SYNOPSIS

```
# OBJECT-ORIENTED INTERFACE (PREFERRED INTERFACE)

# create object
# e.g, compute average elevation in boxes of 2x2
# no data required at this point
my $binner = PDL::NDBin->new(
 axes => [ [ 'x', min => 0, max => 10, step => 2 ],
           [ 'y', min => 0, max => 10, step => 2 ] ],
 vars => [ [ 'elevation', 'Avg' ] ],
);
# or any sort of computation:
#   elevation => sub { (shift->selection->stats)[2] }   # median
#   elevation => sub { shift->selection->min }          # minimum
#   elevation => \&user_defined_function                # anything

# feed and process data
my( $x, $y, $z ) = get_data();
$binner->feed( x => $x, y => $y, elevation => $z );
$binner->process;

# or feed and process in one step
$binner->process( x => $x, y => $y, elevation => $z );

# output
my $average_elevation = $binner->output->{ elevation };

# or as a hash
my %results = $binner->output;
print $results{ elevation }, "\n";

# WRAPPER FUNCTIONS

# bin the values
#    pdl( 1,1,2 )
# in 3 bins with a width of 1, starting at 0:
my $histogram = ndbinning( pdl( 1,1,2 ), 1, 0, 3 );
# returns the one-dimensional histogram
#    long( 0,2,1 )

# bin the values
$x = pdl( 1,1,1,2,2 );
$y = pdl( 2,1,1,1,1 );
# along two dimensions, with 3 bins per dimension:
my $histogram = ndbinning( $x => (1,0,3),
                           $y => (1,0,3) );
# returns the two-dimensional histogram
#    long( [0,0,0],
#          [0,2,2],
#          [0,1,0] )
```

DESCRIPTION

In scientific (and other) applications, it is frequently necessary to classify a series of values in a number of bins. For instance, particles may be classified according to particle size in a number of bins of, say, 0.01 mm wide, yielding a histogram. Or, to take an example from my own work: satellite measurements taken all over the globe must often be classified in latitude/longitude boxes for further processing.

PDL has a dedicated function to make histograms, hist(). To create a histogram of particle size from 0 mm to 10 mm, in bins of 0.1 mm, you would write:

```
my $histogram = hist $particles, 0, 10, 0.1;
```

This will count the number of particles in every bin, yielding the 100 counts that form the histogram. But what if you wanted to perform other computations on the values in the bins? It is actually not that difficult to perform the binning by hand. The key is to associate a bin number with every data value. With fixed-size bins of 0.1 mm wide, that is accomplished with

```
my $bin_numbers = PDL::long( $particles/0.1 );
```

(Note that the formulation above does not take care of data beyond 10 mm, but PDL::NDBin does.) We now have two arrays of data: the actual particle sizes in $particles, and the bin numbers associated with every data value in $bin_numbers. The histogram could now be produced with the following loop, $N being 100:

```
my $histogram = zeroes( long, $N );
for my $bin ( 0 .. $N-1 ) {
 my $want = which( $bin_numbers == $bin );
 $histogram->set( $bin, $want->nelem );
}
```

But, once we have the indices of the data values corresponding to any bin, it is a small matter to extend the loop to actually extract the data values in the bin. A user-supplied subroutine can then be invoked on the values in every bin:

```
my $output = zeroes( long, $N )->setbadif( 1 );
for my $bin ( 0 .. $N-1 ) {
 my $want = which( $bin_numbers == $bin );
 my $selection = $particles->index( $want );
 my $value = eval { $coderef->( $selection ) };
 if( defined $value ) { $output->set( $bin, $value ) }
}
```

(This is how early versions of PDL::NDBin were implemented.) The user subroutine could do anything with the values in the currently selected bin, $selection, including counting the number of elements. But the subroutine could also output the data to disk, or to a plot. Or the data could be collected to perform a regression. Anything that can be expressed with a subroutine, can now easily be plugged into this core loop.

This basic idea can even be extended by noticing that it is also possible to do multidimensional binning with the same core loop. The solution is to 'flatten' the bins, much like C and Perl flatten multidimensional arrays to a one-dimensional array in memory. So, you could perfectly bin satellite data along both latitude and longitude:

```
my( $latitude, $longitude ); # somehow get these data as 1-D vars
my $flattened = 0;
for my $var ( $latitude, $longitude ) {
 my $bin_numbers = long( ($var - $min)/$step );
 $bin_numbers->inplace->clip( 0, $n-1 );
```

```
 $flattened = $flattened * $n + $bin_numbers;
}
```

$flattened now contains pseudo-one-dimensional bin numbers, and can be used in the core loop shown above.

I've left out many details to illustrate the idea. The basic idea is very simple, but the implementation does get a bit messy when multiple variables are binned in multiple dimensions, with user-defined actions. Of course, ideally, you'd like this to be very performant, so you can handle several millions of data values without hitting memory constraints or running out of time. PDL::NDBin is there to handle the details for you, so you can write

```
my $average_flux = ndbin( $longitude, min => -70, max => 70, step => 20,
                          $latitude,  min => -70, max => 70, step => 20,
                          vars => [ [ $flux => 'Avg' ] ] );
```

to obtain the average of the flux, binned in boxes of 20x20 degrees latitude and longitude.

The rest of the documentation goes into more detail on the methods and implementation. You may also want to check out the examples (see *EXAMPLES*), or the comparison of PDL::NDBin with alternative solutions on CPAN (see *SEE ALSO* and further).

Please note that, although I do not anticipate major API changes, the interface and implementation are subject to change.

METHODS

add_axis()

Add an axis to the current object, with optional axis specifications. The argument list must be a list of key-value pairs. The name of the axis is mandatory.

```
$self->add_axis( name => 'longitude', min => -70, max => 70, n => 14 );
```

The following axis specifications are available:

name

: The name of this axis.

min

: The lowest value of the first bin. Values below this minimum will be binned in the first bin. Optional; will be determined from the actual minimum value in the data if not supplied.

max

: The highest value of the last bin. Values above this maximum will be binned in the last bin. Optional; will be determined from the actual maximum value in the data if not supplied.

step

: The width of the bins. Currently only a fixed step size is allowed, which means that all the bins have equal width. Optional; will be determined from the data range and the number of bins if not supplied.

n

: The number of bins. Optional; will be determined from the data range and the step size if not supplied. If the step size is not supplied, *n* will be set to the number of data values, or to 100, whichever is smaller.

round

: Round *min* and *max* to the nearest multiple of this value.

Note that you cannot specify all specifications at the same time, because some may conflict.

add_var()

Add a variable to the current object. The argument list must be a list of key-value pairs. The name of the variable and the action are both mandatory.

```
 $self->add_var( name => 'flux', action => 'Avg' );
```

The following variable specifications are available:

name

The name of this variable.

action

The action to perform on this variable. May be either a code reference (a reference to a named or anonymous subroutine) or a class name.

The action classes that are available as of PDL::NDBin v0.004 are:

- *PDL::NDBin::Action::Avg*
- *PDL::NDBin::Action::Count*
- *PDL::NDBin::Action::StdDev*
- *PDL::NDBin::Action::Sum*

They provide optimized implementations, coded in C, for the corresponding operations. The class names may be abbreviated to the part after the last `::`.

new()

Construct a PDL::NDBin object. The argument list must be a list of key-value pairs. No arguments are required, but you will want to add at least one axis eventually to do meaningful work.

```
 my $obj = PDL::NDBin->new( axes => [ [ 'x', min => -1, max => 1, step =>
.1 ],
                                      [ 'y', min => -1, max => 1, step =>
.1 ] ],
                            vars => [ [ 'F', 'Count' ] ] );
```

The accepted keys are the following:

`axes`

Specifies the axes along which to bin. The axes are supplied as an arrayref containing anonymous arrays, one per axis, as follows:

```
 axes => [
           [ $name1, $key11 => $value11, $key12 => $value12, ... ],
           [ $name2, $key21 => $value21, $key22 => $value22, ... ],
           ...
         ]
```

Only the name is required. All other specifications are optional and will be determined automatically as required. For a list of allowed axis specifications, consult add_axis(). Note that you cannot specify all specifications at the same time, because some may conflict.

At least one axis will eventually be required, although it needn't be specified at constructor time, and can be added later with add_axis(), if desired.

`vars`

Specifies the values to bin. The variables are supplied as an arrayref containing anonymous arrays, one per variable, as follows:

```
vars => [
          [ $name1 => $action1 ],
          [ $name2 => $action2 ],
          ...
        ]
```

Here, both the name and the action are required. In order to produce a histogram, supply `'Count'` as the action.

No variables are required (an *n*-dimensional histogram is produced if no variables are supplied), but they can be specified at constructor time, or at a later time with add_var() if desired.

axes()

Retrieve the axes. Returns a list in list context, and an array reference in scalar context.

vars()

Retrieve the variables. Returns a list in list context, and an array reference in scalar context.

feed()

Set the piddles that will eventually be used for the axes and variables. Arguments must be specified as key-value pairs, the keys being the name, and the values being the piddle for every piddle that is to be set.

```
$binner->feed( latitude => $latitude, longitude => $longitude );
```

Note that not all piddles need be set in one call. This function can be called repeatedly to set all piddles. This can be very useful when data must be read from disk, as in the following example (assuming $nc is an object that reads data from disk):

```
my $binner = PDL::NDBin->new( axes => [ [ x => ... ], [ y => ... ] ] );
for my $f ( 'x', 'y' ) { $binner->feed( $f => $nc->get( $f ) ) }
$binner->process;
```

autoscale_axis()

Determine the following parameters for one axis automatically, if they have not been supplied by the user: the step size, the lowest bin, and the number of bins. Use whatever combination is needed of the specifications that have been supplied by the user, and the data itself. Obviously, the piddles containing the data must have been set before calling this subroutine. Details of the automatic parameter calculation are given in the section on *IMPLEMENTATION NOTES* below.

It is not usually required to call this method, as it is called automatically by autoscale().

autoscale()

Determine the following parameters for all axes automatically, if they have not been supplied by the user: the step size, the lowest bin, and the number of bins. It will use whatever combination is needed of the specifications that have been supplied by the user, and the data itself. Obviously, the piddles containing the data must have been set before calling this subroutine. For more details on the autoscaling, consult autoscale_axis().

```
$binner->autoscale( x => $x, y => $y, z => $z );
```

autoscale() accepts, but does not require, arguments. They must be key-value pairs as for feed(), and indicate piddle data that must be fed into the object prior to autoscaling. Note that the autoscaling applies to all axes, and not only supplied as arguments.

It is not usually required to call this method, as it is called automatically by process().

labels()

Return the labels for the bins as a list of lists of ranges.

process()

The core method. The actual piddles to be used for the axes and variables can be supplied to this function, although the argument list can be empty if all piddles have already been supplied. The argument list is the same as the one of feed(), i.e., a list of key-value pairs specifying name and piddle.

```
# if all piddles have already been set with feed()
$binner->process();
```

process() returns $self for chained method calls.

output()

Return the output computed by the previous call(s) to process(). Each output variable is reshaped to make the number of dimensions equal to the number of axes, and the extent of each dimension equal to the number of bins along the axis.

The return value in list context is a hash, the keys and values of which correspond to the names and data of the variables. The return value in scalar context is a reference to this hash. When no variables have been supplied, a hash with a single key called *histogram* is returned.

```
my $result = $binner->output;
print $result->{average};
```

Note that it is not possible to call process() after having called output(), because the piddle data may have been reshaped.

_consume()

```
_consume BLOCK LIST
```

Shift and return (zero or more) leading items from *LIST* meeting the condition in *BLOCK*. Sets $_ for each item of *LIST* in turn.

For internal use.

_random_name()

Generate a random, hopefully unique name for a pdl.

For internal use.

WRAPPER FUNCTIONS

PDL::NDBin provides the two functions ndbinning() and ndbin(), which are (almost) drop-in replacements for histogram() and hist(), except that they handle an arbitrary number of dimensions.

ndbinning() and ndbin() are actually wrappers around the object-oriented interface of PDL::NDBin, and may be the most convenient way to work with PDL::NDBin for simple cases. For more advanced usage, the object-oriented interface may be required.

ndbinning()

Calculate an *n*-dimensional histogram from one or more piddles. The arguments must be specified (almost) like in histogram() and histogram2d(). That is, each axis must be followed by its three specifications *step*, *min* and *n*, being the step size, the minimum value, and the number of bins, respectively. The difference with histogram2d() is that the axis specifications follow the piddle immediately, instead of coming at the end.

```
my $hist = ndbinning( $pdl1, $step1, $min1, $n1,
                      $pdl2, $step2, $min2, $n2,
                      ... );
```

Variables may be added using the same syntax as the constructor new():

```
my $hist = ndbinning( $pdl1, ...,
                      vars => [ [ $var1, $action1 ],
                                [ $var2, $action2 ],
                                ... ] );
```

If no variables are supplied, the behaviour of histogram() and histogram2d() is emulated, i.e., an *n* -dimensional histogram is produced. This function, although more flexible than the former two, is likely slower. If all you need is a one- or two-dimensional histogram, use histogram() and histogram2d() instead. Note that, when no variables are supplied, the returned histogram is of type *long*, in contrast with histogram() and histogram2d(). The histogramming is achieved by passing an action which simply counts the number of elements in the bin.

Unlike the output of output(), the resulting piddles are output as an array reference, in the same order as the variables passed in. There are as many output piddles as variables, and exactly one output piddle if no variables have been supplied. The output piddles take the type of the variables. All values in the output piddles are initialized to the bad value, so missing bins can be distinguished from zero.

ndbin()

Calculate an *n*-dimensional histogram from one or more piddles. The arguments must be specified like in hist(). That is, each axis may be followed by at most three specifications *min*, *max*, and *step*, being the the minimum value, maximum value, and the step size, respectively.

```
my $hist = ndbin( $pdl1, $min1, $max1, $step1,
                  $pdl2, $min2, $max2, $step2,
                  ... );
```

Note that $min, $max, and $step may be omitted, and will be calculated automatically from the data, as in hist(). Variables may be added using the same syntax as the constructor new():

```
my $hist = ndbin( $pdl1, ...,
                  vars => [ [ $var1, $action1 ],
                            [ $var2, $action2 ],
                            ... ] );
```

If no variables are supplied, the behaviour of hist() is emulated, i.e., an *n*-dimensional histogram is produced. This function, although more flexible than the other, is likely slower. If all you need is a one-dimensional histogram, use hist() instead. Note that, when no variables are supplied, the returned histogram is of type *long*, in contrast with hist(). The histogramming is achieved by passing an action which simply counts the number of elements in the bin.

Unlike the output of output(), the resulting piddles are output as an array reference, in the same order as the variables passed in. There are as many output piddles as variables, and exactly one output piddle if no variables have been supplied. The output piddles take the type of the variables. All values in the output piddles are initialized to the bad value, so missing bins can be distinguished from zero.

EXAMPLES

A few examples are included with this distribution, in the directory *examples/*.

Histogram and stem-and-leaf plot

The basic usage of PDL::NDBin is illustrated below by constructing a histogram. Suppose we have a data table as follows (only the first 8 lines of data are shown):

```
 # Prestige  Income  Education  Occupation
 97          76      97         Physician
 93          64      93         Professor
 92          78      82         Banker
 90          75      92         Architect
 90          64      86         Chemist
 90          80      100        Dentist
 89          76      98         Lawyer
 88          72      86         Civil engineer
 ...
```

(The table is also included in the example files, and is taken from John Fox, Applied Regression Analysis, Linear Models, and Related Methods, SAGE Publications, Inc., 1997). We will now write a script to compute the histogram of the *Income* field, in bins of 10 units wide.

```
 use PDL;
 use PDL::NDBin;
```

Note that loading PDL::NBBin does not automatically export PDL to your namespace, so you need to load PDL explicitly.

```
 my $binner = PDL::NDBin->new( axes => [ [ 'Income', min => 0, max => 100,
step => 10 ] ],
                               vars => [ [ 'Income', 'Count' ] ] );
```

First we build the object with a call to new(). Note that the same name can be used in both axes and variables (in this case, *Income*). *step* signifies the width of the bins. By associating the action *Count* with *Income*, we will produce a histogram of the elements in *Income*. (The action name is actually the name of a class in the PDL::NDBin::Action namespace.)

```
 my( $prestige, $income, $education ) = rcols 'table';
```

Next, we read the data from the data file. The PDL function rcols() is very convenient to read tabular data of the kind shown above.

```
 $binner->process( Income => $income );
```

The data is then 'fed' into the binning object, with a call to process(). Note that you need to specify the name that was given in the constructor call in order to associate the numerical data with the axis and variable.

```
 my %results = $binner->output;
 my $histogram = $results{Income};
```

We now recover the histogram with output(), which returns a hash with the results, keyed by name (again the same name as used in the constructor). To find the number of elements in the bin with 40 <= income < 50, for instance, you could also use the following *awk(1)* script:

```
 $2 >= 40 && $2 < 50 { cnt++ }
 END                 { print cnt }
```

Of course, for this very simple example, the histogram could as well be calculated with the following built-in function of PDL:

```
 my $histogram = hist( $income, 0, 100, 10 );
```

If you'd rather print a stem-and-leaf plot, you could modify the constructor call as follows:

```
 my $binner = PDL::NDBin->new( axes => [ [ 'Income', min => 0, max => 100,
step => 10 ] ],
                               vars => [ [ 'Income', \&stem_and_leaf_plot ]
 ] );
```

Now the action associated with $income is no longer *Count* (which counts the elements in each bin), but a reference to the user-supplied subroutine stem_and_leaf_plot(). The latter could be implemented as shown below.

```
sub stem_and_leaf_plot
{
 my $iter = shift;
 my $bin  = $iter->bin;
 my @list = map { $_ % 10 } sort $iter->selection->list;
 printf "%d | %s\n", $bin, join '', @list;
}
```

The only argument supplied to our callback stem_and_leaf_plot() is an object of the type *PDL::NDBin::Iterator*. This object is used to iterate over the bins of the variable ($income). With the method bin(), we can recover the current bin number. With selection(), we recover those elements of $income that fall in the current bin. Those elements are then printed in a neat list (retaining only the last digit).

To actually produce the stem-and-leaf plot, we still need to call

```
$binner->process( Income => $income );
```

The result is the following neat diagram:

```
0 | 77899
1 | 245667
2 | 1111299
3 | 46
4 | 12224788
5 | 355
6 | 02447
7 | 2256668
8 | 01
9 |
```

Note that it is not necessary to call output(), as we are not interested in the return value of stem_and_leaf_plot().

Local averaging of two-dimensional data

This is a slightly more complicated example, where PDL::NDBin is used to average two-dimensional data in boxes of 1x1. Suppose you have elevation data of a particular area in the form of (x,y)-located samples:

```
# x     y  height
0.3   6.1   870.0
1.4   6.2   793.0
2.4   6.1   755.0
3.6   6.2   690.0
5.7   6.2   800.0
1.6   5.2   800.0
...
```

(The data have been taken from Example 14 of the GMT Cookbook. You can find more information on GMT under *SEE ALSO*.) Note that the samples are not distributed uniformly over the area. We want to compute the *average* elevation in boxes of 1 by 1, replacing multiple samples in any given box by the mean of those samples (e.g., prior to computing a surface through these points). When using the Generic Mapping Tools, you'd do it as follows:

```
 blockmean table -R0/7/0/7 -I1
```

How to do this with PDL::NDBin is shown below.

```
 use PDL;
 use PDL::NDBin;
 my( $x, $y, $z ) = rcols 'table';
```

As in the first example.

```
 my $binner = PDL::NDBin->new( axes => [ [ 'x', min=>-0.5, max=>7.5,
step=>1 ],
                                         [ 'y', min=>-0.5, max=>7.5,
step=>1 ] ],
                               vars => [ [ 'x', 'Avg' ],
                                         [ 'y', 'Avg' ],
                                         [ 'z', 'Avg' ] ] );
```

The constructor call specifies two axes for two-dimensional binning, and will compute the average in each bin of three variables simultaneously: x- and y-coordinate, and elevation ($z). We need to average the coordinates, as we want to replace multiple points with a single, average point; that is why *x* and *y* appear in the axes as well as in the variables.

To produce a table with averaged data, proceed (roughly) like in the first example:

```
 $binner->process( x => $x, y => $y, z => $z );
 my %results = $binner->output;
 my @avg = map { $_->flat } @results{ qw/x y z/ };
 wcols @avg;
```

wcols() is the inverse of rcols() and will print out the data in tabular format.

Average and standard deviation of sampled satellite data

The next example shows how to deal with large data volumes. Suppose you have the following data:

```
 #  longitude        latitude        albedo         flux     windspeed
 -28.5789718628  -17.6553726196  0.0973502323031   84.5   7.1533331871
 -12.5770769119  -20.5219345093  0.094131320715    81     6.69999980927
 -16.9122467041    1.0953686237  0.0729057863355   87.25  6.04666662216
 -16.2659015656  -11.5013151169  0.0838633701205   89     8.14666652679
   0.3412319422  -27.6491680145  0.151734098792    78.75  6.48000001907
 -32.6132278442   39.7315559387  0.128813564777   104.5   6.19333314896
 -33.4954719543   33.6763381958  0.0628560185432   80     5.28666687012
  11.4981594086   35.1409721375  0.0674269720912   84.25  5.2266664505
  ...
```

The data are actual satellite data obtained with the GERB instrument (*http://gerb.oma.be*). The data are located by longitude, latitude, and the task at hand is to assign each sample to boxes of *m* degrees longitude by *n* degrees latitude, and then to average all samples belonging to any given box, as well as computing the standard deviation. An example of this kind of binning in Python is shown *here*. In *awk(1)*, you could compute the average flux in the box bounded by -60 < longitude < -20 and

-60 < latitude < -20 as follows:

```
$1 > -60 && $1 < -20 && $2 > -60 && $2 < -20 { sum += $4; cnt++ }
END { print sum/cnt }
```

For the purpose of this example, the data sets have been stripped down very much, and the number of lat/lon boxes has been reduced greatly. A variant of this script is used to bin and average the samples for a complete month of data, totalling around 4GB of input data and more than 60 million samples.

The constructor call is

```
 my $binner = PDL::NDBin->new( axes => [ [ longitude => min => -60, max =>
60, step => 40 ],
                                       [ latitude  => min => -60, max =>
60, step => 40 ] ],
                               vars => [ [ avg    => 'Avg'    ],
                                       [ stddev => 'StdDev' ],
                                       [ count  => 'Count'  ] ] );
```

In an application, a large volume of data would likely be spread over multiple data files. Suppose that the data are distributed over a number of *.txt* files (in a real application, a binary format would be preferred over plain text). The following loop then processes all files without loading the entire data volume into memory:

```
for my $file ( glob '??.txt' ) {
 my( $longitude, $latitude, $albedo, $flux, $windspeed ) = rcols $file;
 $binner->process( longitude => $longitude,
                   latitude  => $latitude,
                   avg       => $flux,
                   stddev    => $flux,
                   count     => $flux );
}
```

Note how the data are read from disk and immediately processed. After the call to process(), the data are no longer required, and can be discarded! The actions *Avg*, *StdDev* and *Count* (and also *Sum* which is not shown in this example) keep an internal state which allows them to 'chain' multiple calls to process(). Note how the same variable $flux is fed three times to three different actions in order to obtain its average, standard deviation, and count, respectively.

The results are recovered as usual with

```
my %results = $binner->output;
print "Average flux:\n", $results{avg}, "\n";
print "Standard deviation of flux:\n", $results{stddev}, "\n";
print "Number of observations per bin:\n", $results{count}, "\n";
```

Another point to note in this example is that the optimized action classes *Avg* (and similar) are required for performance when processing large volumes of data. The average could in principle also be computed with a coderef:

```
avg => sub { shift->selection->avg }
```

Although the result will be the same, the computation will be much slower, since the call to selection() is very time-consuming.

IMPLEMENTATION NOTES

Lowest and highest bin

All data equal to or less than the minimum (either supplied or automatically determined) will be binned in the lowest bin. All data equal to or larger than the maximum (either supplied or automatically determined) will be binned in the highest bin. This is a slight asymmetry, as all other bins contain their lower bound but not their upper bound. However, it does the right thing when binning floating-point data.

Flattening multidimensional bin numbers

In PDL, the first dimension is the contiguous dimension, so we have to work back from the last axis to the first when building the flattened bin number.

Here are some examples of flattening multidimensional bins into one dimension:

```
(i) = i
(i,j) = j*I + i
(i,j,k) = (k*J + j)*I + i = k*J*I + j*I + i
(i,j,k,l) = ((l*K + k)*J + j)*I + i = l*K*J*I + k*J*I + j*I + i
```

Actions

You are required to supply an action with every variable. An action can be either a code reference (i.e., a reference to a subroutine, or an anonymous subroutine), or the name of a class that implements the methods new(), process() and result().

The actions will be called in the order they are given for each bin, before proceeding to the next bin. You can depend on this behaviour, for instance, when you have an action that depends on the result of a previous action within the same bin.

Code reference

In case the action specifies a code reference, this subroutine will be called with the following argument:

```
$coderef->( $iterator )
```

$iterator is an object of the class PDL::NDBin::Iterator, which will have been instantiated for you. Important to note is that the action will be called for every bin, with the given variable. The iterator must be used to retrieve information about the current bin and variable. With $iterator->selection(), for instance, you can access the elements that belong to this variable and this bin.

Class name

In case the action specifies a class name, an object of the class will be instantiated with

```
$object = $class->new( $N )
```

where $N signifies the total number of bins. The variables will be processed by calling

```
$object->process( $iterator )
```

where $iterator again signifies an iterator object. Results will be collected by calling

```
$object->result
```

The object is responsible for correct bin traversal, and for storing the result of the operation. The class must implement the three methods.

When supplying a class instead of an action reference, it is possible to compute multiple bins at once in one call to process(). This can be much more efficient than calling the action for every bin,

especially if the loop can be coded in XS.

Exceptions in actions

There is no protection from exceptions raised in actions, i.e., exceptions in actions will be propagated to the package that calls PDL::NDBin. This feature protects you from typos inside the action:

```
my $binner = PDL::NDBin->new(
 axes => [ ... ],
 vars => [ [ variable => sub { shift->selection->avearge } ] ]
);
```

In this example, average() is misspelled. If the action were executed in an `eval` block, the typo would go unnoticed, and all values of the output piddle would be undefined. If you want to trap exceptions in actions, use a wrapper action defined as follows:

```
variable => sub {
 my $iter = shift;
 eval { $your_action->( $iter ) };
}
```

Iteration strategy

By default, ndbin() will loop over all bins, and create a piddle per bin holding only the values in that bin. This piddle is accessible to your actions via the iterator object. This ensures that every action will only see the data in one bin at a time. You need to do this when, e.g., you are taking the average of the values in a bin with the standard PDL function avg(). However, the selection and extraction of the data is time-consuming. If you have an action that knows how to deal with indirection, you can do away with the selection and extraction. Examples of such actions are: PDL::NDBin::Action::Count, PDL::NDBin::Action::Sum, etc. They take the original data and the flattened bin numbers and produce an output piddle in one step.

Note that empty bins are not skipped. If you want to use an action that cannot handle empty piddles (e.g., PDL method min()), you can wrap the action as follows to skip empty bins:

```
variable => sub {
 my $iter = shift;
 return unless $iter->want->nelem;
 $your_action->( $iter );
}
```

Remember that returning *undef* from the action will not fill the current bin. Note that the evaluation of `$iter->want` entails a performance penalty, even if the bin is empty and not processed further.

Automatic axis parameter calculation

Range

The range, when not given explicitly, is calculated from the data by calling min() and max() on the data. An exception will be thrown if the data range is zero. autoscale_axis() honours the *round* key to round bin boundaries to the nearest multiple of *round*.

Number of bins

The number of bins *n*, when not given explicitly, is determined automatically. If the step size is defined and positive, the number of bins is calculated from the range and the step size as discussed below. If neither the number of bins, nor the step size have been supplied by the user, the number of bins is taken equal to the number of data values, or equal to 100, whichever is smaller.

The calculation of the number of bins is based on the formula

```
n = range / step
```

but needs to be modified. First, *n* calculated in this way may well be fractional. When *n* is ultimately used in the binning, it is converted to *int* by truncating. To have sufficient bins, *n* must be rounded up to the next integer. Second, the computation of *n* is and should be different for floating-point data and integral data.

For floating-point data, *n* is calculated as follows:

```
n = ceil( range / step )
```

The calculation is slightly different for integral data. When binning an integral number, say 4, it really belongs in a bin that spans the range 4 through 4.99...; to bin a list of data values with, say, *min* = 3 and *max* = 8, we must consider the range to be 9-3 = 6. A step size of 3 would yield 2 bins, one containing the values (3, 4, 5), and another containing the values (6, 7, 8). The correct formula for calculating the number of bins for integral data is therefore

```
n = ceil( (range+1) / step )
```

The modified formula for integral data values leads to more natural results, as the following example shows:

```
my $data = short( 1, 2, 3, 4 );
my( $min, $max, $step ) = ( 1, 4, 1 );

print ndbin( $data, $min, $max, $step );
# prints [1 1 1 1], as expected

print scalar hist( $data, $min, $max, $step );
# prints [1 1 2] at the time of writing (PDL v2.4.11)
```

Step size

The step size, when not given explicitly, is determined from the range and the number of bins *n* as follows:

```
step = range / n
```

for floating-point data, and

```
step = (range+1) / n
```

for integral data.

The step size may have a fractional part, even for integral data. The step size must not be less than one, however. If this happens, there are more bins than distinct numbers in the data, and the function will abort.

Note that when the number of *n* is not given either, a default value is substituted for it by PDL::NDBin, as described above.

TIPS & TRICKS

Find the total number of bins

```
use List::Util 'reduce';
my $binner = PDL::NDBin->new( axes => [ [ 'x', ... ], [ 'y', ... ] ] );
$binner->autoscale( x => $x, y => $y );
my $N = reduce { our $a * our $b } map { $_->{n} } $binner->axes;
```

Hook a progress bar to PDL::NDBin

For long-running computations, you may want to hook a progress bar to PDL::NDBin. There is an example in the *examples/* directory, but here is the gist:

```
use List::Util 'reduce';
use Term::ProgressBar::Simple;


my $progress;
my $binner = PDL::NDBin->new(
 axes => [ [ 'x', ... ], [ 'y', ... ] ],
 vars => [ ...,
           [ 'dummy' => sub { $progress++; return } ] ]
);
$binner->autoscale( x     => $x,
                    y     => $y,
                    dummy => null );
my $N = reduce { our $a * our $b } map { $_->{n} } $binner->axes;
$progress = Term::ProgressBar::Simple->new( $N );
$binner->process();
```

Note that, although we don't care about the return value of the anonymous sub associated with *dummy*, Term::ProgressBar::Simple doesn't like being returned from a function. (Hence the *return*.)

SEE ALSO

- The PDL::NDBin::Action:: namespace
- The *PDL* documentation

There are a few histogramming modules on CPAN:

- *PDL::Basic* offers the histogramming functions hist(), whist()
- *PDL::Primitive* offers the histogramming functions histogram(), histogram2d(), whistogram(), whistogram2d()
- *Math::GSL::Histogram* and *Math::GSL::Histogram2D*
- *Math::Histogram*
- *Math::SimpleHisto::XS*

Other tools:

- *awk(1)* is a fantastic tool that can be used to do many tasks like gridding or averaging with very concise scripts. Working with very large data volumes in plain text can be a bit slow, though.
- The *Generic Mapping Tools* (written in C) are focused on creating high-quality graphics but can also be used for tasks like gridding, local averaging, and more.

The following sections give a detailed overview of features, limitations, and performance of PDL::NDBin and related distributions on CPAN.

FEATURES AND LIMITATIONS

The following table gives an overview of the features and limitations of PDL::NDBin and related distributions on CPAN:

```
+-------------------------------------------------+---------+--------+---
-----+-----------+----------+
```

```
 | Feature                                              | MGH     | MH     |
MSHXS  | PDL       | PND      |

+-----------------------------------------------------+---------+--------+---
-----+----------+----------+
 | Allows piecewise data processing                    | -       | -      | -
     | -         | X        |
 | Allows resampling the histogram                     | -       | -      | X
     | X         | -        |
 | Automatic parameter calculation based on the data | -       | -      | -
     | X         | X        |
 | Bad value support                                   | -       | -      | -
     | X         | X        |
 | Can bin multiple variables at once                  | -       | -      | -
     | -         | X        |
 | Core implementation                                 | C       | C      | C
     | C         | C/Perl   |
 | Define and use callbacks to apply to the bins       | -       | -      | -
     | -         | Perl+C   |
 | Facilities for data structure serialization         | X       | X      | X
     | X         | -        |
 | Has overflow and underflow bins by default          | -       | X      | X
     | -         | -        |
 | Interface style                                     | Proc.   | OO     |
OO     | Proc.     | OO+Proc. |
 | Maximum number of dimensions                        | 2       | N      | 1
     | 2         | N        |
 | Native data type                                    | Scalars | Arrays |
Arrays | Piddles   | Piddles  |
 | Performance                                         | Low     | Medium |
High   | Very high | High     |
 | Support for weighted histograms                     | X       | X      | X
     | X         | -        |
 | Uses PDL threading                                  | -       | -      | -
     | X         | -        |
 | Variable-width bins                                 | X       | X      | X
     | -         | -        |

+-----------------------------------------------------+---------+--------+---
-----+----------+----------+
```

```
   MGH   = Math::GSL 0.26 (Math::GSL::Histogram and Math::GSL::Histogram2D)
   MH    = Math::Histogram 1.03
   MSHXS = Math::SimpleHisto::XS 1.28
   PDL   = PDL 2.4.11
   PND   = PDL::NDBin 0.004
```

An explanation and discussion of each of the features is provided below.

Allows piecewise data processing

> The ability to process data piecewise means that the input data (i.e., the data points) required to produce the output (e.g., a histogram) do not have to be fed all at once. Instead, the input data can be fed in chunks of any size. The resulting output is of course identical, whether the input data be fed piecewise or all at once. However, the input data do not have to fit in memory all at once, which is very useful when dealing with very large data sets.
>
> An example may help to understand this feature. Suppose you want to calculate the monthly

mean cloud cover over an area of the globe, in boxes of 1 by 1 degree. The total amount of cloud cover data is too large to fit in memory, but fortunately, the data are spread of several files, one by day. With PDL::NDBin, you can do the following:

```
my $binner = PDL::NDBin->new(
 axes => [[ 'latitude',    min => -60, max => 60, step => 1 ],
          [ 'longitude',   min => -60, max => 60, step => 1 ]],
 vars => [[ 'cloud_cover', 'Avg' ]],
);
for my $file ( @all_files ) {
 # suppose $file contains the geolocated cloud cover data for
 # one day of the month
 my $lat = $file->read( 'latitude' );
 my $lon = $file->read( 'longitude' );
 my $cc  = $file->read( 'cloud_cover' );
 $binner->process( latitude    => $lat,
                   longitude   => $lon,
                   cloud_cover => $cc );
}
my $avg = $binner->output->{cloud_cover};
```

In this example, only the data of a single day have to be kept in memory. The $binner object keeps a running average of the data, and retains the proper counts until the output $avg must be generated.

Only PDL::NDBin offers this feature. It can be simulated with other libraries for histograms, as long as histograms can be added together. PDL::NDBin extends the feature of piecewise data processing to sums, averages, and standard deviations.

Allows resampling the histogram

To resample a histogram means to put in a histogram of *N* bins, the data that were originally in a histogram of *M* bins, where *N* and *M* are different.

Only Math::SimpleHisto::XS and PDL support this feature. In PDL, the function is known as rebin() (to be found in *PDL::ImageND*).

Automatic parameter calculation based on the data

If a minimum bin, maximum bin, or step size are not supplied, PDL and PDL::NDBin will calculate them from the data. Other libraries require the user to specify them manually.

Bad value support

Bad value support, when it is present, allows to distinguish missing or invalid data from valid data. The missing or invalid data are excluded from the processing. Only the PDL-based libraries PDL and PDL::NDBin support bad values.

Can bin multiple variables at once

When data is co-located, e.g., cloud cover, cloud phase, and cloud optical thickness on a latitude-longitude grid, some time can be saved by binning the cloud variables together. Once the bin number has been determined for the given latitude and longitude, it can be reused for all cloud variables. This is marginally faster than binning the cloud variables separately. Only PDL::NDBin supports this feature.

Core implementation

Math::GSL::Histogram is a wrapper around the GSL library, which is written in C.

Math::Histogram is a wrapper around an *N*-dimensional histogramming library written in C.

Math::SimpleHisto::XS, by the same author as Math::Histogram, is implemented in C.

The core histogramming functions of PDL are implemented in C.

The core loops of PDL::NDBin are implemented partly in Perl, partly in C.

Define and use callbacks to apply to the bins

PDL::NDBin can handle any type of calculation on the values in the bins that you can express in Perl or C, not only counting the number of elements in order to produce a histogram. At the time of writing (version 0.004), PDL::NDBin supports counting, summing, averaging, and taking the standard deviation of the values in each bin. Additionally, Perl or C subroutines can be defined and used to perform any operation on the values in each bin.

This feature, arguably the most important feature of PDL::NDBin, is not found in other modules.

Facilities for data structure serialization

Serialization is the process of storing a histogram to disk, or retrieving it from disk. Math::GSL::Histogram, Math::Histogram, Math::SimpleHisto::XS, and PDL all have built-in support for serialization. PDL::NDBin doesn't, but the serialization facilities of PDL can be used to store and retrieve data. (I usually store computed data in netCDF files with *PDL::NetCDF*.)

Has overflow and underflow bins by default

Data lower than the lowest range of the first bin, or higher than the highest range of the last bin, are treated differently in different modules.

Math::GSL::Histogram ignores out-of-range values.

Math::Histogram and Math::SimpleHisto::XS have overflow bins, i.e., by default they create more bins than you define. These so-called overflow bins are situated at either end of every dimension. Out-of-range values end up in the overflow bins.

The histogramming functions of PDL, and PDL::NDBin, store low out-of-range values in the first bin, and high out-of-range values in the last bin.

To ignore out-of-range values with PDL::NDBin, define an additional bin at either end of every dimension, and disregard the values in these additional bins.

To simulate overflow and underflow bins with PDL::NDBin, define an additional bin at either end of every dimension.

Interface style

Proc. means that the module has a procedural interface. *OO* means that the module has an object-oriented interface. PDL::NDBin has both. Which interface you should use is largely a matter of preference, unless you want to use advanced features such as piecewise data feeding, which require the object-oriented interface.

Math::GSL::Histogram has a somewhat awkward interface, requiring the user to explicitly deallocate the data structure after use.

Maximum number of dimensions

The maximum number of dimensions that can be processed. Math::Histogram and PDL::NDBin can handle an arbitrary number of dimensions.

Native data type

Obviously, deep down, all data values are just C scalars. By 'native data type' is meant the data type used to communicate with the library in the most efficient way.

At the time of writing (Math::GSL version 0.27), Math::GSL::Histogram did not have a facility to enter multiple data points at once. It accepts only Perl scalars, and requires the user to input the data points one by one. Similarly, to produce the final histogram, the bins must be queried one by one.

Math::Histogram and Math::SimpleHisto::XS accept Perl arrays filled with values (although they also accept data points one by one as Perl scalars). Passing large amounts of data in an array is generally more efficient than passing the data points one by one as scalars.

PDL and PDL::NDBin operate on piddles only, which are memory-efficient, packed data

arrays. This could be considered both an advantage and a disadvantage. The advantage is that the piddles can be operated on very efficiently in C. The disadvantage is that PDL is required!

Performance

In the next section (see *PERFORMANCE*), the performance of all modules is examined in detail.

Support for weighted histograms

In a weighted histogram, data points contribute by a fractional amount (or weight) between 0 and 1. All libraries, except PDL::NDBin, support weighted histograms. In PDL::NDBin, the weight of all data points is fixed at 1.

Uses PDL threading

In PDL, threading is a technique to automatically loop certain operations over an arbitrary number of dimensions. An example is the sumover() operation, which calculates the row sum. It is defined over the first dimension only (i.e., the rows in PDL), but it will be looped automatically over all remaining dimensions. If the piddle is three-dimensional, for instance, sumover() will calculate the sum in every row of every matrix.

Threading is supported by the PDL functions histogram(), whistogram(), and their two-dimensional counterparts, but not by hist() or whist(). PDL::NDBin does not (yet) support threading.

Variable-width bins

In a histogram with variable-width bins, the width of the bins needn't be equal. This feature can be useful, for example, to construct bins on a logarithmic scale. Math::GSL, Math::Histogram, and Math::SimpleHisto::XS support variable-width bins; PDL and PDL::NDBin do not and are limited to fixed-width bins.

PERFORMANCE

One-dimensional histograms

This section aims to give an idea of the performance of PDL::NDBin. Some of the most important features of PDL::NDBin aren't found in other modules on CPAN. But there are a few histogramming modules on CPAN, and it is interesting to examine how well PDL::NDBin does in comparison.

I've run a number of tests with PDL version 0.004 on a laptop with an Intel i3 CPU running at 2.40 GHz, and on a desktop with an Intel i7 CPU running at 2.80 GHz and fast disks. The following table, obtained with 100 bins and a data file of 2 million data points, shows typical results on the laptop:

```
 Benchmark: timing 50 iterations of MGH, MH, MSHXS, PND, hist, histogram...
        MGH: 41 wallclock secs (40.83 usr +  0.00 sys = 40.83 CPU) @
1.22/s (n=50)
         MH:  6 wallclock secs ( 5.60 usr +  0.00 sys =  5.60 CPU) @
8.93/s (n=50)
      MSHXS:  2 wallclock secs ( 2.22 usr +  0.00 sys =  2.22 CPU) @
22.52/s (n=50)
        PND:  1 wallclock secs ( 1.43 usr +  0.00 sys =  1.43 CPU) @
34.97/s (n=50)
       hist:  2 wallclock secs ( 1.09 usr +  0.00 sys =  1.09 CPU) @
45.87/s (n=50)
  histogram:  1 wallclock secs ( 1.08 usr +  0.00 sys =  1.08 CPU) @
46.30/s (n=50)


 Relative performance:
             Rate       MGH         MH      MSHXS        PND       hist
histogram
 MGH       1.22/s        --       -86%       -95%       -96%       -97%
```

```
-97% MH         8.93/s       629%         --       -60%       -74%       -81%
      -81%
 MSHXS      22.5/s      1739%       152%         --       -36%       -51%
-51%
 PND        35.0/s      2755%       292%        55%         --       -24%
-24%
 hist       45.9/s      3646%       414%       104%        31%         --
-1%
 histogram 46.3/s      3681%       419%       106%        32%         1%
--
```

From this test and other tests, it can be concluded that PDL::NDBin (shown as 'PND' in the table) is, roughly speaking,

1. faster than Math::GSL::Histogram (shown as MGH in the table)

   Although this module is actually a wrapper around the C library GSL, the performance is rather low. The process of getting a large number of data points into Math::GSL::Histogram's data structures is inefficient, as the data points have to be input one by one.

2. faster than Math::Histogram (shown as MH)

   This library wraps another multidimensional histogramming library written in C. It allows inputting multiple data points at once. It is quite a bit faster than Math::GSL::Histogram, but does not offer the raw performance of PDL or Math::Histogram's cousin Math::SimpleHisto::XS.

3. faster than Math::SimpleHisto::XS (shown as MSHXS)

   Math::SimpleHisto::XS, by the same author as Math::Histogram, is similar to the latter library, but implemented in XS for speed, and limited to one-dimensional histograms. It is generally somewhat slower than PDL::NDBin, but outperforms it for small files or large bin counts (10,000 bins or more).

4. slower than PDL

   Although PDL::NDBin outperforms hist() by 10 to 20% in some of the tests, PDL's built-in functions hist() and histogram() are, on average, the fastest functions. Given that the core of these routines runs in pure C, this is not very surprising. The PDL functions have very low overhead and are very memory-efficient.

Note that, in the tests, various data conversions between piddles and ordinary Perl arrays were required. The timings exclude these conversions, and count only the time required to produce a histogram from the "natural" data structure, i.e. piddles for PDL-based modules, and ordinary Perl arrays for the other modules.

Note also that the histograms produced by the different methods were verified to be equal.

Two-dimensional histograms

Similar conclusions are obtained for two-dimensional histograms. The following table shows results on the laptop for 2 million data points with 100 bins:

```
 Benchmark: timing 50 iterations of MGH2d, PND2d, histogram2d...
       MGH2d: 65 wallclock secs (64.38 usr +  0.09 sys = 64.47 CPU) @
0.78/s (n=50)
       PND2d:  6 wallclock secs ( 5.96 usr +  0.00 sys =  5.96 CPU) @
8.39/s (n=50)
 histogram2d:  2 wallclock secs ( 2.16 usr +  0.01 sys =  2.17 CPU) @
23.04/s (n=50)


 Relative performance:
```

```
                Rate       MGH2d       PND2d histogram2d
MGH2d       0.776/s          --        -91%        -97%
PND2d        8.39/s        982%          --        -64%
histogram2d  23.0/s        2871%        175%          --
```

(It was not possible to run the test with Math::Histogram to completion.)

Scaling w.r.t. number of data points

Performance figures for a few tests on a particular machine don't say much. As PDL::NDBin is intended to handle large amounts of data, it is important to check how well PDL::NDBin's performance scales as the problem size increases.

The first and most obvious way in which a problem may be 'large', is the number of data points. If a given method cannot process a large number of data points, or can only do so with increased effort, it is not suitable for large problems. How large that is, depends on the application, but in the field of satellite data retrieval (where I work), 33 million data points is not exceptional at all (but it is the largest size I could test). In this section, we examine how well PDL::NDBin's performance scales with the number of data points, and compare with alternative modules.

The following table shows timing data on the laptop for 100 bins, but with a variable number of data points:

method	# points	CPU time (s)	n	time/iter. (ms)	time/i./point (ns)
MGH	66,398	38.84	1,500	25.893	389.972
MGH	2,255,838	43.06	50	861.200	381.765
MH	66,398	6.21	1,500	4.140	62.351
MH	2,255,838	5.65	50	113.000	50.092
MSHXS	66,398	2.11	1,500	1.407	21.185
MSHXS	2,255,838	2.26	50	45.200	20.037
PND	66,398	1.79	1,500	1.193	17.972
PND	2,255,838	1.38	50	27.600	12.235
PND	33,358,558	2.28	5	456.000	13.670
histogram	66,398	0.99	1,500	0.660	9.940
histogram	2,255,838	1.12	50	22.400	9.930
histogram	33,358,558	1.65	5	330.000	9.893

Note that the tests couldn't be run with Math::GSL::Histogram, Math::Histogram, and Math::SimpleHisto::XS on the largest data file (33 million points), due to insufficient memory.

The methods show a linear increase in time per iteration with the number of data points, which translates to a fixed time per iteration per data point. This is the desired behaviour: it guarantees that the effort required to produce a histogram does not increase faster than the problem size. Every method examined here displays this behaviour.

Quite notable is the high CPU time per iteration per data point of PDL::NDBin for small data files. For large data files, the time per iteration per data point is more or less constant. This effect is not fully understood, but may indicate high overhead or start-up cost.

The results suggest that PDL::NDBin scales well with the number of data points, and that it is therefore well suited for large data. PDL::NDBin and histogram() (and hist()) are currently the only

methods that allow processing very large data files.

Scaling w.r.t. number of bins

The number of data points may not be the only way in which a problem may be 'large' or hard. The number of bins may also be high. In applications with satellite data, for instance, a latitude/longitude grid with a resolution of only 5 degrees already yields more than 2000 bins, and raising the resolution to 1 degree yields approximately 64,000 bins.

Most of the methods depend in some way on the number of bins. If the execution time depends to a significant extent on the number of bins, the method is not suitable for large numbers of bins. In this section, we examine how well PDL::NDBin's performance scales with the number of bins, and compare with alternative modules.

The following table shows timing data on the laptop for 2 million data points, with a variable number of bins:

method	# bins	CPU time (s)	n	time/iter. (ms)
MGH	10	42.57	50	851.400
MGH	50	42.35	50	847.000
MGH	100	42.53	50	850.600
MGH	1,000	43.06	50	861.200
MGH	10,000	42.96	50	859.200
MGH	100,000	46.60	50	932.000
MGH	1,000,000	78.75	50	1575.000
MH	10	5.53	50	110.600
MH	50	5.51	50	110.200
MH	100	5.53	50	110.600
MH	1,000	5.65	50	113.000
MSHXS	10	2.26	50	45.200
MSHXS	50	2.21	50	44.200
MSHXS	100	2.22	50	44.400
MSHXS	1,000	2.26	50	45.200
MSHXS	10,000	2.30	50	46.000
MSHXS	100,000	2.65	50	53.000
MSHXS	1,000,000	6.22	50	124.400
PND	10	1.41	50	28.200
PND	50	1.40	50	28.000
PND	100	1.40	50	28.000
PND	1,000	1.38	50	27.600
PND	10,000	1.37	50	27.400
PND	100,000	1.40	50	28.000
PND	1,000,000	1.95	50	39.000
histogram	10	1.09	50	21.800
histogram	50	1.09	50	21.800
histogram	100	1.08	50	21.600
histogram	1,000	1.12	50	22.400
histogram	10,000	1.15	50	23.000
histogram	100,000	1.21	50	24.200
histogram	1,000,000	1.45	50	29.000

Note that some data are missing because the associated test didn't run successfully (e.g., segmentation fault).

The methods show more or less constant execution time per iteration, independent of the number of bins. This is the desired behaviour: the overhead of managing the bins does not dominate the execution time.

Quite notable is the behaviour of PDL::NDBin at high bin counts: beyond 1,000 bins, execution time rises significantly. The cause of this problem is not known.

The results suggest that PDL::NDBin scales well with the number of bins up to 1,000. Beyond 1,000 bins, the performance decreases significantly.

BUGS

None reported.

TODO

As is probably obvious from this manual, there are quite a few areas where PDL::NDBin can be improved. In particular:

- PDL:NDBin does not currently have a way to collect and return the values in a bin as a list or piddle; this would be very useful for plotting or output.
- PDL::NDBin does not currently support variable-width bins and weighted histograms.
- PDL::NDBin has some performance issues with very small datasets or large bin counts; some profiling is in order.
- The documentation can be expanded and improved in a few places.
- The axes should be refactored into objects instead of bare hashrefs, with methods such as labels(), n(), step(), etc.
- The action classes *Min* and *Max* would be useful and easy to add.