin,lin'.

interpolationQualifier: [UTF8Text, contingent] Contingency: required when **axis**' style is 'points' or 'boundaries' and interpolationQualifier is other than 'none'.

Body: The list of allowed elements are:

values: Required when **axis**' style is other than 'none'.

11.2 Examples

Example 1: An axes element for multiplicity as a function of energy (i.e., $m(E)$):

```
|       <axes>
|           <axis index="1" label="energy_in" unit="eV"/>
|           <axis index="0" label="multiplicity" unit=""/></axes>
```

Example 2: An axes element for $P(\mu|E)$:

```
|        <axes>
|            <axis index="2" label="energy_in" unit="eV"/>
|            <axis index="1" label="mu" unit=""/>
|            <axis index="0" label="P(mu|energy_in)" unit=""/></axes>
```

Example 3: An axes element for $P(\mu|E)$ where the x_2 (i.e., E) and x_1 (i.e., μ) have style="points" and the x_2 axis has "log,lin" interpolation:

```
|        <axes>
|            <axis index="2" label="energy_in" unit="eV" style="points">
|               <interpolation value="log,lin"/><values> ... </values><axis>
|            <axis index="1" label="mu" unit="" style="points">
|               <values> ... </values><axis>
|            <axis index="0" label="P(mu|energy_in)" unit=""/></axes>
```


12 values container

Data containers have meta-data and data. In most cases the data have three common meta-data. These are 1) the number of data values (i.e., its **length**), 2) the **type** of data and 3) the character used to separate consecutive values (i.e., the **sep** character). For this reason, many of the containers store their data in a **values** container where, if needed, these three meta-data are stored. This container also allows for compressing leading and trailing zero data. If the **start** attribute is defined, it specifies the first non-zero data value. If the sum of the **start** and **length** attributes is less than the value of the **size** attribute, then the the **size** value represents the total number of data values where all non-specified values are zero. All data in this container must be of the same type and separated by 'sep'.

12.1 Specifications

Tag: values

Attributes: The list of allowed attributes are:

index: [Integer32, optional] Defined by the parent.

label: [UTF8Text, optional] Defined by the parent.

style: [UTF8Text, optional] Defined by the parent.

length: [Integer32, optional] Number of data stored in the body of the container.

type: [UTF8Text, default is "Float64"] Specifies the type of data in the body (e.g., Integer32, Float64). Only one type of data can be stored in each instance of a values container.

sep: [FIXME, default is " " (i.e., the space character)]: The character used to separate entities in the body.

start: [Integer32, default is "0"] For start="N", the first N values are zero and are not stored.

size: [Integer32, contingent] The total number of data values including leading and trailing zero values that are not stored. Contingency: required when the sum of **start** and **length** do not added to the total number of data values.

Body: The body contains the data for the parent container.

12.2 Examples

Example 1: values container representing the floating point numbers 1.2, 3.2 and 2.3.

```
|       <values length="3" sep="," type="Float64"> 1.2, 3.2, 2.3</values>
```

Example 2: values container representing the floating point numbers 0, 0, 0, 0, 0, 1.2, 3.2 and 2.3.

```
|       <values start="5" length="3" sep="," type="Float64"> 1.2, 3.2,  2.3</values>
```

Example 3: Same as Example 2 but with no length attribute and uses of the default sep and type attributes.

```
|       <values start="5> 1.2 3.2  2.3</values>
```


Example 4: values container representing the floating point numbers 0, 0, 0, 0, 0, -1.2, 7.3, 2.3, -11, 0, 0 and 0.

```
|       <values start="5" length="4" size="12">-1.2 7.3 2.3 -11</values>
```

13 array container

The array container stores numeric data on a multi-dimensional grid. An array of dimension n is a list containing one or more n-1 dimensional arrays which must all have the same shape (see definitions below). Each of those in turn contain a list of n-2 dimensional arrays, and so on down to the 0-dimensional array which is a number.

For an n-dimensional array with N_i grid points on the i^{th} dimension (where i ranges from 1 to n), there are $N_\mathrm{n} \times N_{\mathrm{n}-1} \times \cdots \times N_2 \times N_1$ total values of the array. The most commonly used array is the matrix (i.e., the 2-dimensional array).

Arrays are used frequently in nuclear data, whenever the underlying data are stored on a uniform grid. For example, a covariance matrix can be stored in a 2-dimensional array while a transfer matrix (for use in deterministic transport codes) requires a 3-dimensional array. While the simplest way to store an array is to explicitly list all the values, substantial space savings can often be achieved by using various compression strategies. One example where compression is valuable in a nuclear data evaluation is the ^{232}Th resonance parameter covariance matrix found in the ENDF-VII.1 [4] neutron sub-library, MF=32 MT=151. The matrix shape is 2781×2781, yet only about 9000 of these values (0.1% of the matrix) are non-zero. If we assume that each zero is stored using two units of ASCII storage including

the separator (i.e., ' 0'), and also assume that each non-zero value takes up a total of 24 units, then the zero values will require about $2781 \times 2781 \times 2$ / (9000×24), or about 71, times more memory than the non-zero values occupy. It is therefore beneficial that the array data container be designed to allow for storing various compression schemes.

13.1 Definitions

shape: the shape of an n-dimensional array is a list of n integers representing the number of values (i.e., length) along each dimension. For example, shape='4,3,6' is a 3-dimensional array with 4 values along the first dimension, 3 values along second dimension and 6 values along the third dimension. An array with shape='5' represents a 1-dimensional list with indices ranging from 0 to 4 inclusive. The length of the **shape** is equal to the number of dimensions in the array.

size: The size of an array is the total number of values it contains. The size is equal to the product of the length of each dimension: for an array with shape '4,6,3', the size is $4 \times 6 \times 3$ or 72.

storage order: The storage order determines how an array is laid out in contiguous memory. For example, the 2×3 array

$$C = \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$$

could be laid out either with the rows stored one after another as '1 2 3 4 5 6' (this is known as **row-major** storage order) or with the columns stored one after another as '1 4 2 5 3 6' (**column-major** storage order). More generally, for an n-dimensional array, row-major storage means that the right-most array index (see **array indexing** description below) changes fastest in contiguous memory, followed by the second-to-right-most and so on. Column-major storage means that the left-most index changes fastest, followed by second-to-left-most and so on.

flattened array: Flattening an n-dimensional array means converting it into a 1-dimensional array with the same total number of elements as the original. The order of those elements depends on the array's storage order. For example, flattening array C (above) produces a 1-d array with six elements. Those elements are 1 2 3 4 5 6 if the storage order is row-major, or 1 4 2 5 3 6 if the storage order is column-major.

array indexing: A list of integer indices can be used to point to any value in an array, or to a sub-array within an array. For an n-dimensional array, the index list contains from 1 to n distinct indices (e.g., $[i_{\mathrm{n}}, i_{\mathrm{n}-1}, ..., i_2, i_1]$). Each item in the index must be a non-negative integer, and must be less than the length of the array along that dimension. Indexing is 0-based. For compactness, arrays are often stored as a list of values with implicit demarcation. For example, an array of shape '3,4' that uses **row-major** storage order may store its 12 values as the string 'v1, v2, v3, v4, v5, v6, v7, v8, v9, v10, v11, v12' (here v1 symbolically represents the first value and so on). The first value v1 is indexed as array[0][0], the second value v2 as array[0][1] and the last value v12 as array[2][3]. Indexing this array with only the index 1, as in array[1], yields the 1-dimensional array 'v5, v6, v7, v8' of shape '4'. This is the same array indexing notation used for the C and Python programming languages.

The element pointed to by an array index does *not* depend on storage order: in the example above, array[1,2] points to v7 no matter whether the array is stored as **row-major** or **column-major**.

Two indexing notations are used and are equivalent. The first notation lists the indices as comma separated values enclosed between '[' and ']'. For example, array[3, 2, 4]. The second notation lists each index enclosed between '[' and ']' (this is the notation used in the C programming language). The prior example in this latter notation is array[3][2][4]. For more details on arrays and array indexing please see `http://en.wikipedia.org/wiki/Array_data_structure`.

storage: An array may contain many values that are 0, and/or it may be symmetric or anti-symmetric about the diagonal. In these cases, the array can often be stored more efficiently (i.e., compressed) by using sparse and/or symmetric forms. The array container includes optional **compression**, **triangular** and **permutation** attributes describing the compression strategies (if any) used to reduce storage size.

compression: The following **compression** types are defined:

none: every value is listed explicitly (unless the **triangular** attribute is other than 'none'; see below).

diagonal: if most non-zero values in an array lie on or near the diagonal, the array can often be stored more efficiently by traversing along the direction of the array diagonal. The simplest example is when *all* off-diagonal elements in the array are zero, so that only diagonal values need to be specified. For an n-dimensional array with N values along each dimension, only the values with indices $[i_n, i_{n-1}, \ldots, i_2, i_1]$ where $i_1 = i_2 = ... = i_n$ are stored. For example, to store a diagonal array with shape = '4,4,4', only the values [0,0,0], [1,1,1], [2,2,2], [3,3,3] need to be stored.

DISCUSSION POINT **Should 'diagonal' arrays be allowed to have different lengths along various dimensions? For example, if the only non-zero elements in a 5x3 array are at [0,0], [1,1], [2,2], should we allow storing it as diagonal?**

Diagonal storage is made more general by permitting the storage of off-diagonal vectors that lie parallel to the **main** diagonal, along with starting indices for each vector. For arrays whose non-zero values lie near the diagonal[15], diagonal storage can be an optimal compression scheme. For example, the 2-dimensional array

$$C = \begin{matrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 6 \\ 0 & 0 & 0 & 7 \end{matrix}$$

can be stored using diagonal compression as two vectors: [1 3 5 7] starting at indices [0,0] and [2 4 6] starting at indices [0,1].

This array can be stored on disk using two **values** elements. One as its **label** attribute set to **startingIndices**, contains the starting indices with length equal the number of diagonal vectors times the number of array dimensions. The other has no **label** attribute and contains the diagonal vectors with length equals the sum of the lengths of all diagonal vectors. In the example above, the startingIndices are [0 0 0 1], and the values are [1 3 5 7 2 4 6]. For higher dimensions see the examples in section 13.3.

[15] For a 2-d array (i.e., a matrix) this is called a banded matrix.

flattened: A general method for storing any sparse n-dimensional array is to start by flattening the array so that there is only one index and then to define three separate **values** elements as: 1) starting indices in the new flattened array, 2) the number of values (**numberOfValues**) that will be given following each starting index, and 3) the list of values, whose length is equal to the sum of the **numberOfValues** values. For example, consider the following matrix:

$$C = \begin{array}{rrrrr} 1 & -1 & 0 & -3 & 0 \\ -2 & 5 & 0 & 0 & 0 \\ 0 & 0 & 4 & 6 & 4 \\ -4 & 0 & 2 & 7 & 0 \\ 0 & 8 & 0 & 0 & -5 \end{array}$$

This matrix can be stored in a flattened **array** using three separate **values** elements as (assuming row major storage order):

- flatIndices = 0, 12, 21, 24
- numberOfValues = 7, 7, 1, 1
- values = 1, -1, 0, -3, 0, -2, 5, 4, 6, 4, -4, 0, 2, 7, 8, -5

The first seven items in **values** correspond to the seven elements starting at index 0 of the flattened array, the next seven items correspond to the seven elements starting at index 12, and the last two items correspond to elements 17 and 20 respectively. Together with the **shape** and **storageOrder** attributes, these three arrays completely describe any n-dimensional array. In this example, some zeros appear in the **values** array. This is not required, but if two non-zero values are separated by a single 0 this helps achieve greater compression.

embedded: Some arrays can be efficiently represented by breaking them into multiple sub-arrays. For example, if the original array is

$$C = \begin{array}{rrrrr} 1 & 2 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 \\ 0 & 0 & 5 & 6 & 7 \\ 0 & 0 & 8 & 9 & 10 \\ 0 & 0 & 11 & 12 & 13 \end{array} \tag{3}$$

it can be represented as two separate blocks:

$$C_1 = \begin{array}{rr} 1 & 2 \\ 3 & 4 \end{array} \quad \text{and} \quad C_2 = \begin{array}{rrr} 5 & 6 & 7 \\ 8 & 9 & 10 \\ 11 & 12 & 13 \end{array}$$

where C_1 starts at indices (0,0) in the original array, and C_2 starts at indices (2,2).
The **embedded** compression scheme allows decomposing arrays in this fashion. An embedded n-dimensional array with shape = $(M_n, M_{n-1}, ..., M_1)$ contains 1 or more n-dimensional sub-arrays. Each sub-array has its own shape $(m_n, m_{n-1}, ..., m_0)$ along with starting indices $(s_n, s_{n-1}, ..., s_0)$ indicating where in the **full** array it begins. The shape and starting indices of sub-arrays are constrained such that $0 \leq s_j \leq M_j - m_j$ along

each dimension j (in other words, sub-arrays must be small enough to fit inside the full array after being offset by their starting indices).
For an n-dimensional array C containing K sub-arrays (labeled $c_0, c_1 ..., c_{K-1}$), C can be computed as follows:

$$C[i_n, i_{n-1}, ..., i_0] = \sum_{k=0}^{K-1} w(k)\ c_k[i_n - s_n, i_{n-1} - s_{n-1}, ..., i_0 - s_0]$$

where the weight function $w(k)$ is equal to 1 if sub-array k satisfies the condition $0 \leq i_j - s_j \leq m_j$ along each dimension j, or equal to 0 otherwise. Any element of C not covered by any of the sub-arrays is equal to 0. Sub-arrays cannot overlap.
Each sub-array can store data using any of the **compression**, **triangular** and **permutation** attributes. Also, no restriction is placed on the storage order used by each sub-array. However, each sub-array must contain the same data type as the parent array.
Embedded arrays are quite flexible, and offer many different ways of decomposing an array into constituents. The choice of the best way to decompose any given array is left up to the user.
DISCUSSION POINT Should we allow sub-arrays to have fewer dimensions than the original? This adds some complexity since we would have to define how to unpack those smaller-dimension arrays into the **full** array. For example, if a 4x4 array has a 1-dimensional sub-array with shape = (3) and starting indices = (1,1), does it get unpacked along the rows or along the columns of the full array? Can this be determined from the storageOrder attribute? Or, does the sub-array need to explicitly have 2 dimensions, with shape = (3,1) or (1,3)?
I can think of some possible cases where it would be nice to use lower-dimensional arrays inside an embedded array. For example, if the first 'sheet' of a 3x3x3 array is a symmetric (or diagonal) 3x3 array, we can't store it as 3x3x1 and still take advantage of the symmetry / diagonal nature since a (3x3) **diagonal** array is treated differently from a (3x3x1) diagonal array (actually, right now we don't allow (3x3x1) arrays to be diagonal, although that is another discussion point).

symmetry: An n-dimensional array is symmetric if

- the value at location [$i_{\mathrm{n}}, i_{\mathrm{n-1}}, \cdots, i_2, i_1$] is the same for all permutations of the indices $i_1, i_2, \cdots, i_{\mathrm{n}}$.

For a 2-dimensional array, this reduces to the requirement that

- array[i_2, i_1] = array[i_1, i_2] for all combinations of i1 and i2.

An n-dimensional array is anti-symmetric if the value is multiplied by -1 when two indices are interchanged.

For a 2-dimensional array, this reduces to the requirement that

- array[i_2, i_1] = -array[i_1, i_2] for all combinations of i1 and i2.

Symmetric and anti-symmetric arrays can be stored in a compact form by only listing one permutation of the indices. Two common ways of storing symmetric arrays are **triangular lower-diagonal**, where only the elements on and below the main diagonal are stored (i.e., indices with $i_{\mathrm{M}} \geq i_{\mathrm{n-1}} \geq ... \geq i_2 \geq i_1$), and **triangular upper-diagonal**, where only

elements on and above the main diagonal are stored (i.e., indices with $i_{\mathrm{M}} \leq i_{\mathrm{n}-1} \leq ... \leq i_2 \leq i_1$).

For example, the following is a symmetric array with shape='4,4'. In this example, array indices are listed explicitly:

```
second index --> | 0  1  2  3
first index    ---------------
     |          0| 1  2  4  7
     |          1| 2  3  5  8
    \|/         2| 4  5  6  9
                3| 7  8  9  0
```

This array can be stored using triangular lower-diagonal symmetry as the list 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 (assuming the storage order is **row-major**). That is, the lower-diagonal triangle is:

```
1
2  3
4  5  6
7  8  9  0
```


The anti-symmetric array:

```
 0  -1   2
 1   0   1
-2   1   0
```

can be stored using the upper-diagonal as the list -1, 2, 1 . The zeros on the diagonal are not stored.

The **triangular** attribute specifies whether only the *lower* or *upper* hyper-triangle is stored. The **permutation** attribute specifies whether the array is symmetric (*permutation="+1"*) or anti-symmetric (*permutation="-1"*). If an array has only non-zero values in the *lower* or *upper* hyper-triangle, it can be stored compactly by setting **triangular** appropriately and with *permutation="none"*.

13.2 Specifications

Tag: array

Attributes: The list of allowed attributes are:

index: [Integer32, optional] Defined by the parent element.

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

shape: [FIXME, required] a comma-separated list of integers representing the length of the array along each dimension. The integer type is Integer32.

compression: [UTF8Text, default='none'] flag indicating whether a sparse storage scheme is being used, and if so which scheme is used. Allowed value is one of the following: 'none', 'diagonal', 'flattened' and 'embedded'. Projects can also define their own compression schemes.

triangular: [UTF8Text, default='none'] Allows for storing values in only the upper- or lower-diagonal hyper-triangle. If **permutation** is 'none', non-stored values are zero. Otherwise, they are determine by the **permutation** attribute.

permutation: [UTF8Text, default='none'] If **triangular** is other then 'none', this attribute specifies whether the permutation of two indices is +1 or -1. Allowed values are 'none', '+1' '-1.

storageOrder: [UTF8Text, default='row-major'] Indicates whether the data are stored in row-major or column-major order (i.e., whether the last or first index is varying the fastest. Allowed value are 'row-major' or 'column-major',

offset: [FIXME, required if this array is nested inside an **embedded** array; not allowed otherwise] gives the starting indices for a sub-array nested inside an array that uses **embedded** compression. A comma-separated list of integers equal in dimension to the parent array. The integer type is Integer32.

Body: A list of **values** elements that depends on the value of the **compression** attribute. The **label** attribute of each **values** element designates the type of data stored that element. For the **values** elements containing the data of the array, label shall not be specified. For each **values** element the **index** and **style** attributes shall not be used. The allowed **values** elements for different combinations are:

compression='none': in this case the array only contains a **values** element that contains the array data. The **label** attribute of the **values** element shall not be used. The number of data in the **values** element depends on the array shape and the value of the **triangular** and **permutation** attributes.

compression='diagonal': if all off-diagonal elements are zero, then the diagonal array only needs to contain a **values** element, with size = N (the smallest array dimension). If some off-diagonal elements are included, then the array contains two **values** elements. One element has label="startingIndices" and contains the starting indices. The other element contains the array data and its **label** attribute shall not be used.

compression='flattened': In this case the array contains three **values** elements. One element contains the flat indices and has **label**="flatIndices". Other element contains the number of values data and has **label**="numberOfValues". The third element contains the array data and its **label** attribute shall not be used. The **flatIndices** and **numberOfValues** both shall have data type Integer32, and must have the same size.

compression='embedded': contains 0 or more child **array** elements. Each child array shall have the same data type as the parent array, must have an **offset** attribute, and must obey size restrictions as described in section 13.1.

13.3 Examples

The following are sample arrays showing how data are stored for several combinations of **compression**, **triangular** and **permutation** attributes. Newlines are used in some of these examples to help the reader visualize the arrays, but newlines are not required:
Example 1: an array used to store 1-dimensional data:

```
|    <array shape="5">
|            <values length="5"> 3.14 1.59 2.65 3.589 7.93 </values></array>
```

Example 2: a 2-dimensional array. In this example the **compression** and **triangular** attributes can be omitted since they are both equal to the defaults:

```
|    <array shape="4,4" compression="none" triangular="none">
|      <values length="16">
|        1.1  2.7  3.6  0.0
|        2.7  4.8  0.7  0.2
|        3.6  0.7  5.4  0.1
|        0.0  0.2  0.1  3.6</values></array>
```


The same data can be stored more compactly by taking advantage of symmetry:

```
|    <array shape="4,4" compression="none" triangular="lower"
|        storageOrder="row-major">
|      <values length="10">
|        1.1
|        2.7  4.8
|        3.6  0.7  5.4
|        0.0  0.2  0.1  3.6</values></array>
```

Example 3:

```
|    <array shape="4,4" compression="none" triangular="none">
|      <values length="16">
|        2.3    0    0    0
|        0    4.6    0    0
|        0      0  5.4    0
|        0      0    0  6.8</values></array>
```

This can be stored more compactly using the compression="*diagonal*" as:

```
|    <array shape="4,4" compression="diagonal">
|      <values> 2.3 4.6 5.4 6.8 </values></array>
```

Example 4: 3-dimensional array with most non-zero elements near the diagonal. Stored first without any compression as:

```
|    <array shape="3,3,3">
|      <values length="27">
|        1 2 0
|        0 3 0
|        0 0 0
|
|        0 0 0
|        0 2 3
|        0 0 4
|
|        0 0 0
|        0 0 0
|        0 0 3</values></array>
```

The same data can also be stored using diagonal compression as:

```
|    <array shape="3,3,3" compression="diagonal">
|      <values label="startingIndices" dataType="Integer32" length="9">
|         0 0 0
|         0 0 1
|         0 1 1 </values>
|      <values>
|         1 2 3  <!-- diagonal array starting at 0,0,0 -->
|         2 3    <!--            -"-             0,0,1 -->
|         3 4    <!--            -"-             0,1,1 --></values></array>
```

Note that the main diagonal (starting at 0,0,0) has three values, while the other diagonals have 2 values.
Example 5: Sparse 3-dimensional array where all non-zero elements are found in one corner.

```
|    <array shape="2,8,15" storageOrder="row-major"
|          compression="flattened">
|      <values label="flatIndices">97 111 221 235</values>
|      <values label="numberOfValues" length="4">8 9 4 4</values>
|      <values length="25">
|       12.2 6.431  0.983   2.121     8.7205 0.511   1.24e-3 4.24e-5
|       8.25 2.154  1.232   0.963     3.126  7.82e-2 6.18e-4 9.29e-6 1.23e-8
|      2.843 0.9261 6.23e-3 7.4502e-6
|      3.126 1.5723 0.3421  4.167e-4</values></array>
```

The same array can also be stored using **embedded** compression:

```
|    <array shape="2,8,15" compression="embedded">
|      <array shape="1,2,9" offset="0,6,6">
|        <values length="18">
|         0.0  12.2   6.431   0.983     2.121  8.7205  0.511   1.24e-3 4.24e-5
|         8.25 2.154  1.232   0.963     3.126  7.82e-2 6.18e-4 9.29e-6 1.23e-8
|            </values></array>
|      <array shape="1,2,5" offset="1,6,10">
|        <values length="10">
|         0.0   2.843  0.9261 6.23e-3  7.4502e-6
|         3.126 1.5723 0.3421 4.167e-4 0.0 </values></array></array>
```

14 XYs container: a Tabulated x0(x1) functional

This container stores a single-valued function $x_0(x_1)$ (i.e., $y(x)$) as a tabulated list of (x_i, y_i) pairs with $x_i < x_{i+1}$.

14.1 Specifications

Tag: XYs

Attributes: The list of allowed attributes are:

index: [Integer32, contingent] Defined by the parent element. Contingency: required when more than one $x_0(x_1)$ container can be embedded in another data container.

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

value: [determined by valueType attribute, contingent] Specifies the value of x_2. Contingency: required when container is embedded in a higher dimensional functional container.

valueType: [UTF8Text, default="Float64"] Specifies the data type of value.

interpolation: [UTF8Text, required] The interpolation rule along the x_1 axis.

Body: The following containers are allowed in the body.

axes: [optional] The axes information for the data. For an embedded container, the axes element is inherited from its parent container if not present.

values: [required] The specifications of the **values** element are given in Section 12.

uncertainties: [optional] Contains a list of **uncertainty** containers.

14.2 Examples

Example 1: A tabulated $f(x)$ with (x, y) points (1e-5, 0), (1e-3, 0.12), (0.1, 2e-5), (1, 3.14e-5) and (2e7, 4.3e-12).

```
|        <XYs interpolation="lin,lin">
|            <values length="10">
|                1e-5 0  1e-3 0.12  0.1 2e-5  1 3.14e-5  2e7 4.3e-12</values></XYs>
```

15 series container:

This container stores a function $x_0(x_1)$ (i.e., $y(x)$) as coefficients of a polynomial sequence. That is, for coefficients C_i and polynomial sequence $P_i(x)$, the function $y(x)$ is defined as

$$y(x) = \sum_{i=\text{iMin}}^{\text{iMax}} C_i \, P_i(x) \tag{4}$$

Predefined polynomial sequences are listed in Table 13.

Table 13: List of predefined polynomial sequences.

	defaults			
Name	iMin	domainMin	domainMax	series
polynomial	0	$-\infty$	∞	$y(x) = \sum_{i=\text{iMin}}^{\text{iMax}} C_i \, x^i$
Legendre	0	-1	1	$y(x) = \sum_{i=\text{iMin}}^{\text{iMax}} C_i \, P_i(x)$ where $P_i(x)$ is the Legendre polynomial of order i.

This is a 1-dimensional functional $x_0(x_1)$ container (i.e., it is a representation of a function of the form $x_0(x_1)$ or $y(x)$).

15.1 Specifications

Tag: series

Attributes: The list of allowed attributes are:

index: [Integer32, contingent] Defined by the parent element. Contingency: required when container is embedded in a higher dimensional container]

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

value: [determined by valueType attribute, contingent] Represents the x_2 value for the container. Contingency: required when embedded in a 2-dimensional functional container.

valueType: [UTF8Text, default="Float64"] Specifies the data type of value attribute.

function: [UTF8Text, required] One of the names listed in Table 13 or a project-defined polynomial series.

iMin: [Integer32, optional: default is 0] iMax is determined from the iMin and the **size** as determined by the **values** element.

domainMin: [UTF8Text, optional] $y(x)$ is only valid from domainMin to domainMax. Must be less than or equal to domainMax.

domainMax: [UTF8Text, optional] See domainMin.

Elements: The list of allowed elements are:

axes: [optional] The axes information for the data. For an embedded container, the axes element is inherited from its parent container if not present.

values: [required] The specifications of the **values** element are given in Section 12. The numeric values are the coefficients C_i listed by consecutive order i starting with iMin.

uncertainties: [optional] Contains a list of **uncertainty** containers.

15.2 Examples

Example 1: An example for Legendre series is

```
|       <series function="Legendre">
|           <values length="4">1 0.1 -0.02 1.3e-5</values></series>
```


Example 2: The polynomial series

$$f(x) = 1 + 0.1x - 0.02x^2 + 1.3 \times 10^{-5}x^3 \tag{5}$$

can be represented as

```
|       <series function="polynomial">
|           <values length="4"> 1 0.1 -0.02 1.3e-5</values></series>
```

16 multiD_XYs container

This container stores the n-dimensional function $x_0(x_n, ..., x_1)$ for $n > 1$ as a list of (n-1)-dimensional functions $x_0(x_{n-1}, ..., x_1)$. Each (n-1)-dimensional function stores its associated x_n value in its **value** attribute. The embedded $x_0(x_{n-1}, ..., x_1)$ containers are sorted by increasing x_n. The index values are incremented by 1 and the first embedded container has index 0.

16.1 Specifications

Tag: multiD_XYs

Attributes: The list of allowed attributes are:

index: [Integer32, contingent] Defined by the parent. Contingency: required when container is embedded in a higher dimensional container

label: [UTF8Text, optional] Defined by the parent.

style: [UTF8Text, optional] Defined by the parent.

value: [determined by valueType attribute, contingent] Represents the x_{n+1} value for the container. Contingency: required when embedded in a (n+1)-dimensional container.

valueType: [UTF8Text, default="Float64"] Specifies the data type of value.

dimension: [Integer32, required] The dimension n of the function, equal to the number of independent axes.

interpolation: [UTF8Text, default="lin,lin"] The interpolation rule along the x_n axis.

interpolationQualifier: [UTF8Text, default="none"] The interpolation qualifier.

Body: The list of the following containers:

axes: [optional] The axes information for the data. For an embedded container, the axes container is inherited from its parent container if not present.

list of 2 or more [required] $x_0(x_{n-1}, ..., x_1)$ functional containers.

uncertainties: [optional] Contains a list of **uncertainty** containers.

16.2 Examples

Example 1: As an example, for an $x_0(x_2, x_1)$ function for $P(\mu|E)$, the XML representation would look like:

```
|       <multiD_XYs dimension="2" interpolation="lin,lin">
|           <axes>
|               <axis index="2" label="energy_in" unit="eV"/>
|               <axis index="1" label="mu"/>
|               <axis index="0" label="pdf(mu|energy_in)"/></axes>
|           <XYs value="1e-05" index="0" interpolation="lin,lin">
|               <values> ... </values></XYs>
|           <XYs value="1e-01" index="1" interpolation="lin,lin">
|               <values> ... </values></XYs>
|           <XYs value="2e7" index="2" interpolation="lin,lin">
|               <values> ... </values></XYs></multiD_XYs>
```

Example 2: An example for the XML representation of a $x_0(x_3, x_2, x_1)$ function for $P(\mu, E'|E)$ would look like:

```
|       <multiD_XYs dimension="3" interpolation="lin,lin">
|           <axes xlink:href="/templates/axes/angularEnergy"/>
|           <multiD_XYs value="1e-5" index="0"
|                       dimension="2" interpolation="log,log">
|               <XYs value="-1" index="0" function="Legendre">
|                   <values> ... </values></XYs>
|               <XYs value="-0.5" index="1" function="Legendre">
|                   <values> ... </values></XYs>
|               ...
```

```
|                <XYs value="1" index="4" function="Legendre">
|                    <values> ... </values></XYs></multiD_XYs>
|            ...
|            <multiD_XYs value="2e7" index="20" dimension="2">
|                <XYs value="-1" index="0">
|                  <values> ... </values></XYs>
|                <XYs value="-0.5" index="1" interpolation="log,log">
|                  <values> ... </values></XYs>
|                ...
|                <XYs value="1" index="4">
|                  <values> ... </values></XYs></multiD_XYs></multiD_XYs>
```

17 regions container

This container stores an n-dimensional function as a list of adjoining n-dimensional functions where each function represents a different region of the highest dimensional axis. The dependent values for the adjoining functions can be discontinuous and their interpolation rules can be different. Here, the word **adjoining** means that for the highest dimensional axis, the upper domain boundary for region i must be the same as the lower domain boundary for region $i+1$. That is, there can be neither gaps nor overlaps in the domain.

17.1 Specifications

Tag: regions

Attributes: The list of allowed attributes are:

index: [Integer32, contingent] Defined by the parent. Contingency: required when container is embedded in a higher dimensional container.

label: [UTF8Text, optional] Defined by the parent.

style: [UTF8Text, optional] Defined by the parent.

value: [determined by valueType attribute, contingent] Represents the x_{n+1} value for the container. Contingency: required when embedded in an (n+1)-dimensional function.

valueType: [UTF8Text, default="Float64"] Specifies the data type of value.

dimension: [Integer32, required] The dimension n of the function, equal to the number of independent axes.

boundary: [optional] Specifies how to evaluate the function at the intersection between two **region**s. Allowed value is one of 'lower', 'average', and 'upper'. If l and u are the values at the boundary for the lower and upper regions, respectively, then the value at the boundary is l, $(l+u)/2$ and u for 'lower', 'average' and 'upper', respectively. **Caleb, do we really want this?**

Body: The list of allowed containers are:

axes: [optional] The axes information for the data. For an embedded container, the axes container is inherited from its parent container if not present.

list of 1 or more [required] $x_0(x_n, ..., x_1)$ functional containers.

uncertainties: [optional] Contains a list of **uncertainty** containers.

17.2 Examples

Example 1: The following is an example for the use of the **regions** container representing a function $x_0(x_2, x_1)$.

```
|        <regions dimension="2">
|            <axes xlink:href="/templates/axes/energy">
|            <multiD_XYs index="0" dimension="2" interpolation="lin,lin">
|                <XYs value="1e-05" index="0">
|                    <values> ... </values></XYs>
|                    ...
|                <XYs value="2" index="11">
|                    <values> ... </values></XYs></multiD_XYs>
|            <multiD_XYs index="1" dimension="2" interpolation="log,log">
|                <regions value="2" index="0" dimension="1">
|                    <XYs index="0"> <values> ... </values></XYs>
|                    <XYs index="1" interpolation="log,log">
|                        <values> ... </values></XYs>
|                    <XYs index="2"><values> ... </values></XYs></regions>
|                ...
|                <XYs value="1e6" index="12">
|                    <values> ... </values></XYs></multiD_XYs>
|            <multiD_XYs index="2" dimension="2">
|                <XYs value="1e6" index="0"><values> ... </values></XYs>
|                    ...
|                <XYs value="2e7" index="9">
|                    <values> ... </values></XYs></multiD_XYs></regions>
```


In this example, 'lin,lin' is used for the incident energy axis (i.e., x_2) for $10^{-5} \leq x_2 \leq 2$ and $10^6 \leq x_2 \leq 2 \times 10^7$ (i.e., within the first and last **multiD_XYs** containers) and 'log,log' interpolation is used for $2 \leq x_2 \leq 10^6$ (i.e., within the middle **multiD_XYs**). The middle **multiD_XYs** container possesses a **regions** container. Furthermore, 'log,log' interpolation is used for the outgoing energy axis (i.e., x_1) inside the '/regions/multiD_XYs[@index=1]/regions[@index=0]/XYs[@index=1] section.

18 gridded container

An n-dimensional **gridded** container stores tabulated data representing a function of n independent variables (i.e., $x_0(x_n, ..., x_1)$) where the data are given on a grid for each independent axis. This container is composed of an n-dimensional array container and an **axes** element composed of $n+1$ **axis** elements.

The **style** attribute for independent **axis** i $(0 < i \leq n)$ shall be either *points*, *boundaries* or *parameters* and the **axis** shall contain a **values** element whose data have the following interpretation:

points: a list of ascending x_i values where the function is evaluated. Behavior of the dependent value (i.e., x_0) between consecutive x_i values is determined by the interpolation for that axis.

boundaries: a list of ascending x_i values where the function is constant between two consecutive values. No interpolation shall be supplied for this style as it is always 'flat'.

parameters: each value corresponds to a coordinate for a polynomial sequence. For example, the parameters may be Legendre orders (possibly starting with the $l > 0$). No interpolation shall be supplied for this style.

DISCUSSION POINT **Currently we are restricting parameters-style grids to only hold polynomial sequences. This makes it easier to associate the grid index with the correct parameter: index 0 = c_0, index 1 = c_1 and so on. However, do we need to be more general?**

No style shall be supplied for the dependent grid.

For an **axis** whose **values** element represents N values, the length N_i of dimension i in the array depends on the **style** attribute for that axis (i.e., dimension). For *points* and *parameters* styles, $N_i = N$. For the *boundaries* style, $N_i = N - 1$.

An example of a use for a gridded container in nuclear data is the covariance matrix, where each matrix element stores, for example, the covariance between a cross section at two different ranges of incident particle energies. The array of covariances by itself is insufficient to fully describe the data; the list of incident energy ranges is also needed.

DISCUSSION POINT For **parameters** style axis, are the values to be integers?

18.1 Specifications

Tag: gridded

Attributes: The list of allowed attributes are:

index: [Integer32, contingent] Defined by the parent element. Contingency: required when container is embedded in a higher dimensional container]

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

value: [determined by valueType attribute, contingent] Represents the x_{n+1} value for the container. Contingency: required when embedded in an (n+1)-dimensional function.

valueType: [UTF8Text, default="Float64"] Specifies the data type of value.

Body: The list of allowed elements are:

axes: [required] The axes information for the data.

array: [required] An **array** container representing the x_0 values.

uncertainties: [optional] Contains a list of **uncertainty** containers.

18.2 Examples

Example 1: In nuclear data a common use for gridded data is for storing covariance matrix data. If the rows and columns of the matrix correspond to the same ranges of incident energy, the second axis can link to the first (i.e., it can use the same *boundaries* defined in the first):

```
|      <gridded>
|        <axes>
|          <axis index="2" label="rows" unit="eV" length="3"> 1e-5 1e5 2e+7</axis>
|          <axis index="1" label="columns" xlink:href="../axis[@index='2']"/>
|          <axis index="0" label="covariances" unit=""/></axes>
|        <array shape="2,2" triangular="lower">
|          <values length="3">
|            0.23
|            0.09 0.18</values></array></gridded>
```

Example 2: Another example is the transfer matrix used in deterministic transport codes. In this example, the transfer matrix is stored as a 3-d array where the array dimensions correspond to (in order): incident energy group 'energy_in', outgoing energy group 'energy_out', and Legendre order l. The first two axis (corresponding to incident and outgoing energies) are **boundaries**-style axis, while the third independent axis (corresponding to l) is **parameters**-style. No interpolation is allowed along any of the independent axes in this example since 1) boundaries-style axis do not require interpolation (they implicitly use flat interpolation), and 2) there is no interpolation between Legendre orders.


```
|      <gridded>
|        <axes>
|          <axis index="3" label="energy_in" unit="MeV" length=88>
|                1.3068e-9 2.0908e-8 ... 20.0</axis>
|          <axis index="2" label="energy_out" xlink:href="../axis[@index='0']"/>
|          <axis index="1" label="Legendre order" unit="" style="parameters"
|                length="9">0 1 2 3 4 5 6 7 8</axis>
|          <axis index="0" label="matrix elements" unit="b"/></axes>
|        <array shape="87,87,9" compression="flattened">
|          <values label="flatIndices" length="...">0 87 ... 67338</values>
|          <values label="numberOfValues" length="...">6 5 ... 3</values>
|          <values length="...">
|            3.145 1.23e-4 -7.4e-6 8.9e-7 6.8e-9 9.2e-11
|            2.843 2.46e-4 -6.4e-6 3.2e-8 -7.8e-11
|            ...
|            1.2e-4 0.23 0.784</values></array></gridded>
```

18.3 Discussion

- We could eliminate the **style** attribute and instead require that **boundaries**- and **parameters**-style grids be given with flat interpolation. This solution has a downside if we want to explicitly list the very top boundary.

Instead of the following:

```
|      <gridded>
|        <axes>
|          <axis index="2" label="foo" unit="" style="boundaries">
|            <values>1 20</values></axis>
|          <axis index="1" label="bar" unit="" style="boundaries">
|            <values>0.5 8.5 15.8</values></axis>
|          <axis index="0" label="vals" unit=""/></axes>
|        <array shape="1,2"><values length="2">
|               7.6 8.4</values></array></gridded>
```

We would need the following:

```
|      <gridded>
|        <axes>
|          <axis index="0" label="foo" interpolation="flat"
|            style="points"><values>1 20</values></axis>
|          <axis index="1" label="bar" interpolation="flat"
|            style="points"><values>0.5 8.5 15.8</values></axis>
|          <axis index="2" label="vals"/></axes>
|        <array shape="2,3"><values length="6">
|               7.6 8.4 8.4
|               7.6 8.4 8.4</values></array></gridded>
```


The array shape grows by one in each dimension (from a 1×2 array to a 2×3 array) in order to explicitly give the value at each point. This adds redundancy and a possibility for discrepancies.

- According to the current specifications, interpolation is required when the **style** is **points**, and not allowed otherwise. Should we be more explicit and always require interpolation?
- Is **boundaries** the correct default axis style? It is a useful default since covariances and transfer matrices are given on boundaries rather than at individual points.
- The length of each axis can be calculated from the array shape + the axis **style**. Should we reduce redundancy by not storing the length explicitly?
- If the axis style is **parameters**, do we still need to supply a unit? We can do 'unit=""', but should we omit it altogether?

19 uncertainty container

A quantity (e.g., a length) has a mean value and an uncertainty (and a unit). Typically, the uncertainty is given as a single value representing a 1-σ uncertainty for a normal distribution (e.g., $12.3 \pm 1.2\ cm$). At times, two uncertainty values are given, one representing the uncertainty below the mean value

and another the uncertainty above the mean value (e.g., $12.3^{+1.4}_{-0.8}$ *cm*). Each uncertainty value may be associated with a different distribution. For example, the -0.8 distribution may be for a log-normal distribution while the 1.4 is for a normal distribution. Furthermore, an uncertainty value may be given as a percent (e.g., $12.3 \pm 12\%$ *cm*). The *General-Purpose Data Containers* support a similar structure for a function's uncertainty where the uncertainty is expressed as one or more functions. For example, for $f(x) \pm d(x)$ the function $d(x)$ represents the uncertainty of $f(x)$ and for $g(x)^{+u(x)}_{-l(x)}$ the functions $l(x)$ and $u(x)$ represent the lower and upper uncertainties, respectively. Each uncertainty function is called an uncertainty component for a function. That is, $g(x)$ has the two uncertainty components $l(x)$ and $u(x)$.

Each uncertainty component is stored in an **uncertainty** container. The **uncertainty** container lists the properties *type*, *pdf* and *relation* of the uncertainty and holds a functional container. The following are pre-defined property values. A project can define addition property values as needed.

type: Pre-defined values are:

- **variance:** The uncertainty component is a function representing the uncertainty $d(x)$ in $f(x) \pm d(x)$.
- **variance−:** The uncertainty component is a function representing the uncertainty $l(x)$ in the expression $f(x)^{+u(x)}_{-l(x)}$.
- **variance+:** The uncertainty component is a function representing the uncertainty $u(x)$ in the expression $f(x)^{+u(x)}_{-l(x)}$.
- **covariance:** The uncertainty component is the self-covariance function for $f(x)$.
- **confidence-interval:** The uncertainty component is a function $d(x)$ which represents the confidence interval for the numerical value given by the **pdf** property.

pdf: Pre-defined values are:

- **normal:** The uncertainty is normally distributed about the mean with standard deviation given by the component's function. For example, for $y(x) \pm d(x)$ the distribution for y at x is given as

$$\text{pdf}(y) = \frac{\exp\left(\frac{-(y-y(x))^2}{2d(x)^2}\right)}{\sqrt{2\pi}\, d(x)}$$

.

- **log-normal:** Like *normal* but with *pdf*(y) given as

$$\text{pdf}(y) = \frac{\exp\left(\frac{-(\log(y)-y(x))^2}{2d(x))^2}\right)}{y\,\sqrt{2\pi}\, d(x)}$$

.

- **A number in the range [0,1]:** For pre-defined **type**s, this is only valid for *confidence-interval*.

relation: The following will use the mean and its uncertainty $f(x) \pm d(x)$ to define the meanings of the various **relation** values. Pre-defined values are:

- **offset:** The uncertainty $u(x)$ is given as an offset from the mean (e.g., $\text{unc}(x) = d(x)$). This is the typical definition for an uncertainty.

absolute: The uncertainty is given as the *offset* plus the mean value (e.g., $\mathrm{unc}(x) = f(x)+d(x)$). As example, if $l(x)$ and $u(x)$ are the lower and upper *offset* uncertainties for $f(x)$ (e.g., $f(x)^{+u(x)}_{-l(x)}$), the *absolute* representation for $l(x)$ and $u(x)$ are $l_{\mathrm{abs}}(x) = f(x) - l(x)$ and $u_{\mathrm{abs}}(x) = f(x) + u(x)$ respectively. This is most useful for **type**="*confidence-interval*".

relative: The uncertainty is given as a ratio of the mean (e.g., $\mathrm{unc}(x) = d(x)/f(x)$).

percent: Like *relative* except uncertainty is given as 100 times the relative uncertainty (e.g., $\mathrm{unc}(x) = 100 \times d(x)/f(x)$).

The examples in the description of **type**, **pdf** and **relation** are for 1-dimensional functions. Multi-dimensional functional uncertainties are support just like for 1-dimensional functions. For example, the 3-dimensional function may be given as $x_0(x_3, x_2, x_1)\pm d(x_3, x_2, x_1)$. For a collated function, uncertainty data can be given in any of its sub-functions. For example, a collated $x_0(x_2, x_1)$ contains a list of $x_0(x_1)$ sub-functions and each of them can contain uncertainty data (See Example 4 in Section 20.2).

19.1 Specifications

Tag: uncertainty

Attributes: The list of allowed attributes are:

index: [Integer32, optional] Defined by the parent element.

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

type: [UTF8Text, default="variance"] Defined types are 'variance', 'variance+', 'variance−', 'covariance' and 'confidence-interval'. Projects can define additional types as needed.

pdf: [UTF8Text, default="normal"] Defined values are 'normal', 'log-normal' and when the **type** attribute is *confidenceinterval*, any number between 0 and 1. Projects can define additional pdfs as needed.

relation: [UTF8Text, default="offset"] Defined values are 'absolute', 'offset', 'relative' and 'percent'.

Body: A functional container with the proper dimension.

20 uncertainties container

Uncertainty data may be given for a functional container. All uncertainty data within a functional container reside in the **uncertainties** element. An **uncertainties** container only possesses a list of **uncertainty** containers, each representing a component of the total uncertainty.

20.1 Specifications

Tag: uncertainties

Attributes: The list of allowed attributes are:

index: [Integer32, optional] Defined by the parent element.

label: [UTF8Text, optional] Defined by the parent element.

style: [UTF8Text, optional] Defined by the parent element.

Body: The list of allowed elements are:

list of uncertainty containers.

20.2 Examples

Example 1: A tabulated $x_0(x_1)$ with uncertainty. The uncertainty is 10% of $f(x)$ at $x = 0$ and goes to 15% at $x = 3$. The function with its uncertainty band is shown in Figure 1.

```
|        <XYs>
|            <axes>
|                <axis index="1" label="energy" unit="eV"/>
|                <axis index="0" label="cross section" unit="b"/></axes>
|            <values> 0.0 1.000 0.1 1.316 0.2 1.447 0.3 1.547 0.4 1.632
|                     0.5 1.707 0.6 1.774 0.7 1.836 0.8 1.894 0.9 1.948
|                     1.0 2.000 1.1 2.048 1.2 2.095 1.3 2.140 1.4 2.183
|                     1.5 2.224 1.6 2.264 1.7 2.303 1.8 2.341 1.9 2.378
|                     2.0 2.414 2.1 2.449 2.2 2.483 2.3 2.516 2.4 2.549
|                     2.5 2.581 2.6 2.612 2.7 2.643 2.8 2.673 2.9 2.702
|                     3.0 2.732</values>
|            <uncertainties>
|                <uncertainty relation="relative">
|                    <XYs>
|                    <values> 0.0 0.1 3.0 0.15</values></XYs></uncertainty>
|                        </uncertainties></XYs>
```

Example 2: A tabulated $x_0(x_1)$ with a lower and upper uncertainty component (i.e., $f(x)^{+u(x)}_{-l(x)}$. Note, the tabulated function and its uncertainties have different x values. The function with its lower and upper uncertainty lines is shown in Figure 2.

```
|        <XYs>
|            <axes>
|                <axis index="1" label="energy" unit="eV"/>
|                <axis index="0" label="cross section" unit="b"/></axes>
|            <values> 0.1 2.9 0.2 2.2 0.4 1 0.6 1.6 1.2 3.8 2.1 6.2</values>
|            <uncertainties>
|                <uncertainty type="variance+" pdf="log-normal">
|                    <XYs>
|                        <values> 0.1 0.1 0.4 0.2 1 0.25 1.5 0.25 2.1 0.1</values>
|                            </XYs></uncertainty>
|                <uncertainty type="variance-" pdf="normal">
```

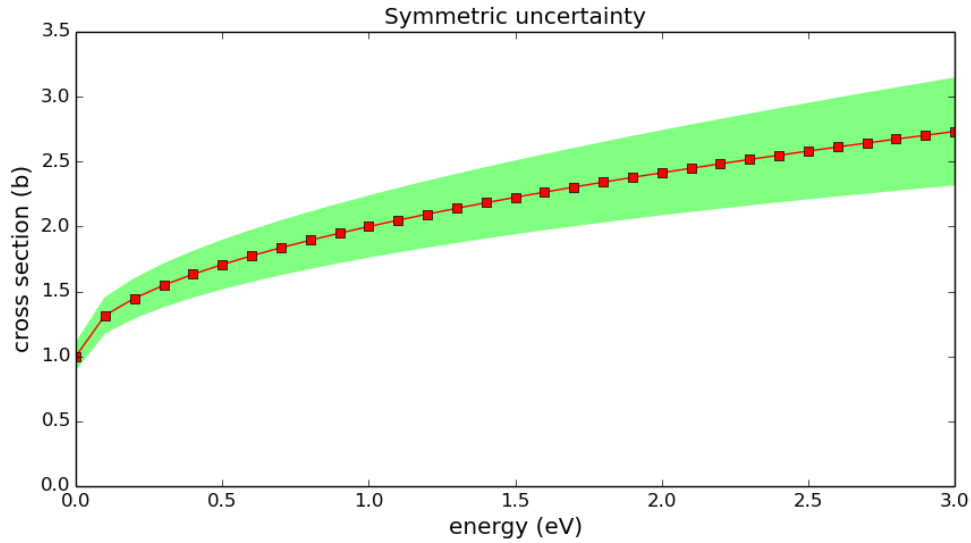

Figure 1: Plot of the function given in Example 1 of Section 20.2 and its uncertainty band.


```
|                    <XYs>
|                        <values> 0.1 0.6 0.4 0.4 2.1 0.1</values></XYs></uncertainty>
|                            </uncertainties></XYs>
```

Example 3: A tabulated $x_0(x_1)$ with a covariance. The covariance is given as a 2-dimensional gridded container.

```
|       <XYs>
|           <axes>
|               <axis index="1" label="energy" unit="eV"/>
|               <axis index="0" label="cross section" unit="b"/></axes>
|           <values> 0.1 2.9 0.2 2.2 0.4 1 0.6 1.6 1.2 3.8 2.1 6.2</values>
|           <uncertainties>
|               <uncertainty type="covariance">
|                   <gridded>
|                       <axes> ... </axes>
|                       <array shape="6,6" triangular="lower">
|                           ...
|                           </array></gridded></uncertainty></uncertainties></XYs>
```

Example 4: A tabulated $x_0(x_2, x_1)$ with with two $x_0(x_1)$ sub-functions, each with uncertainty data.

```
|       <multiD_XYs>
|           <axes>
|               <axis index="2" label="energy" unit="eV"/>
```

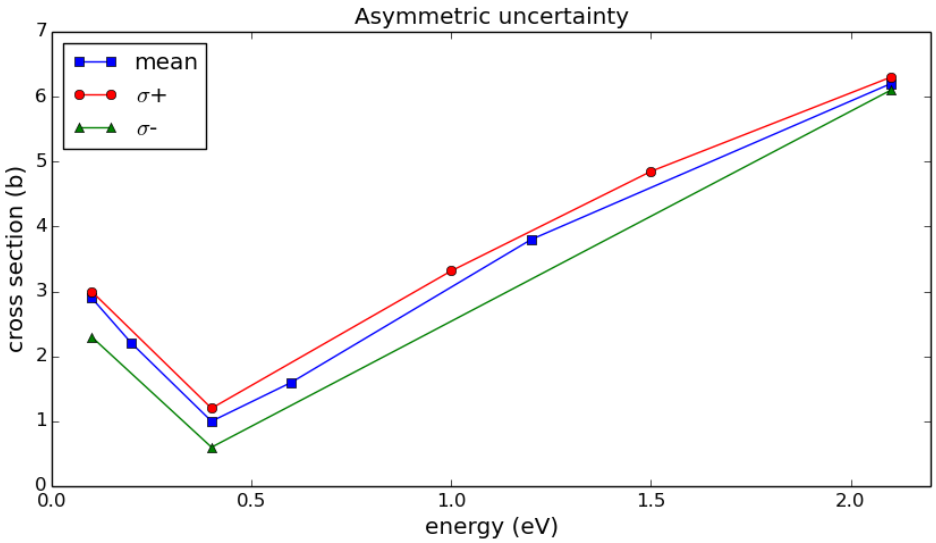

Figure 2: Plot of the function given in Example 2 of Section 20.2 and its lower and upper uncertainty lines.


```
|                <axis index="1" label="mu"/>
|                <axis index="0" label="P(mu|energy)"/></axes>
|            <XYs index="0" value="1e-5">
|                <values> ... </values>
|                <uncertainties> ... </uncertainties></XYs>
|            <XYs index="1" value="2e7">
|                <values> ... </values>
|                <uncertainties> ... </uncertainties></XYs></multiD_XYs>
```

Example 5: A tabulated $x_0(x_2, x_1)$ with an **uncertainties** element. Its sub-functions do not have **uncertainties** elements.

```
|        <multiD_XYs>
|            <axes>
|                <axis index="2" label="energy" unit="eV"/>
|                <axis index="1" label="mu"/>
|                <axis index="0" label="P(mu|energy)"/></axes>
|            <XYs index="0" value="1e-5">
|                <values> ... </values></XYs>
|            <XYs index="1" value="2e7">
|                <values> ... </values></XYs>
|            <uncertainties> ... </uncertainties>></multiD_XYs>
```

Example 1: An example for Legendre series with an equivalent of $C_l^{+b}_{-a}$ for each coefficient is

```
|      <series function="Legendre">
|        <values length="4">1 0.1 -0.02 1.3e-5</values></series>
|        <uncertainties>
|          <uncertainty type="variance+">
|             <series function="Legendre">
|                <values length="4">0.1 0.05 -0.002 3.3e-5</values>
|                    </series><uncertainty>
|          <uncertainty type="variance+">
|             <series function="Legendre">
|                <values length="4">0.1 0.03 -0.001 2.3e-5</values>
|                    </series><uncertainty></uncertainties>></series>
```

21 table container

The **table** container stores spreadsheet-style data as a list of M rows by N columns. The table container is like a 2-dimensional array of shape "M,N". However, there are several differences between a 2-dimensional array and a table:

- a table allows more data types; in particular, it allows for non-numeric data types
- a table allows for an empty cell
- a table allows for mixed data types
- sparse representations for tabular data are not supported.

For a table, the intersection of a row and a column is called a cell and stores a single datum. A datum can be a Integer32, Float64, string, empty (indicated by '<td/>'), valid XML text or any other type defined by a project. Each column has a header that consists of a label and meta-data. The label names the column while the meta-data qualify the data in the column. The meta-data are stored as key/value pairs, such as 'unit="eV"'.

Each cell can contain one of the following five data types (or other types defined by a project):

Integer32: Any valid Integer32 value. For ASCII, the data are stored without quotes (e.g., 3245 and not "3245").

Float64: Any valid Float64 values. For ASCII, the data are stored without quotes (e.g., 3.245e-4 and not "3.245e-4").

string: Any string enclosed by the xml start element '<td>' and its end element '</td>'. For example,
<td>abcd</td>
or
<td>Even spaces and quotes (i.e., " and ') are allowed.</td>.

DISCUSSION POINT **According to this definition, strings must always be enclosed by <td> and </td>. Should we allow an exception if data are stored in column-major**

order, and one column consists entirely of strings (as in the fission product yields example below)? Should we allow comma or space delimiters in that case?

empty: The string <**td**/> is used to denote a cell that is empty (i.e., has no value). Note that while <td/> is valid XML text, it is treated here as an independent type.

Boolean values: The Boolean values for true and false are '<true/>' and '<false/>' respectively. While these are XML text, they are treated as an independent type.

XML text: This is any valid XML text that starts with the <**td**> element and ends with the </**td**> element. Valid XML text must have matching start/end tags. An example of valid XML text is:

DISCUSSION POINT Below we show examples of valid xml text. Should they come up earlier in this document, perhaps in the 'Basic data types' section?

```
<td><area><width value="12.3" unit="cm"/>
    <height value="1.3"><unit>mm</unit></height></area></td>
```

An example of invalid XML text is:

```
<td><area><width value="12.3" unit="cm"/>
    <height value="1.3"><unit>mm</unit></area></td>
```


The latter example is missing the matching end tag for the "<height>" start tag. Another example of invalid XML text is:

```
<area><width value="12.3" unit="cm"/>
    <height value="1.3"><unit>mm</unit></height></area>
```

This example is the same as first example, but is missing the <td> start and </td> end tags.

Specifications

Tag: table

Attributes: The list of allowed attributes are:

columns: [Integer32, required] The number of columns of the table.

rows: [Integer32, required] The number of rows of the table.

storageOrder: [optional: default is 'row-major'] Allowed value is one of 'row-major' and 'column-major'.

Elements: The list of allowed elements are:

columnHeaders: Contains a list of N 'header' elements where N is the number of columns. Each header shall obey the following specifications:

header: One element for each column describing the data in each column.

Tag: 'header'

Attributes: The list of allowed attributes are:

label: [UTF8Text, required] A string representing the name of the column.

index: [Integer32, required] An integer value in the range 0 to (N-1) that represents the order of the column.

unit: [UTF8Text, optional: default is ""] The unit of the data in the column.

types: [UTF8Text, optional: default is "Float64"] A comma separated list of allowed data types for the column. The allowed values are "Integer32", "Float64", "string", "empty", "boolean", "XML" or project specific type. Note that in the types value, the "<td/>" type is given as "empty". Examples for type are 'types="XML"' or 'types="Float64,empty,string"'. If all types for a project are allowed, the string 'all' can be used instead of listing every type.

Elements: The list of allowed elements are:

extras: [optional] Allows for the storage of user specific attributes (i.e., meta-data). DISCUSSION POINT Do these extra attributes (like the 'L' and 'channelSpin' attributes in the example above) need to be stored in a separate element, or can they be stored directly in the 'header' element?

table: The data for the table, consisting of M × N cells.

Attributes: The list of allowed attributes are:

sep: [UTF8Text, optional: default is "whiteSpace"] Valid options are "whiteSpace", ",", "td", "tr" or "tc". The standard rules for "whiteSpace" and "comma" separators apply. The other separator options are described below.

Elements and/or Body: The layout of the table data depends on the values of the **storageOrder** and **sep** attributes.

sep='whiteSpace' or ',': In this case the table has no sub-elements, and the body of the table consists of a **sep**-separated list containing M × N values. The order of data depends on the storageOrder. This option can only be used if all data in the table are numeric types.

sep='td' In this case every cell in the table is explicitly contained inside '<td>' and '</td>' (or '<td/>' to indicate an empty cell). The order of data depends on the storageOrder. This option can handle multiple data types, but if only a few cells contain non-numeric data the following two options may be more concise.

storageOrder='row-major', sep='tr' In this case each row of the table is stored inside '<tr sep="...">' and '</tr>'. Each row can define its own separator, which may be 'whiteSpace' (the default), 'td' or ','. Any row containing non-numeric data must have 'sep="td"'.

storageOrder='column-major', sep='tc' In this case each column of the table is stored inside '<tc sep="...">' and '</tc>'. Each column can define its own separator, which may be 'whiteSpace' (the default), 'td' or ','. Any column containing non-numeric data must have 'sep="td"'. This storage is recommended whenever some columns contain only numeric data while others contain mixed data types, since it helps reduce the size of the numeric columns.

Examples

- In GND, resonance parameters are stored in a table. In the resolved region, each row is one resonance, and the columns store the energy and partial widths of the resonance.

  A sample table storing nuclear resonance parameters might look like the following. Note that the **order** attribute is equal to the default and could be omitted:

```
<table columns="4" rows="4" storageOrder="row-major">
  <columnHeaders>
    <header index="0" label="energy" types="Float64" units="eV"/>
    <header index="1" label="gamma + Cl36 width" types="Float64" units="eV">
      <extras L="0" channelSpin="0.0"/></header>
    <header index="2" label="n+Cl35 width", types="Float64" units="eV">
      <extras L="0" channelSpin="1.0"/></header>
    <header index="3" label="H1+S35 width" types="Float64" units="eV">
      <extras L="0" channelSpin="1.0"/></header></columnHeaders>
  <table>
      54932.0      0.36726    46.4424       0.0
      68236.16     0.39336    217.904      1e-05
      115098.0     0.739      4.30778       0.0
      182523.0     0.74515    1759.74       0.4
  </table></tag>
```

  In this example, each row corresponds to a single resonance, and each column corresponds to one parameter (such as the central energy, width, spin, etc.) of that resonance.

- Fission product yields can be stored in tables, with one column storing fission product names and other columns storing the yields (and possibly uncertainties) for various incident energies:

```
<table columns="3" rows="1262" storageOrder="column-major">
  <columnHeaders>
    <header index="0" label="product" types="string"/>
    <header index="1" label="yield_1" types="Float64">
      <extras energy_in="0.0253 eV"/></header>
    <header index="2" label="d(yield_1)" types="Float64">
      <extras energy_in="0.0253 eV"/></header>
    <header index="3" label="yield_2" types="Float64">
      <extras energy_in="5e5 eV"/></header>
    <header index="4" label="d(yield_2)" types="Float64">
      <extras energy_in="5e5 eV"/></header></columnHeaders>
  <table>
    <column><td>V66</td><td>V67</td><td>V68</td> ...
        <td>Hf171</td><td>Hf172</td></column>
    <column>2.05e-19 0 0 ... 0 0</column>
```

```
<column>1.312e-19 0 0 ... 0 0</column>
<column>4.485e-18 7.338e-19 9 ... 0 0</column>
<column>2.87e-18 4.696e-19 0 ... 0 0</column></table></tag>
```

- In EXFOR, experimental data and uncertainties are stored in tables. Each row typically corresponds to an incident energy, and columns contain information like cross section, ratio to standard, uncertainty, etc.

Discussion

- Do tables need a way to specify interpolation rules between either rows or between columns (or both)?

22 Additional discussion points

DISCUSSION POINT

- Should data containers allow additional attributes (beyond those listed in these specifications) defined by a project?
- Define a physical quantity

A Other meta-languages

The discussion and specifications have so far have used the XML meta-language and examples in it. This section illustrates how other meta-languages can be used to express general-purpose data containers.

The XML meta-language has the following high level concepts:

1. three fundamental components - element, attribute and body. Recall, an attribute is a key with a value.
2. an element's body can contain text and other elements interspersed.
3. multiple elements with the same tag can reside within the same parent element

If a meta-language contains these concepts, expressing general-purpose data containers in it should be easy. However, many meta-languages lack either the **attribute** in concept 1, concept 3 or both. For example, JSON does not support attributes, and JSON and HDF5 do not support concept 3. For these meta-languages, the following two more specification are added to the general-purpose data containers:

1. the element **attributes** is a reserved element for storing an element's attributes. This element contains an element for each attribute. Each contained element shall use the key for its tag name and the value for its body.
2. the name 'tag' is reserved for attribute and element names that shall contain the actual tag name if the meta-language does not support concept 3.

For example, the following XML:

```
<employees>
    <employee>
        <name first="Doc" last="Jones"/></employee>
    <employee>
        <name first="Grumpy" last="Smith"/></employee>
    <employee>
        <name first="Happy" last="Earp"/></employee></employees>
```

can be expressed in XML - although not recommend - as:

```
<employees>
    <employee1 tag="employee">
        <name>
            <attributes>
                <first>Doc</first>
                <last>Jones</last></attributes></name></employee1>
    <employee2 tag="employee">
        <name>
            <attributes>
                <first>Grumpy</first>
```

```
                <last>Smith</last></attributes></name></employee2>
        <employee3 tag="employee">
            <name>
                <attributes>
                    <first>Happy</first>
                    <last>Earp</last></attributes></name></employee3>
                                                        </employees>
```

Or, in JSON as:

```
{ "employees" : {
    "employee1" : {
        "tag" : "employee",
        "name" : {
            "attributes" : {
                "first" : "Doc"
                "last" : "Jones" } } },
    "employee2" : {
        "tag" : "employee",
        "name" : {
            "attributes" : {
                "first" : "Grumpy"
                "last" : "Smith" } } },
    "employee3" : {
        "tag" : "employee",
        "name" : {
            "attributes" : {
                "first" : "Happy"
                "last" : "Earp" } } } } }
```


B Python regular expression syntax

This section describes some of the Python regular expression syntax. A regular expression is a "sequence of characters that forms a search pattern". For this article, a regular expression representing a form is compared to a string and if they match, the string is properly formatted for that form. For the limited use of regular expression syntax in this article, it is sufficient to consider a regular expression built using 4 elements: a repetition character, an expression, the or-ing operator and escape sequences. Here a expression can be another regular expression.

An expression can be:

- a single character except a few special characters which need to be escaped. For this discussion, the important special characters are '.', '[', ']', '(', ')', '|', '?', '*' and '+'.
- any character between the '[' and ']'. For example, the expression representing any of the first 6 lower case English alphabet characters (LCEA) can be the string '[abcdef]' or '[defbca]' (i.e.,

order does not matter). This can be shortened using the '-' character to '[a-e]'. Independent of the number of characters between the '[' and ']', only one character in the string is compared to the list of characters between the '[' and ']'. For example, the regular expression 'c[aeiou]t' matches 'cat' and 'cot' but not 'coat' as the sub-expression '[aeiou]' is only compare to the second character.

- The period character '.' matches any character.
- The regular expression between the '(' and ')'. For example, the string '(ab?cd+)'.

For this article, we only need the following repetition characters and their expressions. If 'C' is an expression then the repetition expressions are:

'C?': 0 or 1 of the preceding 'C' expression. For example, 'a?' will match 0 or 1 'a' character.

'C+': 1 or more of the preceding 'C' expressions. For example, 'a+' will match 1 or more 'a' characters.

'C*': 0 or more of the preceding 'C' expressions. For example, 'a*' will match 0 or more 'a' characters.

'C{m}': m of the preceding 'C' expressions. For example, 'a{4}' will match 4 'a' characters.

'C{m,n}': m to n of the preceding 'C' expressions. For example, 'a{2,4}' will match 2, 3 or 4 'a' characters.

The or-ing character is the '|' character. For example, the regular expression 'a|b' will match either the string 'a' or 'b'. The or-ing character has the lowest precedence so the expression 'abc|def' will match either 'abc' or 'def'.

An escape sequence starts with the backslash character (i.e., '\') followed by a character designating a special mode. For this article, the only relevant escape sequence is '\.' which allows for the matching of the period character (i.e., '.').

A simple regular expression example to match any number of the form '#', '#.', '#.#' or '.#' where # is an integer number of arbitrary size (e.g., '324'). The first three are matched by the regular expression '[0-9]+\.?[0-9]*'. The last representation is matched by the regular expression '\.[0-9]+'. Combining these with the or-ing character yields the expression '[0-9]+\.?[0-9]*|\.[0-9]+' which matches all four number representations.

References

[1] OECD/NEA WPEC Subgroup 38 (SG38), *Beyond the ENDF format: A modern nuclear database structure*, `http://www.oecd-nea.org/science/wpec/sg38/`.

[2] Ed. by A. Trkov, M. Herman, D.A. Brown, ENDF/BVII.1 Manual, *ENDF-6 Formats Manual: Data Formats and Procedures for the Evaluated Nuclear Data Files ENDF/B-VI and ENDF/B-VII*, CSEWG Document ENDF-102, BNL Report BNL-90365-2009 Rev.2 (2011).

[3] M.B. Chadwick, P. Oblozinsky, M. Herman, *et al.*, *ENDF/B-VII.0: Next Generation Evaluated Nuclear Data Library for Nuclear Science and Technology*, Nucl. Data Sheets, 107, Pages 2931-3060 (2006).

[4] M.B. Chadwick, M. Herman, P. Oblozinsky, *et al.*, *ENDF/B-VII.1 Nuclear Data for Science and Technology: Cross Sections, Covariances, Fission Product Yields and Decay Data*, Nucl. Data Sheets, 112, Pages 2887-2996 (2011).

[5] The specifications for IEEE Std 754-2008, IEEE Standard for Floating-Point Arithmetic, can be found at `http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4610935`. Also see `http://en.wikipedia.org/wiki/IEEE_floating_point`.

[6] `http://www.w3schools.com/xml/xml_elements.asp`

[7] See `http://en.wikipedia.org/wiki/ASCII` for a description of the ASCII character set.

[8] UTF-8 is the internationally defined ISO/IEC 10646 standard (see `http://en.wikipedia.org/wiki/UTF-8`).

[9] See `http://en.wikipedia.org/wiki/English_alphabet` and `http://en.wikipedia.org/wiki/ISO_basic_Latin_alphabet`.

[10] *The JSON Data Interchange Format*, `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`

[11] See `http://www.w3.org/TR/REC-xml/`.

[12] See "The HDF5 Group" at `http://www.hdfgroup.org`.

[13] The latest C programming specifications are ISO/IEC 9899:2011.

[14] The latest FORTRAN programming specifications are ISO/IEC 1539-1:2010.

[15] Unless otherwise stated, all Python references and examples are for Python 2.7. See `https://www.python.org`.