Practical Magick with C, PDL and PDL::PP – a guide to compiled add-ons for PDL

Craig DeForest, Karl Glazebrook

February 28, 2017

Preface

This guide is intended to knit together, and extend, the existing PP and C documentation on PDL internals. It draws heavily from prior work by the authors of the code. Special thanks go to Christian Soeller, and Tuomas Lukka, who together with Glazebrook conceived and implemented PDL and PP; and to Chris Marshall, who has led the PDL development team through several groundbreaking releases and to new levels of usability.

Contents

1 Introduction

PDL uses C to make "the magic" happen. PDL variables are represented internally with C structures that carry metadata and the actual data around. You can access those directly with C code that you link to Perl using the perlXS mechanism that is supplied with Perl.

Handling PDL variables directly in C can be tedious, because it involves variable-depth nested looping and direct (and fiddly) pointer access to variable-type arrays. So most PDL functions are written in a metalanguage called PP that wraps your C code snippet with a concise description of how a PDL operator should work. PP is compiled by a macro preprocessor, contained in the module PDL::PP. PDL::PP generates Perl and perlXS code from your concise description; that code is then compiled and dynamically linked using perlXS. We use the word *operator* to distinguish PP-compiled code from a Perl (or XS) *function*. A PP-compiled operator can include several ancillary Perl and/or C functions.

PDL and related vectorized languages like IDL or NumPy work fast because they automatically vectorize large operations out of the high level interpreter: while the language itself is processed by an intepreter, the "hot spot" operations at the core of a scientific calculation run in faster compiled C loops. But all of them are generally slower than custom written C code. The reason for that is all vectorized languages access memory in *pessimal* order. If you write an expression like "`$a = sqrt( $b*$b + $c*$c )`", for example, then all of the elements of $c are accessed twice and written to a temporary variable; then likewise for $b; then all elements of both temporary variables are accessed and the result is written to yet another temporary variable, which is accessed and processed with the `sqrt` operation – making 11 passes through your data. If the data are large enough, then every single operation will break your CPU's cache and require (slow) access to the system RAM. The same operation, written in PP, can carry out the root-sum-squares on each element in order, preserving CPU cache and running *several times* faster.

The point of PP is to make writing custom operators in C *inside* the vectorized threading loops nearly as quick and easy as writing Perl expressions. PP handles most of the argument processing and looping logic that is required to make PDL operations work, so that you only write the interior of the loop. PP code is written in C with some extra macros to access the data and metadata of the arguments. With a small amount of effort, you can dash off new operators almost as fast as you could write them in PDL itself. The point of this document is to give you the ability to do that.

This document is intended to be a comprehensive introduction and reference for PDL data structures and the PP language, unifying and extending much of the half-written documentation that already exists (late 2013). Since PP is basically C, you should be familiar with Perl, PDL, and C. You should also have at least read the *perlguts* man page and preferably be familiar with XS (*perlxs*), which is Perl's interface to C.

1.1 Review of PDL Dimensions

PDL (the language) manipulates structured-array objects. Each PDL/piddle (object) has a *dimlist* that lists 0 or more dimensions and a size for each one. If the dimlist has 0 elements, the piddle is a "scalar piddle". Each element of the dimlist is the size of the corresponding dimension of the array, and can be 0 or more. If any element of the dimlist contains 0, then the piddle is *empty* and contains zero elements.

There is a special case – a *null piddle* – that contains no dimlist at all. Null piddles are placeholders, used mainly for output in PP functions. When PP code encounters a null piddle it automatically resizes and allocates the piddle to the appropriate size for the current operation, based on the shapes of the other piddles in the same function call.

Each PDL function or operator operates on objects with some dimension – 0, 1, 2, etc. These dimensions are taken from the *front* of the operands' dimlists, and are called *active dimensions* or *active dims* of the operator to distinguish them from *thread dimensions* of a particular operation (Thread dimensions are looped over automagically by the PP engine, which is what makes PDL a *vector* language). Scalar operators like + have 0 active dimensions. Matrix multiplication has 2 active dimensions. Operators with more than one argument can have different numbers of active dimensions for each argument – for example, `index` has one active dim in the source variable (since you must have a dimension to index over) and zero active dims in

the index variable (since the value used to do the indexing is itself a scalar). Further, the output can have a different number of active dims than the input. The *collapse operators*, like `sumover` and `average`, have one active dim on their input and zero active dims on their output.

Properly written PDL operators access their arguments as if they only had the active dimensions. A scalar operator treats its arguments as if they were scalars. A 1-D operator treats all of its arguments as if they had only a single dimension, etc. Higher dimensions, if any, are called *thread dims* because they are matched up and handled by the threading engine.

1.2 Threading Rules

Threading is a form of automatic vectorization. After the active dimensions in each argument to an operator are accounted for, any remaining dimensions are looped over by the PP engine. These dims are called *thread dims*. All the thread dims in a given operator call must match. They are handled with three simple rules:

1. *Missing dimensions are added, with size 1.* This deals with a basic ambiguity between an array of length 1 and a scalar, by implictly extending low-dimensional arrays out into higher dimensions with size 1.

2. *Dimensions of size 1 are silently repeated.* This extends the concept of scalar operators in a vector space (e.g. scalar multiplication, scalar addition).

3. *All dimensions that are not size 1 must match sizes exactly.* This prevents trying to add, say, a 2-vector to a 3-vector, which doesn't make sense. Note that $0 \neq 1$, so it is possible for an empty piddle to fail to mix with non-empty piddles.

The threading rules are implemented by code generated by the PP suite for each PP function/operator. The PP code "knows" the difference between active and thread dimensions by inspecting the function's *signature* (§2.1), which is a compact way of expressing the structure and dimensionality of an operator.

There is an additional rule of threading: under normal circumstances there should be *no causal link* between different iterations of the threadloop – i.e. the different computations carried out by the thread loops must be completely independent of one another. This leaves the threading engine open to use multiple CPU cores for a large thread operation. (If you must break this rule for a particular operator, it is possible to mark the operator as non-parallelizable.)

1.3 What is PDL::PP?

PDL::PP is a collection of codewalker/compiler modules that compile PP definitions into XS and Perl code. The XS code normally then gets treated with *perlxs* and turned into C code, which in turn gets compiled and linked into a running Perl.

Because PP is a *compiler* and not a runtime interpreter, the compiled code runs quickly. But it also has certain limitations that aren't present in an interpreter. For example, PP mocks up dynamic typing. It keeps track of a single data type of the entire calculation, and autopromotes all the arguments to the type of the strongest one (except as you specify, see §2.1.3). Because the dynamic data types are resolved into a single type before your code gets called, PP can simply compile multiple copies of your code, one for each possible data type. When the Perl side user calls your PP operator, all the arguments get regularized and the appropriately-typed version of your compiled code gets used. Each PP operator gets compiled into a *family* of C functions that implement it.

Like Perl code, PP operators can be written into a standalone module, or placed inline in a Perl script with the `Inline::PdlPP` module.

PDL::PP is necessary to *build* PP modules, but not to *run* them – though this distinction is blurred if you use `Inline::PdlPP` to insert PP code directly in the middle of a Perl script.

1.4 A trivial example: a linear function

Here's a simple PP routine that performs linear scaling: given input variables a, b, and c, it returns the value $ab+c$. This function is nearly universally useful: with appropriate values of b and c it can convert between Fahrenheit and Celsius, scale between samples of different population size, or (if b is a gain matrix and c is a dark image value) flat-field a scientific image. The following is a snippet of *Perl* code, suitable for pasting directly into a Perl/PDL script or autoload file. It invokes the module `Inline::Pdlpp` to insert code in a different language (PP) directly into your script. The PP code is automagically compiled and linked into your running Perl.

```
no PDL::NiceSlice;  # prevents interference
use Inline Pdlpp => <<'EOPP';
pp_def('linscale',

    Pars => 'a(); b(); c(); [o]o()', // Pars (signature) specs threadable arguments.
    Code =><<'EOC',
    $o() = $a() * $b() + $c();

EOC
);
EOPP
*linscale = \&PDL::linscale; # copy the defined code into the current package
```

The Inline code declares a PP operator called "linscale" that is defined in the PDL package. definition takes the form of a call to a function called `pp_def`, which assembles an operator called `PDL::linscale`. The `pp_def` constructor accepts an operator name, followed by a hash containing declarative fields. The minimum fields required are the `Pars` section, which contains a concise description of the parameters, and the `Code` section, which contains the actual code to build.

This `Pars` section specifies three input parameters with zero active dims each; and one output parameter. The user can pass in either three PDLs or four. If the fourth (marked `[o]` for output) is omitted, then the operator will autocreate it and return it. The `Code` section contains a small snippet of C code with macro substitutions, that carries out the operation for a single iteration of any thread loops. In this case the C code snippet is very simple indeed: it just carries out the linear operation and stuffs the result into the output variable.

You can call this function just by invoking it, as in:

```
print (  linscale(pdl(1, 2, 3), 2, pdl(4, 5, 6))  )."\n";
```

which is equivalent to

```
print (  pdl(1,2,3)*2 + pdl(4,5,6)  )."\n";
```

in normal PDL. Either one will generate the output `[6 9 12]`. The PP version of even this trivial example runs slightly faster, because the high level expression generates several temporary variables while the PP code walks through the input parameters exactly once.

1.5 A non-trivial example: Mandelbrot set visualization

Here's a fast example PP routine for calculating something useful: whether a particular complex number is in the famous Mandelbrot set, or no. The Mandelbrot operator on a complex number z is given by $M_i(z) = M_{i-1}(z)^2 + z$, with $M_0(z) = z$. The Mandelbrot set is the set of all z such that $|M_\infty(z)|$ is bounded. Visualizing the Mandelbrot set requires testing each pixel for membership in the set, by iterating M to some large i. You could make a visualizer with simple threading of elementary operators like this:

```
sub thread_mandel {
```

```perl
        my $z = shift; my $cxy = $z->copy; # expects N=2 in dim 0 (real,imaginary)
        my $n = shift // 1000;              # max iteration count -- defaults to 1000.
        for my $i(0..$n) {
            my $tmp = $cxy * $cxy;
            $cxy->((1)) *= 2 * $cxy->((0));         # imaginary part
            $cxy->((0)) .= $tmp->((0)) - $tmp->((1));  # real part
            $cxy += $z;
        }
        return ((($cxy*$cxy)->sumover - 4)->clip(0)->log);

    }
```

You feed a 2×W×H array of complex numbers to `thread_mandel`, and it returns a W×H PDL indicating whether each input number *might* be in the Mandelbrot set. If the return value is 0, the corresponding input number might be in the set; if it's positive, the corresponding number is out of the set, and the value is a measure of its distance from the set. The routine uses the threading engine to operate on an entire complex "image". This threaded example works much faster than a Perl script, but much slower than it could.

Ideally, we'd like the threading engine to do all the operations for each point at once (and *then* move on to the next point), to preserve CPU cache. Also, we'd like it to stop with each point as soon as it can, to save time on points that diverge fast. Using `Inline::Pdlpp`, we can do just that – from within our Perl script! Right under the `thread_mandel` declaration (or anywhere else in our script or module), we can write this (notice that we "`use Inline`" with arguments, rather than calling `Inline::Pdlpp` directly):

```perl
    no PDL::NiceSlice;  # prevents interference
    use Inline Pdlpp => <<'EOPP';
    pp_def('pp_mandel',

        Pars => 'c(n=2); [o]o()',     // Pars (signature) specs threadable arguments.
        OtherPars => 'int max_it',    // OtherPars specs scalar arguments.
        Code =><<'EOC',
        /* All this code gets wrapped automagically in a thread loop.    */
        /* It starts a fresh C block, so you can declare stuff up top.    */
        /* The $GENERIC() macro is the overall expression type.          */
        int i;                                     // iterator.
        $GENERIC()   rp0 = $c(n=>0),   ip0 = $c(n=>1); // Copy the initial value to rp0/ip0
        $GENERIC()   rp  = rp0,        ip  = ip0;      // Copy again for the initial iteration
        $GENERIC()   rp2 = rp*rp,      ip2 = ip*ip;    // find RP^2 and IP^2 for magnitude and z^2.
        for(i=$COMP(max_it); rp2+ip2 < 4 && i; i--) {  // the OtherPars are in the $COMP macro.

            ip *= 2 * rp;  rp  = rp2 - ip2;  // calculate M_i(z)^2
            rp += rp0;      ip += ip0;        // add z
            rp2 = rp*rp;    ip2 = ip*ip;      // calculate rp^2 and ip^2 for next time
        }
        $o()= i;  // Assign the iterator to the output value
    EOC
    );
    EOPP
```

That code snippet (which can be placed anywhere you'd place a regular Perl subroutine declaration) declares a function called `PDL::pp_mandel`. It gets compiled into C and then into object code, and linked into the running Perl. The C-like code in the core of the declaration *is*, in fact, C – with some preprocessor macros. It handles one instance of the function. It gets wrapped automatically with all the required interface logic and loops to become part of a threaded operator and interface to perl, as a function in the `PDL` package. `PDL::pp_mandel` accepts two inputs: a PDL (called 'c' in the code) with a single *active dim* of size 2; and a separate scalar integer ('`max_it`') saying how many times (at most) to iterate M_i for each point. The PDL

arguments (both input and output) are described in the "`Pars`" section of the declaration, which also points out that the return value is a scalar – the active dim is collapsed and discarded, so a $2 \times W \times H$ input PDL will yield a $W \times H$ output PDL. Since `max_it` doesn't participate in the threading operation, it is declared in the "`OtherPars`" section and gets accessed slightly differently from the main threading variables.

The C code is passed into the declaration as a string, and it has three important macros that are used here:

- `$GENERIC()` is a stand-in for whatever the type of the arguments may be. PP compiles a separate variant of the code for each possible data type of its PDL arguments, so that (e.g.) double type variables run in a version of the code where `$GENERIC()` is "`double`"; but byte PDLs run in a version where `$GENERIC()` is "`unsigned char`".

- `$COMP(max_it)` is an expression that retrieves the integer parameter. The `$COMP()` macro is used to store a bunch of compiled-in values specific to each PP function - it's described more later. All the `OtherPars` (which are typically simple C types) can be found here.

- `$c()` and `$o()` give access to the PDL variables declared in the `Pars` section. Each active dim is declared up in the Pars section, and you can specify which element of each active dim you want, by naming it (as in "`n=>0`"). There's another construct, called `loop`, that you can use to loop over a particular active dim. Inside a `loop`, you don't have to access a particular location along each active dim – it's done for you automagically. Here we're not explicitly looping over any dimension – the only active dim runs across (real, imaginary), and we access the components with the "`n=>`" nomenclature. The parameter name ("`n`" here) is *required,* in order to avoid confusion in positional notation: sometimes there is more than one active dim, and/or some of the active dims might be masked by a surrounding `loop` structure.

After `mandel` is declared (either inline or as part of a module declaration; see below for details) you can call it just like any other PDL method – for example,

```
$vals = (ndcoords(200,200)/50-2)->pp_mandel(1000);
```

will feed a `2x200x200` array into `PDL::pp_mandel` and return a 200×200 array in `$vals`. On return, any point that *might* be in the Mandelbrot set gets a return value of 0, and any point that is definitely not in the set gets the number of remaining iterations at the time it diverged. Unlike the simple threading version, `pp_mandel` doesn't waste iterations on numbers that have already diverged.

You can visualize a piece of the mandelbrot set by enumerating a region on the complex plane and feeding it to `pp_mandel`, then plotting the result:

```
pdl> $cen = pdl(-0.74897,0.05708);
pdl> $coords = $cen + (ndcoords(501,501)/250 - 1) * 0.001;
pdl> use PDL::Graphics::Simple;
pdl> $w=pgswin(size=>[600,600]);
pdl> $w->image( $coords->using(0,1), $coords->pp_mandel(2500), {title=>'Mandelbrot'} );
```

yields the image:

Mandelbrot

2   Basic PP

The PP compiler is implemented as a Perl module (`PDL::PP`). A PP source code file is itself a Perl script that loads the `PDL::PP` module and executes methods in it, to generate .pm and .xs output files. The script calls particular functions exported by the `PDL::PP` module to generate segments of code. Each function call takes some parameters that describe the particular code segment. The core of the system is the function `pp_def()`, which defines a PP operator; but there are many definition functions that let you mix and match Perl code segments (and POD documentation), PP operators, and PerlXS / C code segments.

If you want to produce a standalone module, you put the PP calls into a Perl script (standard suffix ".pd") inside your usual module directory structure. The script can use PDL::PP itself, but more typically that is handled by MakeMaker and/or your makefile (see Section 3.2 for a description of how to set up a PP module with MakeMaker).

You don't have to make a standalone module. The module Inline::PP can insert some compiled inline PP code directly into your Perl script. The `Inline::Pdlpp` module takes care of using `PDL::PP` and generating the intermediate files for linking, so you can just insert some pp_def calls in a "use Inline Pdlpp" segment. That is covered in Section 3.1.

PP operators do not return values in the conventional way. In general, you declare one or more named PDL parameters that are flagged as output variables, and these get autogenerated by PP (See the code in §1.4 and §1.5 for examples of how this looks). Your code assigns to particular values inside the output PDL. If you need to return a non-PDL value, the easiest way is to declare a Perl helper function that passes in a Perl SV and or SV ref, and modify that in-place. Then the Perl helper can return the non-PDL value in the usual way.

2.1 Dimensionality and the signature of an operator

PP keeps track of the dimensionality of each operator with a "signature". PP operator signatures are similar to, but slightly different from, C argument declarations. The signature is a list of specifications and names for PDL arguments to the operator. The arguments are delimited with ';'. Each specifier is an optional type specifier, followed by a comma-delimited set of option flags in square brackets, followed by a variable name and a comma-delimited collection of zero or more active dim symbols, all enclosed in parens '(',')'. An example:

```
a(n); indx b(); [o,nc]c()
```

This signature describes an operator with three parameters `a`, `b`, and `c`. The first, `a`, can be any type and has one active dim, called `n`. Your code can loop explicitly over `n`, or index particular values of `n` directly. The second, `b`, is forced to be an index type and has zero active dims - i.e. your code will treat it as a scalar. The last, `c`, is used for output (that is the purpose of the `o` flag), and has zero active dims. The `nc` flag forces the user to supply a PDL for output - the code will not autocreate one if it is missing.

If you have two or more arguments with more than zero active dims, you can choose whether their active dims are "the same" or "different" by giving them the same or different names. For example, the signature:

```
a(n); b(n); [o,nc]c()
```

describes a function with two arguments and one active dim. The two arguments must agree on the size of the active dim, (although your code has full access along that dim).

You can set the exact size of a dimension by assigning to the index name like a C lvalue in the signature. For example,

```
a(n=3); [o]out()
```

specifies a single input argument (`a`) with one active dim (called `n`) that has size 3. If the actual passed-in argument has a different size at runtime, the code will throw an error.

Sometimes, you might want to calculate a dimension on-the-fly based on context. You can do that either by conditioning your PP call with a Perl subroutine (see the paragraph on PMCode, in Section 2.3.2), or by using a special hook to generate code that calculates the dims on the fly (see the paragraph on RedoDimsCode, in Section 2.3.3).

Repeated Dims (Square Operators) There is a not-too-uncommon special case where you want to force two active dims to have the same size (e.g. if your operator works on a square matrix). For that, you have to specify the same dim name in two locations, for example:

```
a(n,n); b(n,n)
```

That case is autodetected and works as expected – *except* that inside the code block, you refer to the dims with a trailing digit, to distinguish them – the first copy is called `n0` and the second is called `n1`. The order is the same in all square variables if you have more than one: the first copy of `n`, working left to right, is referred to as `n0`; the second is `n1`; etc.

2.1.1 The Importance of Operator Shape: A Cautionary Example

It's very important to understand the shape of any operator you intend to use or build, and to think about how that shape interacts with the threading engine. For many operators shape is trivial, but for some it can strongly affect threading behavior and performance. For example, the built-in `index` routine has the signature

```
src(n); indx dex(); [o]output()
```

and it carries out lookup by a single number (in `dex`) into `src`. Provided that `src` is 1-D, or that the user only grabs one value from `src`, that works fine. But a common use case is to thread over additional dims in `src` – feeding in, say, a 2-D array and selecting column vectors with `dex`. Since both `dex` and `src` have thread dimensions, those dimensions must match. Suppose you want to extract 5 column vectors from a 10×20 `src` variable. The active dimension `n` gets size 10 and the first thread dim. gets size 20. Since `dex` is declared with no active dims, it must match the first thread dim - i.e. its dim 0 must have size 20 to match the first thread dim. Getting your 5 column vectors requires using a 20×5 `dex`. (You could supply a 1×5 dex, of course, and the thread engine would automatically extend it to 20×5.) That indexing operation requires 20 times more offset lookups than are strictly necessary, hurting speed. If you know you are planning to thread both over the index and over the source file, you can instead use `index1d`, which behaves similarly but has the signature:

```
src(n); indx dex(m); [o]output(m)
```

This signature declares the `dex` as a 1-D object (to be treated as a list of indices into the 0 dim of `src`, called `n`), and declares that the output is also a 1-D object. If `src` is a 10×20-piddle, and `dex` is a scalar, then the return value will be a 1×20-piddle (`where index` would return a 20-piddle), since the first thread dimension gets size 10. Similarly, if `src` is a 10×20-piddle and `dex` is a 2-piddle, then the return value will be a 2×20-piddle (where `index` would crash on a thread dim mismatch). In other words, `index1d` threads more appropriately than `index,` to pull threaded vectors out of `src`.

That's not to say that `index` is inferior to `index1d`: if you plan to thread over only one of the two inputs (either to extract a single source value for each index, or extract a single column from source with exactly one index location), `index` handles the output dimensions more conveniently.

2.1.2 Signature option flags

You can modify any of your parameters in the Pars field, with one or more of these options in square brackets:

io Marks an argument as both input and output.

nc Marks an argument as necessary ("never create"). This is the default for normal input parameters; the use case is not obvious, but I include it here for completeness.

o marks the argument as an output argument; if missing, it will be created. (If there is only one output argument, it will be returned as the operator's return value).

oca marks the argument as an output, and also forces it to be created (i.e. it should *not* be passed in when the operator is called). Normal output arguments can be passed in to reuse an existing variable.

t marks the argument as a temporary variable (not passed in or out; just created on-the-fly as a scratch space, and erased on return. The argument is only large enough to hold the active dims; it is overwritten on each iteration of the surrounding thread loops)

phys Forces the argument to be physicalized – i.e. calculated and placed in memory if any calculations are pending, or if the argument is a complex slice of another PDL. This is necessary if you plan to access the argument's memory directly, rather than only through the macros supplied by PP.

2.1.3 Signature data types

You can specify the data type of a particular argument in the signature. Normally, a separate version of the code is compiled for each possible numeric data type, and all arguments are promoted to the "weakest" data type that can contain all of them, before your code gets run. If you specify a type for a particular argument, then it is hammered into that type regardless of the type of the rest of the arguments.

PP can accept these type names (in promotion order): `byte`, `short`, `ushort`, `int`, `indx`, `longlong`, `float`, and `double`. All of these are the same as their basic C type, except `indx` – which is the type used on the

current system for indexing pointers. (32 or 64 bit signed integer). You can append a "+" to the type to force that argument to take *at least* the named type or a stronger one.

If you name the data type of an argument, then that argument is excluded from the overall data type calculation. So if you specify a signature like

```
float+ a(); b(m,n); [o]c
```

then its first argument `a` will always be treated as a `float` or `double`, but the second argument `b` won't be autopromoted to match the first, even if you pass in, say, a `float` and a `byte` type for `a` and `b` respectively. If you use the "+" format, then the generated code for weaker argument types will declare your argument as the specified type, but for stronger types it will be declared as the generic type for the operation. For example: "`short+ a()`" will cause `a` to be a `short` in the C function that handles `byte` as the generic type, and it will cause `a` to be a `float` in the C function that handles `float` as the generic type.

Using data type specification can help cut down on memory overhead – for example, specifying a bulky-type temporary variable and manually promoting data into it inside your function avoids storing the whole promoted data array.

2.2 PP definition functions

Running a PP definition script applies the various definitions in the file, in the order they appear. The definitions take the form of calls to functions exported by the PDL::PP module. Each call either changes the behavior of PP, builds a data structure containing your code, or exports it as a .pm and .xs file.

The most basic PP file is a Perl script that imports the PDL::PP module and makes calls to `pp_def()` and `pp_done()`.

2.2.1 Generating C and XS code

Here are PP definitions you're likely to find useful, even for basic stuff.

pp_def - the core of PP `pp_def` accepts an operator name and a hash ref that describes the operator with several different keywords. It is so important that it gets its own section below (§2.3).

pp_done - finish a PP definition The very last PP call at the bottom of a definition script should be `pp_done()`. It takes no arguments, and causes the definitions to be written out to the .xs and .pm files. You can leave it out of `Inline Pdlpp` scripts, since `Inline::Pdlpp` puts it in for you.

pp_addpm - add Perl or POD material to the .pm file (Perl language) PP produces a .xs and a .pm (Perl module) file. `pp_addpm()` accepts a single string containing Perl language material that gets inserted into the generated .pm file, in order of pp_addpm calls. The added code starts just after the introductory material at the top of the module, and subsequent calls to `pp_addpm()` append more lines. It's useful for adding pure-Perl functions or POD. There are a couple of minor restrictions on the Perl code you can insert (in particular, don't mess with `@EXPORT` or `@ISA` directly; use the definition functions below).

PP keeps track of three different locations in your .pm file – the "Top", "Middle", and "Bottom". These are handled by three separate queues of material. By default, `pp_addpm` adds to the "Middle" section, but you can use a two-argument form to specify which queue you want to add stuff to. Just pass in an initial argument that is a hash ref – one of "`{At=>'Top'}`", "`{At=>'Middle'}`", or "`{At=>Bottom'}`". That way you can organize the .pm file more or less independently of the structure of your .pd file.

pp_addbegin - add Perl or POD material to a BEGIN block in the .pm file (Perl language) You can add BEGIN block material to your .pm file this way. (You can also insert an explicit BEGIN block with `pp_addpm`).

pp_addxs - add extra XS code of your own This is handy for adding an extra XS/C function to the generated .xs file, just like pp_addpm is useful for adding perl functions to the generated .pm file. This could be a utility or interface function that doesn't use threading. You should pass in a string that is a complete perlXS declaration; it will go in the function-declaration part of the XS file.

pp_addhdr - add PerlXS header material (C language) This lets you add stuff up in the PerlXS header segment. It comes *after* the basic includes and XS setup, so you can define whole C functions and such in here, as well as just invoking header material. `pp_addhdr` takes a single argument, which is a string containing the code you want to add. If you wrap your string in a `pp_line_numbers()` call, then any compiler errors will refer to line numbers in the .pd file (and be much more understandable).

pp_add_boot - add XS BOOT code This lets you add lines to the declaration/initialization section of the perlXS file, to set up libraries or initialize globals.

pp_boundsCheck - turn off/on PDL bounds checking in macros By default, the PDL access macros inside your PP code are wrapped in bounds checks. This normally doesn't cost much time, because the index variables are usually present in the CPU cache and many processes are memory-bound anyway – but you can turn off bounds checking for a little extra speed (at the risk of dumping core, instead of just throwing a Perl exception, in case of error). You feed in a single argument that is evaluated in boolean context. You can check the status by feeding in no arguments, in which case the current boundsCheck flag is returned.

pp_line_numbers - insert line number pragmas into some code This is mainly helpful for debugging – anywhere you hand in code as a string (mostly in `pp_def`), you can wrap the string in a `pp_line_numbers` call to insert line number pragmas into the code. So instead of saying "`Code=><<'EOCODE',`" in your `pp_def` call, you can say "`Code => pp_line_numbers(__LINE__, <<'EOCODE'),`", and your generated code will report the actual line number in the .pd file instead of a meaningless line number in an intermediate .xs or .pm file.

2.2.2   Controlling import/export and the symbol table

There are a few PP functions that are specifically for controlling how modules import and export elements. You're not likely to encounter or need them for simple tasks but will find them handy if you declare large modules.

pp_bless - set the module into which operators will be added (default "PDL") Pass in the name of the module you want to add your operators to. It defaults to PDL. If you say "`pp_bless(__PACKAGE__)`" they will go into the current package from which you are running PP (probably "`main`" unless you've declared one).

pp_add_isa - add elements to the module's `@ISA` list Just pass in a string containing a module name to be added to `@ISA`. Use this instead of messing with `@ISA` directly. If you call `pp_add_isa`, you should call `pp_bless` first, to set the module that you are modifying! This is mainly useful if/when you're designing a new type of object using PP constructs.

pp_core_importList - control imports from PDL::Core By default, PP issues a 'use PDL::Core;' line into the .pm file. This imports the exported-by-default names from PDL::Core into the current namespace. If you hand in the string "()" you get none of that, which is handy if you are over-riding one of Core's methods. Alternatively, you can explicitly list the names you want, as in " `qw/ barf / `".

pp_add_exported - export particular Perl functions from the module   You supply a single string with a whitespace delimited collection of Perl function names to export from the module. You should use `pp_add_exported()` instead of messing directly with `@EXPORT` in a `pp_addpm()` block. If you don't want to export any function names to the using package, then don't bother. As with a regular PerlXS module, you can export both Perl function found in the .pm file and also hybrid compiled functions named in the .xs file (i.e. things you created with `pp_def` or `pp_addxs`).

pp_export_nothing - don't export defined routines   By default, PP adds all subs defined using `pp_def` into the output .pm file's `@EXPORT` list. This is awkward if you don't want them exported into any module that uses them – for example if you are creating a subclassed object, or otherwise have a name conflict. Call `pp_export_nothing()` just before `pp_done()`, and nothing will be exported, keeping the namespace clean

2.3   The pp_def definition function

Calling `pp_def` is the main way to declare a new PDL operator. The basic call is simple. You feed in the name of a function and a collection of hash keys, the values of which are metadata and/or code for the operator. Inside snippets of code, you can use special macros to access the data and metadata of the PDL arguments to your operator. In this section we describe the different keys you can hand to `pp_def,` and the macros you can use inside your code sections.

There are two basic forms of operator: calculations (data operations) and connections (dataflow operations). Calculation operators are straightforward: they work just like function calls in any other language, returning zero or more PDLs that are the result of the main calculation but that do not remain connected to their source. Connections connect two or more PDLs so that they maintain an ongoing relationship that is updated automagically – assigning to or modifying one causes a recalculation of the other one. In principle connections can perform calculations (e.g. you can define a "reciprocal" operator that flows data between two PDLs), but in practice only the selection and slicing operators are connections.

Both types of call use a single "generic" data type that is calculated at call time, from the arguments. Multiple copies of your operator are compiled, one for each main generic data type, and given slightly different function names. At call time, all arguments are regularized to an appropriate generic type and the appropriate copy of your code (compiled for that generic type) gets called.

2.3.1   A simple pp_def example: cartND

Here is a simple example of a PP calculation function, `PDL::cartND`, that collapses the 0 dim of its input to find the Cartesian length of each row vector in it.

```
pp_def('cartND',                      # name (goes in PDL:: package by default)
       Pars => 'vec(n); [o]len;',     # Signature of the function
       GenericTypes => ['F','D'],     # Data types to use -- or omit to use all
       Code => q{                     # Single quotes around code block
          $GENERIC() acc = 0;         # Declare 'acc' to be the generic type
          loop(n) %{                  # Loop over the active dim 'n' in the signature
            acc += $vec()*$vec();     # Parameter PDLs are automagically indexed for you
          %}                          #  [outside loop you could use, e.g., '$vec(n=>2)']
          $len = sqrt(acc);           # Assign to the output PDL.
        }
     );
```

The operator has a name (`cartND`), and a collection of hash values that define the rest of the function to the PP compiler. The first value has the name `Pars` and is the *signature* of the operator. The operator accepts a single parameter (`vec`) with a single active dim, and returns a single parameter (`len`) with no active dims. It operates on the data types float and double only – its argument will be promoted to float if it begins life as an integer type. The `Code` section is where the basic operation takes place – it is written in C with some

extra macros, and gets wrapped automatically with a loop that handles all thread dims if `vec` turns out to have more than one dimension.

The `$GENERIC()` at the top of the `Code` section is a PP macro – it expands into a type definition for each of the data types you list by abbreviation in the `GenericTypes` value. So that line declares `acc` to be a holding variable of whatever type is appropriate based on the arguments' types at call time.

The `loop(n)` and matching `%{` and `%}` form another macro that expands to a C style `for` loop running over all possible values of the index `n`, so the loop accumulates the sum-of-squares of all the components of the vector. The parameters are accessed with a leading `$` and a trailing `()` pair. You must specify the value of any active dim indices that aren't explicitly set or looped over (e.g. with "`$vec(n=>ii)`" if you have an index variable `ii`) – but in this example there's no need. The `loop(n)` provides implicit dereferencing for the `n` index.

Note that you don't have to include *any* loops in your PP code if your signature is scalar – in that case, each `$arg()` macro stands on its own. (Note that **you have to include the parentheses** even if you're not setting the value of any index!)

At the end of the accumulation loop, the `Code` takes the square root of the accumulated value and assigns it to `$len()`, which goes into the output PDL.

That's all!

You can insert the above snippet into a .pd file and compile it, or into an Inline block in your script and have it compiled on-the-fly.

2.3.2   Basic pp_def keys

Here are a list of common pp_def keys you need for basic calculations.

`Pars` - the operator signature and argument names   This is the operator signature discussed above (Section 2.1). It lists all the PDL arguments accepted by your operator, and their active dims (if any), access modes, and allowable type(s) for that argument (if different from the generic type). An example for a scalar binomial operator is "`Pars => 'a(); b(); [o]out();'`". Another example, for outer product of two vectors, is "`Pars => 'a(n); b(m); [o]out(m,n)'`".

`OtherPars` - non-PDL argument names   After all the PDL arguments to a PP function you can also pass in other parameters that are not PDLs. These are handled with the OtherPars section. These can be declared as C types or Perl C-side types (including `SV *`) and will be converted on-the-fly from their scalar values on the Perl side of the operator call. A caveat is that you can't easily provide a custom typemap for these arguments. If you have, for example, a structure pointer to pass in you can pass it in as a Perl `IV` and cast it to a pointer when you use it (PDL's interface to the Gnu Scientific Library does that). Alternatively, you can pass in pretty much anything as an `SV *` and allow your `Code` section to handle it directly. An example OtherPars is "`OtherPars=>'double foocoeff; IV foocount; SV * perlsv;'`".

Note that, even if you pass in lots of OtherPars, you must always pass in at least one PDL argument in the Pars section, or PP will choke.

The OtherPars are not available as directly named variables or direct macros. You get them via a macro called `$COMP()` that is accessible in the Code segment. For example, with the above declaration, you could access foocoeff by saying "`$COMP(foocoeff)`". COMP stands for "Compiled", and it is a data structure compiled by PP to hold ancillary data during setup for an operation.

`PMCode` - a Perl-side handler   PMCode lets you define some Perl code that gets executed instead of the `Code` block when your operator gets called. There could be several reasons. For example, you might want to precondition the arguments to your PP function, or want to be sure you don't break dataflow connections by autoconverting the arguments to your function; you might want to prepare default values for optional arguments to the main Code block; or you may even want to mock up the Code block in Perl before implementing it. `PMCode` lets you handle those cases.

If you want to use your Perl code to condition the arguments for the compiled code, you can include a `Code` block **and** a `PMCode` block. That will generate both types of code, but only the `PMCode` gets executed directly. Your PMCode can call the special function "`&<package>::_<name>_int()`" with the arguments given in Pars and OtherPars to run the `Code` block. The *<package>* is either "`PDL`" by default, or the value you specified with a call to pp_bless(). That mechanism lets you do some initial parsing or argument currying with Perl, then jump into your compiled code for faster processing of the algorithmic "hot spot". *Note: This functionality does not work for Inline PP modules, for all versions of PDL through 2.011. That is a "known problem" that may be fixed in a future release of PDL. To curry arguments with inline code, you must (for all versions at least through 2.011) declare a "front-end" perl function and a separate PP operator with no* `PMCode` *block.*

`Code` **- the main code to execute** The compiled code you want to implement. See Section 2.3.4 for details on what goes in here. This is the main Code block, but there can be several code blocks even in a simple calculation function (see `BadCode` and `PMCode`, below). You don't actually need to declare a Code block at all if your operator is written entirely in Perl (you can use the `PMCode` key for that).

`BadCode` **- A variant of** `Code` PDL can handle bad values in data. A particular in-band value (e.g. -32768 for shorts) is chosen to mark data as bad or missing. Bad values are intended to be contagious. Handling them requires additional logic. For example, "`$c() = $a() + $b()`" becomes something like:

```
if( !$ISBAD($a()) && !$ISBAD($b()) )
    $c() = $a() + $b();
else
    $c() = BADVAL;
```

which requires a few more CPU cycles than the direct addition. Specifying a separate BadCode segment lets PP switch between direct operations and their bad-value-handling variants based on whether the arguments' badflags are set (i.e. whether bad values may be present or cannot possibly be present). See Section 4.4 for details.

`GenericTypes` **- set specific types to be compiled** PP normally creates a separate version of your operator for each data type recognized by PDL. When the routine is called by name, the PDL threading engine reconciles the variable types the same way they would be reconciled in an arithmetic expression – and the call is dispatched to the appropriate version of the operator (in which the argument types are declared appropriately, and the `$GENERIC()` macro expands to that type declaration in C). But sometimes you want to support only a subset of the available types. You pass in a Perl array ref containing the single-letter specifiers for the types you want to support: 'B' for byte, 'S' for short, 'L' for long, 'N' for index (32 or 64 bits as appropriate to your system), 'F' for float, and 'D' for double. Unrecognized types will yield a compiler error.

`Inplace` **- Set inplace handling** If this is set to be true, then your routine can handle in-place processing of its arguments (to save memory, or for convenience). When an argument is processed in-place, its result is placed back into the original piddle instead of copied to an output variable. The in-place operation is handled fully by the PP engine, including clearing the inplace flag after the operation completes. If you want to handle in-place operation manually, you can check the flag for yourself by masking the parameter's state field with the constant `PDL_INPLACE` (declared in `pdl.h`) – but remember to clear that flag before returning!

If your operator has only one input and only one output, you can set "`Inplace=>1`", and the input and output variable will be linked if the input has its `inplace` flag set. If you have more inputs and/or more outputs, you must specify a two-element array ref with the names of the input and output parameter to link. For example: "`Pars=>'a(); z(); [o]b();', Inplace=>['a','b']`". Of course, the two parameters must have the same signature in the `Pars` declaration, except that the `[o]` modifier must be set for the second.

You can link at most one input to one output parameter using the built-in inplace handling, though of course you can roll your own by accessing the `inplace` flag of each argument.

`HandleBad` - set bad value handling   If you set `HandleBad=>0`, then any input argument with its BadFlag set will cause an exception. If you set it to `undef`, or don't set it at all, then bad values are ignored and treated as any other value.

If you set `HandleBad=>1`, and declare `BadCode` (below), then the BadCode will get run whenever it's appropriate (i.e. one or more input PDLs has its BadFlag set and therefore might contain specially coded missing values). Also, PP will define all the relevant bad-value handling macros `$ISBAD()`, `$SETBAD()`, etc. See Section 4.4.

`Doc` - POD documentation   You can document your project using POD, Perl's Plain Old Documentation format. You can in principle add POD using a PMCode block (since POD is a valid Perl no-op), but using the Doc field will cause your project to be indexed into the system PDL documentation database if/when you "make install" your module.

`BadDoc` - Special POD for bad value handling   If the installed version of PDL was compiled to handle bad values, this documentation gets appended to the documentation in the `Doc` field.

`NoPThread` - Mark Function as not threadsafe   By default, PP reserves the right to run multiple instances of your code in multiple CPU threads, for extra speed on multiprocessor systems. (If it does this, temporary variables in the signature are properly duplicated across threads). If your code is not threadsafe (for example, if you write to local variables or the `$COMP` structure without your own MUTEX), you can set this flag to avoid multithreading. (Note that multithreading is currently a *language-compile-time experimental option* in PDL 2.009, but it may be enabled by default in a future version.)

Because the nomenclature is confusing, here's a reminder. A PThread is a Perl thread, which is generally implemented as a CPU thread. A CPU thread is an execution environment with a unique program counter – unlike a CPU process, it shares memory with other threads in the same execution context. A PThread is a CPU-like thread that exists in some Perl environments, depending on how `perl` was compiled. These types of thread are distinct from PDL's "threading", which is vectorization over thread dims of a PDL. A MUTEX is a mutual-exclusion gate that prevents multiple threads from writing to the same memory simultaneously, or prevents one from reading from a program variable while a different one is writing to it.

`PMFunc` - Control Where the Function gets Linked   Normally your PP code gets compiled and linked into the PDL module (or whichever module you specified with `pp_bless`, if you used that - see Section 2.3.3). PMFunc adds a hard link to the function name of your choice. It's functionally equivalent to adding a line "*$name=\&PDL::name;" to the Perl part of your definition. It appears, as of PDL 2.009, to not work with inline declarations, though it works with module declarations.

`RedoDimsCode` - Calculate the Dimensionality of the Output at Runtime   This field isn't normally necessary, but you can use it to calculate the dimensionality of an output variable at runtime. That's useful for some applications where the output variable's shape is related in a non-simple way to the input variables'. See Section 4.5 for details.

2.3.3   More advanced pp_def keys

You can pass in many more keys to pp_def than the basics described above. The most important of these are associated with dataflow. They are covered in Chapter 5.

2.3.4 pp_def Code macros

In the C code blocks like `Code` or `BadCode`, you write in C – with preprocessing. The PDL "hooks" are implemented with text replacement macros that give you the functionality you need. The basic ones are below. You can find other macros for handling bad values, in Section 4.4.

$COMP(*name*) - access the compiled data structure and OtherPars parameters   `$COMP` accesses a compiled data structure associated with the operator. In computation operators, `$COMP` is by default pre-loaded with all (and only) the non-threading parameters you declare with an `OtherPars` field in `pp_def`. You can declare additional `$COMP` fields with the `Comp` argument to `pp_def`, and/or perform additional computations up-front to generate `$COMP` values with a `MakeComp` block (See Section 2.3.3). That's primarily useful in dataflow operators. The `$COMP(`*name*`)` C fields are lvalues.

$*name*() - access elements in threaded Pars parameters   To access the value in one of the parameters named in the `Pars` signature of your operator, you just name it, with a prepended `$` and a postpended pair of parentheses. (Note that this means you should not name a parameter, e.g., "COMP"!) If there is a named dimension in the parameter, you must index that dimension with a double-arrow, as in `$a(n=>2)` for a parameter `a` with an active dimension called `n`. The `loop` construct, below, automagically selects the index in the dimension being looped over, so you don't have to. If you have to specify multiple indices, you can separate them with commas, as in `$a(n=>2,m=>3)` to specify location in a variable with two active dims.

If a dim appears twice in the signature, as in "`a(n,n);`" in the Pars section to specify a 2-D square variable, then you access the two dimensions by appending a numeral, starting at 0, as in "`$a(n0=>2,n1=>3)`". The numerals start fresh at 0 for each Par with a square (or greater) aspect. There's currently no way to transpose two square parameters relative to one another: the first copy of the repeated name in each parameter maps to *name*0 for all square parameters, and the second maps to *name*1.

$SIZE(*iname*) - return the size of a named dimension   You often need to know the size of a named dimension. This macro expands to it.

$GENERIC() - the generic C type   `$GENERIC()` expands to a C type declaration for the generic type of the operation. All incoming PDL parameters' types will be resolved to the smallest type that can represent all of them, and converted to that type. So you can declare an internal working variable `x`, for example, as "`$GENERIC() x=0;`" in your `Code` block. PP will compile a separate copy of your code for each possible value of `$GENERIC()`, and dispatch execution accordingly at runtime.

$T*foo*() - type-dependent syntax   The $T macro is used to switch syntax according to the generic type. Each generic type has a one-letter abbreviation: "B" for byte, "U" for unsigned short, "S" for short, "L" for long, "T" for Indx, "F" for float, and "D" for double. You string the letters together after the $T, and give the macro a comma-separated list of words to insert in the code depending on the value of the generic type. For example, "`a = ( $TSLFD( short, long, float, double ) ) b;`" is a limited equivalent to "`a = ($GENERIC()) b;`". $T is useful for dispatching calls to a type-specific library function, or any other situation where you need to make a small difference between each of the generic-type copies of the compiled code. *Note: replacement strings containing commas are not supported!*

$PDL(*var*) - return a pointer to the PDL struct   On occasion you need to dive into the guts of the C struct associated with a particular PDL (see §7.1). This macro returns a C pointer of type "`struct PDL *`", so you can access the internals directly.

17

`threadloop %{ ... %}` - explicitly locate the thread loop   PP autogenerates looping code to repeat your operation for each element along thread (extra) dimensions in the input parameters. By default your entire `Code` block is run separately for each threaded-over element. For some types of operation this is inefficient – for example, you might carry out some initialization that applies to each threaded instance of the operation. If you explicitly place your code between the `%{` and `%}` of a threadloop macro, the thread looping code will be placed there only.

`loop(iname) %{ ... %}` - explicitly loop over a named dimension   This operates like threadloop, but generates an explicit loop over just the named dimension. If you use a parameter macro inside the loop, you do not need to specify the value of the named dimension index. You can override the loop by specifying it anyway. Of course, you are also free to generate loop constructs explictly with temporary index variables and normal C looping structures, but loop can be more convenient and readable.

3   Putting it together: Example PP functions

Declaring a PP function is only a part of the process of compilation. Here are examples of how to create a standalone PP module and how to include PP code inline in your Perl code. You can use these examples as templates for your own PP projects.

3.1   Using PP inline with Perl, via Inline::Pdlpp

`Inline::Pdlpp` is terrific for placing a compiled "hot spot" function right in the middle of your Perl script. We already saw a simple example in §1.5. Here are some more examples: a simple one that generates Spirograph^{TM}-like parameterized hypocycloid curves, and a more complex one that enumerates a fractal path through a 2-D grid.

There is an important wart with Inline::Pdlpp compared to a full module definition: at least through PDL 2.008, the `PMCode` and field for `pp_def` doesn't work correctly. It is simply ignored by the `Inline` compilation process. If you need a perl helper to precondition inputs, you need to write it explicitly and call the PP function from there.

You can use a call to `pp_bless` to place the helper function in the package you want, but it's also common to just use a single `pp_def` call and let PP compile the helper into the default PDL package (as in the examples here).

3.1.1   A simple case: combining compound math operations

This example implements `spiro`, a compound math function that generates a parameterized cyclocycloid in 2-D from a single time parameter and some coefficients. Compound math functions like `spiro` are good candidates for PP, because they require many atomic math functions to implement – which breaks cache if you thread over a large dataset. By merging the operations (in C/PP) *inside* the thread loop, you can make spiro work up to 10x faster on large data sets and multiple elementary math operations. An advantage of Inline::Pdlpp is that you can prototype your code in PDL, then upgrade it in-place later in exactly the same location, and call it in exactly the same way. You do not have to set up a module file structure – Inline handles all the compilation behind the scenes.

Here are both a PDL and a PP version of `spiro`. You could place them as part of a larger module declaration, or paste them into a Perl script to be defined in-place, or place them in autoloader files.

The input `$s` parameterizes length along the spiral. `$r1` and `$r2` are the sizes of the outer and inner "gears" in the cyclocycloid apparatus, and `$n1` and `$n2` are the number of teeth.

The code below implements the same operation in regular PDL idiom, and also in PP using Inline – you can place both definitions together in a single script (or, for that matter, module). Using PP moves memory accesses inside the loop, preserving cache and speeding up the code by ~3× for moderate-sized curves of up to a million points.

```
=head2 spiro - parameterized spirograph-like cyclocycloid

=for usage

 $xy = spiro_pdl($s, $r1, $n1, $r2, $n2);
 $xy = spiro_pp($s, $r1, $n1, $r2, $n2);

=cut
sub spiro_pdl {
    my($s,$r1,$t1,$r2,$t2) = @_;
    my $phase1 = ($s/$r2);
    my $phase2 = $phase1 * $t1/$t2;
    my $xy1 = pdl(cos($phase1), sin($phase1))->mv(-1,0) * $r1;
    my $xy2 = pdl(cos($phase2), sin($phase2))->mv(-1,0) * $r2;
    return $xy1 + $xy2;

}
no PDL::NiceSlice;
use Inline Pdlpp => <<'EOF';
pp_def('spiro_pp',

    Pars=>'s(); r1(); t1(); r2(); t2(); [o]xy(n=2)',
    Code=> q{
        double ph1 = $s() / $r2());
        double ph2 = ph1 * $t1() / $t2();
        $xy(n=>0) = cos(ph1) * $r1() + cos(ph2) * $r2();
        $xy(n=>1) = sin(ph1) * $r1() + sin(ph2) * $r2();
    } );

EOF
*spiro_pp = \&PDL::spiro_pp; # copy to current package
print ''Loaded spiro_pdl and spiro_pp!\n'';
```

The two versions of `spiro` have very similar structure. Both versions accept a collection of points to plot (the "s" parameter), two gear tooth counts, and two gear radii. They calculate the corresponding 2-D location of the cyclocycloid at the given "s". The Perl version uses threaded arithmetic operations to assemble the output. The PP version does the same calculation in C – but, more importantly, the C calculation is performed *inside* the threadloop, so that all of the relevant variables can be loaded into registers and/or CPU cache.

The signature of `spiro_pp` shows that it is a scalar operator with 5 inputs. The output has 1 active dim, of size 2 – because it is a collection of 2-D vectors. The arithmetic operations are exactly the same as in `spiro_pdl`, translated to C – except that the assignments and calculations in `spiro_pp` are normal C language scalar operations.

3.1.2   Calling a C function with Inline::Pdlpp

The `hilbert` routine demonstrates jumping out to a C routine to get useful things done. It calculates an approximation to the Hilbert curve, which is a linear fractal that fills the plane (it has Hausdorff dimension 2). The Hilbert curve is useful for dithering greyscale images to a single bit, or other applications that require traversing a 2-D grid of locations without rasterizing them. The Hilbert curve, being fractal, is infinitely complex – so `hilbert` is recursive, producing an approximation that has been refined through a specified number of levels of recursion.

The C code here populates a string with glyphs that indicate which direction the Hilbert curve steps next; it is recursive and descends *level* steps to refine the curve. The PP code parses the returned string and uses it to populate a collection of 2-vectors describing the vertex locations of the curve. For convenience,

I've interspersed explanatory text with code snippets – but all the code in this section can be pasted into a single file for the PDL AutoLoader.

```
=head2 hilbert - generate a Hilbert ordering of planar points
=for usage
  $line = hilbert($pdl);
  $line = hilbert($w,$h);
=for ref
The Hilbert curve fills the unit square with a linear area-filling fractal curve.  This routine uses approxima
=cut
```

This top part of the code is just POD that you might put in any .pdl script.

```
##### Here is the main entry point -- a Perl subroutine that allocates the
##### output array and jumps into the PP code.
sub hilbert {

    my $w, $h;
    ($w,$h) = (ref $_[0] eq 'PDL') ? shift->dims : splice(@_,02);
    my $dim = ( $w>$h ? $w : $h );
    $n = (log(pdl($dim))/log(2))->ceil; $siz = 2 ** $n;
    $coords = PDL->new_from_specification(long, 2, $siz*$siz);
    PDL::hpp($coords,$n);
    return $coords;

}
```

The actual perl function being defined is hilbert. That Perl function calls a PP function declared below – PDL::hpp – which, in turn, calls a C function. PDL::hpp gets called with a single output PDL ($coords) that has the $2 \times size^2$ dimensions calculated above. Notice that $coords is defined with PDL->new_from_specification(), rather than (say) zeroes(). That eliminates one pass through the memory, since the values in $coords are not actually initialized – they get whatever happened to be sitting in memory.

Here is the actual inline code. The "no PDL::NiceSlice" is necessary because NiceSlice interferes with C code parsing. Everything between the "use Inline Pdlpp" and the "EOF" below is fed to PP to generate a .pm and .xs file that are then linked in on-the-fly. The code defines a C function and a PP interface function.

```
no PDL::NiceSlice;
use Inline Pdlpp=><<'EOF'; # exactly like this (no '::')
```

This is the first PP call (inside the PP block). It is a declaration of a C utility routine. Here is where you could #include external libraries, for example, if you need them. We use pp_addhdr to put this piece of C code at the top of the generated code.

```
pp_addhdr( << 'EOAHD');
PDL_COMMENT(" This stuff goes below the top-of-file declarations and   ")
PDL_COMMENT(" before the XS stuff, in the compiled C code. It's a good ")
PDL_COMMENT(" place to declare a helper C function.                    ")
static char *hre(char *p, char dir, int level) {

    if(level==0) {
        switch(dir) {

            case '<': strcpy(p,">v<"); break;   case '>': strcpy(p,"<^>"); break;
            case '^': strcpy(p,"v>^"); break;   case 'v': strcpy(p,"^<v"); break;
        }
        p += 3;
```

```
        } else {
            int l = level - 1;
            switch(dir) {

                case '<': p=hre(p,'^',l); *(p++)='>'; p=hre(p,'<',l); *(p++)='v';
                          p=hre(p,'<',l); *(p++)='<'; p=hre(p,'v',l); break;
                case '>': p=hre(p,'v',l); *(p++)='<'; p=hre(p,'>',l); *(p++)='^';
                          p=hre(p,'>',l); *(p++)='>'; p=hre(p,'^',l); break;
                case '^': p=hre(p,'<',l); *(p++)='v'; p=hre(p,'^',l); *(p++)='>';
                          p=hre(p,'^',l); *(p++)='^'; p=hre(p,'>',l); break;
                case 'v': p=hre(p,'>',l); *(p++)='^'; p=hre(p,'v',l); *(p++)='<';
                          p=hre(p,'v',l); *(p++)='v'; p=hre(p,'<',l); break;
            }
        }
        return p;

    }
    EOAHD
```

The EOAHD ended the first PP call (we're still inside the PP definition block), so we're ready to call `pp_def`. There is a single PDL parameter in the `Pars` signature - a collection of vectors that we will populate. It's declared "nc" for clarity, because PP can't autocreate it (it would have no way of knowing the size of the named dimension m) The named dimension `n` is forced to have size 2, since it indexes an (x,y) pair for each of the *m* points to be reported. The tag points out that we *only* want a single copy of the code, suitable for running on the `long` type. One non-PDL parameter is accepted – the recursion level to which `hre` should descend. It's autoconverted to `int` by PP and the PerlXS mechanism. The `level` parameter gets accessed with the $COMP(level) macro.

The `hre` code needs a character buffer in which to put the motion directions, so we allocate that. The size of each active dim is available through the $SIZE() macro, and we use that to make the buffer the correct size (the size was precomputed by `hilbert` before entry).

The meat of the code is simple: we call the C routine that does the work, then convert its recursively-generated step code into a sequence of coordinates of the vertices of the curve. Because `n` and `m` are declared active dimensions, they are not automatically looped over – our code has to do that. But we don't have to use a `for` loop - the `loop(m)` macro takes care of that. Take care to use the macro brackets %{ and %}, rather than simple brackets! Inside the loop, the variable `m`, with no modifiers, is predeclared and has the correct value. We access `out` with the macro $out(). Inside the macro parens, we have to give values to all the active dimensions that are not already defined. Since we're in the `loop(m)` construct, `m` is defined – but `n` is not. The $out() macro is an lvalue, and we assign the accumulated x and y values to it.

The `loop(m)` loop is nested inside a construct called `threadloop`. That is important in case of threading – by default, PP wraps the threadloop around the whole code – but we don't want to call `hre()` once for each hyperplane, if the user decides to thread his call. We only need to call it once. Placing the threadloop macro around the hotspot lets the code call `hre()` just once (at the start) and then get down to the business of threading over higher dimensions. The `threadloop` macro can even be placed inside the other loop – we could in principle save even a little more time by putting the `loop(m)` macro on the outside and nesting only the final two assignments inside a tiny `threadloop`.

```
    pp_def('hpp',
          Pars=>'[o,nc]out(n=2,m);', # Note: must be passed in to define m size!
          GenericTypes=>[L],
          OtherPars=>'int level',
          Code=> <<'EOHPP'
        long dex, side, x, y;
        char *s, *buf = (char *)malloc($SIZE(m)+2);
```

21

```
$PDL_COMMENT(" hre is the expensive computation. Since it's exactly the same   ");
$PDL_COMMENT(" for all operations of the same size, we call it *outside* an    ");
$PDL_COMMENT(" explicit threadloop.  That way it only gets executed once, even ");
$PDL_COMMENT(" for a threaded operation.                                       ");
      hre(buf, '>', $COMP(level)-1);
      s = buf;


      threadloop %{     # Threadloop specifies the "hot spot" for threaded operations.
          loop(m) %{
              if(!m){ x=y=0; } else {
                  switch( *(s++) ){
                     case '^': y++; break;
                     case 'v': y--; break;
                     case '<': x++; break;
                     case '>': x--; break;
                  }
              $out( n=>0 ) = x;
              $out( n=>1 ) = y; }
          %}
      %}

      free(buf);
  EOHPP
  );
  EOF
```

This is the end of the inline PP block, and Perl code continues below. There's just a simple print statement, to show that we can pick up the thread. The print executes *after* Inline finishes compiling `PDL::hpp`.

```
print "Compiled hilbert!\n";
```

3.2   A standalone PP Module

If you are coding a complete module that uses PP functions extensively, you can place the entire module - PP, C, and Perl - in a single file that will generate `.xs` and `.pm` files when built. Most of PDL itself is implemented this way. There are two important things that aren't obvious from using PP inline: (1) you need to invoke `PDL::PP` directly to process the source code and generate the files; and (2) most authors choose to merge the Perl and PP segments of their module into a single source file called "*module*.pd". If you use the standard `ExtUtils::MakeMaker` or `Module::Build`, then there are straightforward recipes to use, with routines that are included with PDL in the module `PDL::Core::Dev`.

The various intermediate and compiled files get put into the current working directory when the PP calls are made. If you're using `ExtUtils::MakeMaker`, that is the same directory as your `.pd` file. After compiling your module, you'll find a `.xs`, a `.c`, a `.pm`, and a `.so` file in the same directory. The human readable files (`.xs`, `.c`, and `.pm`) all get warning comments at the top of the file, pointing out that they were generated from your .pd file and should not generally be edited.

This section is a demonstration of how to implement a simple object, `PPDemo::Hello`, as a standalone module that uses Perl, C, and PP code. All the elements are declared inside one file, `hello.pd`.

`PPDemo::Hello` is a bit silly – it keeps track of the number of people in a large guest house, indexed by day of arrival or departure and ID of their group. It tracks arrival and departure of people in events, and integrates those events to arrive at the number of dinner guests each day. The information is kept in a 3-column PDL that contains (day-of-event, party-id, number of people). The events are stored in rows. To avoid too much overhead for shuffling memory, the PDL is allocated 100 rows at a time, and a separate counter in the object keeps track of how many rows are used.

A `PPDemo::Hello` object doesn't have a lot of extra information in it, so it also demonstrates PDL's autohash extensions: the object is stored in a hash ref, and the database is in a key called "`PDL`" – so it can also be manipulated directly as a PDL object by all the usual techniques. To make that happen, we just add "`PDL`" to `@PPDemo::Hello::ISA`.

If you use `ExtUtils::MakeMaker` (or `Module::Build`), you only need two files – a `Makefile.PL` (or `Build.PL`) and a single ".pd" file that invokes PP to generate the program files.

3.2.1 Makefile.PL

Within the context of a MakeMaker module, you can use a simple Makefile.PL to compile your code, drawing functionality from the `PDL::Core::Dev` module. Here's a sample:

```
# Sample Makefile.PL for a PP-based module
use ExtUtils::MakeMaker;
use PDL::Core::Dev;
PDL::Core::Dev->import();
my $package = [

    ''hello.pd'',        # File name containing your module's PP declarations
    ''PPDemo_Hello'',    # Preferred name for your module's intermediate files
    ''PPDemo::Hello''    # Fully qualified name for the module you're declaring
    ];

my %hash = pdlpp_stdargs($package);
# (Modify %hash as you wish here - adding metadata, author name, etc.)
WriteMakefile( %hash );
sub MY::postamble { pdlpp_postamble($package) };
```

The `pdlpp_stdargs` call loads `%hash` with the appropriate values to set up compilation – it loads `%hash` with all the parameters needed to make MakeMaker compile your `.pd` file into `.pm` and `.xs` files, to compile that `.xs` file into appropriate dynamic libraries to which Perl can link, and to put the compiled files in their correct places in the `blib` hierarchy. Your `.pd` file, as with any other Perl file, may or may not actually declare a complete Perl module. For example, in the PDL distribution itself many `.pd` files create additional functionality directly in the `PDL` package rather than declaring their own package.

3.2.2 hello.pd

The single file you use for your PP module is a ".pd" file. It is a perl program that calls PP functions to emit code, so it starts a lot like any other Perl script. The very first call is to `pp_bless`, to make sure PP puts compiled objects into the package we want. (The default is "`PDL`", but if you just dump all your methods in there you lose the benefits of a hierarchical name space). We also call `pp_add_isa` to make sure we can manipulate our object with PDL calls. You probably don't need that for your object, unless you want to be able to manipulate it directly as a PDL. (All PP operators, including the main PDL API, can accept a blessed hash ref containing a piddle in the field named "`PDL`", instead of an ordinary piddle.)

```
#!/usr/bin/perl  # (Or whatever for your system)
# This is hello.pd - a demo file for a simple PP-based module.
use PDL;
pp_bless('PPDemo::Hello'); # equivalent of ''package'' at top of a .pm
pp_add_isa('PDL');         # hook in the PDL hash-extension stuff
```

The next bit is also common – we want to stick some POD and "`use strict`" into the top of our module. So we use `pp_addpm` to place those things into the .pm file that PP will build.

```
pp_addpm( {At=>Top}, <<'EOPM'); # place following perl code at the top of the file

use strict;

=head1 NAME

Hello - sample perl + PP object

=head1 DESCRIPTION

Hello keeps track of long-term guests over a span of days, counted by
day from an epoch.  You tell it when guests enter and leave.  Methods
retrieve information about how full the house is, and about who is the
most frequent visitor.

Methods are ''new'', ''event'', ''tomorrow'', and ''dinner_count''.

=cut

### Do this if you want your POD loaded ino the PDL shell's help database on-the-fly.
$PDL::onlinedoc->scan(__FILE__) if($PDL::onlinedoc);

EOPM
```

So much for Perl header material. We can also add some C header material. Here, it's just a sample comment and a helper C function to print the day of the week. The helper function uses `pdl_malloc`, which is a fire-and-forget version of `malloc` that produces scratch areas on the fly. `pdl_malloc` uses the Perl garbage collector mechanism: its returned pointer is the data field of an anonymous mortal SV, which is cleaned up automagically by Perl whenever you exit the enclosing scope. The C code will get placed near the top of the generated XS file, just above the various function declarations generated by PP itself.

```
pp_addhdr( <<'EOC');

PDL_COMMENT('' This is C.  Both pdl.h and pdlcore.h have been included.  (This '');
PDL_COMMENT('' style is preferred by some over /* */, which doesn't nest.)'');

#include <stdio.h>
PDL_COMMENT('' Return the abbreviated day of the week in a short scratch string '');
static char *dow_data=''Sun\0Mon\0Tue\0Wed\0Tu\0Fri\0Sat\0'';
char *dow( unsigned int n ) {
  unsigned int dex = n % 7;
  char *out = pdl_malloc( 4 * sizeof(char) );
  strncpy(out, dow_data + 4 * dex, 4);
  return out;
}

EOC
```

Most of the methods are Perl-side, so they are placed in a single pp_addpm call. You can break up the call if you choose to. Since we don't really care where this material goes in the module, we don't specify the lcoation to `pp_addpm`. It defaults to the "middle" (after the Top material and before the Bottom material). The Top, Bottom, and Middle queues are maintained separately, so material appears in the order of `pp_addpm` calls for each of those parts of the file. Notice that the methods make a lot of use of the fact that regular PDL methods work on the object itself automagically since it keeps its primary piddle in the hash field named 'PDL'. That shortcut is apparent in the stringifier, in particular, where we use "`slice`" and such on the object itself instead of the piddle that it contains.

```
pp_addpm( <<'EOPM' );

## Simple constructor
sub new {

    return bless( { day=>0, n=>0, PDL=>zeroes(long, 3, 100) }, $_[0] );

}

## Stringifier -- so ''print $a'' makes something useful
use overload '""' => sub {

    my $me = shift;
    my $s = sprintf('PPDemo object -  days: %d;  events: %d;  ids: %d'',
        ($me->slice(0)->max - $me->slice(0)->min + 1),
        $me->{n}, $me->dim(1),
        $me->slice([0,0,0])->qsort->uniq->nelem
        );
    $s .= ($me->{n}) ?
        $me->slice('x',[0,$me->{n}-1]) :
        ''\n[ Empty ]\n'';
    return $s;

};

## event adder - add a row to the event database (or several - it's threadable)
sub event {

    my ($me, $id, $num) = ($_[0], pdl($_[1]), pdl($_[2]));

    ## Extend rows if necessary
    me->{PDL} = $me->{PDL}->glue(1,zeroes(3,100))
      unless( $me->{n} + $id->dim(0)) < $me->{PDL}->dim(1) );

    my $cut = $me->slice('x',[$me->{n}, $me->{n} + $id->dim(0) - 1, 1]);
    $cut->slice([0,0,0]) .= $me->{day};
    $cut->slice([1,1,0]) .= $id;
    $cut->slice([2,1,0]) .= $type;
    $me->{n} += $id->dim(0);
    return $me;

}

## Advance day
sub tomorrow { $_[0]->{day}++; }

EOPM
```

Here's the first PP definition - a simple dinner-guest counter. You can normally get away with just a `Code` section to the PP definition, as in the other examples – but here we use the `PMCode` section to declare a Perl wrapper function that preparses the arguments. The parsing here is pretty trivial – it amounts to just making the `$day` field default to -1 if not specified, since PP itself will autogenerate null PDLs if the output parameter isn't specified. The PP declaration just acccumulates dinner guests by event from the beginning of time until the specified day (or the end of the data set), and returns the sum.

If you specify a `PMCode` field, then the `Code` field generates a function called _<*name*>_int, instead of just <*name*>. The `PMCode` section is a Perl snippet that must declare <*name*>. Warning: If you don't make a `pp_bless` call (as we did above), then the _<*name*>_int ends up in the package `PDL`, while the Perl declaration

executes in the current package. In that case, you'd have to call `PDL::_<name>_int` explicitly. But it's best to call `pp_bless` and get everything in the right place from the start.

```
pp_def(

    'dinner_count',
    Pars=>'in(col=3,m); day(); [o]out()',
    GenericTypes=>['L'],
    Code=><<'EOC',

$out() = 0;
loop(m) %{
 if( ($day() < 0) || ($in( col=>0 ) <= $day()) )
     $out() += $in( col=>2 );
%}
EOC

    PMCode => <<'EOPM',

sub dinner_count {

    my($in, $day, $out) = ( ($_[0]), ($_[1] // -1), ($_[2] // PDL::null()));
    _dinner_count_int($in, $day, $out);
    return $out;

}
EOPM
);
```

Here's the second (and last) PP definition – this one returns a string listing dinner attendees by day. It might be better accomplished using Perl, but this demonstrates how to return a Perl SV from a PP routine. There's no clean mechanism to do so, so it works by modifying an SV in place and passing that out through the PMCode wrapper. The PMCode wrapper generates an empty string SV and passes in a reference to it. `dinner_plan` accesses the referred-to SV, and appends to the contained string. That same sort of hand-back works for other Perl types that aren't PDLs, as well. There is an explicit `threadloop` in the `dinner_plan` PP code, to avoid reallocating the char buffer on every thread iteration.

```
  pp_def( # Returns a Perl scalar by handing a ref into the compiled code

    'dinner_plan',
    Pars=>'in(col=3,m); day()',
    OtherPars=>'SV *rv',
    GenericTypes=>['L'],
    Code=> <<'EOC',

char buffer[BUFSIZ];
threadloop %{

    int day = $day();
    int nguests = 0;
    PDL_COMMENT(" sv is a Perl scalar in which we want to return a value. ")
    PDL_COMMENT(" We do this by passing in a ref to it, and following the ref here.")
    SV *sv = SvRV($COMP(rv));
    if(day<0) {    /* Negative input day means ''use the last day''.  */

        loop(m) %{  /* Accumulate the maximum value of the day field */
            if($in(col=>0) > day)
                day = $in(col=>0);
        %}
```

```
        }
        loop(m) %{  /* Accumulate dinner guests for this day */
            if($in(col=>0) <= day)
                nguests += $in(col=>2);
        %}
        /* Append a summary of the day to the passed-in string */
        sprintf(buffer,"Day %d (%s): plan for %d\n", day, dow(day), nguests);
        sv_catpv(sv, buffer);
    %}
    EOC
    PMCode => <<'EOPM'
    sub dinner_plan {

        my($in, $day) = ( ($_[0]), ($_[1] // -1) );
        my $s = "";                          # Generate an empty string $s
        _dinner_plan_int($in, $day, \$s);   # hand in a ref to $s (to be modified inplace)
        return $s; }

    EOPM
    );
```

Finally, we have to finish the output with:

```
    pp_done();
```

The earlier declarations only modify package-global variables in the PP code generator. `pp_done()` does the actual work of generating the files.

3.2.3 Using PPDemo::Hello

Here's a sample interaction with the perldl shell. User input is in bold.

```
    pdl> use PPDemo::Hello
    pdl> $a = new PPDemo::Hello
    pdl> $a->event( pdl(1,6,7,8), pdl(-1,-1,2,3) );
    pdl> $a->tomorrow;
    pdl> $a->event( 1, 3 );
    pdl> $a->event( 9, 1 );
    pdl> p $a
    PPDemo object -  days: 2;  events: 6/100;  ids: 2
    [
     [ 0   1 -1]
     [ 0   6 -1]
     [ 0   7  2]
     [ 0   8  3]
     [ 1   1  3]
     [ 1   9  1]
    ]
    pdl> p $a->dinner_plan();
    Day 1 (Mon): plan for 7
    pdl> p $a->dinner_plan(pdl(0,1,2));
    Day 0 (Sun): plan for 3
    Day 1 (Mon): plan for 7
    Day 2 (Tue): plan for 7
    pdl> p $a->dinner_count
    7
    pdl>
```

The object works as expected, and the C and `Perl` aspects merge (practically) seamlessly. In particular, note that `dinner_plan` threads normally - but the output is still just a Perl scalar string that happens to contain all the days, summarized by line.

4 Using PP for Calculations (no dataflow)

The most basic use of PP is to produce a single output PDL that is obtained by acting on one or more inputs. The examples in §2-§3 all happen to be simple calculation cases, producing a single output. But you can return any value or multiplet of values you want, by specifying multiple [o] variables in the `Pars` signature, or by accepting refs in the `OtherPars` field. Direct assignment, temporary variables, and Perl wrappers have all been demonstrated in §2-§3 but are described here in §4.1-§4.3 in more depth. Bad value handling is straightforward and in §4.4.

4.1 Using direct assignment

The simplest and most common type of PP operator calculates a value and stuffs it into an appropriate location in an output PDL. You design your algorithm to act on its simplest input data type (e.g. a scalar, a vector, a collection of vectors, or what-have-you), and write a signature to match that data type. You declare an output variable in the signature, assume it is allocated correctly for your input parameter, and assign your computed value directly to the element(s) of the output variable. That basic mode of operation is in most of the examples above, including §2.3.1.

Direct assignment doesn't necessarily have to be simple. Here is an example of a function that accepts an NxM array and returns its average (a scalar), its average-over-columns (an M-vector) and its average-over-rows (an N-vector):

```
pp_def('multisum',
       Pars=>'im(n,m); [o]av(); [o]avc(m); [o]avr(n);',
       Code => <<'EOC',

    $av() = 0;
    loop(n) %{ $avr() = 0; %} // initialize avr
    loop(m) %{
      $avc() = 0;              // initialize avc element
      loop(n) %{
        $GENERIC() pix = $im();
        $av()  += pix;
        $avc() += pix;
        $avr() += pix;
      %}
      $avc /= $SIZE(n);       // normalize avc element
    %}
    loop(n) %{ $avr() /= $SIZE(m); %}  // normalize avr
    $av() /= $SIZE(m) * $SIZE(n);      // normalize av

EOC
Doc => <<'EOD'
=for ref
Sample PP code uses direct assignment to a scalar and two vectors. The input has
two active dims named 'n' and 'm'.  We loop over both, accumulating a
(scalar) average of the whole input matrix, a (m-vector) average across
each row, and a (n-vector) average across each column.

=cut
EOD
```

```
    );
```

Here, we accumulate sums three different ways and then normalize them appropriately. Note that the `m` and `n` values are automatically handled right by the index macros inside the nested loops. The `$av()` macro gets indexed with nothing, the `$avc()` macro gets indexed with the implicit value of `m`, and the `$avr()` macro gets indexed with the implicit value of `n`.

4.2   Temporary Variables to Ease the Pain

Nearly any nontrivial calculation requires holding intermediate values in local variables. For scalar quantities this is simple – you can declare a C variable in the Code section of your PP declaration. If you want to shave a few clock cycles off your calculation, you can even declare the C variable outside an explicit `threadloop %{ %}` block, to prevent it being pushed onto the stack (and then popped off again) every time the thread loop executes. For vector quantities it is not so simple - C requires you to track the size explicitly, and generally to allocate the memory too.

Rather than explicitly declaring an array that is just the right size, you can declare a temporary variable in the signature of your PP function, and use it just like a passed-in PDL. The access macros will work correctly, and the temporary variable is typed just exactly like other arguments (e.g. `$GENERIC()` or other type as declared). The temporary variable is declared just large enough to hold one instance of its declared active dims – any thread dimensions are ignored, so you don't allocate extra memory you don't need. (Warning: Temporary variables are currently not threadsafe in PDL 2.007. Threadsafe temporaries are a planned feature for PDL 3.)

Here's an example that uses a temporary variable to stash the denominator and discriminant of the quadratic equation. These could have been declared as scalar variables - putting them in a temporary PDL demonstrates autodeclaration of a vector quantity.

```
no PDL::NiceSlice;
*solve_quad = \&PDL::solve_quad;
use Inline Pdlpp => <<'EOF';
pp_def('solve_quad',

    Pars=>'coeffs(n=3); [o]sols(s=2); [t]parts(s);',
    GenericTypes=>[F,D,S],
    Code => <<'EOC',

/* Stash the denominator and discriminant */
$parts(s=>0) = 1.0 / ($coeffs(n=>2) * 2);
$parts(s=>1) = $coeffs(n=>1) ** 2 - 4 * $coeffs(n=>2) * $coeffs(n=>0);

if($parts(s=>1) >= 0) {

    /* One or more solutions exist.  Put them in the output. */
    $parts(s=>1) = sqrt($parts(s=>1));
    $sols(s=>0) = $parts(s=>0) * ( -$coeffs(n=>1) - $parts(s=>1) );
    $sols(s=>1) = $parts(s=>0) * ( -$coeffs(n=>1) + $parts(s=>1) );

} else {


    /* No real solutions exist.  Set answer to NaN.  notice the type selection macro! */
    $sols(s=>0) = $sols(s=>1) = $TFDS( union_nan_float.f, union_nan_double.d, -32768 );

}
EOC
);
EOF
```

The 'solve_quad' example solves a quadratic equation analytically. The discriminant and denominator of the quadratic formula are used more than once in valid solutions, so they get stashed in the temporary variable parts, which is declared as a 2-vector. In the event that the discriminant is negative, no real solutions exist, and the routine returns the appropriately typed NaN. In cases where a specific action is needed for a specific type, you can use the $T macro – you list the $T followed by a collection of one-letter type identifiers. The code switches between the strings in the argument list, based on the current value of $GENERIC() – so the floating-point version of the code ("F") gets "union_nan_float.f" and the double-precision version ("D") gets "union_nan_double.d". Each of those expressions contains an appropriate NaN value for that data type. Short types ('S') can't support NaN, so we supply -32768 instead. This would also be a good place to use BAD values instead of NaN (see §4.4).

4.3   Conditioning arguments with Perl

Surprisingly often you might want to change the default behavior for a PP operator, or prepare or curry arguments in some way, before entering the actual algorithm. For non-dataflow PP operators, there's a mechanism to do that. If you specify a field called PMCode in your PP declaration, the compiled stuff from the Code field will go into _subname_int instead of just subname. Your PMCode field can then contain some Perl that declares subname and (optionally) springboards into _subname_int. Here's an example: a routine called minismooth that accepts a 2-D PDL image and returns its input, smoothed by a quasi-minimum operator. We use a temporary variable, too, to demonstrate how to use it from a Perl-wrapped call.

minismooth is useful, for example, for finding a smooth background image in the presence of spiky foreground objects such as a starfield: the stars are eliminated in the local minimum-smoothing process, leaving locally "typical" values for the background.

The arguments to minismooth get conditioned by the PMCode, so that it can be called more flexibly than a naked PP operator can.

The minismooth algorithm itself is straightforward. There are two active dims – the height and width of the image. The code loops over all pixels in the image, and for each one it loops over the entire neighborhood, accumulating the n lowest values in the image. The output pixel at the current location gets the nth lowest value from the neighborhood.

(Note that in versions of PDL up to at least 2.009, this type of declaration (using PMCode) only works inside a standalone module definition. A long-standing wart in Inline::Pdlpp prevents the PMCode from being interpreted.)

```
    pp_def('minismooth',

        Pars=>'im(n,m); [o]sm(n,m); [t]list(n);',
        OtherPars=>'long size; long nth;',
        PMCode => <<'EOPMC',

    =head2 minismooth

    =for usage


        $sm = minismooth( $im, $size, $nth );


    =for ref

    C<minismooth> smooths an image by finding the minimum (or near-minimum) over
    the C<$size>xC<$size> block of pixels around each source pixel.  If you specify
    C<$nth>, then the nth smallest pixel in each neighborhood is used, instead of the
    absolute smallest.

    =cut
    sub PDL::minismooth {
```

```
    my $im = shift;
    die "minismooth requires a PDL!" unless( UNIVERSAL::isa($im,'PDL'));
    my $size = shift // 21;
    my $nth = shift // 1;
    $nth = 1 if($nth  < 1);
    $size=1  if($size < 1);
    my $sm = PDL->null;
    PDL::_minismooth_int($im, $sm, PDL->null, $size, $nth);
    return $sm;

}
EOPMC

    Code => <<'EOC',

{

    long i,j,k,l,ii,jj; // declare index variables
    PDL_Indx = ($COMP(size)-1)/2;
    long nn, mm;
    long n_kept;
    $GENERIC() current;
    for(i=0; i<$SIZE(m); i++) {
        for(j=0; j<$SIZE(n); j++) {
            n_kept = 0;
            PDL_COMMENT("Loop over the neighborhood around the current pixel");
            for(k=-sz; k<=sz; k++) {
              for(l=-sz; l<=sz; l++) {
                nn = j+l;
                if(nn<0) { nn=0; } else if(nn>=$SIZE(n)) { nn=$SIZE(n)-1; }
                mm = i+k;
                if(mm<0) { mm=0; } else if(mm>=$SIZE(m)) { mm=$SIZE(m)-1; }
                current = $im(n=>nn, m=>mm);
                if( (n_kept < $COMP(nth)) || current < $list(n=>0) ) {
                  PDL_COMMENT("The current number fits on the minimum-values list");
                  PDL_COMMENT("Use insertion sort to keep the list sorted");
                  for(ii=0; ii<n_kept && $list(n=>ii) >= current; ii++) { ; }
                  if(n_kept < $COMP(nth)) {
                    PDL_COMMENT("Few enough elements - put 'em on the back");
                    for(jj=n_kept; jj>ii; jj--) { $list(n=>jj) = $list(n=>jj-1); }
                    n_kept++;
                  } else {
                    PDL_COMMENT("List is full - bump something off the front");
                    ii--;
                    for(jj=0; jj<ii; jj++) { $list(n=>jj) = $list(n=>jj+1); }
                  }
                  $list(n=>ii) = current;
                } // end of list manipulation
              } // end of l neighborhood loop
            } // end of k neighborhood loop

            $sm(n=>j, m=>i) = $list(n=>0);
        } // end of j pixel loop
    } // end of i pixel loop

} // end of PP code
EOC
);
```

4.4 BAD values

PDL's BAD value system uses in-band special values to mark bad or missing points in your data. Because bad values cost extra computing time (every value handled by a bad-aware PP operator has to be checked against the bad value by the operator), PP lets you specify two different versions of your code - one that handles bad values and one that doesn't. The two parallel copies of the code should implement exactly the same function, but one with bad value handling and one without.

Each PDL carries a status flag (the badflag) that indicates whether there *might* be bad values in the PDL: if the badflag is 0, then the PDL is known to have no bad values and the normal Code gets used. If it is 1, then the special BadCode gets used.

You can create a BAD-aware PP operator by setting the `HandleBad` field in the hash you pass to `pp_def`. If HandleBad is set to a true value, then PP looks for both a `Code` section and a separate `BadCode` section. The `BadCode` gets used on PDLs with a true BadFlag and the `Code` gets used on everything else.

If you do not do anything about BAD values, then your code will ignore them by default – but will pass its parameters' BadFlag status on to any output PDLs. If BAD values will cause your code trouble, you can set the `HandleBad` field (in the pp_def argument hash) to 0. That will cause PP to throw an error if your code gets a parameter with the BadFlag set.

Inside either your `BadCode` or your `Code` section, the following macros will come in handy:

$ISBAD(*var*) This boolean macro returns TRUE if the contained value in var is bad. The *var* is a regular PDL access macro just like you'd use to access any of the PDLs in the signature, except that it doesn't have a leading `$` – as in "`$ISBAD(a(n=>2))`".

$ISGOOD(*var*) This is the counterpart to $ISBAD, and returns a flag indicating goodness.

$SETBAD(*var*) This assigns the BAD value to the corresponding PDL argument. The *var* uses the same format as `$ISGOOD` and `$ISBAD`. Unlike those two, it makes sense to use in a `Code` section (since you may have to set the first BAD value in a particular PDL). In the `Code` section, if you set a value to BAD you should also set the PDL's badflag with the `$PDLSTATESETBAD()` macro, below

$ISBADVAR(*c_var, pdl*) This works like `$ISBAD`, except that the first argument is a cached value of the PDL in a `$GENERIC()` type C variable. The second argument is the name of the PDL it came from (which must be one of the Pars in the signature).

$ISGOODVAR(*c_var, pdl*) This works like `$ISGOOD`, except that the first argument is a cached value of the PDL in a `$GENERIC()` type C variable. The second argument is the name of the PDL it came from (which must be one of the Pars in the signature).

$SETBADVAR(*c_var, pdl*) This works like `$SETBAD`, except that the first argument is a cached value of the PDL in a `$GENERIC()` type C variable. Just like `$SETBAD`, you should also set the state of the PDL to BAD using `$PDLSTATESETBAD()` if you use this inside a `Code` block.

$PDLSTATEISBAD(*pdl*) This accesses the *pdl*'s badflag, which indicates whether the PDL *may* have BAD values in it (and therefore requires bad-value checking). You don't normally have to read the badflag explicitly, since PP does that for you (and shunts execution into your `Code` or `BadCode` segment accordingly). The *pdl* argument is the name of a parameter in the `Pars` field of `pp_def`.

$PDLSTATEISGOOD(*pdl*) Counterpart to `$PDLSTATEISBAD`.

$PDLSTATESETBAD(*pdl*) This sets the *pdl*'s badflag. You need to do this explicitly if you set a value to BAD inside your `Code` block. The *pdl* argument is the name of a parameter in the `Pars` field of `pp_def`.

$PDLSTATESETGOOD(*pdl*) This clears the *pdl*'s badflag. You shouldn't *need* to do this, but you might like to if you are in a position to *know* that none of the elements of the *pdl* are BAD.

4.4.1 BAD value handling: a simple case

Here's a simple example of BAD value handling. The `countbad` operator collapses a vector PDL by counting the number of BAD values in it.

```
pp_def('countbad',

      Pars=>'in(n); [o]out();',
      HandleBad => 1,
      Code => << 'EOC',

$out() = 0;
EOC

      BadCode => <<'EOBC'

long bc = 0;
loop(n) %{

      if( $ISBAD(n) )

            bc++;

%}
$out() = bc;
EOBC

      );
```

The code is pretty straightforward. If the input PDL has its `badflag` clear, i.e. it has no BAD values, then the Code section gets used. It just returns 0. If the input PDL has its `badflag` set, the BadCode gets used. It loops over the input values, counting the bad values, and returns the count. The count is cached in a local variable, `bc`, to avoid the overhead of making the `$out()` macro call for each iteration of the loop.

4.4.2 BAD value handling: marking values BAD.

Here's an example of how to mark values BAD. The `recip` operator implements reciprocals, with checking for the 1/0 case. There are two copies of the reciprocal code – the `Code` block and the `BadCode` block. In the `Code` block, no bad values are expected in the input, but we may have to generate some for the output. In the `BadCode` block, the input may contain bad values, but the badflag of the output is automatically set before our code block gets called. See also the example in Section 4.5.

```
pp_def('recip',

      Pars=>'in(n); [o]out();',
      HandleBad => 1,
      GenericTypes=>['F','D'], # only makes sense for floating-point types
      Code => << 'EOC',

if ($in()==0) {

      $PDLSTATESETBAD(out());
      $SETBAD(out());

} else {

      $out() = 1.0/$in();

}
EOC

      BadCode => <<'EOBC'
```

```
if ( $in()==0 || $ISBAD(in()) ) {

    $SETBAD(out());

} else {

    $out() = 1.0/$in();

}
EOBC

    );
```

4.5   Adjusting output dimensions (RedoDimsCode)

Sometimes you don't know the dimensionality of an output variable at compile time – that is to say, the `$SIZE()` of one of the named dims in a signature argument (in the `Pars` section of `pp_def`) may require some computation to figure out. PP can handle that, in a limited way. You accomplish that by passing in the `RedoDimsCode` field into `pp_def`. Your `RedoDimsCode` is called during setup for the computation (before the actual computation in the `Code` and/or `BadCode` sections), and you can treat the `$SIZE()` macro as an lvalue (i.e. assign to it).

Many of the same other macros that are available in the `Code` and `BadCode` sections are available to you in the `RedoDimsCode` section. The ones that are unavailable `loop(`*foo*`)` and `threadloop` loop-management macros, and the data-access `$`*var*`(`*dim=>expr...*`)   ` macro.

This lets you adjust the size of your output and/or temporary parameters before carrying out the computation. Any thread dimensions from the overall calculation get added onto the end of the `[o]` output variables' dimlists automatically, after you set the `$SIZE()`. (That doesn't happen for `[t]` temporary variables, since each temporary variable only gets its active dims – so no thread dims get tacked on the end.)

The adjustment is a bit limited, though – you can only compute the new size based on the information available *before* the computation takes place, since no actual computation has happened before the `RedoDimsCode` gets called. That works easily for operations where you know the output dimensions from the input parameters or input dimensions, and it's common to see a one-line `RedoDimsCode` section in those cases. The only catch is that the `$SIZE()` macro is write-only at that stage, so you have to read any dimensions directly from the C structure of the PDL itself.

Operations that output sets (such as `which()`) require actually walking through the data to figure out the output dimensions. That can be tricky, since the threading loop constructs aren't available in `RedoDimsCode`. The `which()` operator and its relatives use a `PMCode` section to flatten the input PDL, and treat it as a single array which can be looped over explicitly with a C `for` loop. That trick can work for n dimensional inputs as well, provided that you know n when you are writing the code.

Here is an example of a simple `RedoDimsCode` that uses just the input dimensions to change the output. For more complex examples (including the `which()` trick and more powerful techniques such as are used by `range()`), see Section 6.

The `increments` operator accepts a collection of n number and returns the differences between adjacent numbers. It has one fewer output elements than input elements. If you feed in a single element, you get back an Empty piddle (size 0) and if you feed in an Empty piddle you get back another Empty piddle. There's both a `Code` and a `BadCode` segment to demonstrate BAD handling as well, but the important elements are (a) the `RedoDimsCode` that sets the size of the `m` dimension, and (b) the explicit loop over the m dimension (which could also have been handled with an explicit C loop and a local index variable). The `RedoDimsCode` uses the `$PDL(`*var*`)` macro to get to the underlying C struct. The struct field `ndims` reports the number of dimensions in the PDL; the `dims` array contains the sizes.

(Note that, because the inputs to `increments` have different dimensionality, it cannot handle inplace processing. If the `inplace` flag for in is set, `increments` will ignore it.)

```
pp_def('increments',
```

```
        Pars=>'in(n); [o]out(m);',
        RedoDimsCode=><<'EORDC',

$PDL_COMMENT(`` Check the dimension of 'in', and calculate output size. '');
if ( $PDL(in)->ndims && $PDL(in)->dims[0]>0 )

        $SIZE(m) = ( $PDL(in)->dims(0)-1 )

else

        $SIZE(m) = 0;

EORDC

        Code=><<'EOC',

loop(m) %{
 $out() = $in(n=>m+1) - $in(n=>m);
%}
EOC

        BadCode=><<'EOBC',

loop(m) %{

        if($ISGOOD(n=>m) && $ISGOOD(n=>i)) {
            $out() = $in(n=>m+1)-$in(n=>m);
        } else {
            $SETBAD(out());
        }

%}
EOBC

        );
```

The `RedoDimsCode` in `increments` explicitly shortens the output dimension by 1 compared to the input dimension; the rest just calculates the difference between adjacent elements and handles BAD values.

5   Using PP With Dataflow

PP does not only allow you to do explicit, immediate calculations - your PP operators can set up dataflow links between the input and output. This is implemented through a quasi-object implemented in C, called a *trans*. A trans contains all the information needed to link two PDLs.

   The dataflow engine was implemented by Tuomas Lukka in the early 2000s, with the intent that you could construct complete behind-the-scenes data pipelines: modify a particular input PDL, and the result could automagically cascade through an entire calculation to change the output PDL. As it turns out, the most common use case for dataflow has been *multiple representations* of the same underlying data. Therefore, all of the built-in selection operators in the language implement dataflow, while the calculation operators do not.

   With the information in this section, you can implement dataflow operations for either selection operators or calculation operators. Dataflow operators in general construct trans structures that relate their input and output PDLs, but their PP declarations are nearly the same as normal computed PP operator declarations. Some additional fields to `pp_def` are required, to set up the additional code structures for the underlying trans object.

5.1 Code/BackCode and a basic dataflow operator

You can implement a calculated dataflow operator with lazy evaluation, simply by marking it as a dataflow operator and writing a BackCode section. Here's an example of a simple two-way computed dataflow operator, to convert Fahrenheit to Celsius and vice versa.

```
pp_def('FtoC',

     DefaultFlow => 1,    # Mark as a dataflow operator
     NoPdlThread => 1,    # Turn off threadsafe generated code (doesn't work)
     P2Child => 1,        # Declares $PARENT and $CHILD parameters
     Reversible => 1,     # Enable BackCode
     RedoDims =><<'EORD' # No RedoDims by default - this just copies $PARENT dims to $CHILD.

          long ii;
          $SETNDIMS($PARENT(ndims));
          for(ii=0; ii<$PARENT(ndims); ii++) {

               $CHILD(dims[ii]) = $PARENT(dims[ii]);
               $CHILD(dimincs[ii]) = $PARENT(dimincs[ii]);

          }
          $CHILD(datatype) = $PARENT(datatype);
          $SETDIMS();

EORD

     Code => '$CHILD() = ($PARENT() - 32) * 5/9; ',
     BackCode => '$PARENT() = ($CHILD() * 9/5) + 32; '

);
```

Most of the characters in the operator are in the `RedoDims` section, which copies the parent dimensionality (and internal `dimincs` fields) into the child. This could/should be done by default, but for PDL versions through 2.009 this particular use case does not work properly in the PP compiler, so an explicit `RedoDims` is required. The `RedoDims` code sets the child's dimensionality and `dimincs` counters to be equal to the parent's. The `Code` converts a single element of the `$PARENT` from Fahrenheit to Celsius, and stores it in the `$CHILD`. The BackCode converts a single element of the `$CHILD` from Celsius to Fahrenheit and stores it in the `$PARENT`. You use FtoC as follows:

```
perldl> $F = pdl(32);
perldl> $C = $F->FtoC;
perldl> p $C;
0
perldl> $C++; p "$F F = $C C\n";
33.8 F = 1 C
perldl> $F++; p "$F F = $C C\n";
34.8 F = 1.55555555555555 C
perldl>
```

Assigning to either `$F` or `$C` with computed assignment (".="), or modifying `$F` or `$C` with side-effect operators, automatically updates the other one. A dataflow connection has been established.

5.2 Structure of a dataflow operator

Dataflow operators make use of several stages of code execution. In a typical use case, at call time the operator defines a trans that involves computing some input parameters and/or examining input data. Then it sets the dimensions of the output PDL. The actual calculation is deferred until the output PDL is used. This "lazy evaluation" is a wart that's left over from the development history of dataflow in PDL, but could

in principle be extended to elementwise lazy evaluation. (At present, all operators calculate the results for every element in a PDL, as soon as the first element is accessed). These different steps are implemented with different `pp_def` fields.

Dataflow operations work by creating a separate structure/object called a "`trans`" that relates two PDLs – the parent and the child. The trans structure includes metadata about the transformation and also pointers to the code blocks you declare in the `pp_def` call. Its elements are accessible with the `$PRIV()` macro, which expands to C code that references the private C struct.

The `pp_def` fields you need to know for a basic dataflow operator are:

Pars The signature (as in a regular computed operator). There are some caveats – see "`P2Child`" and "`DefaultFlow`", below.

OtherPars Any additional parameters not subject to the usual threading rules (which could be the main parameters – see the `rangeb` operators in `slices.pd` for an example)

Doc POD format documentation for the operator

HandleBad Same as for a regular computed operator – ignore for default treatment, set to 1 to enable the `BadCode` (and `BadBackCode`) fields, set to 0 to cause the code to barf if it receives a PDL with the badflag set.

Code Just like a normal PP operator, this computes the output PDL value from the input PDL values.

BackCode This should compute (and set) the value of the input PDL from the output PDL. It gets called if the output PDL is changed by some other operator, to flow data back into the input PDL. You do not need to specify BackCode for one-way flow. (See the `Reversible` flag, below)

DefaultFlow You almost certainly want to set this flag to 1. It sets up a dataflow relationship between two special parameters, "`PARENT`" and "`CHILD`", that should be declared in the `Pars` field (or in the `P2Child` field, see below). If you don't either use the `P2Child` flag or declare `PARENT` and `CHILD` in the Pars field, PP will throw an error.

Reversible This flag indicates whether the transformation is two-way. If it is, then you need to specify both `Code` and `BackCode` segments (or `EquivCPOffsCode` – see below).

P2Child This flag takes the place of `Pars` for simple dataflow. Setting the flag is equivalent to declaring `Pars=>'PARENT(); [oca]CHILD();'`. It declares a zero-dimensional input `PARENT` and a zero-dimensional output `CHILD`.

NoPdlThread This is a flag that prevents the normal threading behavior of PP. If it is set, no threadloop will be executed around (or within) the various `Code` blocks where it would normally happen. That is useful for edge cases where you might want to work independently of the normal threading mechanism (see Chapter 6).

Comp Should be a C declaration of some fields to be computed (and stashed) during setup of a trans. If this is present, then the `OtherPars` parameters are not automatically incorporated in the array as with conventional computed operators. (That happens with a default `Comp` and `MakeComp` field that gets overridden by your `Comp` and `MakeComp`.) See `MakeComp` for details. The `Comp` field has (limited) access to the `$COMP()` macro itself, specifically so that you can declare variable-length arrays here. In particular, constructs like "`int b; int a[$COMP(b)]`" work. Before using those arrays (in the corresponding `MakeComp` code block) you have to set the variables that hold the sizes, then run the macro `$DOCOMPDIMS()`. That allocates all the variable length arrays.

MakeComp Should contain code to compute and stash the contents of the `$COMP()` macro, called the computed stash. This is the place to store any state or parametric information about the trans itself. The

`OtherPars` parameters are accessible in this block of code as regular C identifiers – so if you pass in an OtherPar called `foo`, you access it here as `foo` and not as `$COMP(foo)`. If you don't specify `MakeComp`, you get a default `MakeComp` block that just copies any `OtherPars` into the `$COMP()` structure.

`RedoDims` Should contain code to compute and stash the dimensions of the child PDL, from the dimensions of the parent PDL and the parameters passed in.

`EquivCPOffsCode` This field is useful for simple dataflow relating arbitrary locations in one PDL to arbitrary locations in another, for example inside the built-in `range` and `index` operators. It contains code for relating particular elements between the two linked PDLs (parent and child). Its name derives from "equivalent child/parent offsets", and the code should implement direct-copy dataflow between a single element of the special `$CHILD()` and `$PARENT()` piddles. If present, `EquivCPOffsCode` replaces `Code`, `BackCode`, and (if `HandleBad` is set) `BadCode` and `BadBackCode` – these all end up running the `EquivCPOffsCode`, with the appropriate assignment placed inside to slosh data forward or backward. The `EquivCPOffsCode` should loop over all values for which dataflow should happen. For each of those values, it needs call one of the special macros `$EquivCPOffs($pi,ci$)` or `$EquivCPTrunc($pi,ci,oob$)` for each value. Here, pi is the computed offset in the parent to a particular element, ci is the offset to the corresponding element in the child, and oob is a flag indicating that the offset is out of bounds in the parent (and should therefore be loaded with BAD or 0 in the child for forward flow, or ignored for reverse flow). Note that `EquivCPOffsCode`, unlike `Code` and `BackCode`, does not deal with threading! You have to do the looping yourself.

The calling structure is as follows. When you first invoke the operator, `MakeComp` gets called (making the `$COMP()` variable macros available), then `RedoDims`. Only when the user makes use of the resulting child PDL does the `EquivCPOffsCode` (or `Code` or `BackCode`) get called.

Good examples of dataflow operators are to be found in the PDL distribution itself, in ".../Basic/Slices/slices.pd". Check out the code for `slice` and `sliceb`, or `range` and `rangeb`. Those operators are implemented without the `PMCode` field, because `PMCode` seems to not work correctly for dataflow operators. The `sliceb` operator uses a special kind of trans called an "affine" trans, which doesn't actually use calculation code or `EquivCPOffs` at all – it causes the child PDL to point directly at the parent's data structure. The `rangeb` operator generates a data structure containing information about each range to be extracted, then walks through the source data calling `EquivCPOffs`.

6 PP on the Edge (things PP can't do easily, and how to do them)

PP is great for performing ordinary vectorized calculations where the dimensions are known or computable up front. There are certain problem domains at which it is awkward – for example, allowing an operator to vary its own signature at runtime (changing the number of active dims) is difficult, as is accepting a variable length argument list. Here are some basic strategies for writing operators that are right at the edge of what PP can do easily.

6.1 Variable Active Dims

The most obvious example of an operator with variable active dims is the built-in `indexND`. That operator collapses an index variable by lookup into a source array: the 0 dim of the index is treated as running across dimension in the source array, and each row of the index variable causes one element of the source array to be indexed. Depending on the size of the index variable's active dim, the signature should look like one of the following:

```
PARENT(i0);         index(n=1); [o]CHILD());
PARENT(i0,i1);      index(n=2); [o]CHILD());
PARENT(i0,i1,i2);   index(n=3); [o]CHILD());
```

```
    PARENT(i0,i1,i2,i3); index(n=4); [o]CHILD());
    # etc.
```

PP can't handle that type of operator easily and straightforwardly, because the shape of `PARENT` depends on the size of one of the dimensions in `index`.

The meat behind `indexND` is the slightly more general `rangeb` operator, which pulls a specified rectangular block out of the source PDL at each indexed location. It can be found in the PDL source distribution in the file "`Basic/Slices/slices.pd`". It works by sidestepping the threading behavior entirely. The `PARENT` and `CHILD` are passed in with 0 active dims via the `P2Child` flag to `pp_def` (which autodeclares a basic parent/child signature). All other parameters (in particular the index and size) are declared as `OtherPars` with type "`SV *`" so that the code has direct access to the Perl variable containing the index. PDL provides a function, `PDL->SvPDLV()`, that accepts a `SV *` and returns the corresponding `pdl *`, so you can access the internal structure directly.

The `rangeb` operator has three key parts: the `MakeComp` code gets executed first, and extracts all the relevant size and offset information from the input PDLs into C arrays of known size. Then the `RedoDims` code block calculates the size, dimincs (memory stride for each dimension) and datatype of the `CHILD` (output) variable, from the input. It uses the `$SETDIMS()` macro to allocate the child's actual dimensions and data block, and reproduces the regular threading rules used by PP. Finally, the `EquivCPOffsCode` gets called to copy values into the new `CHILD` piddle. The `EquivCPOffsCode` for `rangeb` iterates explicitly over all input dimensions including thread dimensions. Iterating over a variable number of dimensions is tricky; there's a nice example at the bottom of the EquivCPOffsCode for rangeb. The main loop is a `do/while` construct, with an explicit iterator at the end. The iterator is a `for` loop that handles carry from the fastest-moving to the next-fastest-moving index, and on up to the slowest-moving index.

6.2   Variable Argument Counts

The best way to handle a variable argument count is to use a Perl-side currying function (`PMCode` in a module or a separately declared function in an inline script) that stuffs the variables into a Perl array/list, then pass a ref to the array into the PP operator as an `OtherPars` parameter. Your code can dereference the ref and traverse the array (`AV *`) directly. Each element of the AV will be a separate `SV *` that you can parse with your own code, and/or convert to a `pdl *` . See the `perlguts` and/or `perlapi` man page for details on how to manipulate Perl AVs.

6.3   Returning Perl Variables

There is no explicit SV return mechanism in the `Code` blocks of a PP function If you need to return a single SV value to a Perl function, you can pre-allocate a variable with a Perl-side currying function (`PMCode` in a module or a separately declared function in an inline script), then pass in a ref to the value as an `OtherPars` parameter. Your PP code can dereference the ref to get an lvalue SV, and assign to that SV. Then the currying function can return the variable's value.

The same mechanism works for lists – if you need to return several SVs, you can construct an array in your currying function and pass a ref to it into the actual PP code. Then the C side can populate the referenced AV with SVs to be returned by the currying function.

7   PDL internals: accessing PDLs directly from C

PDL data structures are directly accessible from your C code within PerlXS. Most times you want to use PP to access the structure, but (particularly if you are using an external library) you may want to get in and mess around with the internals yourself. Here is a brief overview to get you going. The most powerful way to access PDLs is via PP (Section 2), but both direct access and a module called `CallExt` offer slightly lighter-weight forms of access.

From within Perl, PDLs are blessed scalar refs. The scalar that is referenced contains an IV (Perl integer value) that is itself a pointer to a C structure (`struct pdl`). If you #include both `<pdl.h>` and `<pdlcore.h>`, you get access to the data structures and to a variety of important access routines. You do *not* need to link to a PDL library to access this rather extensive C API.

When Perl executes "`use PDL;`", a "core" structure (called "`PDL`" on the C side) is created that contains direct function pointers to the utilities. The core structure is a static patchtable that allows all interested modules to call the same C functions – otherwise the Perl dynamic linking code would link a separate copy of the PDL core functions for each module that uses them. If you are using PP, a pointer to this patchtable is automatically placed into a C file-scope static variable called "`PDL`" – hence utility routines are accessed from C as (e.g.) "`PDL->SvPDLV(foo)`".

[If you want to use the API from a non-PP XS module, then the header section of your PerlXS file should declare a `struct pdlcore *` called `PDL`, and the `BOOT:` section of your perlXS file should retrieve the value of this pointer from the Perl global variable `$PDL::SHARE`. (You must also ensure that the pointer has been initialized, e.g. with a "`use PDL;`" in the Perl portion of your module).]

For a full list of the routines in the PDL C API, you currently need to examine the online documentation and source code comments. A separate document is *planned* to describe these PDL internals, but the online documentation is extensive and should get you going.

The very simplest, and preferred, way to access the data in a PDL from C/XS is to wrap your code in a PP call and use the PP macros. For cases where that will not work easily, you can have your routine accept an SV * (Perl scalar) and run `PDL->SvPDLV` on the `SV *`. The return value is a `pdl *`, i.e. a pointer to the actual PDL internal structure. Before accessing the data directly from C without the PP macros, you must check the status flags, non-nullness of the data field, and data type of the PDL. You will also need to cast the `data` field to the appropriate type of pointer before dereferencing it. The data are structured using the `dimincs` array of the PDL structure – you multiply each element of dimincs by the index you want in the corresponding dimension, and summing those products gives you the offset into `data`.

7.1   The struct pdl type

The typedefed "`pdl`" type is a "`struct pdl`", which is declared in `pdl.h` in the PDL distribution. Here is a list of the fields in it. Nonstandard types are declared in `pdl.h`. The `PDL_Indx` type, in particular, is either a 32 bit or 64 bit unsigned value depending on your sysem architecture.

magicno (unsigned long) An unsigned long with a magic number (`PDL_MAGICNO`; declared in `pdl.h`) in it. This is a fence for debugging purposes.

state (int) A bit field containing flags that describe the state of the PDL. The flags are declared in `pdl.h`: use bitwise-and to mask them out of the state.

> **PDL_ALLOCATED** Data have been allocated for this PDL. Also implies that the data is up-to-date (unless the PDL_PARENTDATACHANGED flag is set).
>
> **PDL_PARENTDATACHANGED** This PDL is connected to another one via dataflow, and that other PDL has changed – but this one has not yet been updated.
>
> **PDL_PARENTDIMSCHANGED** This PDL is connected to another one via dataflow, and that other PDL's shape has changed – but this one has not yet been updated.
>
> **PDL_PARENTREPRCHANGED** The representation of the parent changed (for example, `sever()` broke it from its parent), so data access hooks like the `incs` field need to be recalculated.
>
> **PDL_ANYCHANGED** This is the bitwise OR of the previous 3 flags - i.e. the parent has been touched.
>
> **PDL_DATAFLOW_F** Track forward dataflow starting from this PDL into its children.
>
> **PDL_DATAFLOW_B** Track reverse dataflow back from this PDL into its parent.

PDL_DATAFLOW_ANY bitwise OR of the previous 2 flags - i.e. this PDL has a dataflow connection some-where.

PDL_NOMYDIMS this PDL is null (auto-reshaping to match needed shape in an expression).

PDL_MYDIMS_TRANS the dims are received through a trans (transformation structure) from another PDL.

PDL_OPT_VAFFTRANSOK It's okay to attach a virtual-affine trans to this PDL (i.e. to point another PDL at the same data block)

PDL_HDRCPY The hdrcpy flag (causes the header of this PDL to be autocopied into result PDLs when appropriate, in expressions and operators)

PDL_BADVAL The BAD flag for this PDL - if set, the PDL may contain BAD values and the BadCode gets executed instead of the normal Code.

PDL_TRACEDEBUG causes a bunch of PP internal debugging

PDL_INPLACE the inplace flag for this PDL - if set, the PDL has been primed for an in-place operation by the calling entity.

PDL_DESTROYING is set to indicate that this PDL is in the process of being destroyed, so data flow operations should be ignored.

PDL_DONTTOUCHDATA indicates that the `data` pointer is inviolate - don't change it, free it, nor use the datasv. Used mainly for piddles that mmap memory from files.

trans (pdl_trans *) This is a pointer to a transformation structure, which implements dataflow for PDLs that require explicit flow. The transformation structure includes code refs that are called by PP whenever this PDL is updated. It is NULL if the PDL has no dataflow "children". Complicated dataflow (such as indexing, dicing, and ranging) uses this mechanism.

vafftrans (pdl_vaffine *) This is a pointer to a more optimized transformation structure for PDLs with dataflow "children" that are related by an affine transformation. Affine transformations are transformations in which the indices in the child are related linearly to the corresponding indices in the parent (as in simple slicing). They use a different structure and are very fast, because no copying is needed – the same physical array in memory can be indexed by the related PDLs.

sv (void *) If this is non-NULL, it is a pointer back to the Perl SV that contains the PDL.

datasv (void *) This is either NULL or a pointer to a string Perl SV that contains the data in its data field. PDLs use the Perl memory management infrastructure, so the data block of a given pdl * is stored as a byte string in an anonymous Perl SV somewhere. The datasv should have a reference count of 1 for each PDL that accesses it. The ref count gets decremented when the PDL is destroyed, freeing the memory.

data (void *) This is the actual data pointer, and points to the string field in the associated SV. It can be null if the PDL contains no actual data. The type of the data depends on the value of the `datatype` field, below.

badvalue (double) If this PDL uses BAD value logic, then this is the value that is considered BAD. It is stored as a double regardless of data type of the PDL itself.

has_badvalue (int) This is a simple flag – 0 for straight-up values, 1 for PDLs that recognize bad values.

nvals (PDL_Indx) This is the total number of elements contained in the whole PDL – it's the product of the elements of the "dims" array. It is returned to Perl by `PDL::nelem`.

datatype (int) indicates the data type of the PDL. It's declared as an int, but is actually an enum - access it with one of the datatype enums in pdl.h (`PDL_B`, `PDL_S`, `PDL_US`, `PDL_L`, `PDL_IND`, `PDL_LL`, `PDL_F`, or `PDL_D`).

dims (PDL_Indx *) is a pointer to an array of sizes for each dimension of the data – the contents are returned to Perl by `PDL::dims` or `PDL::shape`. This can be allocated separately or point to a small preallocated space farther down, depending on the number of dimensions.

dimincs (PDL_Indx *) is a pointer to an array that caches the memory increments associated with each dimension in the PDL. Like `dims`, it can be separately allocated or simply a part of this `struct pdl`, depending on how many dimensions there are.

ndims (short) is the length of the dim list for this PDL. It is returned to Perl by `PDL::ndims`.