PDL for impatient IDL users

Craig DeForest, deforest@boulder.swri.edu

Last rev: 21-Apr-2006

If you're used to using IDL, you know what you want to do but might find perl itself to be slightly confusing, because the language has a lot of elements that IDL does not. Here's a brief overview of the differences in syntax, to get you up to speed. It's oriented specifically toward IDL users who are trying out PDL to see what it can do. Basic syntax, variable types, file I/O, and string handling get special treatment.in this chapter, which is designed to get you going as quickly as possible.

The reference documentation that comes with perl itself is surprisingly easy to use. On a UNIX system, `man perl` will give you a table of contents, and a myriad of sub-pages such as `man perlfunc` will give you details on different parts of the language. Some of those sub-pages are tutorials that are designed to be easier to use than the reference pages.

PDL documentation comes as additional man pages, a help-and-apropos function at the `perldl` prompt, and a locally browsable HTML tree. All of the component software modules within PDL have command-line man pages. Within `perldl`, the PDL command shell, you can type "`?`<subject>" to get reference material on individual subjects, or "`??`<subject>" to get an apropos list of documentation that contains the keyword <subject>. You can point your web browser into the perl library tree in (by default) `/usr/local/lib/perl5/site_perl/` to find an online version of the same information.

1 Basic syntax

Like IDL, perl uses mainly imperative syntax: you tell a notional daemon what to do in small, successive steps. Unlike IDL, perl also has strong evaluative syntax: most commands return a value that you can use in an expression. Many of the basic constructs are the same, but the syntax is somewhat different.

The emphasis is somewhat different, too: while IDL distinguishes strongly between statements (e.g. procedure calls) and expressions (e.g. function calls), perl/PDL does not. Virtually everything in the language has a value, so you can make your code as procedural (FORTRAN-like) or as evaluatve (Lisp-like) as you prefer. In practice, the clearest, most maintainable code is usually somewhere in between those two extremes.

In addition to imperative and evaluative syntax you will find *pipeline* syntax as well. Pipeline constructs are reminiscent of UNIX shell pipelines, but using "method call" arrows instead of the vertical-bar character. For example:

```
$newvar = $var->clip(-1,1)->acos->pow(2)->sin;
```

is a synonym for

```
$newvar = sin( pow( acos( clip($var,-1,1) ) ), 2 ) ;
```

except that the pipeline form is much more readable, because you don't have to figure out the multiple layers of parentheses.

1.1 A Plea for Good Coding Style

Perl's flexibility means that you can duplicate the coding style of most other languages (including IDL), if you want to. It also means that you can write unbelievably weird and unreadable code if you set out to do so. Good coding is your responsibility! In certain circles, perl has a reputation for looking more like line noise than like programming. This can be attributed to two things: regular expressions (which do look cryptic until you know how to read them) and highly compressed coding style. It's important to code succinctly enough to be clear, and no more.

1.2 Command separators and blocks

In IDL, statements are notionally one line each, with a hack (the '&' connector) to connect them on a single line. Perl cares not at all for line breaks, so all commands are delimited by the ';' separator – just like C or Pascal. Conventionally, there is more or less one statement per line, but that is not enforced by the compiler.

Rather than `BEGIN` and `END`, perl uses `{` and `}` to mark the boundaries of blocks of code.

There's an exception to this, which is that when you are using PDL interactively, in the "perldl" shell, it usually executes whatever perl code you type as soon as you press <ENTER>. See 1.3, below, for details.

1.3 Multiline commands at the shell

In IDL, if you want to enter a multi-command codelet at the shell, you have to enter all the commands on a single line or else enclose them in a ".run" block. In PDL 2.4 and above, the perldl shell automatically senses whether you have any grouping constructs still open, and if you do it keeps accepting input until you close them. So (for example) you can cut-and-paste most code straight from a working script into the perldl shell.

This makes for some differences between the IDL and perldl shell behavior. For example, if you leave a quoted string open at the end of an IDL line of input, the IDL interpreter closes the string for you. If you leave a quoted string open at the end of a perldl line of input, the perldl preprocessor detects it and assumes that you meant to enter a multiline string (ie one that contains newline characters).

The perldl shell prompt looks like this: "`perldl>`".

The multiline prompt looks like this: "`..{ >`". The "`{`" character will be replaced by all of the nesting constructs that are still open (here, just a single brace).

You can abort a multiline command with control-D (the EOF character).

1.4 Variants of `if`: postfix-`if`, `unless`, and `?:`

The perl branch construct has more idioms than IDL's. In particular, there is an `unless` branch that acts exactly like `if`, but executes on false values rather than true. Furthermore, the IDL construction "`else if`" is contracted into a single word: "`elsif`". For additional natural-language flavor there's a postfix version of `if`/`unless` that works like the postfix in English (example: "Go to bed if it's after eight o'clock!" becomes "`go_to_bed() if( $hour >= 20 );`". The postfix structure is used to express (in the code) that you expect a particular statement to get executed most of the time, or to highlight important branchings in the code. An example:

```
while($a = <INPUT>) {    # Read lines from file INPUT, until it's empty.
  chomp $a;              # Kill the newline at the end of the line.
  push(@lines,$a) unless($a =~ m/^#/);
}
```

This loop reads in lines from a file and puts them into the array @lines, skipping lines that begin with the character '#'. The postfix `unless` expresses (to other programmers) that the `push` is the intended usual execution path.

You can use postfix-`if`/`unless` with multiple commands by including a `do` block:

```
do {<stuff>} if( <test> );      # alternative form of if
do {<stuff>} unless( <test> ); # alternative form of unless
```

There's another variant of `if` that C programmers will recognize: the ternary operator "`?:`" is an expression form of `if` that is especially handy for quick exceptions in assignments. For example, you might see the perl code

```
printf "Found %d %s\n", $i, ($i==1 ? "line" : "lines");
```

which handles singular/plural distinctions and is succinct. (You can read more about the ternary operator in section **??**

1.5 Loop constructs: `for`, `foreach`, `while`, `until`, `do`

perl has the usual collection of looping constructs. There is quite an array, so you can pick the one that's most convenient for you.

`for` loops The loop construct that is most similar to IDL's `for` loop is:

```
for $i(1..10) { <do stuff> }
```

which will assign the values `1` through `10` to `$i`, in sequence, and execute the commands in the block each time. There are several variants of `for`. In particular, you can replace the iteration expression with any old list of values:

```
for $i(@list) { <do stuff> }
```

iterates over all the elements of @list, in order, rather than over a numerical range. You can even specify the list explicitly:

```
for $i('foo','bar','baz',17,1) { <do stuff> }
```

will execute your code with `$i` set to each of those comma-delimited values, in order. Because this is similar to the behavior of the UNIX csh's `foreach` construct, you can also use "`foreach`" instead of "`for`" – they're synonyms..

Don't be confused if you encounter a loop with no explicit iterator at all! It's quite legal to say

```
for(1..10) { <do stuff> }
```

That will use perl's default operand `$_` as an iterator. Remember, `$_` works in most cases like a regular variable except that you can sometimes omit its name for brevity.

There's also a three-element version of `for` that follows the C language rather than FORTRAN:

```
for($i=0; $i <50; $i++) { <do stuff> }
```

This three-argument form of for has an *initializer*, a *test*, and an *incrementer*. The initializer sets up the loop, the test (which can be any Boolean expression) is evaluated at the top of the loop, and the incrementer is executed at the end of the loop.

`while` loops The `while` construct looks a lot like the IDL `while`:

```
while( <boolean> ) { <do stuff> }
```

`until` loops The `until` construct is just like `while`, but the sense of the boolean expression is inverted:

```
until( $i==10 ) { print $i++; }
```

will print "`0123456789`" (provided that `$i` is undefined to start with).

Post-checked loops and do To get a loop that always executes at least once, you can use do (which is similar to IDL's do):

```
do {<stuff>} while(test);
do {<stuff>} until(test);
do {<stuff>} if(test);     # alternative form of if
do {<stuff>} unless(test); # alternative form of unless
```

1.6 Loop exits: next and last

You can iterate a loop prematurely, or exit the loop entirely, anywhere within the loop block, by saying next or last. That's handy if you have several exit conditions and want to exit from the middle of the block of code: it prevents your "hot code" from being nested in several levels of if statements. Usually it pops you out of the innermost loop you're in, but you can exercise greater control; see the perl man page for details.

Here's an example:

```
for $i(1..10) {
    next if($i==5);
    print $i;
}
```

will print "1234678910", because the 5 case gets skipped by the next.

Neither next nor last works with do blocks, for reasons that are historical, but they do work in naked blocks, so you can enclose your do block inside another block to make them work properly; this is explained in more detail in the perl man page. If it doesn't make sense to you, you probably don't need to know.

2 Variables and Expressions

Perl variables come in four basic sorts: *scalars*, which hold a single value; *lists*, which can hold a bunch of scalars indexed by number (they're also called *arrays* but are quite unlike IDL arrays; see below); *hashes*, which can hold a bunch of scalars indexed by string (sort of like IDL structures, but more general and without the arbitrary restrictions); and *refs*, which are like pointers with built-in crash protection. PDL adds a new type of variable, also called a *PDL* or, in English, a "piddle", that is closely analogous to an IDL array: it is useful for holding millions of scalar values all of the same type (though you can also use PDLs to hold individual values). PDLs (the variables) are the bread-and-butter variable type for scientific computing with PDL (the language extension).

In IDL, you refer to variables by name alone. In perl/PDL, variables always have a *sigil* in front of them to identify both that they are variables, and what kind of value you expect to get out. In IDL, you'd say

```
A = 5              ; assign 5 to the variable A (short int by default)
```

```
B = indgen(5)     ; assign [0,1,2,3,4] to B (short int by default)
C = findgen(1e6)  ; assign [0..999999] to C (floats)
D = {a:1,b:2,c:3} ; Set D to be an IDL structure</PRE><P>
```

In perl/PDL, the analogous commands are:

```
$a = 5;      # assign 5 to the perl scalar A
@b = (0..4); # assign the list (0,1,2,3,4) to B
$c = xvals(1e6); # Assign the sequence 0..999999 to C.
%d = (a=>1,b=>2,c=>3); # Set %d to be a perl hash.
```

The sigils help sort out what's what, and also divide up the namespace – so you can simultaneously have a scalar variable `$a`, a list `@a`, a hash `%a`, and a subroutine called `a`. The syntax keeps them straight. That can seem confusing if you're used to the idea of "one identifier, one thing" in IDL – but putting sigils in front of the identifier expands the name space and prevents collisions like the infamous array–subroutine ambiguity in IDL. In general, scalar values are denoted by '`$`', lists are denoted by '`@`', and hashes are denoted by '`%`'.

A caveat: while `@a` is a list, its elements are themselves scalars – so `$a[5]` is element number 5 from `@a`, and `@a[5,6,7]` is a list of three elements from `@a`. @a[5] is a trivial list of one element from $a. In some cases there is a difference between a list of one element and the element itself (see §3.1)

As far as perl itself is concerned, PDL variables are special, *magic* (yes, that is a technical term) perl scalars. That is less confusing in practice than it may sound – it just means that, anywhere you can put a single perl scalar value, you can put a PDL that contains (in principle) millions of values. For example, you can create a perl list of PDLs, each of which contains a complete FITS image.

2.1 Scalars vs. IDL variables

IDL is a strictly typed language. Its variables have a particular type (e.g. `short`) that they retain until they are destroyed. Ordinary perl variables (not PDLs) are polymorphic: they change type according to use. So you can treat numeric values as strings and they work correctly; or you can use numeric values without regard for their initial type, and they will work correctly. This is useful, for example, in loops. In IDL, loops crash by default after 32,767 iterations. That's because loop variables are short integers by default, and after 32,767 iterations the loop variable overflows to become -32,768. In perl that never happens – the variable gets promoted to an appropriate type that can contain the value. Similarly, if you have a number and you want to put it into a string, you can just use it in a string context and you will get What You Want.

Like IDL scalars, perl scalars can represent the undefined value. Unlike IDL, perl deals gracefully with undefined values. You can test whether a variable is defined or not, using the built-in boolean function `defined()`, but if you don't bother and just use an undefined value in an expression, it will evaluate as the empty string, 0, or false, whichever is appropriate.

2.2 Lists vs. IDL arrays

Perl lists are designed for handling short to midsized collections of things. Each element of the list has a separate scalar (or ref) value, so that perl lists can be completely heterogeneous. Perl has provision for handling undefined values, so you can leave some elements in the middle of a list undefined. Lists have some very nice features that IDL arrays lack. In particular, you can index elements off the end of the list and they will be automagically created (with the undefined value). Or you can index elements from the back of the list instead of from the front, by using negative indices. In IDL:

```
A = INDGEN(10)
PRINT, A[10] ; this makes IDL throw an exception.
PRINT, A[-1] ; this also throws an exception.
```

In perl:

```
@a = 0..9;
print $a[10]; # prints nothing (element is undefined)
print $a[-1]; # prints '9'.
```

Notice that the elements have the '`$`' sigil rather than the '`@`' sigil. That's because we're asking for a scalar value.

2.3 PDLs vs. IDL arrays

PDLs are very similar in concept to IDL arrays: they have fixed sizes and types. But unlike IDL arrays, PDLs can contain out-of-band BAD values. You can fake this in IDL by sticking 'nan' into floating-point arrays; but the integer types have no bad-value flag. The bad value just marks a particular element as unusable for whatever reason; further calculations that use the bad value just silently return the bad value themselves, so that you can propagate missing values in your data. You can also index and modify PDLs with rather more facility than IDL arrays; that's described in the Chapter **??**

2.4 Hashes vs. IDL structures

Like IDL structures, perl hashes associate keyword/value pairs. The keyword is a string, and the value is something that could be contained in a variable. In PDL, as in IDL, each value can itself refer to any valid data structure in the language.

Perl hashes are considerably more flexible than IDL structures. In particular, if your perl code refers to a hash value that isn't present, you just get the undefined value, rather than throwing an exception (as it would in IDL). You can add new keyword/value pairs to your hash just by assigning to them with '=', rather than having to define a new hash. Here's an example. IDL code: add keyword/value pair ('`B`',`2`) to structure `A`:

```
W = WHERE(STRUPCASE(TAG_NAMES(A)) EQ 'B')
IF W(0) EQ -1 THEN ADD_TAG(A,'B',2) ELSE A.B = 2
```

perl code: add keyword/value pair ('B',2) to hash %a:

```
$a{'B'} = 2;
```

2.5 Perl refs

Refs are very easy to use in perl. They're a common way to "roll up" a complex data structure into a single perl scalar that you can hand around. They're impervious to most of the "gotchas" of C pointers, and they're more flexible than IDL pointers.

In general, you make a ref to something by naming the something, and putting a backslash in front of it. You dereference a ref by putting the appropriate sigil in front of the ref itself:

```
$aref = \$a;       # aref is a ref to the scalar $a
$A = $$aref;       # $A gets the value of $a (case-sensitive!).
$A = ${$aref};     # Another way to say the same thing, less ambiguously
$bref = \@b;       # bref is a ref to the list @b.
@B = @$bref;       # @B gets a copy of @b.
$B = ${$bref}[2]; # $B gets the element #2 of @b
$B = $bref->[2]; #  $B gets the element #2 of @b
```

The last form is common for both array elements and (with curly braces, as in '$hashref->{"KEY"}') hash values.

Perl keeps a reference count for everything living in its memory, so garbage collection is easy and automatic. You also never have to worry about dangling refs, because nothing is ever deallocated until its reference count reaches 0. In practical terms, data hang around until you have eliminated all access paths to them; then the data evaporate silently.[1]

Refs are useful any time you want to avoid making an extra copy of perl data (e.g. if you pass a ref into a subroutine you don't end up making a copy of the whole original list or hash or whatever), or any time you want to encapsulate a hash or list into a single scalar.

PDL arrays are implemented "under the hood" as refs to opaque objects with a language interface written in C.

3 Expressions

Perl expressions are far more powerful than IDL expressions, largely because of the branching constructs that are borrowed from C. In particular, the 'and' and 'or' operators (and their higher-precedence versions '&&' and '||') are lazy (they only evaluate the second argument if it will affect the truth of the result), so you can use them as

[1] Like all reference-based garbage collectors, perl's garbage collector gets confused by recursive structures. So if you have a collection of refs that points to itself, none of them will ever be deallocated unless you explicitly break the recursion before letting go of your last ref to the variable. This comes up surprisingly rarely, but deserves mention.

branching constructs; and (where appropriate) they return their first true argument, so you can very tersely say:

```
$val = $passed || $loaded || $default;
```

and `$val` will get the first true value between the three terms on the right-hand side. The equivalent IDL code is:

```
if ((size(default)(size(default)(0))+1) ne 0) then $
        value = default
if ((size(loaded)(size(loaded))(0))+1) ne 0) then $
        if (loaded ne 0) then value = default
if ((size(pass)(size(pass)(0))+1) ne 0) then $
        if (pass ne 0) then val=pass
```

The IDL code is complicated by the need to explicitly check the validity of each parameter, and the non-laziness of the boolean `.and.` and `.or.` operators.

Like C, perl includes a ternary operator that allows you to insert explicit conditionals into your expression. Some folks don't like it, but the ternary operator can make some odd assignments clearer. Here, the first line returns the arcsine of `$s` or (if `$s` is out of range) the arcsine of 1/`$s`. The second line returns an axis label string depending on whether the variable `$var` contains a FITS header.

```
$arcsin = (abs($s) > 1) ? asin(1/$s) : asin($s);
$title = (defined $var->hdr->{NAXIS}) ? $var->hdr->{CTYPE1} : '(Arbitrary)');
```

The presence of side effects can be surprising at first but is very helpful:

```
$a = $data++;
```

is more compact and clearer than the IDL equivalent:a = data

```
a = data
data = data + 1
```

Of course, you can do it that way if you want:

```
$a = $data;
$data = $data + 1;
```

3.1 Typing and context:

Since perl does behind-your-back typing for normal perl variables, each expression has a *context* in which it is evaluated. The context tells the expression what type is should be. For example, the left-hand argument of `&&` is evaluated in Boolean context, so whatever arithmetic or string value comes from the expression, it gets coerced into a Boolean value. Contexts are *void, scalar* and *list*. Void is the context for values that are

going to be ignored (e.g. function calls that don't actually do anything with their value – where, in IDL, you'd use a procedure call). The scalar context comes in several flavors: *Boolean*, *arithmetic*[2], *string*, and *ref*. The basic idea behind contexts is that most of the time they allow you to pretend that your perl variable or expression is whatever type is appropriate, and let the language take care of the casting and conversion details.

The largest context difference is between list and scalar context. Perl lists act differently depending on whether you are seeking a scalar or list value. For example, if you assign a list value to a scalar variable, the scalar gets the number of elements in the list:

```
@a = (4,5,6); # @a gets the list (4,5,6).
$a = (4,5,6); # $a gets the number of elements (3).
```

That conversion may sound confusing but it's actually quite useful when dealing with lists. For example, you can say

```
if($count == @a) { <do some stuff> }
```

to do some stuff if `@a` has exactly `$count` elements, or

```
if(@a) { <other stuff> }
```

to do other stuff if `@a` has any elements at all. (here, `if` is expecting a boolean value, so `@a` is evaluated in scalar context, yielding the number of elements. If the number is nonzero, it counts as true.

Because perl has no way of knowing if a perl scalar is meant as a number or string, there are different operators for comparing strings and numbers. The string comparison operators are the two-letter comparison operators that you're used to in IDL (like '`eq`'), while the numerical comparisons are more similar to the C language operators (like '==').

Using IDL-like comparisons is a trap for new users: `eq` converts its arguments to strings ("evaluates them in string context", in perlspeak), and then compares the strings lexically. This normally works OK, since the string representation of two equal numbers should themselves be equal. But it's very inefficient compared to == if you are trying to do arithmetic comparison. The related `gt` and `lt` operators don't work properly on numbers, because lexicographic (string) order is different than numeric order.

In general, use the <, ==, >, and <=> operators for numbers, and the related `lt`, `eq`, `gt`, and `cmp` operators for strings.

3.2 PDL variables and context:

PDLs are strongly typed: when you create a PDL, it gets a particular representation and stays that way. The basic types are similar to the IDL types: byte, short, ushort, long, ulong, float, and double. (Complex numbers are supported as a subclass of PDL;

[2] Arithmetic context has subtleties involved with integer and floating-point context. These work the same way as IDL expressions: all elements of an arithmetic expression are promoted to the highest-precision type.

see the appropriate section.) This fixed-type behavior is confusing to many perl users but should be familiar to you from your IDL experience.

Perl has three main contexts that affect the behavior of PDLs: *arithmetic*, *boolean*, and *string*.

Arithmetic context is what you get if you use PDLs in the usual way – adding, subtracting, and such. In numeric context, PDLs act "just like" IDL arrays: normal math operations act elementwise, and each array preserves its storage class (char/byte, short-int, long-int, float, double, etc.).

Boolean context is what you get if you use a PDL in a branch statement like `if` or `while` or even the `&&` and `||` operators. Multi-element PDLs are not allowed in this context; that is similar to IDL. Single-element PDLs are treated as TRUE if they are nonzero and FALSE if they are zero. That is the same behavior as practically every computer language on the face of the planet *except* IDL (which make even integers FALSE and odd integers TRUE), and INTERCAL[3], so don't be confused.

String context is what you get if you use a PDL with a perl operator that normally expects a string argument (like pattern matching or the `eq` operator or, most commonly, the `print` statement). When you use a PDL in string context, it's converted to a human-readable string suitable for printing. The string is, in general, not convertible back into a PDL without some effort. Because string conversion is intended for use with `print`, PDLs that are moderately large (more than 10^4 elements) don't get converted – the string that you get back is "`TOO LONG TO PRINT`". String context is easy to remember as "just" a way to give you direct access to the output of `print`: use a PDL as if it were a string, and you get the string that would be printed.

3.3 Assignments and Dataflow

PDL maintains a notion of "data flow", in which changes in the contents of one PDL automagically flow back to related PDLs. Used sparingly, data flow is both powerful and addictive. In combination with the slicing and indexing operators, data flow lets you parallelize tasks that are inaccessible from IDL's limited vector syntax (and that would hence require slow interpreted loops in IDL).

Lazy copying of data is one aspect of dataflow. Sometimes you want to pass around an array but don't want to make a copy of it. No problem: the normal assignment operator (=) works by making a duplicate pointer to the underlying data structure of your PDL variable, so (for example)

```
$a = $b;
```

is computationally very cheap: `$a` and `$b` now point to the same region of memory.

In addition, there's a separate elementwise assignment operator (`.=`) that actually copies elements into an existing data structure. So the statment

```
$a .= 5;
```

[3]INTERCAL stands for "Computer Language With No Pronounceable Acronym"; some claim that it is a joke, but others do apparently useful work in it. You might want to visit the online Intercal reference manual `http://www.muppetlabs.com/~breadbox/intercal-man/home.html`.

copies the value 5 into every element of `$a`. If you have just said `$a=$b;`, as above, then `$b` changes too. This is surprising to many IDL users, because IDL always makes explicit copies, so the two forms of assignment are merged into a single operation.

If you want an actual copy of a PDL, you can explicitly ask for one:

```
$a = $b->copy;
```

That is more computationally expensive, but isolates `$a` and `$b` from one another.

Dataflow works with most of the slicing and indexing operators, so that subfield operations work correctly:

```
$a->(5:7,2:3) .=0;
```

clears a rectangular subregion of `$a`, just like the related IDL command. But, because the dataflow relationship is preserved, you can do more interesting things:

```
$a = zeroes(3,3);
$a->diagonal(0,1) .= pdl(1,2,3);
print $a;
```

prints a diagonal (scaling) matrix:

```
[
 [1 0 0]
 [0 2 0]
 [0 0 3]
]
```

Think of dataflow as a generalization of the lvalue subfields in IDL. In IDL, you can say "`A(2:5,3:5)=0`" to assign a value to a subfield of the array A.

In PDL, that relationship is taken to its natural extreme: subfields of the PDL `$a` stay connected to `$a` unless you disconnect them explicitly:

```
$b = $a->(2:5,3:5);
$c = $b->(1,1);
$c .= 0;
```

assigns 0 to the (1,1) element of `$b`, which flows back to the (3,4) element of `$a`.

3.4 Headers and ancillary information

Headers: PDL variables contain more information than IDL variables. Each PDL has a "header" attached to it, for use in storing metadata about the PDL itself. The header is a hash ref (that is to say, a pointer to an array that's indexed by keyword; the IDL equivalent is a pointer to a structure). The header is not allocated until you use it for the first time. You can access the header with the `gethdr`, `sethdr`, `hdr`, and `fhdr` methods:

```
$a->sethdr(\%hdr);          # Set the header to an existing hash
$hdr_ref = $a->gethdr       # Get a ref to the header.
$hdr2 = $a->hdr_copy;       # Copy the header.
$a->hdr->{CTYPE1} = "Lon";  # Set a header value (safely)
$a->fhdr->{CtYpE2} = "Lat"; # Set FITS keyword (case insensitive)
```

`hdr` and `fhdr` automatically generate a header if there isn't one present: you just use it as if it exists already. If no header exists, then `gethdr` will return the undefined value. The only difference between `hdr` and `fhdr` is that `fhdr` creates a "tied hash" that takes the place of a hash but behaves more like a FITS header – in particular, it automatically includes `SIMPLE` and `END` cards, and the keywords are case-insensitive (normal hashes are case sensitive).

Inplace flag: In addition to header information, there's a flag (the "inplace" flag) that you can set and clear in the PDL using the `inplace` method. The inplace flag is used to prevent unnecessary memory usage: if you say

```
$a = cos($b);           # normal usage
```

then `$a` gets new memory allocated. Alternatively, you can say

```
$a = cos($b->inplace); # in-place usage saves memory
```

and `$b` will be modified in-place in memory. Rather than assign a new variable, you can even say just

```
$b->inplace->cos;
```

and `$b` will be modified in place. You use `inplace` anywhere that, in IDL, you'd use `temporary --` the difference is that `temporary` undefines its argument, while `inplace` keeps it around.

If you want to make your own functions `inplace`-aware, you can either check the inplace flag explicitly or else use the `new_or_inplace` method to allocate your output variable:

```
$out = $in->new_or_inplace;
... # (fill up $out here)
return $out;
```

3.5 Variable scoping

In IDL subroutines, all variables are local: there's no way (short of using a common block) to get access to global variables. In perl, all variables are global by default – if you want to hide them, you have to declare them local using the `my` modifier (as in "`my $pi = 3;`"). This is something of a trap for IDL users: it's easy to forget to localize your variables, and then scrozzle them somewhere else. On the other hand, this is a

real boon for writing quick-and-dirty routines. If you want an explicit reminder then you can say "`use strict;`" at the top of your subroutine or script – see the perl documentation for details.

As with everything else, you can fine-tune your variable scoping. The `local` modifier does run-time dynamic scoping, which comes in handy in special circumastances; and the `our` modifier makes a variable explicitly global.

3.6 Perl/PDL expressions and operators

Perl has more operators than IDL does, and each operator's behavior has more nuances than those of IDL. Here is a brief catalog most of the perl operators. (See the 'perlop' man page for more). They're listed in descending order of precedence.

Since the operators act slightly differently on PDLs and on perl scalars, they are listed separately. Most operators do more or less what you'd expect: they act elementwise where possible, and by stringifying the PDL where that makes the most sense. There are two notable exceptions: '`x`', the repetition operator, is actually matrix multiplication when applied to PDLs; and '`.=`', the string concatenation operator, has been pre-empted for an elementwise assignment **(see below)**.

Dereference operator ('->'): This is borrowed (and extended) from C's dereference-a-pointer-to-a-struct operator; it is a generic operator to dereference a method call or act on the object pointed to by a ref. Examples:

```
$v1 = $hash->{VAR1};            # Get item from a hash ref
$v2 = $array->[5];              # Get item from an array ref
$pdl->wfits("foo.fits");        # method call
$rad = $xy->pow(2)->sum->sqrt;  # Pipeline syntax
```

PDL also uses the single arrow operator to generate rectangular slices of large arrays:

```
$subfield = $bigarray->(2,3:5); # items 3-5 from col. 2
```

Don't confuse the single-arrow with the double-arrow operator that's used to identify hash fields! (below)

Autoincrement and autodecrement ('++', '--'): These act like the C language autoincrement and autodecrement operators: they return their argument and (as a side effect) either increment or decrement it. Depending on placement, the side-effect happens before or after the value is grabbed:

```
$a = 5;
print ++$a, ", ";
print $a++, ", ";
print $a, "\n";
```

will print "`6, 6, 7`".

Table 1: Perl operators and precedence, and their actions on PDLs. Items with a "*" mark are described in more detail in the text.

Prec.	Op.	Associativity (side of first grouping)	Action on perl scalars	Action on PDLs
1	()	NA	Grouping and function eval.	Identical to perl scalars.
2	->	Left	Dereference a ref or a method	Dereference a method or slice a PDL *
3	++, --	Either (Unary)	Increment/decrement in-place.	Elementwise *
4	**	Right	Exponentiation	Elementwise
5	!	Right (Unary)	Logical negation	Elementwise
5	~	Right (Unary)	Bitwise negation (bit flipping)	Elementwise - gets coerced to long int.
5	\	Right (Unary)	Reference operator	Identical to perl scalars.
5	+, -	Right (Unary)	Arithmetic no-op/negation	Elementwise
6	=~, !~	Left	String operation binding	Acts on stringified PDL
7	*, /, %	Left	Multiply, divide, and modulus	Elementwise *
7	x	Left	Repetition operator	Matrix multiplication *
8	+,-	Left	Addition and subtraction	Elementwise
8	.	Left	String concatenation	Acts on stringified PDL
9	<<,>>	Left	Bit-shift operator	Elementwise - gets coerced to long int.
10	Named	Right (Unary.)	Named operators - various	Acts on stringified PDL *
11	<, >, <=, >=	NA	Arithmetic comparisons	Elementwise *
11	lt, gt, le, ge	NA	String (lexical) comparisons	Forbidden
12	==, !=, <=>	NA	Arithmetic equivalence	Elementwise *
12	eq, ne, cmp	NA	String equivalence	Elementwise *
13	&	Left	Bitwise AND	Elementwise logical AND
13	|,^	Left	Bitwise OR and XOR	Elementwise logical OR and XOR
14	&&	Left	Lazy Boolean logical AND	Scalar (rejects multi-element PDLs)
15	||	Left	Lazy Boolean logical OR	Scalar (rejects multi-element PDLs)
16	.., ...	(none)	Range operators	Acts on stringified PDL
17	?:	Right (Ternary)	Ternary conditional operator	L:Scalar; M,R: identical to perl scalars.
18	=	Right	Assignment	Assignment-by-reference *
19	+=, etc.	Right	In-place arithmetic modifiers	Elementwise *
20	„=>	Left	List separators	Identical to perl scalars *

Arithmetic multiply, divide, and mod ('*','/','%'): The mod (`%`) operator is a true mathematical modulus rather than the more common sign-inverting variety: `$a % $b`. is always on the interval [0,`$b`) provided that `$b` is positive, and on the interval (`$b`,0] provided that `$b` is negative. PDL generalizes mod a little further even than perl does: perl `%` always coerces its arguments to integers, while PDL `%` allows floating point operands. Hence, this little wrinkle:

```
print 3.3 % 2.7, ", ";
print pdl(3.3) % pdl(2.7),"\n";
```

prints "`1, 0.6`". In general, if you use PDLs you get the more general behavior; but if you rely on it you must be careful always to use at least one PDL argument to `%`.

Repetition and matrix multiplication ('x'): When both arguments are ordinary perl scalars, 'x' is a useful repetition operator – e.g.

```
print "-" x 79;
```

will print a full line of 79 dashes – but PDL co-opts it as matrix multiplication. Matrices are addressed in (X,Y) order rather than (R,C) order – so they look correct on screen and the expression "$m x $colvec" acts correctly, but indexing is reversed from standard math textbooks. There is a subclass of PDL called "PDL::Matrix", that interchanges the first two dimensions so that you can index your matrices in (R,C) order if you want.

1-D PDLS are row vectors; you have to transpose them or insert a dummy 0th dimension to get column vectors. see **??**

Named perl operators: Perl has a large number of built-ins that can be called as either functions (using "name(arg1,arg2,...)" syntax) or operators (using "name arg1,arg2,..." syntax). When you use those functions as operators, they get this precedence and associate on the right – so "`print splice @a,2`" does the splice first and then the printing, just like you'd probably hope. In general, the built-ins are string operators so applying them to PDLs simply stringifies the PDL.

Arithmetic comparison and equality ('==', '!=', '<','>','<=','>=','<=>'): These operators act elementwise, so you get out an array of values. For the normal comparison operators, all the elements are either 0 or 1, indicating the truth of the corresponding comparison. The *spaceship operator* ('<=>') deserves special mention. It returns -1, 0, or 1, depending on whether the first argument is smaller then, equal to, or greater than the second argument. You can use these operators with multi-element PDL variables to do threaded comparisons, just not in a Boolean context (which needs exactly one scalar value).

String comparison and equality ('eq', 'ne', 'lt', 'gt', 'le', 'ge', 'cmp'): These operators do not apply directly to PDLs, but they deserve special mention because they resemble the numeric comparison operators in FORTRAN and IDL. The string comparisons operate on perl scalars, and use lexical ordering rather than numerical ordering. That's great if you're (e.g.) sorting file names, but horrible if you're actually trying to do numeric comparisons. Don't be confused.

Assignment by reference ('='): This is the normal assignment operator for perl, and when you apply it to a PDL it will assign the whole PDL at once, as a reference, without copying any actual data.

Loading your programs: .pdl files, .pm files, cut-n-paste

PDL offers two main ways to access code. There's an autoloader that acts similar to the IDL `!path` mechanism (but streamlined), and of course the usual perl module syntax. Although the autoloader has some advantages over the IDL autoloader, it is mainly used for one-off code and project-specific development. General purpose, polished code tends to get collected into perl modules that are explicitly loaded at compile time. Both styles are discussed here.

4 Loading your programs: .pdl and .pm files

4.1 Autoloader (.pdl) files:

The PDL autoloader acts similarly to the IDL autoloader and is a part of the perldl shell: if you are in perldl and run a subroutine (say `foo`) that isn't defined, then the autoloader searches your path for a file called `foo.pdl` and executes it in an attempt to load the subroutine. Note that, as in IDL, it's legal to have side-effects as the subroutine compiles!

PDL's autoloader differs from IDL's in three ways:

- First, every file that you load gets watched and reloaded as necessary, so you don't have to say "`.run foo`" a lot if you're doing development, as you do in IDL. The files are checked just before the command prompt gets printed. Since the checking involves reading directories only, it's fast enough not to affect operations in normal use – though you can switch off this behavior by setting a variable.
- Second, the search path allows metacharacters that shorten the path string considerably. If you want to include a directory *and all its subdirectories* in the search path, you can say "`+dir`" in the path.
- Third, PDL separates the language and shell functionalities (something IDL does not do). The autoloader is part of the shell, and not normally part of the language itself. You can use PDL from simple perl scripts just by saying

  ```
  #!/usr/bin/perl
  use PDL;
  ```

  at the top of the shell script. Then you can run the script from the command line (on UNIX-like systems) or by double-clicking it (on GUI systems). Such scripts do not have access to the autoloader by default. You can make them use the autoloader by saying

  ```
  use PDL::AutoLoader;
  ```

  near the top of the script. Such scripts have less control than many people would like over which module is being run, but they have the advantage that they can use the same ad-hoc autoload tree that you might put together for interactive work on, say, a particular data analysis project. The choice is yours.

4.2 Perl module (.pm) files

When your quick-and-dirty code stabilizes and you consider publishing it, you will probably want to package it into a perl module. Perl modules must be explicitly loaded with the perl `use` statement, but they make it easy to collect related routines into one place, along with their documentation. They also afford a degree of version control that is not possible with an autoloader design.

Object-oriented efforts are most naturally organized as modules; see the `perltoot` man page for details..

There are standardized forms for documentaton and version tracking within modules, and provisions for avoiding namespace pollution: the perl name space is hierarchical, so your module can hide its routines within its own part of the namespace, and not collide with other projects.

5 File I/O

I/O from PDL is extremely flexible. The basic I/O routines built into perl are designed to make your life easy for ASCII I/O – but there are several specific routines that help you read and write binary data, FITS files, and most standard image formats.

5.1 Basic perl I/O

In IDL, you use FORTRAN file numbers to describe files on disk. In perl, you use *filehandles* – ASCII identifiers that are associated with the file. Filehandles have no sigil, unlike variable names. They are case-sensitive. By convention, they are written in all-caps, to distinguish them from reserved words and such. When you start your perl program you have three filehandles open: `STDOUT`, `STDERR`, and `STDIN`. UNIX users will recognize those as the names of the three standard UNIX streams.

The perl built-in file handling functions are described in detail in the perl documentation. The basic function calls are `open` and `close` for file handling, `print` and `printf` for unformatted and C-style formatted ASCII output, the "`<FILE>`" operator for unformatted ASCII input, `write` and `read` for fixed length (ala FORTRAN) ASCII I/O, and `syswrite` and `sysread` for unformatted binary I/O.

6 Images and collections of images

Image I/O, including FITS files, is via `rpic` and `wpic`, generalized image read/write routines that handle most standard formats including PNG, JPEG, GIF, PPM, FITS, Encapsulated PostScript, and others. Some simple ways to read and write FITS files are:

```
$a = rpic("filename.fits");         # Works for other file formats too
$a->wpic("another-filename.fits"); # method syntax
wpic($a,"another-filename.fits");  # function syntax
```

You can read in a long list of FITS files with:

```
@files = <directory/*.fits>;    # Example of UNIX-style globbing
@imgs = mrfits(@files);         # @files is a list of file names.
$cube = rcube( \&rpic, @files );
```

The output of `mrfits` is a perl list, each element of which is the corresponding FITS file loaded into a PDL variable. The output of `rcube` is a single data cube loaded with all of the images stacked in the highest dimension, so it is more useful for a collcetion of uniformly sized files.

Unlike IDL, PDL can keep a metadata header attached to each variable, so there is no need to keep track of an additional header variable for each image you load. To access fields in `$a`'s FITS header in the example above, use (e.g.) "`$a->hdr->{TELESCOP}`".

6.1 Raw binary data

There are several ways to manipulate raw binary data with no in-file header at all. One way is with the `PDL::IO::FlexRaw` module, which defines the commands `readflex` and `writeflex`. see Chapter **??** for details.

6.1.1 Arbitrary collections of data

There are at least two simple ways to dump a collection of PDL variables to disk for later retrieval. The method closest to IDL's save-file format is with the `PDL::IO::Storable` module, which lets you write and read collections of variables to disk in a fast, opaque, binary format, using the functions `store` and `retrieve`. The `PDL::IO::Dumper` module translates arbitrary perl data structures into perl source code, which is a convenient quasi-human-readable format that is completely portable, but much slower and larger than the `Storable` format.

7 Other useful tools

7.1 PDL::Transform

PDL contains a module, "PDL::Transform", that handles general coordinate transformations on data. FITS headers are used to separate the pixel and scientific coordinate systems. PDL::Transform defines a new type of object, a Transform, that represents a coordinate transformation; you can apply a Transform to a list of vectors (in which case they are converted to the new system) or to an image (in which case it is resampled to the new coordinate system).

PDL::Transform uses FITS headers to keep track of the scientific coordinate system independently of the pixel coordinate system of image arrays, so you can mix and match data from independent sources, provided that each data file has a WCS FITS header (most solar observatories include that these days).

7.2 Graphing/plotting

PDL has several graphical front-ends; take your pick. The currently most popular one is **PDL::Graphics::PGPLOT**, which uses the venerable but stable PGPLOT routines; the most actively supported one is **PDL::Graphics::PLplot**, which uses the newer PLplot library (that runs faster and supports RGB color better). Three-dimensional graphics are supported via the OpenGL library, using **PDL::Graphics::TriD**.

7.3 Solar tools

SSWPDL contains a few solar-physics-specific utilities that you might want. You can browse the **auto** tree to find them, but these specific tools are particularly handy:

t_derotate_image and **t_diff_rot** are Transform constructors that are handy for co-aligning images.

pb0r and **sun_pos** duplicate the corresponding functionality in the rest of SSW.

align_and_center provides sub-pixel co-alignment of image sequences.

zspike implements the ZSPIKE time-domain despiking algorithm (which works not only for EUV and FUV solar images but also for more challenging data such as SOHO/MDI magnetograms).

units converts between familiar SI , CGS, and other physical units.

amoeba implements a simple parameter fitter.

rk4 is a Runge-Kutta integrator.