Perl 6 Now: The Core Ideas Illustrated with Perl 5

SCOTT WALTERS

Apress®

Perl 6 Now: The Core Ideas Illustrated with Perl 5

ISBN (pbk): 1-59059-395-2

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Chris Mills
Technical Reviewers: Randal Schwartz and James Lee
Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Dominic Shakeshaft, Jim Sumser
Project Manager: Tracy Brown Collins
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Production Manager: Kari Brooks-Copony
Production Editor: Janet Vail
Compositor: Kinetic Publishing Services, LLC
Proofreader: Linda Seifert
Indexer: Kevin Broccoli
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013, and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders@springer-ny.com`, or visit `http://www.springer-ny.com`. Outside the United States: fax +49 6221 345229, e-mail `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 7

Multidimensional Arrays

Corresponding to at least some of these [type names], there will also be lowercase intrinsic types, such as int, num, str, and ref. Use of the lowercase type name implies you aren't intending to do anything fancy OO-wise with the values, or store any run time properties, and thus Perl should feel free to store them compactly. (As a limiting case, objects of type bit can be stored in 1 bit.)

—Larry Wall, Apocalypse 2

Perl 5 arrays hold sequences of scalars, which can be strings, numbers, references, or the special `undef` value. The scalar variable is rich in metadata and has the right amount of power for normal use, but it isn't a good fit for representing individual pixels or samples in a large data set. It's just too much metadata, and it's metadata that isn't needed. This chapter covers an alternative for Perl 5 that works better in these cases. I'm calling the alternative a *multidimensional array* to distinguish it from Perl's built-in single-dimensional arrays, though it'd be just as correct to call it a *lightweight array*.

Perl 6 arrays can contain scalars, just like Perl 5, but they're also able to directly hold a list of numbers. An *array of numbers* saves memory as opposed to Perl 5's *arrays of scalars containing numbers*.

Perl 5 accomplishes this with the Perl Data Language (PDL) module, called `PDL`. (See the "Perl Data Language" section for a lot more details on this.) Perl 6 uses the normal array syntax combined with traits. Lightweight types are variables declared with the `int`s and `num`s type traits in Perl 6 or created using the `PDL` module in Perl 5. In both cases, the effects are the same; both do the following:

- Efficiently store large data sets
- Store multidimensional information
- Efficiently perform operations on all members of a multidimensional array

Image data is notoriously large when uncompressed, and of course it must be uncompressed to perform any meaningful operation on it. Data for sounds, data sets for statistics, and data from hardware data-capture boards are other applications that have in the past required programmers to drop down to C or step up to a specialized platform.

Perl eschews optimizations that save a little memory and require programmer attention. In Perl 6, lightweight arrays require very little programmer awareness, and they can save quite a lot of memory. Perl 5 takes some work, but either way, they open up worlds to Perl.

Besides efficiently storing large data sets, math and logic operations quickly consider the entire set of data. Working in parallel, standard operators, such as addition, work on each element of both sets, but other operators and methods exist to perform operations that are useful only on data sets, such as averaging.

New for Perl 6

Perl 6 offers lightweight and normal scalars, arrays, and hashes. Lightweight types may not have properties, but they require far less storage. Simple types have names in all lowercase letters.

```
# Lightweight storage, Perl 6
# simple values:
my int $scalar;
# Arrays
my int @array is Dim(1000, 1000);
# or:
my @array of int is Dim(1000, 1000);
```

Compare this to `my Int $array is Dim(1000, 1000)`, with `Int` having an initial capital letter.

■ **Naming variables** Please don't actually name your scalars `$scalar` or your arrays `@array`. Also, don't name them `$data` unless you're really unsure what they're going to hold, and then you've got larger problems to worry about than variable naming. Few programs are smart enough to process raw, typeless, free-format data in any meaningful way. Thank you.

Chapter 5 documented new features for the standard, nonlightweight variables. This chapter is about lightweight variables as used in arrays. You can use Perl 6 lightweight variables as scalars, but there's little reason to do so unless your program declares millions of scalar variables (in which case it has larger problems). However, if you did want a lightweight scalar, you'd declare it in Perl 6 with the syntax `my int $foo` or `my $foo is int`.

Lightweight types in Perl 6 don't accept properties or additional traits. The variables can't remember anything except their value according to the limits of the storage type. Specifically, they can't remember any meta-information about themselves, but the compiler tracks a certain amount while compiling (such as their datatype and scope). However, arrays containing lightweight types do accept traits, and the datatype being stored in the array is specified using them, such as with `my int @foo` or `my @foo is array of int`. Note that `my int @foo` means that `@foo` stores `int`s, not that `@foo` itself is an `int`; hence, it's the same as `my @foo is array of int`.

■ **PDL and undef** PDL Arrays don't recognize the special undef value in that they don't return it, and assigning it is equivalent to assigning in 0. However, PDL has optional bad value support with a similar purpose to undef. Unlike operations between undef and valid values, any operation between a bad value and any other value always results in another bad value. Bad values occupy individual positions in an array. Unless things go horribly wrong, an entire PDL array won't be a bad value, but it may contain several. See perldoc PDL::Bad for details.

Perl Data Language

> *In any event, it is absolutely my intent that the built-in array types of Perl 6 support PDL directly, both in terms of efficiency and flexibility.*
>
> —Larry Wall, to the perl6 language mailing list

PDL stores numbers compactly in RAM and operates efficiently on them. Arrays are composed of elements, and each element is the same size. Elements may be bytes, short integers, long (regular) integers, floats, or double-precision floats. Integers, short or long, may be signed or unsigned. Unlike normal Perl arrays, PDL arrays don't contain references, strings, or "tieable" (using Perl's tie feature, documented in `perldoc perltie`) values. None of the standard scalar metadata exists for elements of a PDL array.

Arrays created with `PDL` may have multiple dimensions. Whereas a standard Perl array accepts only one subscript, such as with `@arr[10]`, an array created with PDL can have any number of subscripts, such as with `$array->at(10, 5, 37)`. A one-dimensional array takes a single subscript and can be imagined as a single row of values. A two-dimensional array takes two subscripts and can be visualized as a square, with the first index traversing one axis and the second index traversing the second axis. A three-dimensional array is a cube, and so on.

Beyond a small amount of fixed overhead, the memory requirements of a PDL array are easy to compute: multiply each of the dimensions together, and then multiply the result by the number of bytes in each cell. A long integer is either 4 or 8 bytes depending on your hardware. A 1,000 by 1,000 array (a two-dimensional array) of `long` integers would be just short of 4 megabytes of memory, for example. An array of regular Perl scalar variables holding numbers would be at least eight times as large. They could grow much larger if used as strings, as Perl caches the string rendering of numeric values when non-PDL scalars are used.

Lightweight numeric types in Perl 5 and Perl 6 don't cache string representations of themselves. In Perl 6, `str`s don't cache numeric representations of themselves. In Perl 5 and Perl 6, normal scalars, including array subscripts, do cache numeric and string renderings of themselves for speed. Perl still automatically converts between integer and string representation for you, but applications that repeatedly use a single variable as a string and then as a number will suffer a performance hit compared with scalars. Lightweight variables can't store special values such as `undef`. These lightweight types do work with autoboxing, a feature introduced in Chapter 14. In Perl 5, multidimensional arrays can't have aliases or references taken to individual elements.

Some operators are *matrix operators*, which operate on two arrays according to the rules of matrix algebra. Some built-in methods, such as `average()`, *collapse* a PDL array by performing some operation on each array in the last dimension of the PDL array and by replacing the last dimension with the results of that operation.

Operators *vectorize* on PDL arrays. Adding one to a PDL array with `$pdl + 1` adds one to *every element* of the array. When an operator is used on two PDL arrays, the operation is performed on each *corresponding element* of the arrays.

A logic test operator, such as `<`, tests every element of a PDL array, either against a single value or against another PDL array. The `any()` method is defined to test whether some operation is true for at least one element of an array. The `all()` method is defined to test whether some operation is true for all elements of an array.

■ **Hyper operators** Perl 6's hyper operators, documented in Chapter 6, perform an operation on each element of an array. Normal operators used on `PDL` arrays cause that operator to behave as a hyper operator.

■ **Any and all** `any()` and `all()` in the PDL sense are similar to Perl 6 junctions, implemented in Perl 5 by `Quantum::Superpositions`. See Chapter 18 for more on this stuff.

PDL-Bundled Documentation

The `perldoc PDL` manual page lists the dozen or so other PDL manual pages available. Included and external modules operate on the data structures, doing statistical, arithmetic, and image-processing operations. Interfaces exist to windowing toolkits, such as `Tk` and `Prima`, that use `PDL::PrimaImage`.

`PDL::Impatient` is the best starting point to get a feel for PDL. Also see `PDL::Indexing`, `PDL::FAQ`, and `PDL::Indexing`.

The following are all the modules loaded by default when you `use PDL`:

`PDL::Core` documents the constructor and routines, providing meta-information such as dimensions and index sizes. Type conversion routines are also in `PDL::Core`. Type `perldoc PDL::Core` at the command shell to pull up the documentation.

`PDL::Basic` is noncore PDL array creation and reporting logic. Consider it an extension of `PDL::Core`.

`PDL::Ops` documents the overloaded operators and their method call equivalents. This tells what happens when you use +, *, <, and so forth, on two PDL arrays.

`PDL::Ufunc` contains routines that "reduce" PDL arrays. Information from two dimensions (one dimension may hold a huge number of parallel arrays) are totaled, multiplied, averaged, or subjected to other logical or mathematical operations and compiled into a single dimension. This is like adding columns in a spreadsheet. See also `PDL::Reduce` for writing your own functions to reduce data.

`PDL::Slices` lets you slice a multidimensional subarray to create a new PDL array from an existing one and has the routines to modify indexing. In its own words, it's "stupid index tricks." One PDL data structure, or *PDL array* (pronounced *piddle array*), may be a view of another, not requiring memory of its own. Most of this is beyond the scope of this chapter, but slicing is useful for getting data into and out of PDL arrays. See also `perldoc PDL::NiceSlice` for a source filter that provides a simplified syntax for slicing PDL arrays.

`PDL::MatrixOps` performs operations on matrices. This is done in signal processing and 3D graphics, among other things. By the way, if you have a processor with "multimedia extensions," it has a small matrix engine built in.

`PDL::Math` imports trigonometric functions such as `sin()`, old standards such as `ceil()`, and operations for polynomials.

`PDL::IO::Misc` reads and writes various scientific data formats, as well as raw and ASCII.

`PDL::Primitive` has an assortment of routines built by other operations or used to build other operations. If you can't find something you're looking for elsewhere, try here.

PDL has a Web site at `http://pdl.perl.org` with some excellent screen shots of 3D plots that are easy to do with a few commands and the right modules installed.

Creating a PDL Array

Use the `Dim` trait in Perl 6 or the `PDL->zeroes()` class method in Perl 5.

```
# These two are equivalent for creating a single-dimensional array
my int $array is Dim(1000);                 # Perl 6
my $array = PDL->zeroes(1000)->long;        # Perl 5 with use PDL
# These two are equivalent for creating a two-dimensional array:
my int $array is Dim(1000, 1000);           # Perl 6
my $array = PDL->zeroes(1000, 1000)->long;  # Perl 5 with use PDL
```

For the two-dimensional array, indices from `0, 0` through `999, 999` are valid, though PDL wants you to fetch data with `$array->at(999, 999)` rather than the more familiar `$array->[999][999]` syntax. You can get more dimensions by adding numbers to the list (and installing additional sticks of RAM to keep up with the exponential memory requirements of the additional dimensions).

You can build PDL arrays from existing data structures as well.

```
# Building a PDL array from an existing data structure
use PDL;
my $array = PDL->pdl( [ [ 0 .. 2 ], [ 3 .. 5 ], [ 6 .. 9 ] ] );
my $array = PDL->pdl( $perl_array_reference );
```

From here on in this chapter, everything is Perl 5 unless marked otherwise. Starting here, the examples also omit `use PDL;`. It's assumed to be there.

Type is one of `byte`, `short`, `ushort`, `long`, `longlong`, `float`, or `double`. If you're not familiar with C, `double` is a double-precision `float`, and a `longlong` is usually a double-wide `long`, but the smaller versions may be the same as the larger versions. Things are guaranteed to be of *at least* a certain size but are permitted to be larger.

A PDL array may be converted to any of the following types by calling a method of the same name as used to construct a type:

```
$array = $array->short(); # make into an array of shorts
my $new_PDL array = PDL->short( [1, 200, 1000, 15000] ); # new a PDL array of shorts
```

A `short` is a *short integer* that's usually 16 bits (half of what's required for a `long`), but it's quite possibly 32 bits (the same size as a `long`) and may be other sizes on truly bizarre systems. PDL's 32-bit `long` corresponds to Perl 6's `int`.

zeroes vs. pdl, long, short, and company `zeroes`, `ones`, and `sequence` initialize a PDL array to be of a certain dimension. The arguments are the desired dimensions. `pdl`, `short`, `long`, `ushort`, `longlong`, and so forth, expect the data itself to be passed in, and the resulting dimensions of the PDL array depends on the dimensions of the data. All of them construct PDL arrays, but the two groups take different information as arguments. Want empty space of a certain datatype? Do something such as `PDL->zeroes(100, 100, 100)->short()`, creating the empty space and then changing the datatype. Or, to avoid allocating memory twice, write `PDL->zeroes('short', 100, 100, 100);`. Just remember that `pdl` and company don't create empty PDL arrays.

Namespace pollution alert These examples pretend that `PDL` doesn't export any functions into your namespace, but `PDL` *does* export hundreds of symbols. If two modules try to export the same symbol, you get the one defined by the last module used. It's best to explicitly specify the package, writing `PDL->sequence()` or `$pdl->sequence()` instead of `sequence()`. If all your code is written in this object-oriented style, you may request that `PDL`, like most modules, *not* export symbols to your namespace. To do this, write `use PDL ();` instead of `use PDL;`. The empty list, `()`, is the list of symbols requested from the `PDL` module. While the various PDL operations are available as both methods and functions, I'm referring to them as *methods* under the assumption they'll be used as such.

Handy for debugging and experimenting, `sequence()` initializes a PDL array with a numeric sequence that keeps counting as it crosses rows. This creates easy-to-recognize data.

```
# sequence() creates easily recognizable data to test with
my $zero_to_ninety_nine = PDL->sequence(10, 10);
print $zero_to_ninety_nine;
```

This outputs the following:

```
[
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
 [40 41 42 43 44 45 46 47 48 49]
 [50 51 52 53 54 55 56 57 58 59]
 [60 61 62 63 64 65 66 67 68 69]
 [70 71 72 73 74 75 76 77 78 79]
 [80 81 82 83 84 85 86 87 88 89]
 [90 91 92 93 94 95 96 97 98 99]
]
```

Several examples in this chapter initialize PDL arrays using `sequence()` for simple demonstrations of other features.

Getting Data In and Out of a PDL Array

`set` stores a value at a given coordinate.

```
$array->set(int rand 1000, int rand 1000, 3);
```

This stores the value `3` at some random coordinate. A coordinate value is required for each dimension of the array with the value coming last on the argument list. The PDL-computed assignment operator, `.=`, writes in a way that modifies, rather than entirely overwrites, another PDL array. Here it's used with the `slice` syntax to assign into the middle of PDL. Use `=` to assign the result of PDL operations to a variable, ignoring the previous contents of the variable. Use `.=` to assign data into the `lvalue` result of other operations or replace data in a PDL that a variable contains a window into.

■ **Computed-assignment operator** The concatenation operator in Perl 5 is `.`, and `.=` is the mutating (assigning) version of that operator. When used with PDL, `.=` doesn't concatenate but replaces data. (The `.=` operator had to be used instead of the `=` operator for technical reasons related to operator overloading in Perl, and Perl 5 doesn't allow for the creation of entirely new operators.) This example happens to be using it to concatenate data, in a sense, even though it's really just assigning data into the middle of an existing buffer. See the `cat()`, `append()`, and `glue()` PDL methods for other concatenate-like behaviors.

`at` returns a scalar that holds a number. Note that PDL arrays contain only numeric data and never contain string data (short of encoding characters as bytes). A PDL array is usually zero initialized (as in the initialization example) or one initialized (useful for matrices).

`at` takes one argument for each dimension to traverse. The following picks a random element out of the PDL array just created:

```
my $random_element = $array->at(int rand 1000, int rand 1000);
```

This is equivalent to `my $random_element = @array[int rand 1000][int rand 1000]` in Perl 6 or Perl 5 using normal arrays and `Perl6::Variables`. Had I created a three-dimensional array, picking a random element would instead look like this:

```
my $random_element = $array->at(int rand 1000, int rand 1000, int rand 1000);
```

Each additional dimension means another argument. The first argument to `at()` traverses the first dimension, the second argument to `at()` traverses the second dimension, and so on. The dimensions are numbered *from the inside out*. When a PDL array is printed, the first dimension corresponds to values inside the innermost set of brackets, the second dimensional is the set of brackets just outside that, and so on.

```
# PDL coordinates are numbered inside out respective what is printed
# to the screen
use PDL;
my $pdl = PDL->zeroes(3, 3, 3);
$pdl->set(0, 1, 2, 100);
$pdl->set(0, 1, 2, 100);
print $pdl;
```

This outputs the following (without the comments):

```
[
 [
  [0 0 0]
  [0 0 0]       # This would read [0 0 100] if dimensions were
  [0 0 0]       # numbered reverse of what they are
 ]
 [
  [0 0 0]
  [0 0 0]
  [0 0 0]
 ]
 [
  [  0   0   0]
  [100   0   0]  # Here's the 0, 1, 2 coordinate
  [  0   0   0]
 ]
]
```

The coordinates `0, 1, 2` correspond to the 0th value in the first set of brackets in the second block. This arrangement is backward from what you may expect.

If you wanted to generalize the process of randomly subscripting an arbitrarily sized PDL array, use the `dims()` method without an argument, which returns the number of dimensions an array has. `dim($dim)` returns the size of any given dimension. You could `map { }` through a `rand` call, like so:

```
# Extract a random element from a PDL array using dims() and dim(n)
my $random_element = $array->at(
    map { int rand $_ }
    map { $array->dim($_) } 0 .. $array->dims()
);
```

This loops from `0` to `$array->dims()`. On each iteration, it fetches the size of that dimension using `$array->dim($_)` and builds a list of those dimensions. These sizes are mapped through `rand`. The result of this is used as the list coordinates of the actual random element.

■ **Indexing multidimensional PDL arrays** While `set()` and `at()` use lists of values to index a multidimensional array, many PDL methods (specifically those with names ending in `ND`) don't. The "Indexing Flattened PDL Arrays" section near the end of this chapter contains details.

`print` or anything else asking for a string representation of a PDL array gets data out in a way that's useful for debugging. Merely print the PDL array, like so:

```
# Dump a nicely formatted, human-readable rendering of a PDL array
print $array;
```

`list` sends back a nice, normal Perl array containing all the data in the PDL array. You'll want to do this with small PDL arrays, data sets after you've finished performing operations, or, more typically, a slice of a PDL array.

```
# Collapse all dimensions into one and return it as a plain Perl list
my $array = PDL->sequence(10, 10);
my @array = $array->list();
print "@array\n";
```

That's a roundabout way of writing `my @array = (0 .. 100); print "@array\n";`.

■ **The list() method** This method flattens multidimensional arrays. Perl 6 adds an operator for explicitly flattening arrays, *, introduced in the next chapter. The PDL list() method is similar. A flattened array has no structure (it's no longer organized into rows and columns) but is useful anywhere a list is accepted.

`set()` and `at()` are useful for experimenting with PDL to get a feel for the coordinate system (which was the point of this previous example), but they're not particularly useful for actually moving data into and out of a PDL array. For that, `slice()` and `range()` are used in conjunction with the `.=` operator to get data in, and `slice()` and `range()` are used with the `list()` method to get data out.

Let's say you wanted to work on a small piece of a larger PDL array.

```
my $array = PDL->sequence(10, 10);
my @array = $array->slice('1:8,1:8')->list();
print "@array\n";
```

A `slice()` creates a window into a PDL array, returning another PDL array. Except for the `slice()` method, this example is identical to the previous example. This `slice()` example shaves off the top row, bottom row, left column, and right column, cropping a border off from around the data. The slice is built from positions 1 through 8 in the first dimension and positions 1 through 8 in all the rows in the second dimension. A sliced-out PDL has the same dimensions as the PDL from which it was extracted. The format of the previous slice could be read as follows:

```
# Meaning of slice fields in the simple example
# This is not valid PDL
slice('dimension 1 start:dimension 1 end,dimension 2 start:dimension 2 end');
```

Slicing before taking a `list()` is an important idiom. I want to draw special attention to it, but it may be helpful to see what the slice actually looks like before it's flattened.

```
print PDL->sequence(10, 10)->slice('1:8,1:8');
```

These are simple examples of `slice()`. It has its own little metalanguage passed as its sole argument that describes *how* to take the slice. If you had a tile-based game with a large world map but wanted to show only a small portion of it at a time, `slice()` would be perfect.

When slicing off a multidimensional part of a PDL array, use the `flat()` method to create a single-dimensional PDL array. This is similar to `list()` except it returns a PDL array instead of a plain old list. This is the first part of an idiom for assigning a block of data into the middle of an existing PDL array.

```
print PDL->sequence(10, 10)->slice('1:1,:'); # needs to be flattened
print PDL->sequence(10, 10)->slice('1:1,:')->flat(); # that's better
```

Now that you know how to take a slice, the assignment operator, `.=`, will overwrite data in that slice with new data.

```
# Flattened slices allow easy assignment into a "window" into a PDL array
my $pdl = PDL->sequence(10, 10);
$pdl->slice('1:1,:')->flat() .= PDL->sequence(10);
print $pdl;
```

This assigns one single-dimensional sequence right into the middle of a multidimensional one. Rather than write out `'1:1,0:9'` in the argument to `slice()`, I've dropped the `0` and the `9`, and `slice()` defaults to picking an entire dimension where it sees a bare `:`. If you extract a PDL array of the dimensions `(3, 3, 3)`, you may assign another PDL array of the dimensions `(3, 3, 3)` into it without needing to flatten either PDL array.

The `range()` method generalizes the idea of slicing to accept the criteria of values to splice out as two PDL arrays.

```
# range() also extracts a PDL out of the middle of another PDL
my $pdl = PDL->sequence(10, 10);
print $pdl->range(PDL->pdl(2, 3), PDL->pdl(5, 5));
```

This generates the following output:

```
[
 [32 33 34 35 36]
 [42 43 44 45 46]
 [52 53 54 55 56]
 [62 63 64 65 66]
 [72 73 74 75 76]
]
```

The first argument to `range()` is the coordinates of the top-left corner of the area to extract, and the second argument is the number of elements required from each dimension. In this example, the first argument is `(2, 3)`, indicating two rows should be skipped and three columns should be skipped to seek to the desired block of data. The second argument is `(5, 5)`, indicating five rows and five columns should be returned. Rather than PDL arrays, you could use anonymous arrays, created with `[ ]`, to specify the location and size of the window to create. For dimensions greater than two, add more numbers, just as with `set()`, `at()`, and other PDL methods. Both arguments to PDL arrays could be single dimensional, and if they're multidimensional, the operation is repeated for each row. If the second argument is neglected, a PDL array containing a single value is returned, which has the same effect as if the second argument was passed and contained all `1`s. `range()` also takes an optional third argument that indicates how boundary conditions are to be handled. See `perldoc PDL::Primitive` for details. Various other PDL methods, such as `whichND()`, return coordinates, and you can use their output directly as input to `range()`.

Craig DeForest gave me an example of using `range()` to splice icons onto a larger image. If you're rendering a bunch of spacecraft icons onto a bitmap of Earth and its environs, for example, you can write the following:

```
$image->range($locations, $iconsize, 'truncate') .= $icon;
```

This places onto the image a copy of `$icon` at each location listed in `$locations`. In the simple case, `$locations` contains a single-dimensional PDL that holds the coordinates of where a single copy of `$icon` should be placed. `'truncate'` as a boundary condition tells PDL to ignore the situation of an icon placed partially off the image `$image`.

Processing Data in a PDL Array

The MOD file format (files with the extension `.mod`) was popular before Internet music went mainstream. Several music genres thrived in the MOD file scene, and the community surrounding the format has influenced several in turn. These files were originally associated with the SoundTracker program for the Commodore Amiga, but other programs on the Amiga and other systems were created to compose and play them. They're normally about a tenth of the size of an MP3 file. However, extremely large ones may be half the size of an equivalent MP3 file, and extremely tiny ones (a few kilobytes) were created as an amusement. They're composed of a music score, much like MIDI files, but they include their own samples, and MOD file authors love inventing new instruments and sampling things. The length of the file has little bearing on how long it plays for; it affects only the quality and the number of instruments. Play rate lets a sample be adjusted to different keys, and looping on the middle of a sample allows for different attack, sustain, and decay times to create realistic notes when a sample is used as an instrument. Numerous other effects can be applied to samples as they're played. Virtually all MOD files are licensed to be freely redistributed. Since they're composed of samples, it's easy to remix a track, change instruments, or even steal instruments you like from other MOD files. Some groups collect obscure and antique keyboard samples for this purpose. None of this is possible with MP3 files, so there's still an underground where MOD files thrive.

At the beginning of this chapter, I mentioned processing audio data as an example of where a program would need to deal with a large quantity of numeric samples. Throughout the rest of this chapter, I'll use operations on sound data as examples for PDL.

Representing Data in Memory

It's hard to implement a good MOD file player, or *tracker*, but it's easy to implement a bad one. Ultimately four channels of digital audio are scaled and mixed. The Commodore Amiga did this in hardware, but I wanted to mix it down to a WAV file.

```
# Compute the playing time of a mod file and multiply that by the sample rate to
# decide how much memory is required to hold the audio data.
my $wav_length =  int(scalar(@positions) * (64 / (60/10)) * $sample_rate) + 1;
# Create four tracks of byte resolution data
# large enough to hold the entire wav file.
my $audio = PDL->zeroes(4, $wave_length)->byte();
```

This program fragment allocates four tracks, each as long as what I'm computing the MOD file to be, taken from the number of bars of music, the number of positions in each bar, how many bars a second played, and the number of samples per second. Since I'm writing a plain old WAV file, I picked 44,100 as a good standard sample rate. So, 44,100 times a second, the speaker will be asked to move into a new position. That new position should be very near the last one,

unless something goes horribly wrong (such as when a data CD is put in a stereo by mistake). The result is a smooth, continuous sound wave. The sound wave itself is composed of lots of tones, high and low.

■ **Samples** *Sample* means both a chunk of audio data and an individual value from such a sample. It's also both a noun and a verb. Someone recording audio data takes samples of air pressure using a microphone. Data can be taken only at a certain rate (though it may be very high). Each time the air pressure is measured, it's a sample. To record sound data by sampling air pressure is *to sample* (the verb form of the word). From the verb form came the name for the result of taking samples: a *sample*.

Splicing in Data

The `mod` format reuses not only sample data but also musical score data. Each position specifies which frame is to be played. (A *frame* is similar to a bar of music.) Each frame has 64 ticks. Each tick, a new note can be started on a channel. If a note is already playing and no new one starts, the note continues playing. Ticks happen ten times a second. Four channels exist, and any may get a new note at each tick. The song structure amounts to several nested loops. The variable `$offset` keeps track of where the program is in the output PDL array, `$audio`.

```
# Loop over the MOD file data structures, splicing each note encountered into
# the wav data
use PDL;
use Perl6::Variables;
my $offset = 0;                       # Output wav data offset
my $offset_step = $sample_rate / 10;  # 10 times a second notes may change
for my $position (@positions) {
    for my $frame ( 0 .. 63 ) {
        for my $channel ( 0 .. 3 ) {
            my $note = @patterns[$position][$frame][$channel];
            if($note->{sample}) {
                # the note for this channel changed
                my $scaled_sample = scale_sample($note, \@samples);
                my $scaled_sample_length = $scaled_sample->dim(0);
                $audio->range(
                    [$channel, $offset], [0, $scaled_sample_length]
                ) .= $scaled_sample;
            }
        }
        $offset += $offset_step;
    }
}
```

This demonstrates using the PDL-computed assignment operator, `.=`, to modify data through a window in a PDL array. In this case, it's used to insert a block of data into the middle of a PDL array where the window into the array was created with `range()`. `=sample_rate` is the

number of individual samples (bytes) generated per second for output. `$offset_step` is the number of sample values needed to fill one-tenth of a second at the current sampling rate. Every frame, the program advances this much in the audio file. Notes always start on a frame boundary.

`$scaled_sample->dim(0)` returns the size of the first (0th) dimension, which is the length of the sample, as it's one dimensional. A following routine generates `$scaled_sample`. As one instrument sample is used to generate different notes, you must scale the sample up and down to change the pitch. `$scaled_sample_length` is the length of the sample being spliced in.

`@patterns` is a data structure representing the MOD file song structure. It's an ordinary Perl array of arrays (array of array references, technically), not a PDL array. When subscripted as `@patterns[$position][$frame][$channel]`, it returns the number of any note to be played at that moment in the song. The syntax of data structures or the structure of this data structure isn't important to this example.

Scaling and Oversampling Data

Part of this work is scaling samples. Since the samples in MOD files are usually sampled at a low rate, sampling them up is the normal case for this program. Samples are scaled to change the *pitch* (which is the primary frequency audible in a sample). Scaling the sample creates different notes from just one sample of an instrument, so they're not even scaled by a fixed rate. The amount of scaling changes with each note played.

This example, after obtaining the song data it needs, computes a number to scale the sample by (`$stretch_factor`), computes an interval with which to subsample the sample to shrink or expand it to the desired length (`$inverted_stretch`), builds an array of these subsample positions (`@sample_indices`), and then uses the PDL `interpolate()` method to interpolate values for the subsample positions from the audio sample data.

```
sub scale_sample {

    my $note = shift;
    my $samples = shift;
    my $period = $note->{period};  # speed to play at
    my $waveform = $samples->[ $note->{sample} ]->{waveform}; # audio data
    my $sample_khz = $amiga_constant / ( $period * 2 );
    my $stretch_factor = $sample_rate / $sample_khz;
    my $inverted_stretch = 1 / $stretch_factor;

    my $sample_index = 0;  # current subsample position
    my @sample_indices;    # compiled list of subsample positions

    while($sample_index < $samples_requested) {
        push @sample_indices, $sample_index;
        $sample_index += $inverted_stretch;
    }

    my $zero_to_a_lot = PDL->sequence($waveform->dim(0));
```

```
    (my $stretched_wav, my $err) =
        PDL->float(\@sample_indices)->interpolate($zero_to_a_lot, $waveform);

    return $stretched_wav;
}
```

The variable `$amiga_constant` is a constant known to MOD file players and editors. It's the clock speed of the Amiga's sound chip. Samples in the MOD file are set to be played at multiples of this speed.

Samples in MOD files are sampled at a pretty low rate, and it must scale these up—or perhaps down if it's playing a high-quality sample for a low-pitch note. `$stretch_factor` is a multiplier by the size of the sample. This number times the size of the input sample is the size of the desired output sample.

`$inverted_stretch` is the inverse of this. When stretching, it will be a floating-point number less than one. When shrinking a sample down, it will be a floating-point number greater than one. Since you advance through the input sample this amount each loop and it's less than one, you wind up partway between samples.

Truncating the values in `@sample_indices` (rather than using floating-point values and interpolating) would work. This would find the nearest sample, but a trick called *oversampling* can improve the quality of the low sample rate input sample by approximating the sample value between two actual samples. This example uses `interpolate()` to oversample. This technique isn't specific to audio processing. It's used in all fields of data analysis as well as computer graphics (for example, to smooth out image maps and texture data in 3D graphics). It uses some mathematical method to guess where a true waveform would go given the limited resolution of the sample data.

Figure 7-1 shows the effect supersampling has on audio data.

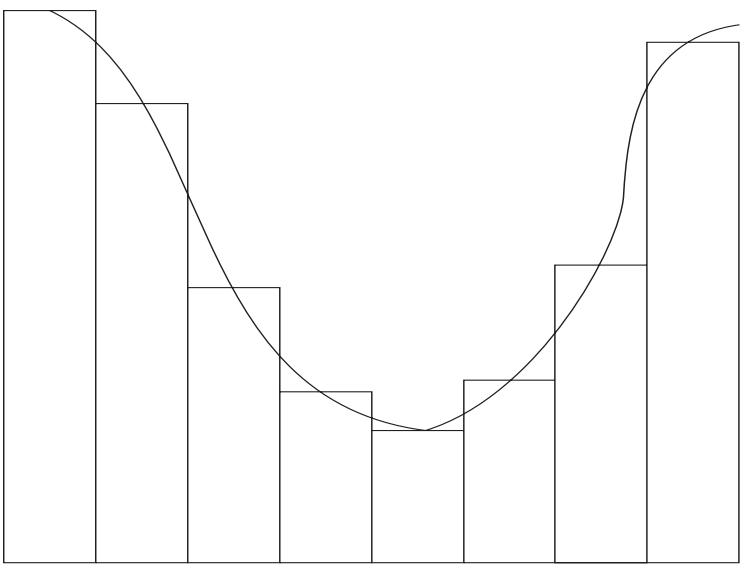

Figure 7-1. *Sine wave interpolated from discrete samples*

The rectangles are limited-resolution samples (the input). The line going through their tops is a *spline* that approximates the probable sound wave from which the sample was recorded. Some data is lost and there's no way to recover it, but this interpolated waveform avoids the buzzing noises, static, and speaker pops associated with low-resolution samples.

The following is a simple example of interpolation, based on the previous example:

```
# Interpolation smooths out data when the sampling rate is changed
use PDL;
# Subsample positions: output sample will have this number samples (6)
# from these places in the input sample (0.3, 0.6, 1.1, 1.5, 1.9, 2.0):
my @sample_indices = (0.3, 0.6, 1.1, 1.5, 1.9, 2.0);
# Common case is the input data represents a series of evenly spaced
# samples starting at 0 counting up to the number of samples in the
# input data.
my $waveform = PDL->pdl( [ map { int rand 10 } 1 .. 3 ] );
my $zero_to_a_lot = PDL->sequence($waveform->dim(0));
print "zero to a lot: ", $zero_to_a_lot, "\n";
print "waveform: ", $waveform, "\n";

(my $stretched_wav, my $err) =
    PDL->float(@sample_indices)->interpolate($zero_to_a_lot, $waveform);
print "stretched wav: ", $stretched_wav, "\n";
```

`$waveform` is randomly generated data. (It was three samples long in this example, and in the previous example it was a PDL array containing the low-resolution-input sample data.) In Figure 7-1, that's the blocky data that would sound crummy coming out of a speaker.

The floating-point values in `@sample_indices` subscript `$waveform`. `interpolate()` takes two arguments: the X coordinate data of subsample positions and the Y coordinate data PDL array of sample data. `$zero_to_a_lot` serves as the X data in these examples, and `$waveform` serves as the Y data. The `interpolate()` method is called as a method on the PDL array containing the subsample positions, which serves as X positions at which to subsample.

In both these examples, `$zero_to_a_lot` communicates to PDL that `$waveform` is a series of evenly spaced samples starting with the index `0` and counting up to however many samples you have (six in this case). The `sequence()` method computes this sequence as a PDL array. If the samples in `$waveform` weren't evenly spaced, `$zero_to_a_lot` would contain floating-point values describing the exact spacing between them, and a plain old sequence couldn't be used. If you were plotting temperature throughout the day, and you got up to log the temperature whenever you remembered, the values in `$zero_to_a_lot` would be the time of day you took a reading. Regardless, the values in `$zero_to_a_lot` must be in numeric order. This isn't really a handicap, but the Y data must be kept with the X data in case the Y data needs to be sorted.

`interpolate()` returns a PDL array with the cleaned-up, smoothed-out, higher-resolution waveform. This data also has limited resolution, but it's based on the infinite resolution of the waveform approximated through interpolation. You also get a flag telling you if any of the values were out of bounds. I'm ignoring this in the example.

Merging Multichannel Audio to One Track

After building the data structures and then going through the score and rendering the samples into the different channels, scaled and looped as needed, the last step is to mix them, and the `average()` method does just that. I've mentioned that operators and methods will perform an operation on each element of an array, perform operations on corresponding elements of two arrays, perform matrix operations between two arrays, and remove a dimension from an array. The `collapse()` method is a case of the latter: it removes a dimension.

```
# Average together four parallel tracks into one track and dump it as bytes
$wav->write( $audio->xchg(0, 1)->average()->list() );
```

`$wav` is a `wav` file opened with `Audio::Wav`. `list()` dumps that as Perl data. `xchg()` swaps the dimensions (logically, without actually moving any data). The following is the heart of that statement again, with comments:

```
# Average together four parallel tracks into one track and dump it as
bytes $audio                # 4 by 100,000 array two-dimensional array
      ->xchg(0, 1)   # 100,000 by 4 array two-dimensional array
      ->average()    # 100,000 element long single-dimensional array
```

Assuming the decompressed song is 100,000 samples long (it'd actually be much larger), `$audio` would be four long, narrow tracks with the dimensions `(4, 100000)`. The `xchg(0,1)` call rearranges the dimensions to be `(100000, 4)`, where there are thousands of arrays, each containing only four elements, with each element a single sample from each track. That gets it ready to invoke `average()` on. `average()` removes the last dimension of an array, leaving an array of the dimension `(1000000)`. In the simplest case, averaging a single-dimensional array, `average()` yields a zero-dimensional array (a single scalar). On a two-dimensional array, as in the MOD player example, it yields a single-dimensional array, containing a list of averages. The data must be averaged instead of totaled, or else the poor 8-bit bytes will overflow and the audio will buzz badly. This example dumps the entire PDL array all at once, first to Perl memory as a plain old list and then to file. This is unacceptable in cases where the PDL array is large, because the native Perl representation of the data is several factors larger than the PDL representation and both representations are kept in memory at the same time. Process the data incrementally; otherwise, when possible, use one of the `PDL::IO` methods (something I'll cover in the section "Comparing PDL Arrays").

Autoleveling

The MOD format was designed to be played easily by hardware. The Amiga has four-channel digital sound, and the sound chip will stream audio from memory at a variable rate and mix the four channels into stereo. Since I'm mixing in software, not hardware, I could use something larger than a byte to hold data for each channel and add notes, using addition, rather than assign over the top what's there. This strategy mixes as it goes. This approach doesn't require four separate audio tracks at all; it requires merely a datatype large enough to hold the numbers without overflow. A 32-bit signed integer can hold the sum of 8 million bytes.

```
# Demonstrate that 31 bits can hold the sum of 8 million bytes
use Math::BigInt;
print(0x7fffffff / 256, "\n");
```

That's plenty, and a short (16-bit integer) may be too little, safely holding only 256 overlapping notes, each note being 8 bits large. Any number of notes may be playing at the same time, and their values are all summed up, easily overflowing an 8-bit value, but you're supposed to be writing an 8-bit value. Even if I write a 16-bit WAV file, 32 bits could overflow it, but on the other hand, it would be barely audible as a 16-bit wave file. That presents something of a problem. When the data was on four tracks, you knew you could just average those tracks together, but now an arbitrary number of tracks have been merged with no count of how many were merged at the point where the most notes were playing at once. Autolevel to the rescue! The waveform, at its highest point, should be exactly the largest value in the output data, and the rest of the waveform should be relative to that. (However, the average volume could be raised if the tops of some of the largest waveforms were clipped off, but I'll leave this as an exercise for the reader, major record labels, and radio stations.) Autoleveling consists of two things: identifying the maximum value and scaling the rest of the data relative to that value. In PDL, this is easier done than said.

```
# Autolevel a single track of audio data
use Math::BigInt;
my $max_int = 0x7fffffff;   # 31 bits
my $max_level = $audio->max();
my $scale_factor = int($max_int / $max_level) - 1;
$audio = $audio * $scale_factor;
```

Take another `max()` reading on `$audio` after this scaling operation. It should be close to what `use Math::BigInt; print 0x7fffffff, "\n";` prints. You can create some sample data with something such as `use PDL; my $audio = PDL->sequence(1000);` to test this example. `$scale_factor` is the ratio between the largest value found in the data stream and the largest value you can represent with an unsigned 32-bit value. If unsigned shorts had been used instead of unsigned ints, then you'd have to use `0xffff` as the maximum constant. As always, judiciously using `print` statements will help you understand what's happening.

This approach requires changes to the other example. `$audio` is a one-dimensional array rather than two, and data is added to the `$audio` track rather than replaced into it. `+=` is used instead of `.=`.

```
# Modification to the "Loop over the mod datastructure" example to accommodate
# a single, scaled track
$audio->range( [$offset], [$scaled_sample_length] ) += $scaled_sample;
```

As previously, this creates a window into the data that's the size of the scaled note. Instead of assigning the scaled note in, it adds (with arithmetic addition) each sample. You no longer have to merge the tracks before writing the data file, because the code merges them as it goes.

Comparing PDL Arrays

PDL plots to numerous graphing libraries you probably don't have. It also writes the NetPBM formats (bitmapped, grayscale, and true color). NetPBM includes compressions and converters to generate GIF, JPEG, PNG, MPEG, TIFF, and numerous other formats. This makes sense because PBM is a go-between format. Rather than having GIF-to-PNG conversions, TIFF-to-GIF conversions, and so on, you need only converters to and from PBM.

```
my $audio_length = $audio->dim(0);
my $pbm = PDL->zeroes(255, $audio_length)->long;
my $steps = 254;
my $chunk = 0x7fffffff / $steps;
for my $i (reverse 0 .. $steps) {
   my $range_start = PDL->long(($chunk * $i) x ($audio_length - 1));
   my $range_stop  = PDL->long(($chunk * $i + $chunk) x ($audio_length -1));
   $pbm->range([$i], [$audio_length]) .=
       (($audio > $range_start) & ($audio < $range_stop));
}

$pbm = $pbm->xchg(0, 1);

$pbm->wpic("piddle.pbm");
```

`$steps` sets the Y resolution of the output image. The possible range of values is divided by this number. Each line of graphic output encompasses `$chunk` values. In other words, the image is drastically scaled down from the resolution of the actual data. You make `$steps` pass through the data, each time identifying everything in range. Two PDL arrays are created: `$range_start` and `$range_stop`. `$range_start` contains a PDL array as long as the audio data that contains the lower-range value repeated through its whole length. `$range_stop` is the same, except it contains the cutoff value for the line of graphic output. Using comparison operators, you can find everything that's between the two ranges.

```
$pbm->range([$i], [$audio_length]) .=
    (($audio > $range_start) & ($audio < $range_stop));
```

This draws in a row of the image where each pixel is off unless the wave data is in range. Remember that PDL arrays always operate on all their values at once, so no loops are required to do a single line of graphic output. `$range_start` and `$range_stop` are the same length (have the same size in their first dimension) as `$audio`. `<` and `>` each return a PDL array as long as their two input PDL arrays. The returned PDL array is composed entirely of `0`s and `1`s. Each output value is `1` if the corresponding input values have the described relationship. `&` also takes two PDL arrays of the same length and returns a PDL array, but the bitwise *and* operation is performed, not the logical *and* operation. This works exactly like the boolean *and* when input values are `0` or `1`. `PDL::Ops` describes logical, less-than, and greater-than operations, as well as others.

The following are some easier examples to help understand what's happening:

```
# Simple demonstration of vectorized logic operators in PDL
my $garbage = PDL->long(map { int rand 10 } (1 .. 20));
my $range_start = PDL->long((3) x 20);
my $range_stop  = PDL->long((7) x 20);
print 'garbage contains:    ', $garbage, "\n";
print 'garbage above start: ', $garbage > $range_start, "\n";
print 'garbage below stop:  ', $garbage < $range_stop, "\n";
print 'intersection:        ',
    ($garbage > $range_start) & ($garbage < $range_stop), "\n";
```

Since it's random, your data is almost certain to be different, on the first run at least, but the following is the first run for me:

```
garbage contains:    [0 9 1 7 8 5 8 2 9 5 3 0 9 9 2 3 8 9 7 4 9]
garbage above start: [0 1 0 1 1 1 1 0 1 1 0 0 1 1 0 0 1 1 1 1 1]
garbage below stop:  [0 0 1 0 0 1 0 1 0 1 1 1 0 0 1 1 0 0 0 1 0]
intersection:        [0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 1 0]
```

The `intersection` line is where the `garbage above start` and `garbage below stop` lines both have true values. For a bit to be set in the graphical output, the audio at any given X point must fall between the upper and lower bounds for the current Y line being drawn. Figure 7-2 shows a nice bass wave for your viewing pleasure.

Figure 7-2. *Example audio data plotted as an image*

NetPBM is a family of formats: PBM, PGM, and PPM. The `wpnm()` method takes one of those three strings as its second argument. PBM stands for *Portable Bit Map* (1 bit that's black and white), PGM stands for *Portable Gray Map* (black and white), and PPM stands for *Portable Pixel Map* (true color, 8 bits per channel, 24 million colors). `perldoc PDL::IO::Pnm` has more information on reading and writing PNMs. Get more information on the PBM family of filters from `http://netpbm.sourceforge.net` and `http://acme.com`.

■ **Signed and unsigned waveforms** Mixing signed and unsigned data causes the speaker to pop between samples, as zero is in a different place. This chapter uses unsigned data. Signed data is normal for the PC, but the Amiga uses unsigned. In the name of oversimplification, the examples here are unsigned.

Swapping Dimensions

PDL has an assortment of methods for exchanging dimensions, removing dimensions, and remapping dimensions. None of these is easily or quickly possible with normal arrays in Perl 5. `perldoc PDL::Slices` and `perldoc PDL::Core` document the methods.

`transpose()` swaps the rows into columns, rotating the data 90 degrees. This isn't what actually happens, but it's what effectively happens. The data doesn't actually move, but the metadata in the PDL array that contains information about the indices is updated, so transposing arrays is extremely quick in PDL.

```
# Transposing an array effectively rotates as columns and rows are swapped
my $zero_to_ninety_nine = PDL->sequence(5, 5);
print $zero_to_ninety_nine;
print $zero_to_ninety_nine->transpose();
```

This example, when run, outputs the following:

```
[
 [ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]
]

[
 [ 0  5 10 15 20]
 [ 1  6 11 16 21]
 [ 2  7 12 17 22]
 [ 3  8 13 18 23]
 [ 4  9 14 19 24]
]
```

`xchg()` swaps one dimension with another.

```
# Data may be reshaped by swapping dimensions
my $zero_to_ninety_nine = PDL->sequence(5, 5);
print $zero_to_ninety_nine;
print $zero_to_ninety_nine->xchg(0, 1);
```

Arguments are the numbers of the two dimensions you want to swap. They can be any two dimensions. If you have 100-dimensional data, you can swap dimensions 37 and 58, for instance. For a two-dimensional array, exchanging those two dimensions has the same effect as `transpose()`.

`reorder()` lets you specify the order of the dimensions, potentially moving all of them around. `reorer()` is from `PDL::Core`.

`clump()` merges several dimensions. It's documented in `PDL::Core`.

`dummy()` adds a new dimension to data after the fact. Use it to repeatedly to aggregate existing data with new data for use with collapsing operators such as `average()`. Dummy dimensions use no memory.

Indexing Flattened PDL Arrays

Indices in multidimensional arrays may be single values rather than lists of coordinates. The `at()` and `set()` methods take one argument for each dimension of the PDL array, but methods ending with `ND`, such as `whichND()` and `indexND()`, operate on a flattened PDL array, which may have been any number of dimensions before being flattened. `whichND()` and other functions return PDL arrays full of indices, and it's far easier to operate on flattened copies of a PDL array and represent each element's position as a single value rather than as a variable-sized list of indices. `flat()` makes a logical copy of the original that shares memory but merely has a different coordinate system, so there's no need to permanently throw away dimensions. The operation is extremely efficient. This is an advanced usage of PDL.

Earlier in the chapter, I gave the following example for selecting a random element from a PDL array with exactly two dimensions, each with a size of `20`:

```
# Previous example
use PDL;
my $array = PDL->sequence(20, 20);
my $random_element = $array->at(int rand 20, int rand 20);
print $random_element;
```

The following is a version that works on PDL arrays with any number of dimensions:

```
use PDL;
my $array = PDL->sequence(20, 20);
my $random_element = $array->flat->indexND(int rand $array->nelem());
print $random_element;
```

The `indexND()` method, documented in `perldocPDL::Slices`, operates on a PDL array and takes as arguments indices of values to return from the PDL array. This behavior resembles regular Perl array slices. `indexND()` also accepts a PDL array of positions to look up, in which case the PDL array returned contains multiple items.

Not Covered

PDL is worthy of a whole book. This chapter picks and chooses what *kinds* of things about PDL are most interesting.

One PDL array may serve as a window into another. The `slice()` syntax, among many others, can create a data structure shaped to provide a specialized view of the data while storing it in a format that another algorithm requires. You'll find this covered in the PDL documentation but not here.

PDL has built-in data persistence as it dumps to numerous formats used by other systems. See `PDL::IO::FastRaw` and `PDL::IO::Pnm`. `PDL::Impatient` mentions other I/O interfaces.

`PDL::MatrixOps` documents matrix operations. `PDL::Impatient` has a good, quick introduction to them as well. These are specialized in applications, and people who need them generally know they need them.

`PDL::Graphics::TriD` renders data as 3D objects using OpenGL or MesaGL in a window and lets the user rotate it in real time. See its documentation for details.

The `PDL` CPAN distribution installs a command shell, `perldl`. `perldl` interactively accepts commands, executing them immediately, features a history facility, and defines shortcuts. See `perldoc perldl` for information.

The PDL method `where()` allows operations to be performed on a subset of items from a PDL array. `where()` works with `<`, `>`, `==`, and other logic operators. It returns a PDL composed of items for which the logic condition is true, and the resulting PDL may be modified using operators and methods to alter the corresponding values in the original PDL array. See `perldoc PDL::Primitive`'s synopsis of `where()` for a code example that finds values over a certain value and caps them at that value.

Resources

Apocalypse 9 at `http://dev.perl.org` covers the mixture of lightweight storage and arrays as it fits into the Perl 6 world. PDL has its own site at `http://pdl.perl.org` with documentation and screen shots. CPAN's search tool at `http://search.cpan.org` lists page after page of modules

that work with PDL, interfacing the graphics packages, file formats, statistical systems, and other things. If something doesn't come with the main PDL distribution, CPAN is the next stop. Finally, `perldoc PDL::Math` has an excellent example you can try that plots a 3D graph using GL, OpenGL, or MesaGL.

This chapter was written to complement `perldoc PDL::Impatient`, not replace it, so I also strongly recommend reading that documentation. The examples on using overloaded operators alone are worth cracking the cover.

The PDL `any()` and `all()` methods work with matrix operators to allow the programmer to compute a large number of possibilities and then verify that all of them meet some criteria (for example, checking that nothing can go wrong and that all tolerances are within specification) or checking that one of them meets some criteria (for example, if done correctly, the goal is possible). While Chapter 18 deals with these ideas in depth, remember that you can use PDL for a similar sort of mass computation and evaluation of possibility sets.

PDL vectorizes operations, which fits with Perl 6's concept of hyper operators. Speed benefits aside, hyper operators simplify working on sets of data. An expression that can otherwise be written as loops nested two, three, four, or twenty deep becomes simple expressions involving an operator and two PDL arrays. Chapter 6 explains hyper operators.

Summary

PDL is a bit like a closet that's full of toys and in that way is a bit like Perl itself. Chances are good that you can find what you need, and even if you don't, you'll find a lot of other features you'll use later. It's evident that a huge amount of work went into PDL. I've outlined in the chapter's introduction a few domains that PDL helps with: digital signal processing, processing audio and graphic data, and statistics. But that's a simple-minded view. PDL's arrays complement Perl's native arrays perfectly and naturally have boundless uses in day-to-day programming. Any application processing batches of numbers is a candidate for PDL.

Interpreted languages such as Perl 5 aren't suitable for processing large batches of audio and video data directly. The virtual machine is suited to executing high-level, complex operations with good speed. Perl 5's virtual machine isn't suited to executing billions of tiny, low-level operations at blazing speed. Compiled languages, and to a lesser degree Just In Time–compiled languages, are perfect for this, and PDL is an XS Perl extension written in C. This enables PDL to do the tiny, low-level, repetitious number crunching quickly and frees Perl to do the high-level, complex operations.

You can consider PDL to be a language built on top of Perl. Commercial number-crunching environments offer many of the same primitives, but building an environment on top of Perl allows Perl's expressive syntax to shine through, and that's to say nothing of CPAN.

PDL is production-quality software and is widely deployed; it's used for research, data processing, and visualization.