Chapter 1

Introduction

This histogram package originated from the ARCS [1] reduction package, which can be used to reduce inelastic neutron scattering data to $S(Q, E)$.

Histograms are common in scientific and business computations. In experimental science, results of measurements are usually histograms. The purpose of the histogram package is to provide a python representation of histogram. One difficulty of implementing histogram is that a histogram usually contains a large chunk of data, and such data can be represented in various ways (c/fortran/c++). A goal here is to factor out that detail so that the histogram class is independent of array implementations.

Histogram itself is mostly a container of information, including axes, data, error bars, etc. Histogram package is mostly intended for providing this data structure, but it also provides a simple GUI interface to look at the data in the histogram.

For convenience, numerical operators like $+, -, \times, \div$, and simple math functions like *sum* and *average* should be provided for histogram class. Multiple-dimensional histogram is quite common in scientific computing, and should be supported. Slicing is an important operation that is very useful [2] and should be supported too. It is very important that error bars are propagated in any of these operations.

[1] A direct-geometry time-of-flight neutron spectrometry

[2] For example, high dimensional histogram could be reduced to lower dimension for better understanding using slicing.

Chapter 2

Usage of histogram (Use cases)

Programatically

We expect histogram to be used in basic data analysis procedures. Reduction is one such procedure that transform measured histograms to histograms that are more human-understandable. A typical case of reduction is to reduce a diffraction raw data, $I(pixel)$, to a diffraction pattern, $I(2\theta)$. In the procedure of reduction, various kind of operations will be done to histograms, including numerical operations, slicings, and directly accessing big data arrays in the histogram:

Numerical operations:

```
h *= 3.1
h += h1
h3 = h1 - h2
```

Slicing:

```
sh1 = h.getSubHist( Range(1.5, 3.5) )
sh2 = h.getSubHist( Range(1, 3.5), 10.0 )
```

Acessing data:

```
datastorage = h.data().zs().values.numbers # an NdArray instance
errsstorage = h.errors().zs().values.numbers # an NdArray instance
```

Interactively

We expect that users might want to manipulate histograms interactively and perform numerical operations and slicings. Users will be able to perform those operations in the python command line environment or the histogram GUI.

Users of histogram might also like to interactively view and investigate a small portion of a histogram. Plotting and zooming functionalities are needed. Slicing capability is useful too. Users will be able to perform those operations in the python command line environment or the histogram GUI.

Here is a list of operations that are useful:

- Load histogram from file
- View 1-D histogram
- View 2-D histogram
- Customize plot
- Change title
- Change axis labels
- Save image to a file
- Make slices of histograms

Chapter 3

Design

To further this discussion, we need to first clearly define what do we mean here by "histogram". The result of any measurement is actually a histogram, by which we mean we have data in some bins. For example, if we measure a spectrum with x-axis being time-of-flight, we will get an array of counts, while each element in that array represents the number of counts measured in a predefined time slot (bin). This array of counts can be approximated by

$$\frac{dI}{dx}(x)\Delta x \tag{3.1}$$

where $\frac{dI}{dx}$ is a density function and Δx is bin size. This observation forms the base of our design of histogram classes.

In a more mathematical form, we can describe a histogram as a mapping from an area (rectagular) in a phase space to a $\Re^2$:

$$\mathcal{D} \to \Re^2$$

This statement is not complete without following constraints:

1. in each direction (axis) of the phase space, the axis is discretized to **bins**;
2. the $\Re^2$ represents the 2-tuple of the data and the error bar;

It is apparent that two kinds of information are critical here:

1. the big multiple-dimensional array that keeps the data and the error bars. It maps integer indexes to floating numbers of data or error bar;
2. the axis that holds the information about bins. It maps bins to integer indexes.

Therefore, we need at least two data structures: NdArray and Axis:

- NdArray: map N-tuple of integer indexes to a floating number. Here N is the number of dimension.

- Axis: map bins of physical quantity to index.

Let us rethink this break-down a little bit more to see if it is reasonable. First off, NdArray is a data structure that is pretty fundamental. It is just a multiple-dimensional array. I don't think we can break it down more. We may want to add capabilities like numerical operators, iterators, slicing mechanisms to this data structure, because all of them are needed by histogram, and they certainly can be useful for other applications. The remained data structure, Axis, deserves more thoughts. Axis is one thing that brings physical meaning to histogram. How can we attach physical meaning to an axis? An axis is usually associated with a name and a unit to denote its physical meaning. For example, name ="Neutron Energy", unit="meV" should give us a pretty good idea of the meaning of an axis, while at the same time it perfectly describes a physical quantity. So we probably catch one very important concept here: "physical quantity".

There is another problem that we have not been able to think through carefully. Up to this point, we think of an axis as a mapping from bins to indexes. But if we think about Axis without the context of "histogram", then we find that the notion of "bins" are actually special for Histogram. In the most common scenario, we are dealing with ticks instead of bins for axis. How should we handle that?

Let us leave the second question here and first start thinking about "physical quantity".

3.1 Physical Quantity

What is a physical quantity? Things like t (time), x (position), V (electric potential) are physical quantities. It sounds a simple concept to us, but in programming we always want to have a more rigorous analysis.

Let us starts with a simple physics problem:

> Problem 1: A free falling ball starts falling at $t = 0s$ with initial speed $v = 0m/s$. We know its acceleration is $g = 9.8m/s^2$. Please calculate the distances this ball falls at $t = 1,\ ,3\ ,7$seconds.

Clearly, t is a physical quantity, time. An expression

$$t = 0s$$

tells us that the physical quantity t takes the value **0 seconds**. It sounds a good idea to define a class *PhysicalQuantity*. Aparently a physical quantity has a name, for example, "t".

A physical quantity can take a value. For physical quantity t, $0s$ is a valid value, while "hello" is not. So a *PhysicalQuantity* instance better has a method to verify whether a value is valid. Please note, however, **we don't want to put a value (0seconds) into the PhysicalQuantity (t) class**. The binding of value to physical quantity will happen later.

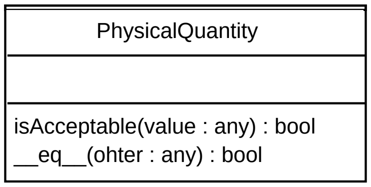

Figure 3.1: Initial draft of PhysicalQuantity class diagram

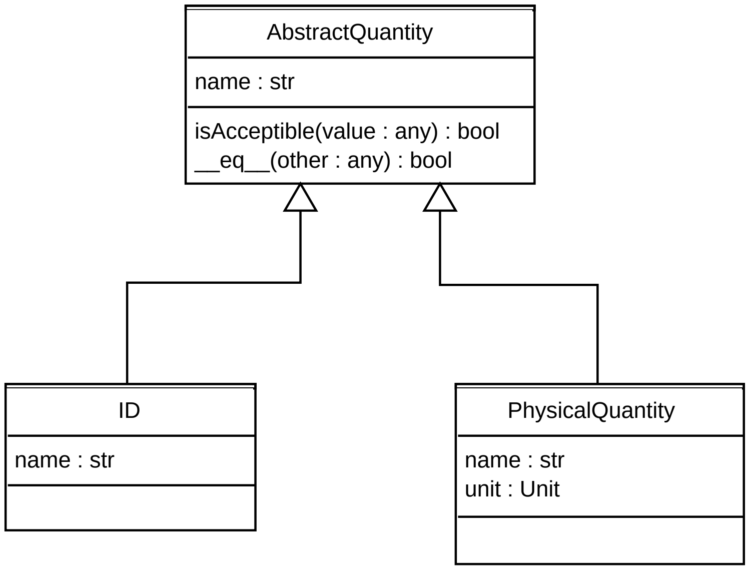

Figure 3.2: A more complete class diagram for Quantity classes

The UML class diagram for *PhysicalQuantity* is shown in Figure 3.1, in which we added a method to compare physical quantities (not values).

In the scope of our histogram package, there are quantities that are not physical. For example, a detector ID is not really a physical quantity. It would be a better design if we consider physical quantity in a broader sense. So we introduced AbstractQuantity, and made PhysicalQuantity a subclass. Figure 3.2 has a more complete picture.

3.2 Physical value containers

Now let us get back to Problem 1. The expression

$$t = 0s$$

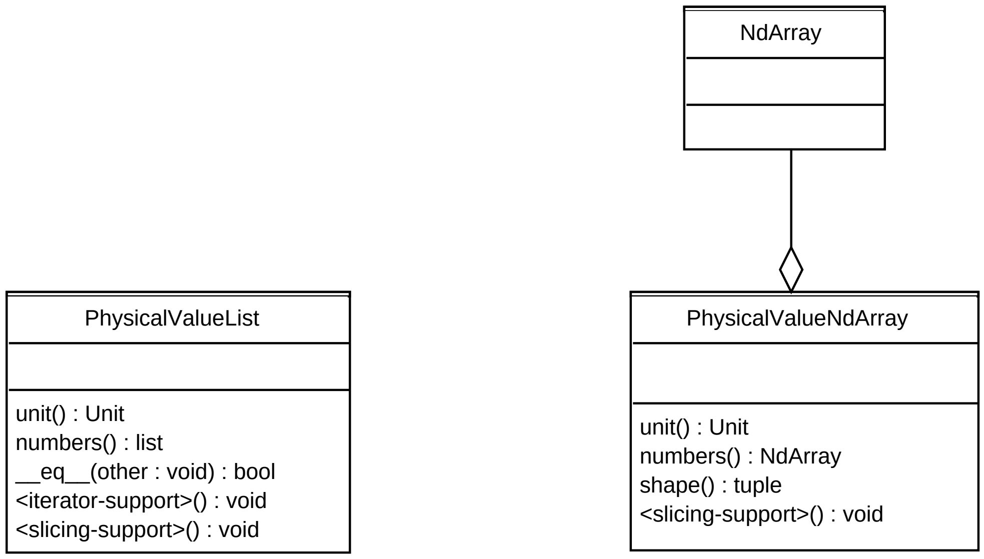

Figure 3.3: Container of values of physical quantity

means the physical quantity t takes the value $0s$. How do we do it programatically? In python (with pyre), we could say

```
>>> from pyre.units.SI import second
>>> t = 0 * second
```

which sounds pretty good (we will get back to this later...)

Sometimes, we might need a list of values for a quantity. For example in Problem 1, we have a list of values for t: $t = 1, , 3 , 7$seconds, for which we need to calculate traveling distances of the ball. We could represent such values as

```
>>> ts = [1*second, 3*second, 7*second]
```

but we could optimize that by saving the unit "second" separately. We introduced class *PhysicalValueList* for this purpose, and it should have the same behavior as a normal list. Similarly, we will need a multiple-dimensional array of physical values, and that is handled by class *PhysicalValueNdArray*. Those two classes are shown in Figure 3.3.

3.3 Binding of physical quantity and its value(s)

Let us stare at this expression again:

$$t = 0s \tag{3.2}$$

The python code

```
>>> from pyre.units.SI import second
>>> t = 0 * second
```

does not really form a full description of expression 3.2. It actually just represents a value, and does not establish a relationship between a quantity and a value.

Thus, we introduce several "binding" classes to bind a (physical) quantity with its value(s): *QuantitySingleValue*, *QuantityValueList*, *PhysicalQuantityValueNdArray*. Their class diagrams are in Figure 3.4.

A few things to note here:

- A *QuantitySingleValue* binds any quantity and a value. The quantity can be a physical quantiy or any other quantity we like. Same for *QuantiyValueList*, in which we bind any quantity to a list of its values.
- For physical quantity, it is better to bind a physical quantity with a *PhysicalValueList* instance (instead of a normal list of physical values) to form a *QuantityValueList* instance, in order to improve performance.
- *PhysicalQuantityValueNdArray* binds a physical quantity with a multiple-dimensional array of physical values. Please note that there is no *QuantityValueNdArray*, so an aribitrary quantity may not be binded to a multiple-dimensional array of its values. Only a physical quantity can be binded with a multiple-dimensional array of its values. This is because we are not yet supporting a generic NdArray that can take values of any quantity, although we could simply use nested python list.

3.4 Axis

Now we have many underlying data structures. Is it now natural to define axis?

An axis represents a quantity, be it a physical quantiy like energy, or a non-physical quantity like detector ID. An axis has ticks, which is a list of values of the corresponding physical quantity. So, an axis is a binding of physical quantity and its values. Now that we already have *QuantityValueList*, it is simple. An axis definitely can use a *QuantityValueList* instance. Figure 3.5 gives details of the base class AbstractDiscreteAxis and its subclasses.

A few more points:

- An axis should be able to map value to index, that is what the method *index* for.
- Some quantities are discrete by nature, including detector ID, pixel ID, and quantized physical quantities. The Axis class of such quantity is named *GenuineDiscreteAxis*.
- Physical quantities that are continuous by nature are represented by *DiscretizedAxis*. Differentiate this case from the *GenuineDiscreteAxis* will

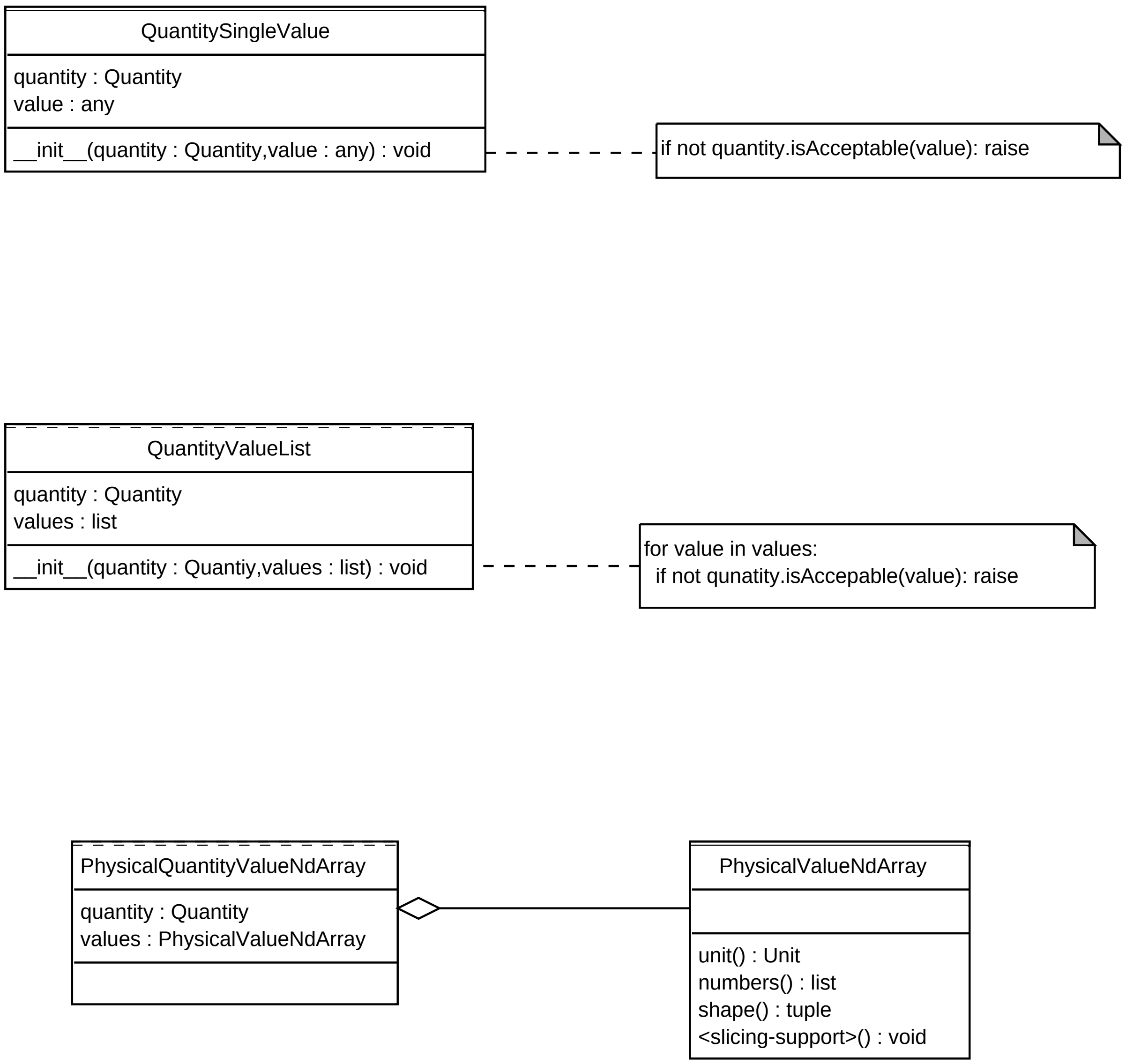
QuantitySingleValue
quantity : Quantity
value : any
__init__(quantity : Quantity,value : any) : void
if not quantity.isAcceptable(value): raise
QuantityValueList
quantity : Quantity
values : list
__init__(quantity : Quantiy,values : list) : void
for value in values:
if not qunatity.isAccepable(value): raise
PhysicalQuantityValueNdArray
quantity : Quantity
values : PhysicalValueNdArray
PhysicalValueNdArray
unit() : Unit
numbers() : list
shape() : tuple
<slicing-support>() : void

Figure 3.4: Binding of physical quantiy and its value(s)

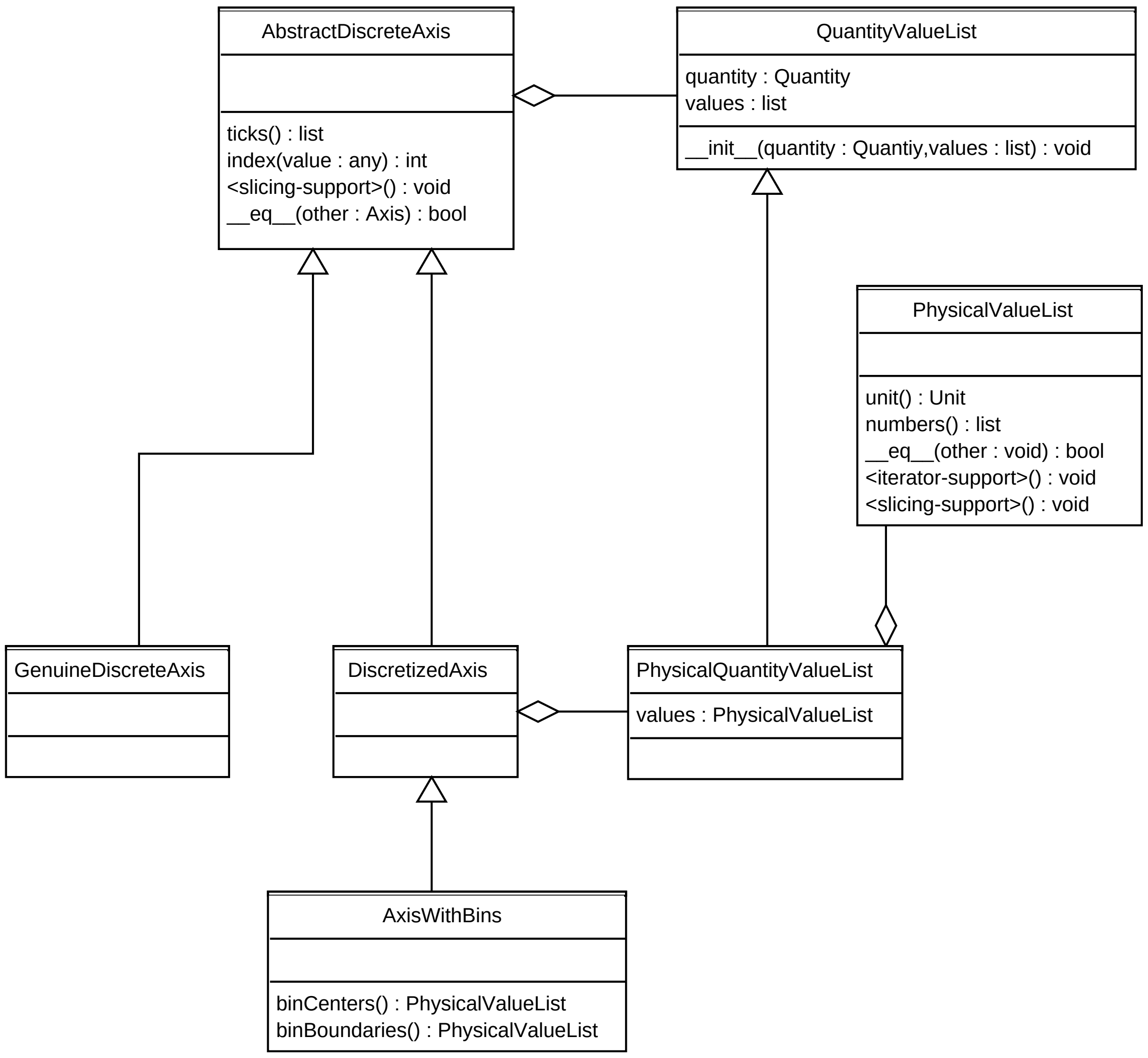
AbstractDiscreteAxis
ticks() : list
index(value : any) : int
<slicing-support>() : void
__eq__(other : Axis) : bool
QuantityValueList
quantity : Quantity
values : list
__init__(quantity : Quantiy,values : list) : void
PhysicalValueList
unit() : Unit
numbers() : list
__eq__(other : void) : bool
<iterator-support>() : void
<slicing-support>() : void
GenuineDiscreteAxis
DiscretizedAxis
PhysicalQuantityValueList
values : PhysicalValueList
AxisWithBins
binCenters() : PhysicalValueList
binBoundaries() : PhysicalValueList

Figure 3.5: UML diagram for Axis classes

make it difficult for us to make mistakes in, for example, doing interpolation.

- Axis could be defined as bins instead of ticks, this is what *AxisWithBins* for. So, for AxisWithBins, we are more concerned with bins. But we can difine ticks for AxisWithBins too. Here is the definition: for AxisWithBins, ticks in the usual meaning are centers of bins.

3.5 GridData

In summary, we have identified several data structures up to this point. We would like to build something useful on top of them. Let us see how far away we are from *Histogram*. Mathematically, we can denote a histogram as a collection of

$$I(x_i,\ y_j,\ z_k,\ ...)$$

and

$$\sigma_I^2(x_i,\ y_j,\ z_k,\ ...)$$

where $i, j, k, ...$ are indexes of values for all axes, $x_i, y_j, z_k, ...$ are values of quantities. Either of them takes a form that looks like

$$f_{ijk...} = \phi(x_i,\ y_j,\ z_k,\ ...)$$

Here, f; $x,\ y, z,\ ...$ all represent some quantities. It is a common situation in physics that we know values of all of the following

$$\{x_i\},\ \{y_j\},\ \{z_k\},\ ...$$
$$f_{ijk...}$$

and we want to be able to approximate the mapping $\phi:\ x, y, z, ... \rightarrow f$ from those data. We ask the class GridData (maybe we should find a better name) to do partially this work for us. Class diagram for GridData and its subclasses is presented in Figure 3.6.

Details of *GridData*:

1. Constructor takes two inputs:
   (a) a list of axes, and
   (b) the binding of physical quantity f with a multiple dimensional array of values of f, i.e. $f_{ijk...}$ (we actually call it z in *GridData* class).
2. Method "axes" return the list of axes.
3. Method "zs" returns the binding of physical quantity f and values $f_{ijk...}$.
4. Method getz evaluates f given coordinates. No interpolation is available.

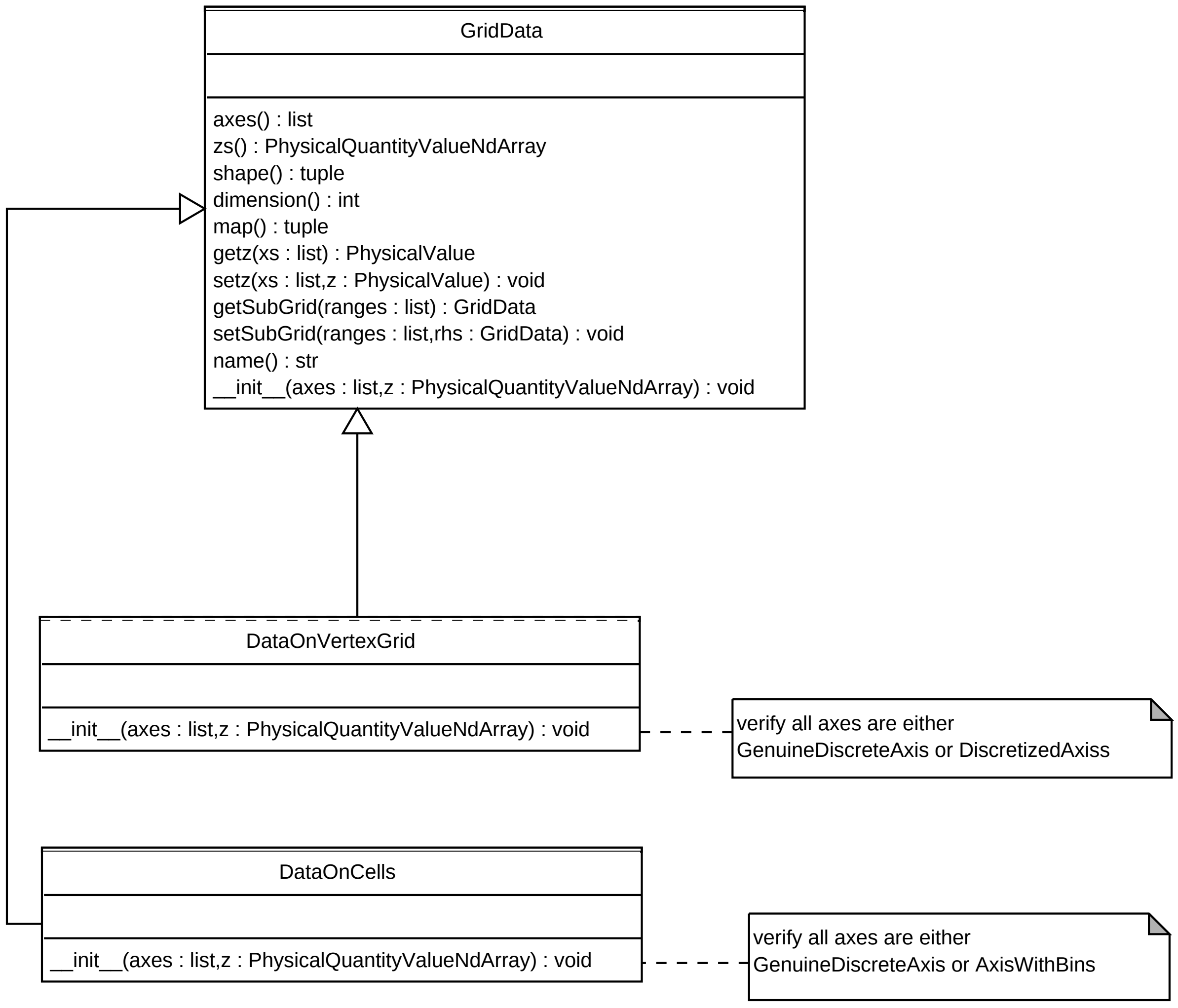
GridData
axes() : list
zs() : PhysicalQuantityValueNdArray
shape() : tuple
dimension() : int
map() : tuple
getz(xs : list) : PhysicalValue
setz(xs : list,z : PhysicalValue) : void
getSubGrid(ranges : list) : GridData
setSubGrid(ranges : list,rhs : GridData) : void
name() : str
__init__(axes : list,z : PhysicalQuantityValueNdArray) : void
DataOnVertexGrid
__init__(axes : list,z : PhysicalQuantityValueNdArray) : void
verify all axes are either GenuineDiscreteAxis or DiscretizedAxiss
DataOnCells
__init__(axes : list,z : PhysicalQuantityValueNdArray) : void
verify all axes are either GenuineDiscreteAxis or AxisWithBins

Figure 3.6: UML diagram for GridData classes

5. Method setz set f to a new value at given coordinates
6. Method getSubGrid returns a new GridData in the specified phase space.
7. Method setSubGrid set f in a specified phase space to be equal to the given GridData instance

Two subclasses of DataGrid is defined. A DataOnCells instance represent data defined on axes that have bins, or are discrete by nature. This is useful for the following two cases:

1. $f_{ijk...}$ represents an averaged value of some physical quantity in the cell bounded by the corresponding bins
2. $f_{ijk...}$ represents integrated value of some physical quantity in the cell bounded by the corresponding bins

A DataOnVertexGrid is useful for the case in which $f_{ijk...}$ represents the value of some physical quantity at the vertex.

3.6 Histogram

Now we finally have histogram. A histogram is simply a container of two DataOnCells instances. It will need some convenient functions, but generally it is pretty simple (Figure 3.7).

Things to note:

- Numerical operations on histogram will be implemented by performing numerical operations on PhysicalValueNdArray. Numerical operations on PhysicalValueNdArray are implemented by performing numerical operations on NdArray instances. Error propagations will be automatically performed for histograms.
- NdArray has an abstract interface. Histograms and PhysicalValueNdArray only use those interfaces. Solid implementations of NdArray must provide those methods defined in the abstract interface.
- Slicing on GridData are implemented by performing slicing on its "*zs*" and axes.
- Histogram can only represent a rectangular region in a phase space. This is usually good enough. Sometimes, however, we have to represent a dataset in a non-rectangular region in a phase space. In those cases, if the non-rectangular region can be easily divided to several rectangular regions, we can use HistCollection. HistCollection is a collection of histograms.

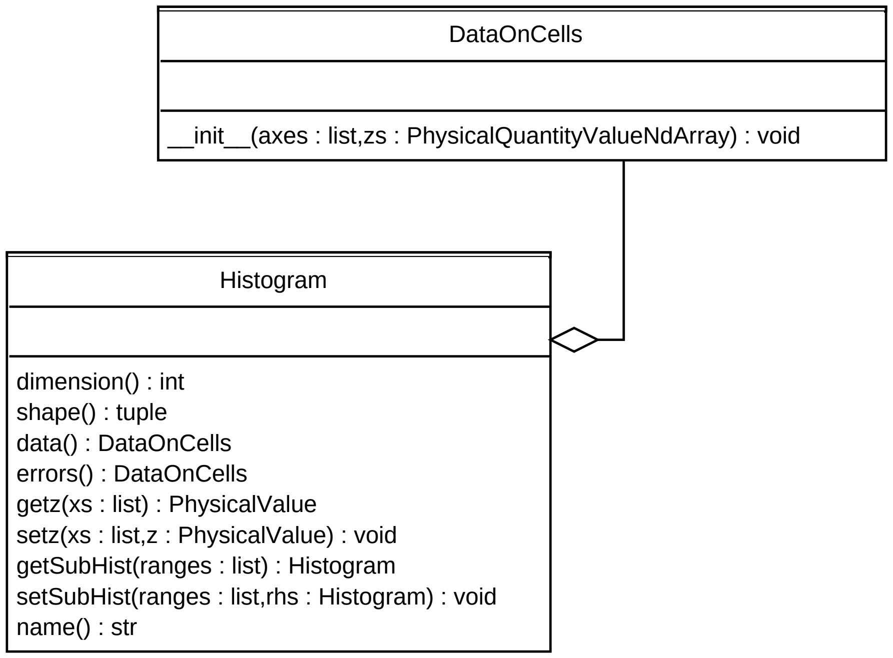
DataOnCells
__init__(axes : list,zs : PhysicalQuantityValueNdArray) : void
Histogram
dimension() : int
shape() : tuple
data() : DataOnCells
errors() : DataOnCells
getz(xs : list) : PhysicalValue
setz(xs : list,z : PhysicalValue) : void
getSubHist(ranges : list) : Histogram
setSubHist(ranges : list,rhs : Histogram) : void
name() : str

Figure 3.7: Histogram class