# PyHEP Tutorial

Alex Samuel

December 1, 2004

**Abstract**

This manual describes how to use the PyHEP Python tools for High Energy Physics. Rather than documenting all APIs exhaustively, it provides examples of common use cases, and describes APIs and object protocols in general terms. See the automatically-generated API documentation, doc comments, or the source code for detailed information about the APIs.

**Warning:** This document, and the software it describes, are incomplete. In particular, the APIs and protocols described here and implemented in the software may change, as the author gains experience with how the software is best used.

# CONTENTS

CHAPTER

**ONE**

# Introduction

This manual assumes you are familiar with modern versions of the Python language, at least version 2.2. PyHEP makes use of several newer Python features, such as true division and list comprehensions.

While you don't strictly need to be familiar with these features to use PyHEP, some of them will help you use PyHEP to its greatest advantage. In particular, PyHEP is designed to support a functional programming style, for which these more advanced Python features are very useful: the `map`, `reduce`, and `filter` built-in functions; iterators; `lambda` expressions; iterators; and list comprehensions.

This tutorial presents a tour of PyHEP's most important features, and shows how to perform some common tasks.

CHAPTER

**TWO**

# Histograms

A histogram is a tool for measuring an *N*-dimensional statistical distribution by summarizing samples drawn from it.

The histogram divides a rectangular region of the *N*-dimensional samples space into rectangular sub-regions, called *bins*. Each bin records the number of recorded samples whose values fall in that sub-region; when a sample is added into the histogram, or *accumulated*, the bin's value is incremented.

The histogram designates an *axis* for each dimension of the sample space, which specifies the range of coordinate values in that dimension which the histogram will accept. Coordinate values that fall below or above that range are called *underflows* and *overflows*, respectively.

Each axis also specifies the edges of the bins in the corresponding dimension, which forms rectangular bins. For *evenly-binned* axes, the positions of the bin edges are evenly spaced along the range of the axis. For *unevenly-binned* axes, the bin edges may be positioned arbitrarily; this allows smaller bins to be used in regions of the coordinate value where samples are more likely to fall more densely.

In some cases, samples are not drawn from the distribution which is being measured, but from a modified distribution (for example, when importance sampling). The modified distribution can be transformed back to the original distribution by assigning each sample a *weight*. The value of the corresponding bin will then be increased by the sample weight. When no weight is specified, unit weight is assumed.

If the samples are statistically independent, the contents of the bins will provide an approximation of the probability density function from which the samples are drawn. This approximation is subject to statistical uncertainty or *error*. The error is drawn from a Poisson distribution; for large statistics, a Gaussian distribution is a good approximation.

## 2.1 Introductory example

This example creates a one-dimensional histogram with five evenly-spaced bins between zero and ten, fills some sample values into it, and dumps the resulting contents.

Import the histogram module.

```
>>> import hep.hist
```

Create a histogram object.

```
>>> histogram = hep.hist.Histogram1D(5, (0, 10))
```

Show the histogram object we just created.

```
>>> print histogram
Histogram(EvenlyBinnedAxis(5, (0, 10)), bin_type=int, error_model='poisson')
```

Accumulate samples into the histogram.

```
>>> for sample in (0, 4, 5, 6, 7, 7, 9, 10, 11):
```

```
...   histogram << sample
...
```

Dump the histogram contents.

```
>>> hep.hist.dump(histogram)
Histogram, 1 dimensions
int bins, error model 'poisson'

axes: EvenlyBinnedAxis(5, (0, 10))

      axis 0             bin value / error
-------------------------------------------
[   None,      0)         0 +  1.148 -      0
[      0,      2)         1 +  1.360 -  1.000
[      2,      4)         0 +  1.148 -      0
[      4,      6)         2 +  1.519 -  2.000
[      6,      8)         3 +  1.724 -  2.143
[      8,     10)         1 +  1.360 -  1.000
[     10,   None)         2 +  1.519 -  2.000
```

## 2.2 Creating histograms

The class `hep.hist.Histogram` provides an arbitrary-dimensional histogram with binned axes. Optionally, a histogram can store errors (uncertainties) of the contents of each bin, or can compute these automatically from the bin contents.

To construct a `Histogram`, provide a specification of each axis, one for each dimension. You can specify an `Axis` object for each axis (described later), but the easiest way to specify an evenly-binned axis is with a tuple containing the number of bins, and a '(lo, hi)' pair specifying the low edge of the first bin and the high edge of the last bin. For example,

```
>>> import hep.hist
>>> one_d_histogram = hep.hist.Histogram((20, (0, 100)))
```

creates a one-dimensional histogram with 20 bins between values 0 and 100. Why all the parentheses? The outermost pair are for the function call; the next pair group parameters of the (single) axis; the innermost group the low and high boundaries of the axis range.

Similarly,

```
>>> two_d_histogram = hep.hist.Histogram((20, (-1.0, 1.0)), (100, (0.0, 10.0)))
```

creates a two-dimensional histograms with 20 bins on the first axis and 100 bins on the second axis.

To create a one-dimensional histogram you may use `Histogram1D`, which is equivalent except the axis's parameters are specified directly as arguments instead of wrapped in a sequence, so that one pair of parentheses may be omitted. Thus, the first example above may be written,

```
>>> one_d_histogram = hep.hist.Histogram1D(20, (0, 100))
```

You can access the sequence of axes for a histogram from its `axes` attribute. You'll see that the tuple arguments you specified have been converted into `EvenlyBinnedAxis` objects.

```
>>> two_d_histogram.axes
(EvenlyBinnedAxis(20, (-1.0, 1.0)), EvenlyBinnedAxis(100, (0.0, 10.0)))
```

For one-dimensional histograms, you may also use the `axis` attribute.

```
>>> one_d_histogram.axis
EvelyBinnedAxis(20, (0, 100))
```

For evenly-binned axes such as these, the `range` and `number_of_bins` attributes contain the axis parameters. The `dimensions` attribute contains the number of dimensions of the histogram. This is always equal to the length of the `axes` attribute.

```
>>> one_d_histogram.axis.number_of_bins
20
>>> one_d_histogram.axis.range
(0, 100)
>>> one_d_histogram.dimensions
1
```

If you specify additional keyword arguments when creating the histogram, they are added as attributes to the new histogram object. For example,

```
>>> histogram = hep.hist.Histogram(
...   (40, (0.0, 10.0), "momentum", "GeV/c"),
...   (32, (0, 32), "track hits"),
...   title="drift chamber tracks")
>>> histogram.title
'drift chamber tracks'
```

This creates a two-dimensional histogram of 40 bins of track energy between 0 and 10 GeV/c, and 32 integer bins counting track hits. Note that this histogram has $(40 + 2) \times (32 + 2) = 1428$ bins, including overflow and underflow bins on each axis. The histogram has an additional attribute `title` whose value is `"drift chamber tracks"`. You can, of course, set or modify additional attributes like `title` after creating the histogram.

### 2.2.1 Axis type

By specifying the range for each axis, you also implicitly specify the numeric type for values along the axis. (If the low and high values you specify for the range are of different types, a common type is obtained by Phython's standard `coerce` mechanism.) So, for instance, if you specify two integer values for the low and high boundaries for the axis, the axis will expect integer values, but if you specify one integer and one floating-point value, the axis will expect floating-point values.

You can find out the type of an axis by consulting its `type` attribute. For example,

```
>>> histogram = hep.hist.Histogram((20, (-1.0, 1.0)), (20, (-10, 10)))
>>> histogram.axes[0].type
<type 'float'>
>>> histogram.axes[1].type
<type 'int'>
```

If the type for an axis is `int` or `long` (because you specified integer or long values for the axis range), the difference between the high and low ends of the range must be an even multiple of the number of bins. If this is not desired, use `float` values for the axis. For example, this construction raises an exception:

```
>>> histogram = hep.hist.Histogram1D(8, (-50, 50))
[... stack trace ...]
ValueError: number of bins must be a divisor of axis range
```

because 50 - (-50) = 100 is not a multiple of 8. However, any of these will work:

```
>>> histogram = hep.hist.Histogram1D(10, (-50, 50))
>>> histogram = hep.hist.Histogram1D(8, (-60, 60))
>>> histogram = hep.hist.Histogram1D(8, (-50.0, 50.0))
```

### 2.2.2 Additional axis parameters

In addition to the number of bins and axis range, you may optionally provide additional elements to a tuple specifying an axis (or as the arguments to `Histogram1D`). The third argument is the string name of the axis quantity, and the fourth argument is a string describing the units of this quantity. For example,

```
h = hep.hist.Histogram1D(30, (0.05, 0.20), "pi0 mass", "GeV/$c^2$")
```

creates a histogram of 30 bins of pi0 mass between 0.05 and 0.2 GeV/$c^2$.

LaTeXmarkup may be used in the axis name and units; these will be rendered appropriately when plotting the histogram. See the section on LaTeXmarkup for details of the syntax.

### 2.2.3 Bin type and error model

A histogram stores for each bin the total number of samples, or the sum of weights of samples, that have fallen in this bin. The keyword argument *bin_type* specifies the numerical type used to store these totals. The histogram's `bin_type` attribute contains this type.

```
>>> histogram = hep.hist.Histogram1D(10, (-50, 50))
>>> histogram.bin_type
<type 'int'>
>>> histogram = hep.hist.Histogram1D(10, (-50, 50), bin_type=float)
>>> histogram.bin_type
<type 'float'>
```

By default, bin contents are stored as `int` values. *Therefore, if you plan to use non-integer weights, you should specify `bin_type=float` when creating the histogram; otherwise, weights will be truncated to integers when filling.*

A histogram can also esimate the statistical counting uncertainty on each bin. This error is represented by a 68.2% confidence interval around the bin value. The interval is represented by a pair of values specifying the sizes of the low and high "error bars", *e.g.* if the bin value is `20` and the errors are `(5.5, 4.5)`, the 68.2% confidence interval is the range (14.5, 24.5).

Several models are available are available that control how the errors are stored or computed:

- `"none"`: Each bin is assumed to have zero uncertainty.
- `"gaussian"`: The errors are computed assuming symmetrical Gaussian counting errors. The bin value is interpreted as a number of counts, and the low and high errors are both the square root of the bin content.
- `"poisson"`: The errors are computed assuming Poisson counting errors. The bin value is interpreted as a number of counts, and the low and high errors are chosen to cover 68.2% of the Poisson cumulative distribution around the bin value. The confidence interval is chosen to cover 34.1% on either side of the central value where possible. However, for a central value of zero, one, or two counts, this would produce a confidence interval with a lower edge below zero, so the lower edge is fixed at zero and the upper edge is chosen to capture 68.2%.
- `"symmetric"`: For each bin, the histogram stores a single value representing both the lower and upper errors. The error value may be specified explicitly for each bin with the `setError` method. When a sample is accumulated into the histogram, the sample weight is added in quadrature to the bin error.

- `"asymmetric"`: For each bin, the histogram stores two values representing the lower and upper errors. The two error values may be specified explicitly for each bin with the `setError` method. When a sample is accumulated into the histogram, the sample weight is added in quadrature to both bin errors.

Specify the error model for a histogram with the *error_model* keyword argument. The default is `"poisson"` if the bin type is `int` or `long`, or `"gaussian"` otherwise. If you will specify bin errors explicitly using the `setError` method, you must specify the `"symmetric"` or the `"asymmetric"` error model. The `error_model` attribute contains a histogram's error model.

For example,

```
>>> histogram = hep.hist.Histogram1D(10, (-50, 50))
>>> histogram.error_model
'poisson'
>>> histogram = hep.hist.Histogram1D(10, (-50, 50), bin_type=float)
>>> histogram.error_model
'gaussian'
>>> histogram = hep.hist.Histogram1D(10, (-50, 50), error_model="symmetric")
>>> histogram.error_model
'symmetric'
```

## 2.3 Other kinds of axes

The histogram axes we've used up to now have been evenly binned, *i.e.* all the bins on each axis are the same size. It is also possible to create a histogrma with unevenly-binned axes, using the `hep.hist.UnevenlyBinnedAxis` class.

Create an `UnevenlyBinnedAxis` instance, specifying a list of bin edges as its argument. You can then specify this axis when creating a histogram. Note that each histogram axis is independently specified, so that one, several, or all may be unevenly binned.

For example, to create a 2-D histogram with an unevenly-binned x axis and one evenly-binned y axis,

```
>>> import hep.hist.axis
>>> x_axis = hep.hist.UnevenlyBinnedAxis(
...     [0, 2, 4, 5, 6, 8, 10, 15, 20], name="count")
>>> histogram = hep.hist.Histogram(x_axis,
...     (10, (0., 5.), "time", "sec"))
```

## 2.4 Filling histograms

To "fill" a histogram is to accumulate samples from the sample distribution into it. A sample is represented by a sequence of coordinate values, where the length of the sequence is equal to the number of dimensions of the histogram. Each item in the sequence is the coordinate value along the corresponding axis of the histogram.

For each bin, the histogram keeps track the number of accumulated samples whose coordiantes fall within the bin. Optionally, samples may be specified a "weight"; in that case, the histogram keeps track of the sum of weights of samples. The numeric type used to store the number of samples or sum of weights for each bin is given by the `bin_type` attribute. The histogram also tracks the total number of accumulations (the "number of entries") that you have made.

The `accumulate` method accumulates an event into the histogram. Specify the coordinates of the sample, and optionally the event weight (which otherwise is taken to be unity). The left-shift operator `<<` is shorthand for `accumulate` with unit weight.

If any of the coordinate values passed to `accumulate` is `None`, the sample is not accumulated into the histogram and the number of entries is not changed.

For example, to accumulate a the sample whose coordinates are `x` and `y` into a two-dimensional histogram `histogram`,

```
>>> histogram.accumulate((x, y))
```

or

```
>>> histogram << (x, y)
```

To accumulate the same sample with weight `weight`,

```
>>> histogram.accumulate((x, y), weight)
```

For one-dimensional histograms, you may specify the sample coordinate by itself, instead of as a one-element sequence. So, to accumulate the sample with coordinate `x` into one-dimensional `histogram`, you may use any of these:

```
>>> histogram.accumulate((x, ))
>>> histogram.accumulate(x)
>>> histogram << (x, )
>>> histogram << x
```

Plotting histograms is described later in the plotting chapter. A quick way of displaying histogram contents in text format is with the `hep.hist.dump` function.

This script below creates a one-dimensional histogram with eleven integer bins between 2 and 12, inclusive, and fills the histogram with the result of simulating 1000 rolls of two dice.

```
import hep.hist
import random

histogram = hep.hist.Histogram1D(11, (2, 13), "rolls", bin_type=int)
for count in xrange(0, 1000):
    roll = random.randint(1, 6) + random.randint(1, 6)
    histogram << roll

hep.hist.dump(histogram)
```

## 2.5 Accessing histogram contents

You may specify a particular bin of a histogram with a “bin number”, which is a sequence of bin positions along successive axes. The length of the sequence is equal to the histogram’s number of dimensions. Each item is the index of the bin along the corresponding axis.

Along each axis, the coordinate in the bin number ranges between zero and one less than the number of bins on the axis. It may also take the values `"underflow"` and `"overflow"`, which denote the underflow and overflow bins, respectively

For example, consider this histogram:

```
>>> histogram = hep.hist.Histogram((20, (-1.0, 1.0)), (24, (0, 24)))
```

The corner bin numbers are for this histogram `(0, 0)`, `(9, 0)`, `(0, 23)`, and `(9, 23)`. The bin whose number is `(12, "underflow")` is for any samples whose first coordinate is between 0.2 and 0.3, and whose second coordinate is less than 0. The bin whose number is `("underflow", "overflow")` is for any sample whose first coordinate is less than -1 and whose second coordinate is greater than 24.

To get the bin number corresponding to a sample point, use the `map` method, passing the sample coordinates.

Just as with sample coordinates, for a one-dimensional histogram you may specify either the bin number as a one-element sequence, or simply the bin number along the (only) axis.

To obtain the contents of a bin, use the `getBin` method, passing the bin number. To obtain the 68.2% confidence unterval on a bin, use the `getError` method, which returns two values specifying how far the interval extends below and above the central value.

For example,

```
>>> histogram = hep.hist.Histogram1D(10, (0.0, 1.0), error_model="gaussian")
>>> histogram.accumulate(0.64, 17)
>>> histogram.map(0.64)
(6,)
>>> histogram.getBin((6, ))
17
>>> histogram.getError((6, ))
(4.1231056256176606, 4.1231056256176606)
```

In the `"gaussian"` error model, the errors on the bin are the square root of the bin contents, here, $\sqrt{17}$=4.123106. Since the histogram is one-dimensional, we just as easily could have used,

```
>>> histogram.getBin(6)
```

To set the contents of a bin, use the `setBin` method, specifying the new value as the second argument. To set the error estimate on a bin, use the `setError` method, specifying a `'(lo, hi)'` pair for the size of the confidence interval as the second argument. Note that you may only call the `setError` method of a histogram with `"asymmetric"` or `"symmetric"` error model, and in the latter case, the average of the `lo` and `hi` values you specify is stored as the single symmetric error estimate.

To obtain the range of coordinate values spanned by a single bin, use the `getBinRange` method, passing the bin number. The return value is a sequence, each of whose items is a `(lo, hi)` pair of coordinate values along on axis spanned by the bin. For example, to print the bin range and value for bins in a one-dimensional histogram,

```
>>> for bin in range(histogram.axis.number_of_bins):
...     (lo, hi), = histogram.getBinRange(bin)
...     bin_value = histogram.getBin(bin)
...     print "bin (%f,%f): %f" % (lo, hi, bin_value)
```

Note that the return value from `getBinRange` is a one-element sequence, since the histogram is one-dimensional. The single element is the pair `(lo, hi)` of the bin's range along the histogram's axis.

A histogram's `number_of_samples` attribute contains the number of times the `accumualte` method (or the `<<` operator) was invoked.

## 2.6 More histogram operations

The following functions are provided in `hep.hist`. Invoke `help(hep.hist.`*function*`)` for a description of a function's parameters.

- The function `scale` produces a copy of a histogram with its contents scaled by a constant factor.
- The function `integrate` returns returns the sum over all bins of a histogram. Specify `overflows=True` to include underflow and overflow bins in the integral.
- The function `normalize` returns a copy of a histogram, scaled such that its integral is unity (or another constant value specified as the optional second argument).

- The functions `add` and `divide` add or divide two histograms binwise, respectively. The histograms must have the same axis ranges and binning.
- The function `getMoment` computes the Nth moments of a histogram in each of its dimensions.
- The `mean` and `variance` functions compute those two statistics.
- The function `slice` produces an (N-1)-dimensional histogram by slicing or projecting out one dimension of an N-dimensional histogram.
- The function `rebin` produces a copy of a histogram with groups of adjacent bins combined together.
- The function `getRange` determines the range of bin values in a histogram. Optionally, the range can accommodate the error intervals on each bin.
- The function `dump` prints the contents of a histogram to standard output or another file.
- The function `project` accumulates samples simultaneously into several histograms from an array of sample events. This function is described later in the chapter on tables.

You may also add or divide two histogram with the ordinary addition and division operators, respectively, and you may scale a histogram with the ordinary multiplication operator. For example,

```
>>> combined_histogram = 3 * histogram1 + histogram2
```

To iterate over all bins in a histogram, use these functions. Each takes an optional second argument `overflow`; if true, underflow and underflow bins are included (by default false).

- `AxisIterator` takes an `Axis`. It returns an iterator that yields the bin numbers for that axis.
- `AxesIterator` takes a sequence of `Axis` objects, such as the value of a histogram's `axes` attribute. It return s an iterator that yields the bin numbers for the multidimensional bin space specified by the axes.
- `BinValueIterator` takes a histogram. It returns an iterator that yields the contents of each bin in the histogram.
- `BinErrorIterator` takes a histogram. It returns an iterator that yields the error estimate on each bin in the histogram.
- `BinIterator` takes a histogram. It returns an iterator that yields triplets `(bin_number, contents, error)` for each bin in the histogram.

For example, the code below show how you might find the largest bin value in a histogram, including underflow and overflow bins. (You could also use the `getRange` function.)

```
>>> max(hep.hist.BinValueIterator(histogram, overflow=True))
```

## 2.7 Scatter plots

A `hep.hist.Scatter` object collects sample points from a bivariate distribution. The samples are typically displayed as a "scatter plot".

The sample points are not binned in any way; rather, the two coordinates of each samples is stored. Unlike with a histogram, the memory usage of a `Scatter` object increases as additional samples are accumulated. However, the resulting object contains the full covariance information for the two coordinate variables, and can be plotted precisely.

To create a `Scatter`, specify information about the two axes. Each may be an instance of `hep.hist.Axis`, or more simply a sequence '(type, name, units)', where 'type' is the Python type of coordinate values along this axis, and 'name' and 'units' are strings describing the coordinate values (which may be omitted). If no axes are specified, they are taken to have 'float' coordinates

For example, this code creates a `Scatter` object.

```
>>> scatter = hep.hist.Scatter(
...     (float, "energy", "GeV"), (int, "number of hits"),
...     title="candidate tracks")
```

Use the `accumulate` method to add a sample to the `Scatter`. The argument must be a two-element sequence containing the two coordinates. The left-shift operator `<<` is a synonym for `accumulate`. The sample values are stored in a sequence attribute `points`.

For example,

```
>>> for track in tracks:
...     scatter << (track.energy, track.number_of_hits)
...
```

CHAPTER

**THREE**

# Tables

PyHEP provides a flat-format database facility, similar to “ntuples” in other HEP analysis systems.

A PyHEP table is similar to a table in a typical database system. A table is a sequence of rows, each of which has the same number of columns and values for each column of the same type. The number and types of columns constitutes the table’s schema.

PyHEP’s tables are subject to the following restrictions:

- A table’s schema is set when it is created, and cannot subsequently be changed. (Creating another table with a different schema using the same data is straightforward.)
- The size in bytes of each table row is a constant. This limits the types of table columns, but allows the implementation of fast seeks in a large table.
- Rows are appended to the table. An entire row must be appended at one time. Rows may not be inserted into the middle of a table, modified, or removed.

## 3.1 Table implementations

Several table implementations are provided. Other than for creating new tables or opening existing tables, the interfaces of these implementations are identical.

1. A extension-class implementation written in C++ in the `hep.table` module. The table resides in a disk file, and rows are loaded into memory as needed. The implementation is simple and efficient. Other programs may write and access these tables via a simple C++ interface.
2. An implementation which uses HBOOK ntuples (column-wise or row-wise) stored in HBOOK files. Not all features of HBOOK ntuples are supported.
3. An implementation which uses ROOT tress stored in ROOT files. Not all features of ROOT trees are supported.

The extension-class implementation in `hep.table` is used in this tutorial.

## 3.2 Creating and filling tables

A table schema is represented by an instance of `hep.table.Schema`. The schema collects together the definitions of the columns in the table. Each column is identified by a name, which is a string composed of letters, digits, and underscores.

A new instance of `Schema` has no columns. Add columns using the `addColumn` method, specifying the column name and type. For instance,

```
>>> import hep.table
>>> schema = hep.table.Schema()
>>> schema.addColumn("energy", "float64")
>>> schema.addColumn("momentum", "float64")
>>> schema.addColumn("hits", "int32")
```

The second argument to `addColumn` specifies the storage format used for values in that column. These column types are supported in `hep.table` (note that other table implementations may not support all of these):

- three signed integer types: `"int8"`, `"int16"`, and `"int32"`
- two floating-point types, `"float32"` and `"float64"`
- two floating-point complex types, `"complex64"` and `"complex128"`

More concisely, you may specify columns as keyword arguments, so the above schema may be constructed with,

```
>>> schema = hep.table.Schema(energy="float64", momentum="float64", hits="int32")
```

You can also load or save schemas in an XML file format using `hep.table.loadSchema` and `hep.table.saveSchema`. The schema above would be represented by the XML file

```
<?xml version="1.0" ?>
<schema>
 <column name="energy" type="float64"/>
 <column name="momentum" type="float64"/>
 <column name="hits" type="int32"/>
</schema>
```

Call the `create` function to create a new table. The parameters will vary for each implementation; for the `hep.table` implementation, the parameters are the file name for the new table, and the table's schema.

```
>>> table = hep.table.create("test.table", schema)
```

The return value is a *connection* to the newly-created table, which resides on disk.

To add a row to the table, call the table's `append` method. You may pass it a mapping argument (such as a dictionary) that associates column values with column names, and/or you may provide column values as keyword arguments. One way or another, you must specify values for all columns in the table. The return value of `append` is the index of the newly-appended row.

For instance, to add a row to the table created above,

```
>>> row = {
...   "energy": 2.7746,
...   "momentum": 1.8725,
...   "hits": 17,
>>> }
>>> table.append(row)
```

or equivalently,

```
>>> table.append(energy=2.7746, momentum=1.8725, hits=17)
```

### 3.2.1 Example: creating a table from a text file

The script below converts a table of values in a text file into a PyHEP table. The script assumes that the file contains floating-point values only, except that the first line of the text file contains headings that will be used as the names of

the columns in the table.

```
import hep.table

def textFileToTable(input_file_name, table_file_name):
    """Convert a text file containing tabular values to a table.

    'input_file_name' -- The file name of a text file containing a table
    of floating-point values.  The first line is assumed to contain the
    column names.  All additional lines are assumed to contain values
    for each column.

    'table_file_name' -- The file name of the table to create."""

    lines = iter(file(input_file_name))

    # Read the first line in the file, and split it into column names.
    heading_row = lines.next()
    column_names = heading_row.split()

    # Construct the schema from these column names.
    schema = hep.table.Schema()
    for column_name in column_names:
        schema.addColumn(column_name, "float32")
    # Create the table.
    table = hep.table.create(table_file_name, schema)

    # Scan over remaining lines in the file.
    for line in lines:
        # Split the line into values and convert them to numbers.
        values = map(float, line.split())
        # Make sure the line contains the right number of values.
        if len(values) != len(column_names):
            raise RuntimeError, "format error"
        # Construct a dictionary mapping column names to values.
        row = dict(zip(column_names, values))
        # Add the row to the table.
        table.append(row)

    del row, table


if __name__ == "__main__":
    # This file was invoked as a script.  Convert a text file as
    # specified on the command line.
    import sys
    textFileToTable(sys.argv[1], sys.argv[2])
```

Here is a table containing parameters for tracks measured in a detector. The first column is the track's energy; the other three are the x, y, and z components of momentum.

```
energy         p_x            p_y            p_z
1.180304e+00   -4.751910e-01  -7.889777e-01  -7.305207e-01
2.080963e+00   1.452333e+00   -4.622423e-01  1.412906e+00
1.743646e+00   1.421417e+00   -3.871017e-01  9.267482e-01
1.541235e+00   -1.059296e-01  -1.395949e+00  -6.358849e-01
1.617172e+00   1.401164e+00   -8.951803e-03  8.004645e-01
2.035846e+00   -4.445600e-02  1.965886e+00   5.165461e-01
1.202767e+00   -1.163040e+00  1.832017e-01   -2.219467e-01
```

```
2.567302e+00   -2.095504e+00  8.135615e-02   1.477200e+00
2.603921e+00   -9.120621e-01  -1.848932e+00  1.587083e+00
2.523255e+00   2.622356e-01   -1.991126e+00  1.523911e+00
2.233162e+00   -1.252426e+00  1.779797e+00   4.894894e-01
2.276338e+00   -1.044715e+00  5.443484e-01   1.944943e+00
2.208343e+00   1.969946e+00   -3.294630e-01  9.361521e-01
1.679185e+00   1.460185e+00   7.744986e-01   2.766050e-01
1.097246e+00   2.343778e-01   -1.065456e+00  -5.151948e-02
1.192835e+00   -9.905712e-01  5.303552e-01   3.862431e-01
3.180005e+00   -7.034522e-01  1.516622e+00   2.703014e+00
2.583735e+00   7.747896e-01   -1.303546e+00  2.089256e+00
1.953654e+00   1.482216e-01   1.877128e+00   5.099221e-01
2.217959e+00   5.964227e-01   1.990123e+00   7.693304e-01
2.307544e+00   -1.608457e+00  5.832562e-01   1.544757e+00
1.705716e+00   -4.596864e-01  1.543895e+00   5.508014e-01
2.343682e+00   1.468491e+00   -4.265070e-01  1.772938e+00
2.371860e+00   -1.794360e+00  1.057422e+00   1.129905e+00
1.370261e+00   4.528853e-01   1.226287e+00   -3.969452e-01
2.294908e+00   2.271728e-02   -2.027759e+00  1.069166e+00
2.875299e+00   1.594167e+00   1.849570e+00   1.514564e+00
2.917657e+00   1.757843e+00   5.340980e-01   2.264131e+00
2.766238e+00   -1.591074e+00  1.628313e+00   1.567799e+00
2.030203e+00   -1.662516e+00  -4.085008e-01  1.086154e+00
2.698105e+00   -5.955992e-01  -1.737854e+00  1.973255e+00
2.095454e+00   5.345570e-01   1.171157e+00   1.649971e+00
2.118398e+00   1.290140e+00   1.337788e+00   1.011093e+00
1.129607e+00   -6.164498e-01  7.138193e-01   -6.126172e-01
1.987723e+00   7.570301e-01   -1.814894e+00  2.700781e-01
2.018107e+00   -6.774516e-02  -1.672229e+00  1.122789e+00
```

If you save the script as ‘txt2table.py’ and the table as ‘tracks.table’, you would invoke this command to convert it to a table:

```
> python txt2table.py tracks.txt tracks.table
```

## 3.3 Using tables

To open an existing table, use the `open` function. The first argument is the table file name. The second argument is the mode in which to open the table, similar to the built-in `open` function: `"r"` (the default) for read-only, `"w"` for write.

For instance, to open the table we created in the last section,

```
>>> tracks = hep.table.open("tracks.table")
```

The table object is a sequence whose elements are the rows. Each row is has a row index, which is equal to its position in the table. Thus, the `len` function returns the number of rows in the table.

```
>>> number_of_tracks = len(tracks)
```

To access a row by its row index, use the normal sequence index notation. This returns a `Row` object, which is a mapping from column names to values.

```
>>> track = tracks[19]
>>> print track["energy"]
```

The table's `schema` attribute contains the table's schema; the schema's `column` attribute is a sequence of the columns in the schema (in unspecified order). You can examine these directly, or use the `dumpSchema` function to print out the schema.

```
>>> hep.table.dumpSchema(tracks.schema)
name             type
----------------------------
energy           float32
p_x              float32
p_y              float32
p_z              float32
```

## 3.4 Iterating over rows

For most HEP applications, the rows of a table represent independent measurements, and are processed sequentially. An *iterator* represents this sequential processing of rows. Using iterators instead of indexed looping constructs simplifies code, opens up powerful functional-programming methods, and enables automatic optimization of independent operations on rows.

Since a table satisfies the Python sequence protocol, you can produce an iterator over its elements (*i.e.* rows) with the `iter` function. The Python `for` construction does even this automatically. The simplest idiom for processing rows in a table sequentially looks like this:

```
>>> total_energy = 0
>>> for track in tracks:
...     total_energy += track["energy"]
...
>>> print total_energy
74.7496351004
```

Since `iter(tracks)` is an iterator rather than a sequence of all rows, each row is loaded from disk into memory only when needed in the loop, and is subsequently deleted. This is critical for scanning over large tables. (Note that after the loop completes, the last row remains loaded, until the variable *track* is deleted or goes out of scope. Also, the table object itself is deleted only after any variables that refer to it, as well as any variables that refer to any of its rows, are deleted.)

Table iterators may be used to iterate over a subset of rows in a sequence. Most obviously, you could implement this by using a conditional in the loop. For instance, to print the energy of each track with an energy greater than 2.5,

```
>>> for track in tracks:
...     if track["energy"] > 2.5:
...         print track["energy"]
...
```

While this is straightforward, it forces PyHEP to examine each row every time the program is run. By using the selection in the iterator, PyHEP can optimize the selection process, often significantly. The selection criterion can be any boolean-valued expression involving the values in the row. The selection expression is evaluated for each row, and if the result is true, the iterator yields that row; otherwise, that row is skipped. (This is semantically similar to the first argument of the built-in `filter` function.) The expression can be a string containing an ordinary Python expression using column names as if they were variable name (with certain limitations and special features), or may be specified in other ways. Expressions are discussed in greater detail later.

For instance, the same high-energy tracks can be obtained using the selection expression `"energy > 2.5"`. Notice that *energy* appears in this expression as if it were a variable defined when the expression is evaluated. The `select` method returns an iterator which yields only rows for which the selection is true.

```
>>> for track in tracks.select("energy > 2.5"):
...     print track["energy"]
...
```

Python's list comprehensions provide a handy method for collecting values from a table. For instance, to enumerate all values of *energy* above 2.5 instead of merely printing them,

```
>>> energies = [ track["energy"] for track in tracks.select("energy > 2.5") ]
```

Note here that `tracks.select("energy > 2.5")` returns an iterator object, so it may only be used (i.e. iterated over) once. However, if you really want a sequence of `Row` objects, you can use the `list` or `tuple` functions to expand an iterator into an actual sequence, as in

```
>>> high_energy_tracks = list(tracks.select("energy > 2.5"))
```

Such a sequence consumes more resources than an iterator. You can iterate over such a sequence repeatedly, or perform other sequence operations.

## 3.5 Projecting histograms from tables

If you have a sequence or iterator of values, you can use the built-in `map` function to accumulate them into a histogram. For instance, if *energies* is a sequence of energy values, you could fill them into a histogram using,

```
>>> energy_hist = hep.hist.Histogram1D(20, (0.0, 5.0))
>>> map(energy_hist.accumulate, energies)
```

Generally, though, you will want to project many histograms at one time from a table. Use the `hep.hist.project` function to project multiple histograms in a single scan over a table. Pass an iterable over the table rows to project—the table itself, or an iterator constructed with the `select` method—and a sequence of histogram objects. Each histogram should have an attribute `expression`, which is the expression whose value is accumulated into the histogram for each table row. The expression are similar to those used with the `select` method, except that their values should be numerical instead of boolean.

For example, this script projects three histograms – energy, transverse momentum, and invariant mass, from the table of tracks we constructed previously. Only high-energy tracks (those with energy above 2.5) are included. The dictionary containing the histograms is stored in a standard pickle file.

```
from   hep.hist import Histogram, Histogram1D, project
import hep.table
import pickle

histograms = {
    "energy":   Histogram1D(20, (0.0, 5.0), "energy", "GeV",
                expression="energy"),

    "pt":       Histogram1D(20, (1.0, 3.0), "p_T", "GeV/c",
                expression="hypot(p_x, p_y)"),

    "mass":     Histogram1D(20, (0.0, 0.2), "mass", "GeV/c^2",
                expression="sqrt(energy**2 - p_x**2 - p_y**2 - p_z**2)"),

    "px_vs_py": Histogram((8, (-2.0, 2.0), "p_x", "GeV/c"),
                          (8, (-2.0, 2.0), "p_y", "GeV/c"),
                expression="p_x, p_y"),
    }
```

```
tracks = hep.table.open("tracks.table")
project(tracks.select("energy > 2.5"), histograms.values())
pickle.dump(histograms, file("histograms.pickle", "w"))
```

Observe that the last histogram is two-dimensional, and its expression specifies the two coordinates of the sample using a comma expression.

With this scheme, you can determine later how the values in a histogram were computed, by checking its `expression` attribute.

## 3.6 Using expressions with tables

Expressions are described in more detail in a later chapter. This section presents techniques for optimizing expressions used with tables.

As described above, you can use Python expressions encoded in character strings with `select` method and `hep.hist.project` function. When such an expressions are evaluated, unbound symbols (*i.e.* variables) are bound to the value of the corresponding column.

Since a table row satisfies the Python mapping protocol, you can pass a table row directly to an expression's `evaluate` method. For example, this constructs an expression object to compute the transverse momentum of a track in the tracks table constructed above.

```
>>> import hep.expr
>>> p_t = hep.expr.asExpression("hypot(p_x, p_y)")
```

Its `evaluate` method computes transverse momentum for a track by binding *p_x* and *p_y* to the corresponding values of the table row.

```
>>> print p_t.evaluate(tracks[0])
0.92102783203
```

To find the largest transverse momentum in the track table,

```
>>> print max(map(p_t.evaluate, tracks))
2.44177757441
```

The chapter on expressions, below, describes how to compile an expression into a format for faster evaluation. When the expression is used with a table, you can produce an even faster version by compiling it with the table's `compile` method. This sets the symbol types correctly based on the table's schema, and applies additional optimizations specific to tables. For example,

```
>>> p_t = tracks.compile("hypot(p_x, p_y)")
>>> print p_t.evaluate(tracks[0])
0.92102783203
```

When you evaluate an expression on many rows of a large table, performance will be substantially better if you compile the expression first for that table. Note that an expression compiled for a table should only be used with that table.

### 3.6.1 Caching expressions in tables

You can also configure a table to cache the results of evaluating a boolean expression on the rows. The first time you evaluate the expression on a row, the row is loaded and the expression is evaluated as usual. On subsequent times, the table reuses the cached value of the expression, instead of reloading the row and re-evaluating the expression.

To instruct a table to cache the value of an expression, pass the expression to the table's `cache` method. Do not pass a compiled expression; pass the uncompiled form instead. You may pass a string or function here as well. Once you add the expression to the table's cache, the expression cache will automatically be used when you compile the expression or use it with `select` and other table functions.

For example, suppose your table file 'tracks.table' was extremely large, and you expect to select repeatedly all tracks with energy above 2.5. You could cache this selection expression like this:

```
>>> cut = "energy > 2.5"
>>> tracks.cache(cut)
```

```
>>> cut = tracks.compile(cut)
```

Optionally, compilar the expression and evaluate it on each row once, to fill the cache.

```
>>> cut = table.compile(cut)
>>> for track in tracks:
...   cut.evaluate(track)
...
```

### 3.6.2 Row types

As we have seen above, the object representing one row of a table satisfies Python's read-only mapping protocol: it maps the name of a column to the corresponding value in the row. While in many ways, they behave like ordinary Python dictionaries (for instance, they support the `keys` and `items` methods), they actually instances of the `hep.table.RowDict` class.

```
>>> track = tracks[0]
>>> print type(track)
<type 'RowDict'>
```

If you ever need an actual dictionary object containing the values in a row, Python will produce that for you:

```
>>> dict(track)
{'energy': 1.1803040504455566, 'p_z': -0.73052072525024414,
 'p_x': -0.47519099712371826, 'p_y': -0.78897768259048462}
```

For some applications, however, it is more convenient to use a different interface to access row data. You can specify another type to use for row objects by setting the table's `row_type` attribute (or with the *row_type* keyword argument to `hep.table.open`). The default value, as you have seen, is `hep.table.RowDict`.

PyHEP includes a second row implementation, `hep.table.RowObject`, which provides access to row values as object attributes instead of items in a map. The row has an attribute named for each column in the table, and the attribute's value is the corresponding value in the row.

For example,

```
>>> tracks.row_type = hep.table.RowObject
>>> track = tracks[0]
>>> print track.p_x, track.p_y, track.p_z
-0.475190997124 -0.78897768259 -0.73052072525
```

You may also derive a subclass from `RowObject` and use that as your table's row type. This is very handy for adding additional methods, get-set attributes, etc. to the row, for instance to compute derived values.

For example, you could create a `Track` class that provides the mass and scalar momentum as "attributes" that are computed dynamically from the row's contents.

```
>>> from hep.num import hypot
>>> from math import sqrt
>>> class Track(hep.table.RowObject):
...   momentum = property(lambda self: hypot(self.p_x, self.p_y, self.p_z))
...   mass = property(lambda self: sqrt(self.energy**2 - self.momentum**2))
...
```

Now set this class as the row type.

```
>>> tracks.row_type = Track
>>> track = tracks[0]
>>> print type(track)
<class '__main__.Track'>
```

You can now access the members of `Track`:

```
>>> print track.momentum
1.17556488438
>>> print track.mass
0.105663873224
```

Setting a table's row type to `RowObject` or a subclass will not break evaluation of compiled expressions on row objects. Expressions look for a `get` method, which is provided by both `RowDict` and `RowObject`.

Note that computing complicated derived values in this way is less efficient than using compiled expressions, as described above. However, you can create methods or get-set members that evaluate compiled expressions.

Only subclasses of `RowDict` and `RowObject` may be used as a table's `row_type`.

## 3.7 More table functions

The function `hep.table.project` is a generalization of `hep.hist.project`. Like the latter, it iterates over table rows and evaluates a set of expressions on each row. Instead of accumulating the expression values into histograms, passes them to arbitrary functions. Invoke `help(hep.table.project)` for usage information.

The `hep.table.Chain` class concatenates multiple tables into one. Simply pass the tables as arguments. Of course, any columns accessed in the chain must appear in all the included tables. For example, this code chains together all table files (files with the ".table" extension) in the current working directory.

```
>>> import os
>>> tables = [ hep.table.open(path)
...            for path in os.listdir(".")
...            if path.endswith(".table") ]
>>> chain = hep.table.Chain(*tables)
```

CHAPTER

**FOUR**

# Expressions

We have already seen expressions used as predicates in a table's `select` method, and as formulae for computing the values to accumulate into histograms. Generally, operations using expressions execute much faster than the same logic coded directly as Python, since expressions are compiled for the specific table into a special format from which they are evaluated very quickly.

## 4.1 Building and evaluating expressions

To parse an expression into a Python expression object representing its parse tree, use `hep.expr.asExpression`.

```
>>> import hep.expr
>>> ex = hep.expr.asExpression("p_x ** 2 + p_y ** 2")
```

The expression object's string representation looks similar to the original formula:

```
>>> print ex
(p_x ** 2) + (p_y ** 2)
```

The expression objects's `repr` shows its tree structure:

```
>>> print repr(ex)
Add(Power(Symbol('p_x', None), Constant(2)), Power(Symbol('p_y', None), Constant(2)))
```

To evaluate an expression, you must provide the values of all symbols. For the expression above, the symbols *p_x* and *p_y* must be specified. You may either call the expression object directly, passing symbol values as keyword arguments:

```
>>> ex(p_x=1, p_y=2)
5.0
```

or you may call its `evaluate` method with a map providing values of the symbols mentioned in the expression:

```
>>> ex.evaluate({"p_x": 1, "p_y": 2})
5.0
```

## 4.2 Compiling expressions

Expression objects, as well as expression coded directly in Python, execute quite a bit slower than similar mathematical expressions implemented in a compiled language like C or C++. However, PyHEP can often execute an expressions

much faster, by compiling it to an internal binary format and then evaluating it with an optimized, stack-based evaluator implemented in C++.

To compile an expression to this optimized form, use `hep.expr.compile`. It returns an expression object that can be used the same way as the original expression, but which executes faster.

```
>>> cex = hep.expr.compile(ex)
>>> cex(p_x=1, p_y=2)
5.0
```

There is no need to use `asExpression` before `compile`; just pass it the expression formula directly.

```
>>> cex = hep.expr.compile("p_x ** 2 + p_y ** 2")
>>> cex(p_x=1, p_y=2)
5.0
```

### 4.2.1 Expression types

PyHEP attempts to determine the numeric type of the result of an expression. To do this, it needs to know the types of the symbol values in the expression. For constants, this is obvious, but since Python is an untyped language, the expression compiler cannot automatically determine the types of symbolic names in an expression, and treats them as generic objects.

PyHEP expressions understand the types `int`, `float`, and `complex`. The value `None` indicates that the type is not known, so the value should be treated as a generic Python object.

For example, in these expressions, PyHEP can infer the type of the expression value entirely from the types of constants.

```
>>> print hep.expr.asExpression("10 + 12.5").type
<type 'float'>
>>> print hep.expr.asExpression("3 ** 4").type
<type 'int'>
```

However, a symbol's type is assumed to be generic, so the whole expression's type cannot be inferred.

```
>>> print hep.expr.asExpression("2 * c + 10").type
None
```

The expression compiler can do a much better job if it knows the numerical type of the expression's symbols. When you call `compile`, you can specify the types of symbols as keyword arguments. For example,

```
>>> cex = hep.expr.compile("2 * c + 10", c=int)
>>> print cex.type
<type 'int'>
```

If you provide a second non-keyword argument, this type is used as the default for all symbols in the expression.

```
>>> cex = hep.expr.compile("a**2 + b**2 + c**2", float)
>>> print cex.type
<type 'float'>
```

By specifying symbol types, you can construct compiled expressions that execute much faster than expressions with generic types.

You can also construct an uncompiled expression with types specified for symbols using the `hep.expr.setTypes`, `hep.expr.setTypesFrom`, and `hep.expr.setTypesFixed` functions.

## 4.3 Using expressions with tables

Since a table row object is a map from column names to values, you may specify a row object as the argument to `evaluate`. In the expression, the name of a column in the table is replaced by the corresponding value in that row. Using the 'tracks.table' table we created earlier,

```
>>> import hep.table
>>> tracks = hep.table.open("tracks.table")
>>> print ex.evaluate(tracks[0])
0.848292267373
```

This works well for small numbers of rows. However, if the expression is to be evaluated on a large number of rows in the same table, it should be compiled. Use the table's `compile` method, which sets the symbol types according to the table's schema and performs other necessary expansions before compiling the expression. The compiled expression object behaves just like the original expression object, except that it runs faster.

```
>>> cm = tracks.compile(ex)
>>> print cm.evaluate(tracks[0])
0.848292267373
```

You may also pass an expression as a string to `compile`. Functions provided by PyHEP which work with expressions, such as a table's `select` method or `hep.hist.project` will compile expressions automatically, where possible.

## 4.4 Expression syntax

An expression is specified using Python's ordinary expression syntax, with the following assumptions:

- Arbitrary names may be used in expressions as variable quantities, functions, etc. Other than the built-in names listed below, all names must be resolved when the expression is evaluated.
- The forward-slash operator for integers is true division, i.e. it produces a `float` quotient. Use the double forward-slash operator (e.g. "x // 3") to obtain the C-style truncated integer division.

The following names are recognized in expressions:

- Built-in Python constants `True`, `False`, and `None`.
- Built-in Python types `int`, `float`, `complex`, and `bool`.
- Built-in Python functions `abs`, `min`, and `max`.
- Constants from the `math` module: `e` and `pi`.
- Functions from the `math` module: `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `floor`, `log`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.
- From the `hep.lorentz` module, `Frame` and `lab`.

In addition, expressions may use these numerical convenience functions. (They are also availble in Python programs in the `hep.num` module.)

**`gaussian`**(*mu, sigma, x*)
: Returns the probability density at *x* from a gaussian PDF with mean *mu* and standard deviation *sigma*.

**`get_bit`**(*value, bit*)
: Returns true iff bit *bit* in *value* is set.

**hypot**(**terms*)
A generalization of `math.hypot` to arbitrary number of arguments. Returns the square root of the sum of the squares of its arguments.

**if_then**(*condition, value_if_true, value_if_false*)
Returns *value_if_true* if *condition* is true, *value_if_false* otherwise. Note that in a compiled expression (only), the second and third arguments are evaluated lazily, so that if *condition* is true, *value_if_false* is not evaluated, and otherwise *value_if_true* is not evaluated.

**in_range**(*min_value, value, max_value*)
Returns true if *min_value* is less than or equal to *value* and *value* is less than *max_value*.

**near**(*central_value, half_interval, value*)
Returns true if the absolute difference between *central_value* and *value* is less than *half_interval*.

## 4.5 Other ways to make expressions

You may specify a constant instead of a string when constructing an expression with `asExpression` or `compile`. The resulting expression simply returns the constant.

```
>>> ex = hep.expr.asExpression(15)
>>> print ex
15
>>> print ex.type
<type 'int'>
```

You may also specify a function that takes only positional arguments. The resulting expression calls this function, using symbols for the function arguments matching the parameter names in the function definition. If the function has a parameters `xyz`, the expression will evaluate the symbol `xyz` and call the function with this value. For example,

```
>>> def foo(x, a):
...   return x ** a
...
>>> ex = hep.expr.asExpression(foo)
>>> print ex
foo(x, a)
>>> ex(a=8, x=2)
256
```

The type of the value returned from such a function is not known.

```
>>> ex = hep.expr.asExpression(foo)
>>> print ex.type
None
```

You may specify it by attaching an attribute `type` to the function, containing the expected type of the function's return value.

```
>>> foo.type = float
>>> ex = hep.expr.asExpression(foo)
>>> print ex.type
<type 'float'>
```

You may also construct expressions programatically from the classes used by PyHEP to represent expressions internally. Each class represents a single operation. Invoke `help(hep.expr.classes)` for a list of these classes and their interfaces.

Here's an example to give you an idea.

```
>>> from hep.expr import Add, Divide, Constant, Symbol
>>> mean = Divide(Add(Symbol("a"), Symbol("b")), Constant(2))
>>> print mean
(a + b) / 2
>>> mean(a=16, b=20)
18.0
```

CHAPTER

**FIVE**

# Working with HBOOK files

The `hep.cernlib.hbook` module provides access to HBOOK files. HBOOK is a part of the CERNLIB library, and provides a structured file format containing histograms and ntuples. Note that not all HBOOK features are supported.

PyHEP's module `hep.cernlib.hbook` is linked statically against CERNLIB version 2001.

## 5.1 Creating, opening, and examining HBOOK files

The functions `hbook.create` and `hbook.open` create or open HBOOK files. The argument is the path to the file; `open` also takes an optional *update* argument, which is true if the file is to be opened in read-write mode for modification. The resulting object represents the top directory of the open HBOOK file.

An HBOOK directory object acts as a map, whose keys are entries in the directory. You can use the usual map methods to examine and access the contents of the directory. For example, the `keys` method returns a sequence of names of items in the directory.

This sample demonstrates how you might open an HBOOK file and extract a histogram.

```
import hep.cernlib.hbook
hbook_file = hep.cernlib.hbook.open("histograms.hbook")
energy_hist = hbook_file["ENERGY"]
```

To load a data item from an HBOOK directory, use the subscript operator (square brackets) or the `get` method. The type of the resulting object depends on the type of the data item in the HBOOK file. Use the same method to descend into subdirectories—the resulting object will be an HBOOK directory object corresponding to the subdirectory. You may similarly access an item by its RZ ID.

As a shortcut to loading subdirectory objects one by one, you may get an item in a subdirectory by specifying its path, with directory names separated by slashes. For instance, the code below shows three ways of accessing the same item.

```
mc_dir = hbook_file.get("MC")
batch2_dir = mc_dir.get("BATCH2")
eff_hist = batch2_dir.get("EFFICIENCY")

eff_hist = hbook_file["MC"]["BATCH2"]["EFFICIENCY"]

eff_hist = hbook_file["MC/BATCH2/EFFICIENCY"]
```

To find out information about an item in a directory without loading it, use the `getInfo` method, passing the item's name. The resulting `Info` object has these attributes, which describe the directory entry:

**`name`**
The RZ title of the entry.

**rz_id**
The RZ ID number of the entry.

**type**
A string describing the type of the object: one of `"1D histogram"`, `"2D histogram"`, `"table"`, or `"directory"`.

**is_directory**
True if the entry corresponds to a subdirectory.

This script print a full listing of the contents of an HBOOK file, including the title, RZ ID, and type of each entry.

```
from   hep.cernlib import hbook

def ls(hbook_dir, indent=0):
    # Loop over directory elements in 'path'.
    for name in hbook_dir.keys():
        info = hbook_dir.getInfo(name)
        print "%s%s%s%5d  %s" % (" " * indent, name,
                                 " " * (40 - indent - len(name)),
                                 info.rz_id, info.type)
        # If it's a directory, list its contents recursively.
        if info.is_directory:
            ls(hbook_dir[name], indent + 1)


if __name__ == "__main__":
    import sys
    file = hbook.open(sys.argv[1])
    ls(file)
```

To create a directory, use the `mkdir` method, specifying the full path to the new directory.

An HBOOK file is not closed until the file object is destroyed, i.e. all references to it are released. Especially when writing an HBOOK file, be careful to release all references to the file object.

## 5.2 Loading and saving histograms

To load a histogram from an HBOOK directory, simply obtain it by name using the subscript operator or `get`. This returns a Python object representing the histogram, which may be modified freely. Note that unlike HBOOK itself, where histograms are always stored in a global "PAWC" memory region, PyHEP constructs ordinary Python objects for histograms. There is no need to manage "PAWC" explicitly.

To save a histogram to an HBOOK file, simply assign it to the subdirectory in which you wish to store it, using subscript notation or the `set` method.

Not all histogram features supported in HBOOK are also supported in PyHEP, and visa versa. Therefore, if a histogram is saved to an HBOOK file and later loaded, it may differ in some of its characteristics. The basic histogram binning, and bin contents and errors (including overflow and underflow bins), are stored correctly, however. Note that PyHEP currently does not provide profile histograms.

The following script copies histograms from an HBOOK file to a new HBOOK file, preserving directory structure. Ntuples are ignored.

```
from   hep.cernlib import hbook

def copy(src_dir, dest_dir):
    # Loop over directory elements in 'path'.
    for name in src_dir.keys():
```

```
            info = src_dir.getInfo(name)
            if info.is_directory:
                # It's a directory.  Make the destination directory, and
                # call ourselves recursively to copy its contents.
                copy(src_dir[name], dest_dir.mkdir(name))
            elif info.type in ("1D histogram", "2D histogram"):
                # It's a histogram.  Load it, and save it to the destination.
                dest_dir[name] = src_dir[name]
            # Ignore other types of entries.


    if __name__ == "__main__":
        import sys
        copy(hbook.open(sys.argv[1]), hbook.create(sys.argv[2]))
```

## 5.3 Ntuples

An HBOOK ntuple is represented in PyHEP by a table. The table satisfies the same protocol as the default table implementation (see `hep.table`), except in the method to create or open tables. Note that because of HBOOK's limitations, certain table features are not supported.

To access an ntuple in an HBOOK file, use the file object's subscript operator or `get` method, just access the ntuple's name, just as you would for a histogram. Note that unlike a histogram, though, the table object is still connected to the ntuple in the HBOOK file. A new row appended to the table is incorporated immeidately into the ntuple. Also, the table object carries a reference to the HBOOK file, in its `file` attribute, so the HBOOK file is not closed as long as there is an outstanding table object for an ntuple in the file.

To create a new ntuple in an HBOOK file, use the `hep.cernlib.hbook.createTable` function. The arguments are the name of the ntuple, the HBOOK directory object in which to create the ntuple, and the schema (as for `hep.table.create`). You may use the optional *rz_id* argument to specify the ntuple's RZ ID.

By default, a column-wise ntuple is used for the table; to create a row-wise ntuple, pass a false value for the optional *column_wise* argument to `createTable`. When creating a column-wise ntuple, the schema may only contain columns of types `"int32"`, `"int64"`, `"float32"`, and `"float64"`. The schema for a row-wise ntuple may use only `"float32"` columns.

This program creates an HBOOK file containing a row-wise ntuple filled with random values. It then re-opens the file, creates a histogram from the values, and stores it in the file.

```
from   hep.bool import *
from   hep.cernlib import hbook
import hep.hist
import hep.table
from   random import random

# Construct a schema with three columns.
schema = hep.table.Schema()
schema.addColumn("a", "float32")
schema.addColumn("b", "float32")
schema.addColumn("c", "float32")

# Create a new HBOOK file.
hbook_file = hbook.create("test.hbook")
# Create a row-wise ntuple in it.
table = hbook.createTable("NTUPLE", hbook_file, schema, column_wise=0)
# Fill 100 random rows into the ntuple.
for i in xrange(0, 100):
    table.append(a=random(), b=random(), c=random())
```

```
# Release these to close the HBOOK file.
del table, hbook_file

# Reopen the HBOOK file.
hbook_file = hbook.open("test.hbook", update=True)
# Get the table.
table = hbook_file["NTUPLE"]
# Project a histogram of the sum of the three values in each row.
histogram = hep.hist.Histogram1D(30, 0.0, 3.0, expression="a + b + c")
hep.hist.project(table.rows, (histogram, ))
# Write the histogram to the HBOOK file.
hbook_file["HISTOGRAM"] = histogram
```

CHAPTER

**SIX**

# Working with Root files

The `hep.root` module provides access to Root files. Root set of libraries and programs for high energy physics analysis. Amongh other things, Root provides a file format for histograms, ntuples (which are called "trees" in Root), and other data objects. PyHEP provides partial data and file compatibility with Root.

PyHEP's module `hep.root` is built and linked against Root shared libraries which are distributed with PyHEP. The module does not depend on any other version of Root installed on your system.

The API and capabilities of `hep.root` are very similar to those of `hep.cernlib.hbook`. A program written for one can be used with the other with minimal modification, and it is easy to write functions, scripts, or programs that can work files from either format.

## 6.1 Opening and examining Root files

The functions `hep.root.create` and `hep.root.open` create or open Root files. The argument is the path to the file; `open` also takes an optional *update* argument, which is true if the file is to be opened in read-write mode for modification. The resulting object represents the top directory of the open Root file.

A Root directory object acts as a map, whose keys are entries in the directory. You can use the usual map methods to examine and access the contents of the directory. For example, the `keys` method returns a sequence of names of items in the directory.

This sample demonstrates how you might open a Root file and extract a histogram.

```
import hep.root
root_file = hep.root.open("histograms.root")
energy_hist = root_file["energy"]
```

To load a data item from a Root directory, use the subscript operator (square brackets) or the `get` method. The type of the resulting object depends on the class of the object instance in the Root file. Use the same method to descend into subdirectories—the resulting object will be a Root directory object corresponding to the subdirectory.

As a shortcut to loading subdirectory objects one by one, you may get an item in a subdirectory by specifying its path, with directory names separated by slashes. For instance, the code below shows three ways of accessing the same item.

```
mc_dir = root_file.get("MC")
batch2_dir = mc_file.get("batch2")
eff_hist = batch2_dir.get("efficiency")

eff_hist = root_file["MC"]["batch2"]["efficiency"]

eff_hist = root_file["MC/batch2/efficiency"]
```

To find out information about an item in a directory without loading it, use the `getInfo` method, passing the item's name. The resulting `Info` object has these attributes, which describe the directory entry:

**`name`**
The entry's name.

**`title`**
The entry's title.

**`class_name`**
The name of the Root class of which this entry is an instance.

**`type`**
A string describing the type of the object: one of `"1D histogram"`, `"2D histogram"`, `"3D histogram"`, `"table"` (for a tree), or `"directory"`.

**`is_directory`**
True if the entry corresponds to a subdirectory.

The following script lists the contents of a Root file specified on the command line. Subdirectory contents are displayed in tree format. Each line shows an entry's name and class name.

```
import hep.root

def ls(root_dir, indent=0):
    # Loop over directory elements in 'path'.
    for name in root_dir.keys():
        info = root_dir.getInfo(name)
        label = "%s (%s)" % (name, info.title)
        print (" " * indent) + label \
              + (" " * (64 - indent - len(label))) + info.class_name
        # If it's a directory, list its contents recursively.
        if info.is_directory:
            ls(root_dir[name], indent + 1)


if __name__ == "__main__":
    import sys
    file = hep.root.open(sys.argv[1])
    ls(file)
```

To create a directory, use the `mkdir` method, specifying the full path to the new directory.

Note that a Root file is not closed until the file object is destroyed, i.e. all references to it are released. Especially when writing a Root file, be careful to release all references to the file object.

## 6.2 Loading and saving histograms

To load a histogram from a Root directory, simply obtain it by name using the subscript operator or `get`. This returns a Python object representing the histogram, which may be modified freely. To save a histogram to a Root file, simply assign it to the subdirectory in which you wish to store it, using subscript notation or the `set` method.

Not all histogram features supported in Root are also supported in PyHEP, and visa versa. Therefore, if a histogram is saved to a Root file and later loaded, it may differ in some of its characteristics. For instance, any additional attributes added to the histogram will be lost. The basic histogram binning, and bin contents and errors (including overflow and underflow bins), are stored correctly, however.

## 6.3 Trees

An Root tree is represented in PyHEP by a table. The table satisfies the same protocol as the default table implementation (see `hep.table`), except in the method to create or open tables. Note that because of Root's limitations, certain table features are not supported.

To open a tree in a Root file as a table, use the file object's subscript operator or `get` method, just as you would for a histogram. Note that unlike a histogram returned from `load`, though, the table object that `load` returns is still connected to the tree in the Root file. A new row appended to the table is incorporated in the tree. Also, the table object carries a reference to the Root file, in its `file` attribute, so the Root file is not closed as long as there is an outstanding table object for a tree in the file.

To create a new tree in a Root file, use the `hep.root.createTable` function. The arguments are the name of the new tree, the Root directory in which to create it, and the schema (as for `hep.table.create`). You may specify a title for the table with the *title* argument.

This program creates a Root file containing a row-wise ntuple filled with random values. It then re-opens the file, creates a histogram from the values, and stores it in the file.

```
from   hep.bool import *
import hep.root
import hep.hist
import hep.table
from   random import random

# Construct a schema with three columns.
schema = hep.table.Schema()
schema.addColumn("a", "float32")
schema.addColumn("b", "float32")
schema.addColumn("c", "float32")

# Create a new Root file.
root_file = hep.root.create("test.root")
# Create a tree in it.
table = hep.root.createTable("tree", root_file, schema, title="PyHEP test")
# Fill 100 random rows into the ntuple.
for i in xrange(0, 100):
    table.append(a=random(), b=random(), c=random())
# Release these to close the Root file.
del table, root_file

# Reopen the Root file.
root_file = hep.root.open("test.root", update=True)
# Get the table.
table = root_file["tree"]
# Project a histogram of the sum of the three values in each row.
histogram = hep.hist.Histogram1D(30, 0.0, 3.0, expression="a + b + c")
hep.hist.project(table.rows, (histogram, ))
# Write the histogram to the Root file.
root_file["histogram"] = histogram
```

CHAPTER

**SEVEN**

# Drawing and Plotting

PyHEP includes a drawing layer, designed to provide device-independent output for simple two-dimensional figures and line drawings. Routines for producing plots are based on this. Currently, the drawing layer supports on-screen output using the GTK library, and PostScript output to a file. The drawing layer is in the module `hep.draw`.

In PyHEP, a drawing surface is represented by a `Draw` object. The setup procedure for a draw object depends on what kind of output it will create (in a window or to a file, the whole page or a subsection of it, etc.) but all `Draw` objects satisfy the same protocol.

In the standard coordinate system used by `hep.draw`, the positive X axis extends rightward and the positive Y axis extends upward; the origin is in the lower-left corner of the drawing region. The default units of measurement are meters, regardless of the output device. The constants `hep.draw.pt` and `hep.draw.inch` are conversion factors from standard points (1/72") and inches, respectively, to meters.

## 7.1 Creating `Draw` objects

The procedure to create an initial `Draw` object depends on the output device.

- The `hep.draw.x11win` module implements a drawing surface in a pop-up X window. Note that only X servers that provide TrueColor visuals are supported. To create a drawing window, instantiate `x11.Window`, passing the window width and height (in meters) as arguments.
- The `hep.draw.gtkwin` module implements a drawing surface in a pop-up window using the GTK toolkit. You must have the GTK2 bindings for Python installed to use this module. To create a drawing window, instantiate `gtkwin.Window`, passing the window width and height (in meters) as arguments.
- The `hep.draw.postscript` module provides drawing to a PostScript file. Two variants exist: the `postscript.PSFile` class draws to a multi-page ADSC PostScript file, and the `postscript.EPSFile` class draws to a file containing an encapsulated PostScript figure. When instantiating either, pass the path to the output file and the width and height of the output as arguments.

  For `postscript.PSFile`, the width and height may be omitted; the default is standard US letter page size.

Some `Draw` objects have a `clear` method. For GTK window objects, this clears the window. For PostScript file objects, this advances output to the next page.

In addition, you create a new `Draw` object from another `Draw` object when you apply a coordinate transformation. Call a `Draw` object's `transform` method, specifying the arguments below; the return value is another `Draw` object using the transformed coordinates. You may still draw using the original object.

- The *range* argument to `transform` is a four-element sequence containing the range along the X and Y axes of the new, transformed `Draw` object, as `(x0, y0, x1, y1)`.

- The *origin* argument is a two-component sequence indicating the position in the old, untransformed `Draw` object at which to position `(x0, y0)` in the new, transformed `Draw` object. The default value for *origin* `(0, 0)`.
- The *scale* argument is a two-component sequence indicating the scale factor of the new, transformed `Draw` object relative to to the old, untransformed one. The default value for *scale* is `(1, 1)`.

This example demonstrates how to create a PostScript `Draw` object. The output is a PostScript file with US letter page size, but drawing is restricted to a four-inch square centered on the page. The coordinates for *ps_draw* range from zero to one along both axes.

```
import hep.draw.postscript
from   hep.draw.postscript import inch

ps_page = hep.draw.postscript.PSFile("ps_page.ps")
ps_draw = ps_page.transform(
    range=(0, 0, 1, 1),
    origin=((8.5 - 4) / 2 * inch, (11 - 4) / 2 * inch),
    scale=(4 * inch, 4 * inch))
```

## 7.2 Drawing Primitives

A `Draw` object provides methods for drawing primitives such as lines, text, and symbols.

A `Draw` object does not carry any state information, so drawing styles such as colors, line thickness, and fonts must be specified with each call. Such styles are specified using instances of these classes:

- `Color(red, green, blue)` produces an object representing a color with the specified red, green, and blue components, each between zero and one. The constants `black` and `white` are also provided. The default color for drawing operations is black

  You may also specify a color using an instance of `Gray(level)`, where *level* is a gray level, or an instance of `HSV(hue, saturation, value)`.
- An instance of `LineStyle(color, width)` specifies the color and line width for drawn lines. The default line style for drawing operations is black with a 1 pt line width.
- An instance of `Fill(color)` specifies the fill pattern for solid objects. The default fill is solid black.
- An instance of `Marker(shape, color, size)` specifies the style of drawn markers (see below). Available shapes are `"filled dot"` (the default), `"empty dot"`, `"filled square"`, `"empty square"`, `"filled diamond"`, `"empty diamond"`, `"asterisk"`, `"+"`, `"X"`. The default color is black and the default size is 1 mm.
- An insance of `Font(family, size)` specifies the drawing style for text. Available shapes are `"Courier"`, `"Courier-Bold"`, `"Courier-Oblique"`, `"Helvetica"`, `"Helvetica-Bold"`, `"Helvetica-Oblique"`, `"Symbol"`, `"Times"` (the default), `"Times-Bold"`, and `"Times-Italic"`. The default size is 11 pt.

Below are listed the available methods for drawing primitives. You may always omit color, font, and style arguments, in which case the defaults specified above are used.

- `drawLine(x0, y0, x1, y1, line_style)` draws a line.
- `drawRectangle(x0, y0, x1, y1, fill_style)` draws a rectangle.

- `drawText(x, y, text, font, color, alignment)` draws text. The *alingment* argument is a sequence of two values between zero and one which specify the X and Y alignment of the text relative to the specified position. The default alignment `(0, 0)` specifies left- and bottom-justified text. The alignment value `(1, 0.5)`, for instance, specifies text that extends rightwards and is vertically centered around the specified position. Note that *text* should be Latin-1 encoded.
- `drawMarker(x, y, marker_style)` draws a marker symbol, such as for a point in a scatter plot. A variety of marker shapes are available; see the description of `MarkerStyle` above.

## 7.3 Drawing accessories

The `hep.draw` module provides these convenience functions:

- `drawFrame(draw, range, line_style)` draws a rectangular frame; *range* is a sequence `(x0, y0, x1, y1)`.
- `addBorder(draw, width, frame_style)` returns a transformed `Draw` object obtained by adding a blank margin of specified *width* around *draw*. If *frame_style* is provided, a border is drawn inside the margin as well.
- `divide(draw, columns, rows, frame_style, gutter)` divides a `Draw` into a rectangular array of smaller drawing regions. The arguments *columns* and *rows* specify the number of drawing regions in the array. If *frame_style* is provied, each drawing region is framed by a rectangle drawn with that line style. If *gutter* is specified, the array includes blank space of this thickness between drawing regions.

  The return value is an array of arrays of `Draw` objects simulating a two-dimensional array indexed by column number and row number.

## 7.4 Plotting

The module `hep.hist.plot` provides functions for plotting histograms and scatter plots.

A plot is represented by a `Plot` object. It contains format and style information for plotting one or more histograms onto a single set of axes, and references to the histograms themselves. Each histogram drawn in a single plot is called a “series”; many series may be included in the same plot, but all share the same scales on their axes.

A `Plot` object is not tied to a particular output device; rather, its `draw` method draws the plot onto any `Draw` object.

The `hep.hist.plot.plot` function is a convenient way to construct a plot of single histogram. It creates a new `Plot` object using a style appropriate for the histogram. You may subsequently alter the plot’s style, add additional series to it, and then draw it.

This example creates a 6 cm by 4 cm encapsulated PostScript file showing the plot of a histogam.

```
import hep.draw.postscript
from   hep.hist.plot import plot

eps_file = hep.draw.postscript.EPSFile("plot.eps", 0.06, 0.04)
plot(eps_file, histogram).draw(eps_file)
```

The function `drawPlot` is equivalent to `plot`, except that it draws the plot as well. The last line of the above example may equivalently be written,

```
from   hep.hist.plot import drawPlot
drawPlot(eps_file, histogram)
```

You may pass additional keyword arguments to `plot` to override style attributes used in the plot. For example, to include overflow and underflow bins in the plot, you woud call,

```
plot(eps_file, histogram, show_overflow=True)
```

The next section describes the various style attributes.

## 7.5 Plotting Styles

The visual format of a plot is stored in a "style dictionary." This is a mapping of names of style attributes (see below for a list). The values in the style dictionary are boolean flags, lengths, colors, and other settings that control how the plot is drawn. The default style dictionary, `hep.hist.plot.default_style`, contains default values for all style attributes used by the plotting code. Each plot has its own style, and each series in a plot has its own style as well; however, these styles need not define every style attribute. Instead, styles are cascading: a style attribute not found in a series's style dictionary is obtained from the plot's style dictionary; if it isn't there either, it is taken from the default style dictionary.

These are the most important style attributes:

- `"show_overflow"` specifies whether or not to show underflow and overflow bins.
- `"show_errors"` specifies whether or not to display errors (uncertainties) in the plot.
- `"bin_style"` specifies how to draw bins of a histogram. The value `"default"` uses the default bin style for each type of histogram. Other styles are described below.
- `"line_style"` is the `hep.draw.LineStyle` used when drawing data and error bars.
- `"marker"` is a `hep.draw.Marker` instance, which specifies the marker style used to plot data.
- `"x_axis_range"` and `"y_axis_range"` are pairs of values that specify the range to plot on the axes. If the value is `"default"`, the range is determined automatically.
- `"x_axis_line_style"` and `"y_axis_line_style"` are `LineStyle` objects specifying how to draw the axis lines. If one is `None`, no axis line is drawn.
- `"tick_size"` and `"tick_line_style"` control the drawing of tick marks on axes. The latter is a `LineStyle` object, or `None` to suppress tick marks.
- `"label_font"` is a `hep.draw.Font` object specifying the font to use for labelling the plot.

For one-dimensionaly histograms, these bin styles may be used:

- `"filled"` draws bins as a landscape of filled rectangles. The height of each rectangle represents the contents of a bin. The `"data_fill"` style attribute is a `hep.draw.Fill` object that specifies how the rectangles are filled.
- `"marker"` represents bins as markers. If errors are shown, they are represented by vertical lines through the markers. The `"marker"` style attribute is a `hep.draw.Marker` object that specifies how the markers are drawn
- `"cross"` represents a bin's as a cross. The horizontal line indicates the bin value and bin width; the vertical line shows bin errors.

For two-dimensional histograms, these bin styles may be used:

- `"box"` draws a box plot. The `"data_fill"` style attribute is specifies how the boxes are filled.
- `"box_twotone"` draws a box plot where boxes for positive bin values are drawn in black and boxes for negative bin values are drawn in blue. The area of the box is proportional to the absolute value of the bin contents.
- `"density"` draws a density plot. The value of each bin is represented by a colored rectangle, where the color is interpolated between white (the z-axis minimum value) and the color of `"data_fill"` (the z-axis maximum value).
- `"density_twotone"` draws a density plot where positive bin values are represented by shades of grey and negative bin values are represented by shades of blue. Solid black represents the z-axis maximum value; white represents zero, and solid blue represents the z-axis minimum value.

The function `makeStyle` function constructs a style dictionary suitable for displaying a particular histogram or scatter plot in a particular format and size. Pass the histogram or scatter plot as the first argument. The `format` parameter may be `"print"` (optimized for hardcopy output), `"screen"` (optimzied for screen display), or `"presentation"` (optimized for large-format presentation). The `scale` parameter is the size (either vertical or horizontal) in which the plot will be drawn. Additional keyword arguments are added as style attributes to the returned style dictionary.

## 7.6 `Plot` objects

The return value of the `plot` function is a `hep.hist.plot.Plot` objects. You can also create an empty `Plot` object explicitly, passing a style dictionary for the plot, and add one or more series manually. The plot's style is accessible as its `style` attribute.

To add a histogram series to a plot, use the `addSeries` method. An option second argument is the style dictionary for that series. (Remember, any style not specified in a series's style dictionary is inherited from the plot's style dictionary, or from the default.) The series are accessible as the plot object's `series` attribute, which is a sequence of `Series` objects. Each of these has a `data` attribute, referring to the histogram itself, and a `style` attribute, containing the series's style dictionary. Note that adding series of different types (histograms of different dimensionality and/or scatter plots) to a single plot may produce unpredictable results.

To draw a plot, call its `draw` method, passing a draw object.

If you wish to draw additional annotations onto a plot after drawing it, call the `buildLayout` method, passing the draw object. The returned layout object has attributes containing draw objects with which you can draw onto various parts of the plot:

- `draw` is the entire draw object used for the plot
- `x_axis_draw` is used to draw the x axis; its x scale is set to the actual x axis scale in the plot
- `y_axis_draw` is correspondingly for the y axis
- `data_draw` is used to draw the plot data; its x and y scales are set to the actual axis scales in the plot

This example shows how to make a plot displaying two one-dimensional histograms, and draw it to an EPS file. One histogram (say, results of simulation) is drawn as a filled-in contour; the other (say, data) is drawn as filled dots with error bars.

```
import hep.draw
import hep.draw.postscript
import hep.hist.plot
```

```
# Create a PostScript file.
ps_file = hep.draw.postscript.PSFile("histograms.ps", border=0.03)
# Add a plot showing two histograms.
plot = hep.hist.plot.plot(ps_file, simulation_hist, bin_style="filled")
plot.addSeries(data_hist,
               show_errors=True,
               bin_style="marker", marker=hep.draw.Marker("filled dot"))
# Draw it.
plot.draw(ps_file)
```

CHAPTER

**EIGHT**

# Lorentz geometry and kinematics

PyHEP provides an implementation of Lorentz vectors, momenta, and transformations. The implementation classes do not simply provide quadruplets of numbers equipped with a Minkowski inner product; instead, they represent coordinate-independent objects. Ultimately, though, the coordinates of a vector must be specified and obtained; this is done using a *reference frame*, which specifies the coordinate system to use.

Coordinates are given in the order $(t, x, y, z)$, the metric signature $(+, -, -, -)$ is employed, and the speed of light $c$ is assumed to be unity.

## 8.1 Reference frames

A four-vector is a geometric object, which can be used to represent, for instance, the space-time position of an event, or the energy-momentum of a particle. The typical representation of a four-vector is a quadruplet of four coordinates, but the coordinate values for a particular four-vector depends on the basis used, or equivalently on the reference frame in which the coordinates are specified.

The Python class `hep.lorentz.Frame` represents a reference frame. The principle of relativity implies that there is no absolute way of specifying a frame; the frame may only be specified in relation to another. In PyHEP, a frame is specified in relation to a special frame, the canonical *lab frame*. The lab frame is `hep.lorentz.lab`, an instance of `Frame`.

## 8.2 Vectors

A four-vector is represented by an instance of `hep.lorentz.Vector` class. A `Vector` instance represents the geometric object, which is independent of reference frame, so you can specify or obtain its coordinate values only in reference to a frame. You may use the lab frame for this, or any other frame which you create.

To create a vector by specifying its coordinates in a particular frame, use the `Vector` method of that frame. For instance,

```
from hep.lorentz import lab
vector = lab.Vector(5.0, 1.0, 0.0, -2.0)
```

creates a four-vector whose time coordinate is 5.0 and whose space coordinates are (1.0, 0.0, -2.0) in the lab frame.

To obtain the coordinates of a four-vector in a reference frame, use the `coordinatesOf` method of that frame. For instance,

```
t, x, y, z = lab.coordinatesOf(vector)
```

You may negate (invert) a vector or scale it by a constant. You may also add or subtract two vectors. Since these are geometric operations, no frame is specified. For example,

```
vector3 = - vector1 / 2 + 3 * vector2
```

Use the ^ operator to obtain the inner product of two vectors. Each vector also has an attribute `norm`, its Lorentz-invariant normal. For example,

```
c = (vector1 ^ vector2) / (vector1.norm * vector2.norm)
```

You may also use `hep.lorentz.Momentum`, a subclass of `Vector` that represents a four-momentum. It provided an additional attribute `mass`, which is equivalent to `norm`, plus an attribute `rest_frame`, which is a `Frame` object representing the rest frame of a particle with that four-momentum.

## 8.3 Transformations and frames

A `hep.lorentz.Transformation` object represents a general Lorentz transformation. It can be used to transform either a geometric object, such as a four-vector, or a reference frame.

Typically, a transformation is specified as a rotation or a boost. A rotation is specified by the Euler angles $\phi, \theta, \psi$ in a particular reference frame. A boost is specified by the vector $\vec{\beta}$ in a particular reference frame. The frame object's `Rotation` and `Boost` methods, respectively, create these transformations. For example,

```
from hep.lorentz import lab
from math import pi
rotation = lab.Rotation(pi / 4, pi / 4, 0)
boost = lab.Boost(0.0, 0.0, 0.5)
```

The arguments to `Rotation` are the Euler angles, and the arguments to `Boost` are the components of $\vec{\beta}$.

Transformations may be composed using the  operation. Be careful about the frame in which you specify each one; generally, for sequential transformations, you will want to apply the previous transformation to your starting frame of reference before specifying the next one.

You can apply a transformation to a four-vector using the ^ operator; this returns a different geometric four-vector.

FIXME

A transformation can also be used to create a new reference frame.

CHAPTER

**NINE**

# Particle properties

The `hep.pdt` module provides code to access a *particle data table*, which contains measured properties of particles studied in high energy physics.

PyHEP includes a particle data table in `hep.pdt.default`. This table contains particle data from the XXXX edition of the *Review of Particle Properties* published by the Particle Data Group (PDG). The function `hep.pdt.loadPdtFile` can be used to load particle data from a file in the PDG's text file format.

A particle data table is represented by a `hep.pdt.Table` instance, which acts as a dictionary keyed with the plain-text names of particles. The text (ASCII) names are set by the PDG. These names are generally the same as the conventional particle designations, with Greek letters spelled out and underscores used to denote subscripts. For particle names associated with more than one charge state, the charge muse be indicated. Neutral antiparticles conventionally designated with an overbar are denoted by the "`anti-`" prefix. For example, a positron is spelled "`e+`", and the short-lived kaon eignstate "`K_S0`". The usual `keys` method will return the plain-text names of all particles in the table. A particle's name may have alternate common text spellings; these are stored as aliases, and a particle may be accessed in the table by any of its aliases.

The values in a particle data table are `hep.pdt.Particle` instances. Each represents a siggle species of particle, with unique quantum numbers. (Some special particle codes representing bound states, particles that have not been established, Monte Carlo internal constructs, etc. are also included.)

A `Particle` object has these attributes. Central values are given for measured properties.

- `name` is the particle's text name.
- `aliases` is a sequence of alternate text names for the particle.
- `charge_conjugate` is the `Particle` object for the particle's charge conjugate.
- `mass` is the particle's nominal mass, in GeV.
- `width` is the particle's width, in GeV.
- `charge` is the particle's electric charge.
- `spin` is the particle's spin, in units of half h-bar.
- `is_stable` is true if the particle is considered stable.
- `id` is the particle's Monte Carlo ID number in the PDG's numbering scheme.

The `Table` object also as a method `findId`, which returns the particle corresponding to a Monte Carlo ID number.

The following is a demonstration of using the default particle data table to look up some particle properties.

```
>>> from hep.pdt import default as particle_data
>>> print particle_data["e-"].mass
0.000510999
>>> print particle_data["J/psi"].spin
1.0
>>> print particle_data.findId(22).name
gamma
```

CHAPTER

**TEN**

# Using EvtGen

PyHEP provides a simple interface to the EvtGen event generator. You can easily generate randomized decays of particles, and examine the decay products. The EvtGen interface is in the `hep.evtgen` module.

To create a particle decay, follow these steps:

1. Create a `Generator` instance. The two arguments to its constructor are the path to the particle data listing file, which contains particle property information, and the path to the main decay file, which contains decays and branching fractions. The default EvtGen particle data and decay files are used if these arguments are omitted. You may specify paths to user decay files, which override the main decay file, as additional arguments. See the EvtGen documentation for information about these files.
2. Create a `Particle` object to represent the initial-state particle. Specify the name of the particle, as listed in the particle data file, as the argument. The particle is originally at rest at the origin of the lab frame.
3. Produce the decay by calling the generator's `decay` method on the particle object.

A `Particle` object's momentum is stored in its `momentum` attribute, as a `hep.lorentz.FourMomentum` object. Use `hep.lorentz.lab.coordinatesOf` to obtain its lab-frame coordinates. Similarly, its production position is stored in its `position` attribute, as a `hep.lorentz.FourVector`. The name of the Particle's species is in its `species` attribute.

Use the `decay_products` attribute to access the decay products of a decayed particle. That value is a sequence of `Particle` objects representing the particle's decay products.

The following script produces a single decay of an Upsilon(4S), using a particle data listing file and a decay file in the current directory. It prints out the decay tree of the Upsilon(4S), with the lab components of each product's momentum, using the `printParticleTree` function.

```
import hep.evtgen
from   hep.lorentz import lab

def printParticleTree(particle, depth=0):
    indentation = depth * " "
    name = "%-12s" % particle.species
    pad = (8 - depth) * " "
    momentum = "%5.2f %5.2f %5.2f %5.2f" % lab.coordinatesOf(particle.momentum)
    print indentation + name + pad + momentum
    for child in particle.decay_products:
        printParticleTree(child, depth + 1)


generator = hep.evtgen.Generator("evt.pdl", "DECAY.DEC")
upsilon4s = hep.evtgen.Particle("Upsilon(4S)")
generator.decay(upsilon4s)
printParticleTree(upsilon4s)
```

CHAPTER

**ELEVEN**

# Interactive PyHEP

This chapter describes how to use interactive PyHEP. Interactive PyHEP is simply the ordinary Python interpreter with certain PyHEP modules preloaded, commonly-used names imported into the global namespace, and some additional interactive functions and variables are provided.

## 11.1 Invoking interactive PyHEP

The `pyhep` script launches interactive PyHEP. This is installed by default in `/usr/bin`; check your installation for your specific location.

In addition to the Python interpreter's usual startup message, the PyHEP version is displayed.

## 11.2 Imported names

These names are imported into the global namespace:

- The contents of the standard `math` module.
- The contents of the `hep.num` module.
- If your version of Python doesn't include built-in `bool`, `True`, and `False`, these names are added.
- The `hep.lorentz.lab` reference frame object, as `lab`.
- The `default` particle data dictionary from `hep.pdt`, as `pdt`.

## 11.3 Interactive functions and variables

The names of interactive PyHEP functions always begin with '`i`', and the names of variables always begin with '`i_`'.

The `ihelp` function displays some help information for interactive PyHEP.

### 11.3.1 Plotting functions

Interactive PyHEP plots histograms in a pop-up plot window, similar to PAW. The plot window may be divided into rectangular "zones", each displaying one plot. All plots are shown in the same window, replacing other plots where necessary.

**iplot**(*histogram*[, *over, **style*])
: Plots a histogram or scatter plot in the plot window. The argument is a histogram or `Scatter` object. The `over` parameter specifies whether the histogram is overlayed onto the current plot. If false (the default), the histogram is displayed in a new plot in the next plot zone, replacing the plot there. If `over` true, it is added as an additional series to the current plot. Any additional keyword arguments are used as style attributes for the new plot or plot series.

**idivide**(*columns, rows*[, *gutter*])
: Divide the plot window into a grid of rectangular plot zones. The arguments are the number of columns and rows in the grid. The optional third `gutter` argument is the size of the gutter between plot zones, by default 1 cm. Each call to `iplot` uses the next plot zone, proceeding left-to-right and then top-to-bottom, unless the `over` argument is true. Note that calling `idivide` always removes all plots and clears the plot window.

**iselect**(*column, row*)
: Select a plot zone. Its arguments are the column and row coordinates of the zone. The next plot is drawn in this zone.

**iredraw**([*all*])
: Redraw the plot in the current zone. If the optional `all` argument is true, all zones are redrawn. Changes to the plot style or histogram contents are reflected only when the plot is redrawn.

**iprint**(*file_name*)
: Print the contents of the plot window to a file. The type of the file is inferred from the file name extension: ".ps" produces a PostScript file, and ".eps" produces an encapsulated PostScript file.

The global variable `i_plot` always refers to the current `Plot` object (or `None` if the current plot zone is empty). The global variable `i_plots` is a sequence of sequences of `Plot` objects for all zones in the plot window.

The global variable `i_window` is a draw object for the plot window, and `i_zone` is a draw object for the current plot zone.

You may modify any plot displayed in any plot zone. Select the zone with `iselect`, access and modify the plot with `i_plot` (for instance, adjust its style or add a series), and then call `iredraw` to redraw it.

### 11.3.2 Table functions

**iproject**(*table, expression*[, *selection, number_of_bins, range*])
: Project a histogram from a table. The first argument is the table object, and the second argument is the expression to accumulate in the histogram. The optional `selection` argument is a selection expression; only table rows for which this expression is true are projected into the histogram. The number of bins and histogram range are automatically determined, but to override these, use the `number_of_bins` and `range` (a pair of values) arguments, respectively. The return value is a histogram object.

**idump**(*rows, *expressions*)
: Dumps values from a table. The first argument is a table or an iterator over table rows. One or more additional arguments are expressions whose values are to be displayed. The output is a table with the row number followed by the values of the specified expressions.