PDL Datatypes

All piddles store matrices of data in the same data type. PDL supports the following datatypes:

Datatype	Internal 'C' type	Valid values
`byte`	unsigned char	Integer values from 0 to +255
`short`	short	Integer values from -32,768 to +32,767
`ushort`	unsigned short	Integer values from 0 to +65,535
`long`	int	Integer values from -2,147,483,648 to +2,147,483,647
`longlong`	long	Integer values from –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807
`float`	float	Real values from -1.2E-38 to +3.4E+38 with 6 decimal places of precision
`double`	double	Real values from 2.3E-308 to +1.7E+308 with 15 decimal places of precision

`pdl` Examples

Row vector from explicit values:	`$v = pdl($type, [1,2]);`
Column vector from explicit values:	`$v = pdl($type, [[1],[2]]);` or `$v = pdl($type, [1,2])->(*1);`
Row vector from scalar string:	`$v = pdl($type, "1 2 3 4");`
Row vector from array of numbers:	`$v = pdl($type, @a);`
Matrix from explicit values:	`$M = pdl($type, [[1,2],[3,4]]);`
Matrix from a scalar:	`$M = pdl($type, "[1 2] [3 4]");`

Piddle Helper Creation Functions

In the following functions, where arguments are marked as `...`, accept arguments in the following form:

- `$type` - an optional data type (see above)
- `$x,$y,$z,...` - A list of n dimensions for the resulting piddle, OR
- `$M` - Another piddle, from which the dimensions will be re-used

Sequential integers, starting at zero:	`$M = sequence(...);`
Sequential Fibonacci values, starting at one:	`$M = fibonacci(...);`
	`$M = zeros(...);`

Of all zeros:	
Of all ones:	`$M = ones(...);`
Of random values between zero and one:	`$M = random(...);`
Of Gaussian random values between zero and one:	`$M = grandom(...);`
Where each value is it's zero-based index along the first dimension:	`$M = xvals(...);`
Where each value is it's zero-based index along the second dimension:	`$M = yvals(...);`
Where each value is it's zero-based index along the third dimension:	`$M = zvals(...);`
Where each value is it's zero-based index along dimension `$d`:	`$M = axisvals(..., $d);`
Where each value is it's distance from a specified centre:	`$M = rvals(..., {Centre=> [x,y,z,...]);`

The following functions create piddles with dimensions taken from another piddle, `$M` and distribute values between two endpoints (`$min` and `$max`) inclusively:

Linearly distributed values along the first dimension:	`$N = $M->xlinvals($min, $max);`
Linearly distributed values along the second dimension:	`$N = $M->ylinvals($min, $max);`
Linearly distributed values along the third dimension:	`$N = $M->zlinvals($min, $max);`
Logarithmically distributed values along the first dimension:	`$N = $M->xlogvals($min, $max);`
Logarithmically distributed values along the second dimension:	`$N = $M->ylogvals($min, $max);`
Logarithmically distributed values along the third dimension:	`$N = $M->zlogvals($min, $max);`

Co-ordinate Piddles

Finally the `ndcoords` utility function creates a piddle of co-ordinates for the supplied arguments. It may be called in two ways:

- `$coords = ndcoords($M);` - Take dimensions from another piddle
- `$coords = ndcoords(@dims);` - Take dimensions from a Perl list

Piddle Conversion

A piddle can be converted into a different type using the datatype names as a method upon the piddle. This returns the converted piddle as a new piddle. The `inplace` method does not work with these conversion methods.

Operation	Operator

Convert to byte datatype:	`$M->byte;` or `byte $M;`
Convert to short datatype:	`$M->short;` or `short $M;`
Convert to ushort datatype:	`$M->ushort;` or `ushort $M;`
Convert to long datatype:	`$M->long;` or `long $M;`
Convert to longlong datatype:	`$M->longlong;` or `longlong $M;`
Convert to float datatype:	`$M->float;` or `float $M;`
Convert to double datatype:	`$M->double;` or `double $M;`

Obtaining Piddle Information

PDL provides a number of functions to obtain information about piddles:

Description	Code
Return the number of elements:	`$M->nelem;`
Return the number of dimensions:	`$M->ndims;`
Return the length of dimension `$d`:	`$M->dim($d);`
Return the length of all dimensions as a Perl list:	`$M->dims;`
Return the length of all dimensions as a piddle:	`$M->shape;`
Return the datatype of a piddle:	`$M->type;`
Return general information about a piddle (datatype, dimensions):	`$M->info;`
Return the memory used by a piddle:	`$M->info("%M");`

Indexing, Slicing and Views

Points To Note

PDL internally stores matrices in column major format. This affects the indexing of piddle elements.

For example, take the following matrix `$M`:

```
    [
     [0 1 2]
     [3 4 5]
     [6 7 8]
    ]
```

[download]

In standard mathematical notation, the element at $M_{i,j}$ will be *i* elements down and *j* elements across, with the elements `0` and `3` at $M_{1,1}$ and $M_{2,1}$ respectively.

With PDL indexing, indexes start at zero, and the first two dimensions are 'swapped'. Therefore, the elements `0` and `3` are at PDL indices `(0,0)` and `(0,1)` respectively.

Views are References

PDL attempts to do as little work as possible in that it will try to avoid memory copying of piddle values when it can. The most common operations where this is the case is when taking piddle slices or views across a piddle matrix. The piddles returned by these functions are *views upon the original data*, rather than copies, so modifications to them *will affect the original matrix*.

Slicing

A common operation is to view only a subset of a piddle. This is called *slicing*.

As slicing is such a common operation, there is a module to implement a shorter syntax for the slice method. This module is PDL::NiceSlice. This document only uses this syntax.

A rectangular slice of a piddle is returned via using the default method on a piddle. This takes up to `n` arguments, where `n` is the number of dimensions in the piddle.

Each argument must be one of the following forms:

`""`	An empty value returns the entire dimension.
`n`	Return the value at index `n` into the dimension, keeping the dimension of size one.
`(n)`	Return the value at index `n` into the dimension, eliminating the entire dimension.
`n:m`	Return the range of values from index `n` to index `m` inclusive in the dimension. Negative indexes are indexed from the end of the dimension, where -1 is the last element.
`n:m:s`	Return the range of values from index `n` to index `m` with step `s` inclusive in the dimension. Negative indexes are indexed from the end of the dimension, where -1 is the last element.
`*n`	Insert a dummy dimension of size `n`.

The following examples operate on the matrix `$M`:

```
    [
     [0 1 2]
     [3 4 5]
     [6 7 8]
    ]
```

[download]

Description	Command	Result
Return the first column as a 1x3 matrix:	`$M->(0,);`	`[ [0][3][6] ]`
Return the first row as a 3x1 matrix:	`$M->(,0);`	`[ [0 1 2] ]`
Return the first row as a 3 element vector:	`$M->(,(0));`	`[0 1 2]`

Return the first and second column as a 2x3 matrix:	`$M->(0:1);`	`[ [0 1] [3 4] [6 7] ]`
Return the first and third row as a 3x2 matrix:	`$M->(,0:-1:2);`	`[ [0 1 2] [6 7 8] ]`

Dicing

Occasionally it is required to extract non-contiguous regions along a dimension. This is called *dicing*. The `dice` method accepts an array of indices for each dimension, which do not have to be contiguous.

The following examples operate on the matrix `$M`:

```
    [
     [0 1 2]
     [3 4 5]
     [6 7 8]
    ]
```
[download]

Description	Command	Result
Return the first and third column as a 2x3 matrix:	`$M->dice([0,2]);`	`[ [0 2] [3 5] [6 8] ]`
Return the first and third column and the first and third row as a 2x2 matrix:	`$M->dice([0,2], [0,2]);`	`[ [0 2] [6 8] ]`

Which and Where Clauses

The other common operation to perform over a piddle is to apply a boolean operation over the entire piddle elementwise. This is achieved in PDL with the `where` method.

The `where` method accepts a single argument of a boolean operation. The element is referred to within this argument with the same variable name as the piddle. The values in the returned piddle are references to the values in the initial piddle.

In a similar mannor to `which` clauses outlined above, there is the `where` method. The difference between these two methods is that `which` returns the values, while `where` returns the indices.

This is best explained with examples over a matrix `$M`:

Description	Return values	Return indices
Obtain all positive values:	`$M->where($M > 0);`	`which($M > 0);`
Obtain all values equal to three:	`$M->where($M == 3);`	`which($M == 3);`
Obtain all values which are not zero:	`$M->where($M != 0);`	`which($M != 0);`

Note that there is also the `which_both` function. This function returns an array of

two piddles. The first is a list of indices for which the boolean operation was true, the second for which the result was false.

Again, as where clauses as so common PDL::NiceSlice has syntatic support for it through the default method. This is acheived through an argument modifier, which is appended to the single argument.

The modifiers are seperated from the original argument via a `;` character, and the following modifiers are supported:

Modifier	Description
`?`	The argument is no longer a slice, but rather a `where` clause
`_`	`flatten` the piddle to one dimension prior to the operation
`-`	`squeeze` the piddle by flattening any dimensions of length one.
`|`	`sever` the returned piddle into a copy, rather than a reference

Using this syntax, the following where commands are identical:

```
    $M-&gt;where($M &gt; 3);
    $M-&gt;($M &gt; 3;?);
```

[download]

View Modification

PDL contains many functions to modify the view of a piddle. These are outlined below:

Description	Code
Transpose a matrix/vector:	`$M->transpose;`
Return the multidimensional diagonal over the supplied dimensions:	`$M->diagonal(@dims);`
Remove any dimensions of length one:	`$M->squeeze;`
Flatten to one dimension:	`$M->flat;`
Merge the first `$n` dimensions into one:	`$M->clump($n);`
Merge a list of dimensions into one:	`$M->clump(@dims);`
Exchange the position of zero-indexed dimensions `$i` and `$j`:	`$M->xchg($i, $j);`
Move the position of zero-indexed dimension `$d` to index `$i`:	`$M->mv($d, $i);`
Reorder the index of all dimensions:	`$M->reorder(@dims);`
Concatenate piddles of the same dimensions into a single piddle of rank n+1:	`cat($M, $N, ...);`
Split a single piddle into an array of piddles across the last dimension:	`($M, $N, ...) = dog($P);`
Rotate elements with wrap across the first dimension:	`$M->rotate($n);`

Given a vector `$v` return a matrix, where each column is of length `$len`, with step `$step` over the entire vector:	`$M->lags($dim, $step, $len);`
Normalise a vector to unit length:	`$M->norm;`
Destructively reshape a matrix to *n* dimensions, where *n* is the number of arguments and each argument is the length of each dimension. Any additional values are discarded and any missing values are set to zero:	`$M->resize(@dims);`
Append piddle `$N` to piddle `$M` across the first dimension:	`$M->append($N);`
Append piddle `$N` to piddle `$M` across the dimension with index `$dim`:	`$M->glue($dim, $N);`

Matrix Multiplication

PDL supports four main matrix multiplication methods between two piddles of compatible dimensions. These are:

Operation	Code
Dot product:	`$M x $N;`
Inner product:	`$M->inner($N);`
Outer product:	`$M->outer($N);`
Cross product:	`$M->crossp($N);`

As the `x` operator is overloaded to be the dot product, it can also be used to multiply vectors, matrices and scalars.

Operation	Code
Row x matrix = row	`$r x $M;`
Matrix x column = column	`$M x $c;`
Matrix x scalar = matrix	`$M x 3;`
Row x column = scalar	`$r x $c;`
Column x row = matrix	`$c x $r;`

Arithmetic Operations

PDL supports a number of arithmetic operations, both elementwise, over an entire matrix and along the first dimension. Double precision variants are prefixed with `d`.

Operation	Elementwise	Over entire PDL	Over 1st Dimention
Addition:	`$M + $N;`	`$M->sum;`; `$M->dsum;`	`$M->sumover;`; `$M->dsumover;`
Subtraction:	`$M - $N;`		
Product:	`$M * $N;`	`$M->prod;`; `$M->dprod;`	`$M->prodover;`; `$M->dprodover;`
Division:	`$M / $N;`		

Modulo:	`$M % $N;`	
Raise to the power:	`$M ** $N;`	
Cumulative Addition:		`$M->cumusumover;`; `$M->dcumusumover;`
Cumulative Product:		`$M->cumuprodover;`; `$M->dcumuprodover;`

Comparison Operations:

PDL supports a number of different elementwise comparison functions between matrices of the same shape.

Operation	Elementwise
Equal to:	`$M == $N;`
Not equal to:	`$M != $N;`
Greater than:	`$M > $N;`
Greater than or equal to:	`$M >= $N;`
Less than:	`$M < $N;`
Less than or equal to:	`$M <= $N;`
Compare (spaceship):	`$M <=> $N;`

Binary Operations

PDL also allows binary operations to occur over piddles. PDL will convert any real number datatype piddles (float, double) to an integer before performing the operation.

Operation	Elementwise	Over entire PDL	Over 1st Dimention
Binary and:	`$M & $N;`	`$M->band;`	`$M->bandover;`
Binary or:	`$M | $N;`	`$M->bor;`	`$M->borover;`
Binary xor:	`$M ^ $N;`		
Binary not:	`~ $M;` or `$M->bitnot;`		
Bit shift left:	`$M << $N;`		
Bit shift right:	`$M >> $N;`		
Logical and:		`$M->and;`	`$M->andover;`
Logical or:		`$M->or;`	`$M->orover;`
Logical not:	`! $M;` or `$M->not;`		

Trigonometric Functions

These PDL functions operate in units of radians elementwise over a piddle.

Operation	Elementwise

Sine:	`$M->sin;`
Cosine:	`$M->cos;`
Tangent:	`$M->tan;`
Arcsine:	`$M->asin;`
Arccosine:	`$M->acos;`
Arctangent:	`$M->atan;`
Hyperbolic sine:	`$M->sinh;`
Hyperbolic cosine:	`$M->cosh;`
Hyperbolic tangent:	`$M->tanh;`
Hyperbolic arcsine:	`$M->asinh;`
Hyperbolic arccosine:	`$M->acosh;`
Hyperbolic arctangent:	`$M->atanh;`

Statistical Functions

PDL contains many methods to obtain statistics from piddles. Double precision variants are prefixed with `d`.

Operation	Over entire PDL	Over 1st Dimention
Minimum value:	`$M->min;`	`$M->minover;`
Maximum value:	`$M->max;`	`$M->maxover;`
Minimum and maximum value:	`$M->minmax;`	`$M->minmaxover;`
Minimum value (as indicies):		`$M->minover_ind;`; `$M->minover_n_ind;`
Maximum value (as indicies):		`$M->maxover_ind;`; `$M->maxover_n_ind;`
Mean:	`$M->avg;`; `$M->davg;`	`$M->avgover;`; `$M->davgover;`
Median:	`$M->median;`; `$M->oddmedian;`	`$M->medover;`; `$M->oddmedover;`
Mode:	`$M->mode;`	`$M->modeover;`
Percentile:	`$M->pct;`; `$M->oddpct;`	`$M->pctover;`; `$M->oddpctover;`
Elementwise error function:	`$M->erf;`	
Elementwise complement of the error function:	`$M->erfc;`	
Elemntwise inverse of the error function:	`$M->erfi;`	
Calculate histogram of `$data`, with specified `$min`imum bin value, bin `$step` size and `$count` bins:	`histogram($data, $step, $min, $count);`	
Calculate weighted histogram of `$data`		

with weights `$weights`, specified `$minimum` bin value, bin `$step` size and `$count` bins:	`whistogram($data, $weights, $step, $min, $count);`
Various statistics:	`$M->stats;` `$M->statsover;`

The 'various statistics' described above are returned as a Perl array of the following items:

- mean
- population RMS deviation from the mean
- median
- minimum
- maximum
- average absolute deviation
- RMS deviation from the mean

Zero Detection, Sorting, Unique Element Extraction

Operation	Over entire PDL	Over 1st Dimention
Any zero values:	`$M->zcheck;`	`$M->zcover;`
Any non-zero values:	`$M->any;`	
All non-zero values:	`$M->all;`	
Sort (returning values):	`$M->qsort;`	`$M->qsortvec;`
Sort (returning indices):	`$M->qsorti;`	`$M->qsortveci;`
Unique elements:	`$M->uniq;`	`$M->uniqvec;`
Unique elements (returning indices):	`$M->uniqind;`	

Rounding and Clipping of Values

PDL contains multiple methods to round and clip values. These all opererate elementwise over a piddle.

Operation	Elementwise
Round down to the nearest integer:	`$M->floor;`
Round up to the nearest integer:	`$M->ceil;`
'Round half to even' to the nearest integer:	`$M->rint;`
Clamp values to a maximum of `$max`:	`$M->hclip($max);`
Clamp values to a minimum of `$min`:	`$M->lclip($min);`
Clamp values between a minimum and maximum:	`$M->clip($min, $max);`

Set Operations

PDL contains methods to treat piddles as sets of values. Mathematically, a set cannot contain the same value twice, but if this happens to be the case with the piddles, PDL takes care of this for you.

Operation	Code
Obtain a mask piddle for values from `$N` contained within `$M`:	`$M->in($N);`
Obtain the values of the intersection of the sets `$M` and `$N`:	`setops($M, 'AND', $N);` or `intersect($M, $N);`
Obtain the values of the union of the sets `$M` and `$N`:	`setops($M, 'OR', $N);`
Obtain the values which are in sets `$M` or `$N`, but not both (union - intersection):	`setops($M, 'XOR', $N);`

Kernel Convolusion

PDL supports kernel convolution across multiple dimensions:

Description	Code
1-dimensional convolution of matrix `$M` with kernel `$K` across first dimension (edges wrap around):	`$M->conv1d($K);`
1-dimensional convolution of matrix `$M` with kernel `$K` across first dimension (edges reflect):	`$M->conv1d($K, {Boundary => 'reflect');`
2-dimensional convolution of matrix `$M` with kernel `$K` (edges wrap around):	`$M->conv2d($K);`
2-dimensional convolution of matrix `$M` with kernel `$K` (edges reflect):	`$M->conv2d($K, {Boundary => 'reflect');`
2-dimensional convolution of matrix `$M` with kernel `$K` (edges truncate):	`$M->conv2d($K, {Boundary => 'truncate');`
2-dimensional convolution of matrix `$M` with kernel `$K` (edges repeat):	`$M->conv2d($K, {Boundary => 'replicate');`

Miscellaneous Mathematical Methods

Here is all the other stuff which doesn't fit anywhere else:

Description	Code
Elementwise square root:	`$M->sqrt;`
Elementwise absolute value:	`$M->abs;`
Elementwise natural exponential:	`$M->exp;`
Elementwise natural logarithm:	`$M->log;`
Elementwise base 10 logarithm:	`$M->log10;`
Elementwise raise to the power `$i`:	`ipow($M, $i);`