Slicing, Dicing and Threading dims with PDL

Fundamental to any vectorised data language such as PDL is the ability to manipulate subsets of data in convenient ways. PDL provides the facilities to change the size and dimensionality of data, to take contiguous and non-contiguous subsections of data along dimensions and to take completely arbitrary subsets of data meeting arbitrary criteria.

A key powerful feature is the ability to manipulate these subsets of data, and if desired to propagate these changes back to the original data **automatically**. This includes passing data to user-written subroutines, which may call standard external C code, which do not know or care about whether the data is a subset or not.

That sounds pretty abstract - but here is a concrete example: with PDL one could for example select all the pixels in an image greater than a certain value or meeting some other condition. This might serve to isolate a bright star or galaxy. One could then pass the pixel values and their locations to a photometry subroutine (which is just written to work on data arrays not caring whether it is a subset or not) which would fit the pixels with some model and replace them in the array. These changed pixels would then be automatically changed in the original image.

This sort of abstraction is extremely powerful as it allows for very concise and clear code. We'll start by looking at the simplest operations to extract simple slices of piddles, and look at increasingly more complex kinds of slices.

Finding piddle dimensions.

PDL data arrays can take arbitrary sizes and dimensions. Finding the current dimensions is straight-forward with the `dims` function which returns a list:

```
$data = zeroes(100,20,3);
print dims($data);
($nx, $ny, $nz) = dims($data);
```

The number of elements in a piddle is equally easy:

```
print nelem($data);
```

The slice function - regular subsets along axes

Earlier we saw how to extract a rectangular subset of a piddle:

```
$section = $gal(337:357,178:198);
```

The piddle `$gal` was a 2D image, we used array syntax (compliments of `PDL::NiceSlice`) to extract a contiguous subset ranging from pixel 337 to 357 along the first dimension, and 178 through 198 along the second. Behind the scenes, this is implemented by the `slice` function.

Use the on-line documentation:

```
perldl> help slice
```

to explore the full set of options. `slice` is probably the most frequently used PDL function so we will explore it in some detail. But first we notice that `slice` is implemented via a named function. Through the magic of `PDL::NiceSlice` and source filtering you can access `slice` functionality in a form very similar to the vector array syntax found in many array computation languages such as FORTRAN-90 and Matlab.

The basic slicing specification.

The argument syntax to `slice` is just a list of ranges, the simplest if of the form `A:B` to specify the start and end pixels. This generalises to arbitrary dimensions;

```
$data = zeroes(1000);
```

```
$sec = $data(0:20);
$data = zeroes(100,100,20);
$sec = $data(0:20,40:60,1:3);
```

Note that PDL, just like Perl and C, uses **ZERO OFFSET** arrays. i.e. the first element is numbered 0, not 1. Just like Perl you can use `-N` to refer to the last elements:

```
$data = zeroes(1000);
$sec = $data(-10:-1); # Elements 990 to 999 (last)
```

One can also specify a step in the slice using the form `A:B:C` where `C` is the step. Here is an example:

```
perldl> $x = sequence(24); # Create a piddle of increasing value
perldl> print $x
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
perldl> print $x(16:22:2)
[16 18 20 22]
```

Quite often one wants all the elements along many of the dimensions, one can just use ` `:` ' or just omit the specifier altogether:

```
perldl> $a = zeroes(10,20,3)
perldl> print dims $a(:,5:10,:)
10 6 3
perldl> print dims $a(,5:10,)
10 6 3
```

Omitting the range allows specification of just one index along the dimension:

```
$z = zeroes 100,200;
$col = $z(42,:); # Column 42 (Dims = 1x200)
$row = $z(:,42); # Row 42 (Dims = 100x1)
```

Since the second argument to slice is just a Perl character string it is easy to manipulate:

```
$x1 = 2; $x2 = 42;
$sec = $data($x1:$x2);
```

Modifying slices.

Here's the biggy:

```
perldl> $x = sequence(24);
perldl> print $x;
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
perldl> $slice = $x(4:20:2);
perldl> print $slice;
[4 6 8 10 12 14 16 18 20]
```

All very well. But now we modify the slice using the assignment operator.

```
perldl> $slice .= 0;
perldl> print $slice;
[0 0 0 0 0 0 0 0 0]
perldl> print $x;
[0 1 2 3 0 5 0 7 0 9 0 11 0 13 0 15 0 17 0 19 0 21 22 23 24]
```

Modifying the slice automatically modfies the original data! However it is done (`$slice++` etc. work just as well).

All the PDL slicing and dicing functions work this way, from the simplest rectangular slices to the most complex conditional slices. This is because they use a fundamental PDL feature known as **dataflow**.

Does a slice consume memory?

What if we have a big array and make a slice of most of it:

```
$x = zeroes (2000,2000);
$slice = $x(10:1990,10:1990);
$slice++;
```

If you monitor the memory consumed by the PDL process on your computer (UNIX/Linux users can try the `top` command) you will see that the amount of memory consumed does not go up - **even when the slice is modified** . This is because the way PDL is written allows many of the simple operations on slices to be optimised - i.e. a temporary internal copy of the slice need not be made. Of course sometimes - for example when passing to an external subroutine - this optimisation is not possible. But the book-keeping of propagating the changes back to the original piddle is handled automatically.

Advanced slice syntax

`slice` has some advanced syntactical features which allow dimensions to be inserted or removed (this comes in quite useful when passing 2D arrays to functions expecting 1D arguments and visa-versa, this comes in extremely useful when using PDL's advanced **threading** features (see *PDL threading and the signature* later.

If a dimension is of size unity it can be removed using `()` :

```
$z = zeroes 100,30;
$col = $z(42,:); # Column 42 - 2D (Dims = 1x30)
$col = $z((42),:); # Column 42 - 1D (Dims = 30)
```

And then one can put them back again using ` `*` ':

```
$col2 = $col(*,:,*); # Dims now = 1x30x1
```

This can even be used to insert more than one element along the dimension:

```
$t = $z(:,*3,:); # Dims now 100x3x30
```

This sort of thing is very useful for advanced threading trickery.

PDL's Method notation

At this point we would like to introduce the varied notations for calling `slice` and it's friends. This is because it will be commonly seen in PDL code and is very handy. While at first unfamilar to C and FORTRAN users it is not rocket science, PDL users will quickly become used to it.

As we mentioned in Chapter 2 piddles are implemented as Perl objects. Objects can have their own personal functions, known as methods. The difference between a method and a function is that a method can only be used on the class of object it belongs too. And methods have a new notation for calling them. This means names (which can get in short suppply) can be re-used for different objects.

Many of PDL's functions are available as methods too, in fact once you started using the more advanced features you will find that many of them are only available as methods. (PDL by default defines a lot of functions, which while useful do clutter Perl's namespace, at some point we had to stop!).

For example here are 3 different ways of calling `slice` :

```
$t = slice($z,":,*3,:"); # Function call (old style)
$t = slice $z,":,*3,:"; # Function call (old style)
$t = $z->slice(":,*3,:"); # Method call (old style)
$t = $z(:,*3,:); # Vector syntax (NiceSlice style)
$t = $z->(:,*3,:); # Method call (NiceSlice style)
```

The `PDL::NiceSlice` style vector syntax is the most concise and readable. The method call syntax (either old style or `PDL::NiceSlice` style) is also readable. You may need to understand the original slicing syntax to understand legacy PDL codes and for the cases where `PDL::NiceSlice` syntax can not or is not used. See the on-line docs for `PDL::NiceSlice` for details.

The dice and dice_axis functions - irregular subsets along axes

As well as take regular slices along axes via the `slice` function, another common requirement is to take **irregular slices** , by which we mean a list of arbitrary coordinates. This operation is referred to in PDL as **dicing** a piddle.

The `dice_axis` function performs a dice along a specified axis:

```
$a = sequence(10,20);
$b = dice_axis $a, 0, [3,7,9]; # Dice along axis 0
$b .= 42; # Alters columns [3,7,9];
print $b;
```

For a 2D piddle dicing along axis 0 selects columns, dicing along axis 1 selects rows. In general in N-dimensions dicing along a given axis reduces the number of elements along that axis, but the number of dimensions remains unchanged. The `dice` function allows all axes to be specified at once:

```
$z = zeroes 10,20,50;
print dims dice $z,[2,3,5],[10,11,12],[30..35,39,40];
```

The list of axes in the dice can be specified using Perl's ` `[]` ' list reference notation or using a 1D piddle:

```
$z = sequence 10,20;
$dice = long(random(10)*10); # Select random columns
$sel = $z->dice_axis(0,$dice);
```

Using mv, xchg and reorder - transposing dimensions

We saw earlier how arguments to `slice` can be used to add and remove dimensions. More sophisticated tricks can be performed with a whole suite of PDL methods.

`xchg` simply swaps two dimensions:

```
$z = zeroes(3,4);
$t = $z->xchg(0,1); # Axes 0 and 1 swapped, dims now = 4,3
```

This is a simple matrix transpose. The method `$z->transpose` and the equivalent operator ~`$z` also do this, though they also make a copy (i.e. return a new piddle) not a slice and can operate on 1D piddles (i.e. convert a row vector into a column vector). Somethimes this is what you want. `xchg` works like `slice` and `dice` - changes affect the original. Also `xchg` generalises to N-dimensions:

```
$z = zeroes(3,4,5,6,7);
$t = $z->xchg(1,3); # Dims now 3,6,5,4,7
```

A different way of switching dimensions around is provided by `$z-`mv(A,B) > which justs moves the axis `A` to posiition `B` :

```
$z = zeroes(3,4,5,6,7);
$t = $z->mv(1,3); # Dims now 3,5,6,4,7
```

Finally one can completely re-order dimensions:

```
$z = zeroes(3,4,5,6,7);
$t = $z->reorder(4,3,0,2,1); # Dims now 7,6,3,5,4
```

Note `reorder` is our first example of a pure PDL method - it does not exist as a function and can only be called using the `$z->reorder(...)` syntax.

Combining dimensions with clump

We've now seen a whole slew of functions for changing the ordering of dimensions. It is now time to look at some more complicated operations. The first of these is something we have already seen in Chapter 1. This is the `clump` function for combining dimensions together. Suppose we have a 3-D datacube piddle:

```
perldl> $a = xvals(5,3,2);
perldl> print $a;


[
 [
  [0 1 2 3 4]
  [0 1 2 3 4]
  [0 1 2 3 4]
 ]
 [
  [0 1 2 3 4]
  [0 1 2 3 4]
  [0 1 2 3 4]
 ]
]
```

We have seen before we can apply a 1-D function like `sumover` to the rows - and using dimension manipulating functions to any of the axes.

But say we wanted to sum over the first **TWO** dimensions? i.e. replace our datacube with a 1-D vector containing the sums of each plane. What we need to do is to `clump' the first two dimensions together to make one dimension, and then use `sumover`. Surprisingly enough this is what `clump` does:

```
perldl> $b = $a->clump(2); # Clump first two dimensions together
perldl> print $b;
[
 [0 1 2 3 4 0 1 2 3 4 0 1 2 3 4]
 [0 1 2 3 4 0 1 2 3 4 0 1 2 3 4]
]
perldl> $c = sumover $b;
perldl> print $c;
[30 30]
```

Now we know about `mv` it is also easy to sum over the last two dimensions:

```
    perldl> print sumover $a->mv(0,2)->clump(2)
    [0 6 12 18 24]
    It is also possible using the special form C< clump(-1) > to clump B<
all > the dimensions together:
```

```
perldl> $x = sequence(10,20,30,40);
perldl> print dims $x->clump(-1);
240000
perldl> print sumover $x->clump(-1); # Same as sum($x)
28799880000
```

Uncannily this is almost exactly how the `sum` function is implemented in PDL.

Adding dimensions with dummy

After our first look at threading in Chapter 2 we know how to add a vector to rows of an image:

```
perldl> print $a = pdl([1,0,0],[1,1,0],[1,1,1]);
[
 [1 0 0]
 [1 1 0]
 [1 1 1]
]
perldl> print $b = pdl(1,2,3);
[1 2 3]
perldl> print $a+$b;
[
 [2 2 3]
 [2 3 3]
 [2 3 4]
]
```

But say we wanted to add the vector to the columns. You might think to transpose $a :

```
perldl> print $a->xchg(0,1)+$b;
[
 [2 3 4]
 [1 3 4]
 [1 2 4]
]
```

But the result is the transpose of the desired result. We could of course just transpose the result but a cleaner method is to use `dummy` to change the dimensions of $b :

```
perldl> print $b->dummy(0); # Result has dims 1x3
[
 [1]
 [2]
 [3]
]
```

`dummy` just inserts a `dummy dimension' of size unity at the specified place. `dummy(0)` put's it at position 0 - i.e. the first dimension. The result is a column vector. Then we easily get what we want:

```
perldl> print $a + $b->dummy(0);
[
 [2 1 1]
 [3 3 2]
 [4 4 4]
]
```

Because of the threading rules the unit dimension makes `$b` implicitly repeat along axis 0. i.e. it is as if `$b->dummy(0)` **looked** like:

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

`dummy` can also be used to insert a dimension of size >1 with the data **explicitly** repeating:

```
perldl> print dims $b->dummy(0,10);
10 3
perldl> print $b->dummy(0,10);
[
 [1 1 1 1 1 1 1 1 1 1]
 [2 2 2 2 2 2 2 2 2 2]
 [3 3 3 3 3 3 3 3 3 3]
]
```

Completely general subsets of data with index , which and where

Our look at advanced slicing concludes with a look at completely general subsets, specified using arbitrary conditions.

Let's make a piddle of real numbers from 0 to 1:

```
perldl> print $a = sequence(10)/10;
[0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
```

We can make a conditional array whose values are 1 where a condition is true using standard PDL operators. For example for numbers below 0.2 and above 0.9:

```
perldl> print $a<0.25 | $a>0.85;
[1 1 1 0 0 0 0 0 0 1]
```

We'll use this as an example of an arbitrary condtion. Using `which` we can return a piddle containing the positions of the elements which match the condition:

```
perldl> $idx = which($a<0.25 | $a>0.85); print $idx;
[0 1 2 9]
```

i.e. elements 0..2 and 9 in the original piddle are the ones we want. We can select these using the `index` function:

```
perldl> print $a->index($idx);
[0 0.1 0.2 0.9]
```

So here we have an arbitrary, non-contiguous slice. However thanks to the magic of PDL we can still modify this as if it was still a more boring kind of slice and have our results affect the original:

```
perldl> $a->index($idx) .= 0; # Set indexed values to zero
perldl> print $a;
[0 0 0 0.3 0.4 0.5 0.6 0.7 0.8 0]
```

In fact PDL posesses a convenience function called `where` which actually lets you combine these steps at once:

```
$a = sequence(10)/10;
$a->where($a<0.25 | $a>0.85) .= 0;
print $a; # Same result as above
```

i.e. we make a subset of values **where** a certain condition is true. You can of course use index with explicit values:

```
# Increment first and last values


$a = sequence(10);
$a->index(pdl(0,9))++;
```

What if you had a 2-D array? `index` is obviously one-dimensional. What happens is an implicit `clump(-1)` (i.e. the whole array is viewed as 1-D):

```
perldl> $a = sequence(10,2);
perldl> $a->index(pdl(0,9)) .= 999;
perldl> print $a;
[
 [999 1 2 3 4 5 6 7 8 9]
 [ 10 11 12 13 14 15 16 17 18 999]
]
```

You can of course use `where` too for any number of dimensions:

```
# e.g.  make a cube with a sphere of 1's in the middle:
$cube = rvals(100,100,100);
$tmp = $cube->where($cube<20);
$cube .= 0;
$tmp .= 1;
```

PDL threading and signatures

Slicing and indexing arbritary subsets of data is certainly a fundamental aspect of any array processing language and PDL is no exeption (as you can tell from the preceding examples). In PDL those functions might be even more important since they are absolutely vital in using PDL **threading** , the fast (and automagic) vectorised iteration of "elementary operations" over arbitrary slices of multidimensional data. First we have to explain what we mean by **threading** in the context of PDL, especially since the term **threading** already has a distinct meaning in computer science that only partly agrees with its usage within PDL. In the following we will explain the general use of PDL threading and highlight the close interplay with those slicing and dicing routines that you have just become familiar with (`slice` , `xchg` , `mv` , etc). But first things first: what is PDL threading?

Threading

Threading already has been working under the hood in many examples we have encountered in previous sections. It allows for fast processing of large amounts of data in a scripting language (such as `perl`). And just to be sure, PDL **Threading** ist **not** quite the same as threading in the computer science sense. Both concepts are related but more about that later.

A simple example

As a starting point, we look at one of the PDL projection operators (they make N-1 dimensional piddles from N dim input piddles). So we need some data to try our code on. This time, we use the image of a tiny fluorescent bead that was recorded with a fluorescent microscope:

```
use PDL::IO::Pic;
$im = rpic 'beads.jpg'; # image stored in the JPEG format
```

The following code snippet calculates the maxima of all rows of this image `$im` :

```
$max = $im->maximum;
```

We rewrite this example slightly so that we can see the dimensions of the piddles involved using a little helper routine (see box) to print out the shape of piddles in the course of computations:

($max = $im->pdim('Input')->maximum)->pdim('Result');

that generates the following output:

```
Input Byte D [256,256]

Result Byte D [256]

\begin_inset Float figure
placement H
wide false
sideways false
status open

\begin_inset Box Boxed
position "c"
hor_pos "c"
has_inner_box 1
inner_pos "t"
use_parbox 0
width "110text%"
special "none"
height "55theight%"
height_special "none"
status open
```

Since is important to keep track of the dimensions of piddles when using (and especially when introducing) **PDL threading** we quickly define a shorthand command (a method) that lets us print the dimensions of piddles nicely formatted as needed:

```
{ package PDL;
sub pdim { # pretty print type+dimensions and
              # allow for optional string arg
   my ($this) = @_;
   print (($#_ > 0 ?
$_[1]\t
:
\begin_inset Quotes eld

\begin_inset Quotes erd

).

       $this->info("%T %D
\backslash
n")); # use info to print type and dims

   return $this;

}}
```

Two observations: note how we temporarily switched into the package PDL so that pdim can be used as a method on piddles and we made the function return the piddle argument so that it can be seamlessly integrated into method invocation chains:

```
$a->pdim("Dims")->maximum;
```

A small utility routine

So let's dissect what has happened. If you look at the documentation of `maximum` it says This function reduces the dimensionality of a piddle by one by taking the maximum along the 1st dimension.

In this respect `maximum` behaves quite differently from `max` . `max` will always return a scalar with a value equal to that of the largest element of a (possibly multidimensional) piddle. `maximum` , however, is by definition an operation that takes the maximum only over a one-dimensional vector. If the input piddle is of higher dimension this **elementary operation** is automatically iterated over all one-dimensional subslices of the input piddle

And, most imporatantly, this automatic iteration (we call it the **threadloop**) is implemented as fast optimized C loops. As a convention, these subslices are by default taken along the first dimensions of the input piddle. In our current example the subslices are one-dimensional and therefore taken along the first dimension. All results are placed in an appropriately sized output piddle of N-1 dimension s, one value for each subslice on which the operation was performed.

Now it should be no surprise that

```
$im3d = sequence short, 5,10,3; # a 3D image (volume)
$max = $im3d->pdim('Input')->maximum;
print $max->pdim('Result') .  " \n"; generates


Input   Short D [5,10,3]
Result  Short D [10,3]
[
 [  4   9  14  19  24  29  34  39  44  49]
 [ 54  59  64  69  74  79  84  89  94  99]
 [104 109 114 119 124 129 134 139 144 149]
]
```

As expected the above command sequence creates a 2D piddle (size `[10,3]`) of maxima of all rows of the original volume data.

Why bother?

Why should we go through this at length? Quickly you will realize that many more complicated operations can be assembled from the iteration of an elementary operation (that is if you keep reading this chapter). Those elementary operations that ship with the basic PDL distribution make the building blocks for your more complicated real world applications; threading just makes sure it will all happen quickly enough and without too much syntactical effort from your side (you still will have to get your head round the idea). So let's expand our example a little further and postpone the why and how for a small while.

More examples

Now suppose we do not want to calculate the maxima along the first dimension but rather along the second (the column maxima). However, we just heard that `maximum` works by definition on the first dimension. How can we make it do the same thing on the second dimension? Here is where the dimension manipulation routines come in handy: we use a to make a piddle in which the original second dimension has been moved to first place. Guess how that is done: yes, using `xchg` we get what we want

If you check `pdim`'s output you see how the originally second dimension of size 10 has been moved to the first dimension (step 1->2) and, accordingly, maximum now does its work on all the columns of the original input piddle `$im3d` (step 3).

Again PDL has automatically iterated the elementary functionality of maximum (calculate the maximum of a one-dimensional vector) over all subslices of the data and created an appropriately sized piddle (here of shape `[5,3]`) to hold the resulting elements.

This general scheme works for most PDL functions. For example, let's say you have a stack of images (represented by a 3D piddle) and you want to convolve each image with the same kernel. That's easy. Make sure the image dimensions (x and y) are the leading dimension in your piddle:

```
$convolved = $stack->conv2d($kernel);
```

And if your image stack is organized differently, e.g. the leading dimension is the z dimension, say in a [8,256,256] shaped piddle just use `mv` to obtain the desired result:

```
$convolved = $stack->mv(0,2)->conv2d($kernel);
```

These (admittedly simple) examples show the general principle: an elementary operation is iterated over subslices of one or several multidimensional piddles. Sometimes the dimensions of the input piddles involved need to be manipulated so that iteration happens as desired (e.g. over the intended subslices) and the result has the intended dimensionality. Formulating your problem at hand in a way that makes use of threading rather than resorting to nested `for` -loops at the perl level can make the difference between a script that is executed faster than you can type and one that is crawling along and giving you plenty of time to have your long overdue lunch break.

Why threading and why call it threading ?

So what are the advantages of relying on threading to perform things you can achieve in perl also with explicit `for` loops and the `slice` command? There are several (very good) reasons. The more you use PDL for your daily work the quicker you will appreciate this.

Before we get into the details of the why and how let's admit: PDL is by no means the first data language that supports this type of automatical implicit looping facility: the authors have in fact been inspired by several previous data language implementations, most notably `Yorick`

Similar concepts are also implemented in APL and J, although well hidden by a wealth of terminology and notation very different from that of most other conventional computer languages . What we think distinguishes PDL from these previous languages is the consistent support of threading throughout PDL, the tight integration with the PDL preprocessor (dealt with in a separate chapter) and the conceptual interplay with the dimension manipulation routines.

The first and most important reason to use **PDL threading** is simply **speed** . The alternative to threading are loops at the perl level. That is certainly a viable alternative, however, if we rewrite our maximum routine along these lines a quick benchmark test will prove our point. First of all, here is the code that does the equivalent of `maximum` on 2D input without using threading

```
sub mymax {
   # we only cover the case of 2D input
   my ($pdl) = @_;
   die "can only deal with 2D input" unless $pdl->getndims == 2;
   $result = PDL->zeroes($pdl->type,$pdl->getdim(1));
   my $tmp;
   for (my $i=0;$i<$pdl->getdim(1);$i++) {
      ($tmp = $result->slice("($i)")) .= $pdl->slice(",($i)")->max;
   }
   return $result;
}
```

We have written it so that `mymax` can just deal with 2D input piddles. A routine for the general n-dimensional case would have been more involved . Note that we explicitly have to create an output piddle of the desired type and size. By comparison, the corresponding threading routine is much more concise:

```
    sub mythreadmax {
       my ($pdl) = @_;
       return $pdl->maximum;
    }
```

In fact, we only wrapped `maximum` in another subroutine to have the same calling overhead as `mymax` . We are trying to be fair (even though we are biased). So let's compare the performance of `mymax` versus `mythreadmax` . How? Remember that we are using perl, after all, and that there is (almost) always a module that does just what you need. Here and now `Benchmark`

Our benchmarking script looks like this

```
    use Benchmark;
    use PDL;
    $a = sequence(10,300);
    timethese(0, { # run each for at least 3 CPU secs
                'Perl loops' => '$pl = mymax $a;',
                'PDL thread' => '$pt = mythreadmax $a;',
    });
```

If we run this script it generates

```
    PDL thread:  4 wallclock secs ( 2.48 usr +  0.64 sys =  3.12 CPU) @
12802.88/s (n=40009)
    Perl loops:  3 wallclock secs ( 1.80 usr +  1.23 sys =  3.03 CPU) @
12.86/s (n=39)
```

That proves our point: while the example using threading is executed at a rate of nearly 13,000 per second using explicit loops has brought down the speed to less than 13/s, a very significant difference. Obviously, the difference between threading and explicit looping depends somewhat on the nature of the elementary operation and the piddles in question. The difference becomes most striking the more elementary operations are involved and the faster an individual elementary operation can be performed. The advantage of threading will level off as the time for performing the elementary operation becomes comparable or even greater than that required to execute the explicit looping code.

Another distinct advantage becomes apparent when comparing the code required to implement the equivalent of the `maximum` functionality explicitly in perl code. We have to write extra code to create the right size ouput piddle, explicitly handle dimension sizes, etc. All in all the code is much less concise and also less general.

With the requirement to deal with all dimensions, loop limits, etc yourself you increase the probability of introducing errors into your code. When using threading, PDL checks all dimensions for you, makes sure it loops over the correct indices internally and keeps you from having to do the bookkeeping: after all, **that** is what computers are good at.

Even though PDL threading makes your life much easier in one respect by taking care of some of the "messy" details it leaves you with another task: you have to find the places in your algorithm/problem where threading can effectively be used and help to make for speedy execution even when using an (almost inevitably slower) scripting language. But finding such places and making use of these vectorised features is the key to using an array-oriented high level language like PDL successfully. This is what the programmer new to PDL and used to low-level programming has to learn: avoid explicit loops where possible and try to use automatically performed **thread loops** instead.

There is yet another benefit that comes with the threading approach. By looking at places where threading can be efficiently used you are also rethinking your problem in a way so that it can be very effectively parallelize d! The keen reader has probably already observed that those internal automatic loops of elementary operations over subslices do not have to be performed sequentially.

```
use Benchmark;
use PDL;
$a = sequence(10,300);
timethese(0, { # run each for at least 3 CPU secs
            'PDL mulithreaded' => '$pl = maximum $a;',
            'PDL singlethreaded' => '$a->add_threading_magic(1,10);
                                           $pt = maximum $a;',
   });
```

The command

$a->add_threading_magic(1,10);

explicitly tells PDL to use 10 (possibly concurrently run depending on your computer hardware) threads when performing the threadloop over dimension 1. For example, on an Cygwin PC with 2 AMD Opteron CPUs this script yields the following output:

The general case: PDL functions and their signature

Having made the case for PDL threading let's study its own messy details. PDL threading is a powerful tool. And as usual you have to pay a price for power: complexity. The general rules for PDL threading can be confusing at first. But there is hope: you can first study the more simple cases and work up to more difficult examples as you go. So let's continue our tour of threading.

The first question arises naturally: how can one find out about the dimension of subslices in a elementary operation of a function in PDL? We know from the preceding examples that some PDL functions work on a one-dimensional subvector of the data and generate a zero-dimensional result (a scalar) from each of the processed subslices, for example: `maximum` , `minimum` , `sumover` , `prodover` , etc. Two-dimensional convolution (`conv2d`), on the other hand, consumes a 2D subslice in an elementary operation. But how do we get this information in general for any given function? It is easy: you just have to check the function's **signature** !

The signature is a string that contains this information in concise symbolic form: it names the parameters of a function and the dimensions of these parameters in an elementary operation. Additionally, it specifies which of these parameters are input parameters and which are output parameters. Finally, for some functions it contains information about special type conversions that are to be performed at run-time.

Generally, you can find the signature of a function using the perldl online help system. Just type `sig <funcname` > at the command prompt, e.g.:

```
perldl> sig maximum

   Signature: maximum(a(n); [o]c())
```

The interesting part is the formal argument list in parentheses that follows the function name:

```
a(n); [o]c()
```

This signature states that `maximum` is a function with two arguments named `a` and `c` . Wait a minute: above it seemed that `maximum` only takes one argument and returns a result! The apparent contradiction is resolved by noting that the formal argument `c` is flagged with the `[o]` option identifying `c` is an output argument. This seems to suggest that we could `maximum` also call as maximum($im, $result);

This is in fact possible and an intended feature of PDL that is useful in **tight loops** where it helps to avoid unneccesary reallocation of variables (see below). In general, however, we will call functions in the usual way that can be written symbolically as:

```
    output_arg_list = function(input_arg_list)
```

or equivalently, using the method notation:

```
    output_arg_list = input_piddle_1->function(rest_of_arg_list)
```

The other important information supplied by the signature is the dimensionality of each of these arguments in an elementary operation. Each formal parameter carries this information in a list of formal dimension identifiers enclosed in parentheses. So indeed `a(n)` marks `a` as a one-dimensional argument. Additionally, each dimension has a **named** size in a signature, in this example `n . c()` has an empty list of dimension sizes: it is declared to be zero-dimensional (a scalar).

If piddles that are supplied as runtime arguments to a function have more dimensions than specified for their respective formal arguments in the signature then these dimensions are treated by PDL as **extra dimensions** and lead to the operation being **threaded** over the appropriate subslices, just what we have seen in the simple examples above.

As mentioned before a higher dimensional piddle can be viewed as an array (again **not** in the perl array sense) of lower dimensional subslices. Anybody who has ever worked with matrix algebra will be familiar with the concept. For some of the following examples it will be useful to illustrate this concept in somewhat more detail. Let's make a piddle first, a simple 3D piddle:

```
    $pdl = sequence(3,4,5);
```

A boring piddle, you say? Yes, boring, but simple enough to clearly see what is going on in the following. First we look at it as a 3D array of 0D subslices. Since we know the syntax of the `slice` method already we can write down all 0D subslices, no problem:

```
    $pdl->slice("($i),($j),($k)");
```

Well, obviously we have not written down all 3*4*5 = 60 subslices literally but rather in a more concise way. It is understood that `$i` can have any value between 0 and 2, `$j` between 0 and 3 and `$k` between 0 and 4. To emphasize this we sometimes write

```
    $pdl->slice("($i),($j),($k)") $i=0..2; $j=0..3; $k=0..4
```

With the meaning as above (and '..' `not` meaning the perl list operator). In that way we enumerate all the sublices. Quite analogously, when dealing with an elementary operation that consumes 1D slices we want to view `$pdl` as an `[` 4,5 `]` array of 1D subslices:

```
    $pdl->slice(":,($i),($j)") $i=0..3; $j=0..4
```

And similarly, as a `[` 5 `]` array of 2D subslices

```
    $pdl->slice(":,:,($i)") $i=0..4
```

You see how we just insert a ':' for each complete dimension we include in the subslice. In fig. XXX the situation is illustrated graphically for a 2D piddle. Depending on the dimensions involved in an elementary operation we therefore often group the dimensions (what we call the **shape**) of a piddle in a form that suggests the interpretation as an array of subslices. For example, given our 3D piddle above that has a shape `[` 3,4,5 `]` we have the following different interpretations:

```
    ()[3,4,5] a 3,4,5-array of 0D slices
    (3)[4,5] a 4,5-array of 1D slices (of shape [3])
```

```
(3,4)[5] a 5-array of 2D slices (of shape [3,4])
(3,4,5)[] a 0D array of 3D slices (of shape [3,4,5])
```

The dimensions in parentheses suggest that these are used in the elementary operation (mimicking the signature syntax); in the context of threading we call these the **elementary dimensions** . The following group of dimensions in rectangular brackets are the **extra dimensions** . Conversely, given the elementary/extra dims notation we can easily obtain the shape of the underlying piddle by appending the extradims to the elementary dims. For example, a [3,6,1] array of 2D subslices (3,4)

```
(3,4)[3,6,1]
```

identifies our piddle's shape as [3,4,3,6,1]

Alright, the principles are simple. But nothing is better than a few examples. Again a typical imaging processing task is our starting point. We want to convert a colour image to greyscale. The input image is represented as a two-dimensional array of triples of RGB colour coordinates, or in other words, a piddle of shape `[3,n,m]` . Without delving too deeply into the details of digital colour representation it suffices to note that commonly a grey value **i** corresponding to a colour represented by a triple of red, green and blue intensities **(r,g,b)** is obtained as a weighted sum:

\begin_inset Formula \[i=\frac{77}{256}r+\frac{150}{256}g+\frac{29}{256}b\]

A straight forward way to compute this weighted sum in PDL uses the `inner` function. This function implements the well-known **inner product** between two vectors. In a elementary operation `inner` computes the sum of the element-by-element product of two one-dimensional subslices (vectors) of equal length:

\begin_inset Formula \[c=\sum_{i=0}^{n-1}a_{i}b_{i}\]

Now you should already be able to guess `inner` 's signature:

```
perldl> sig inner

  Signature: inner(a(n); b(n); [o]c())
```

`a(n); b(n); [o]c();` : two one-dimensional input parameters `a(n)` and `b(n)` and a scalar output parameter `c()` . Since `a` and `b` both have the same named dimension size `n` the corresponding dimension sizes of the actual arguments will have to match at runtime (which will be checked by PDL!). We demonstrate the computation starting with a colour triple that produces a sort of yellow/orange on an RGB display:

```
$yel = byte [255, 214, 0]; # a yellowish pixel
$conv = float([77,150,29])/256; # conversion factor
$i = inner($yel,$conv)->byte; # compute and convert to byte
print "$i \backslash n";
202
```

Now threading makes extending this example to a whole RGB image very straightforward:

```
use PDL::IO::Pic; # IO for popular image formats
$im = rpic 'pdllogo.jpg'; # a colour image from the book dataset
$grey = inner($im->pdim('COLOR'),$conv);
   # threaded inner product over all pixels
$gb = $grey->byte; # back to byte type
COLOR Byte D [3,500,300]
```

The code needs no modification! Let us analyze what is going on. We know that `$conv` has just the required number of dimensions (namely one of size 3). So this argument doesn't require PDL to perform threading. However, the first argument `$im` has two **extra dimensions** (shape `[ 500,300 ]`). In this case threading works (as you would probably expect) by iterating the inner product over the combination of all 1D subslices of `$im` with the one and only subslice of `$conv` creating a resulting piddle (the greyscale image) that is made up of all results of these elementary operations: a 500x300 array of scalars, or in other words, a 2D piddle of shape `[ 500,300 ]`.

We can more concisely express what we have said in words above in our new way to split piddle arguments in elementary dimensions and extra dimensions. At the top we write `inner`'s signature and at the bottom the `slice` expressions that show the subslices involved in each elementary operation:

```
Piddles $im $conv $grey
Signature a(n); b(n); [o]c()
Dims (3)[500,300] (3)[] ()[500,300]
Slices ":,($i),($j)" ":" "($i)($j)"
```

Remember that the slice notation at the bottom does not mean that you have to generate all these slices yourself. It rather tells you which subslices are used in a elementary operation. It is a way to keep track what is going on behind the scenes when PDL threading is at work. Threading makes it possible that we can call the greyscale conversion with piddles representing just one RGB pixel (shape `[ 3 ]`), a line of RGB pixels (shape `[ 3,n ]`), RGB images (shape `[ 3,m,n ]`), volumes of RGB data (shape `[ 3,m,n,o ]`), etc. All we have to do is wrap the code above into a small subroutine that also does some type conversion to complete it:

```
sub rgbtogr {
   my ($im) = @_;
   my $conv = float([77,150,29])/256; # conversion factor
   my $grey = inner $im, $conv;
   return $grey->convert($im->type); # convert to original type
}
```

You can write your own threading routines

Did you notice? By writing this little routine we have created a new function with its own signature that will thread as appropriate. It has **inherited** the ability to thread from `inner`. So what is the signature of `rgbtogr`? It is nowhere written explicitly and we can't use the `sig` function to find out about it

`sig` will only know about functions that were created using `PDL::PP` or if we explicitly specified the signature in the PDL documentation but from the properties of `inner` and the definition of `rgbtogr` we can work it out. As input it takes piddles with a size of the first dimension of 3 and returns for each of the 1D subslices a 0D result (the greyvalue). In other words, the signature is

```
a(tri = 3); [o] b()
```

There is some new syntax in this signature that we haven't seen before: writing `tri = 3` signifies that in a elementary operation `rgbtogr` will work on 1D subslices (we have encountered this before); additionally, the size of the first dimension (named suggestively `tri`) **must** be three. You get the idea.

What we have just seen is worth keeping in mind! By using PDL functions in our own subroutines we can make new functions with the ability to thread over subslices. Obviously, this is useful. We will come back to this feature when we talk about other ways of defining threading functions using PDL::PP below.

Matching threading dimensions

After this small digression, back to the subject at hand: what happens when both piddle arguments have extra dimensions? Well, the extra dimensions have to match. Otherwise we wouldn't know how to sensibly pair the subslices, right? So when do extra dimensions match? It is quite simple:

corresponding extra dimensions have to have the same size in both piddle arguments. Corresponding extra dimensions are those that occur in both piddles. However, one piddle can have more extra dimensions than the other without causing a mismatch. That sounds strange? Ok, here is an example. We use one of the fundamental arithmetic operations in PDL, addition implemente d by the ' + ' operator. You know already that in an array-oriented language like PDL addition is performed element-by-element on scalars. So the signature of ' + ' comes as no surprise

```
    a(); b(); [o] c()
```

two scalars are summed to yield a scalar result. And when we use higher dimensional piddles in an addition this elementary operation is performed over all 0D subslices, as before. So let's go through a few cases. First make some simple piddles

```
    $a = pdl [1,2,3];
    $b = pdl [1,1,1];
    $c = ones 3,2;
    $d = pdl [3,4];
    print $a + $b, "\n";
```

No big deal. Extradims for both piddles have shape `[ 3 ]` obviously matching, resulting in

```
    [2 3 4]
```

Next,

```
    print $a + $c;
    [
     [2 3 4]
     [2 3 4]
    ]
```

Alright, this probably is exactly what you expected but let us go through our new terminology and check that we can formally agree with what we intuitive ly expected anyway.

`$a` 's **extradim** (s) has shape `[ 3 ]` , those of `$c` shape `[ 3 2 ]` . The **corresponding extradim** (s) in this case is just the first one for the piddles involved. It is equal to 3 in both input piddles, so clearly matches.

```
    $a $c
    a(); b(); [o] c()
    ()[3] ()[3,2] ?????
```

Now, here is something we have not explicitly discussed yet: what is the shape of the automatically created output piddle given the shape of the extradims of the input piddles involved? Well, the result is created so that it has as many extradims as that input **piddle** (s) with the most extradims. Additionally, the shape will match that of the input piddles. In our current example that leaves us with a result with extradim shape `[ 3,2 ]` : [o] c() ()[3,2] Remembering that we obtain the shape of the output piddle by appending the shapes of the extradims to that of the elementary dimensions (here a scalar, i.e. 0D) that leaves us with a result piddle of shape `[ 3,2 ]` .

In the next example we want to multiply `$c` with `$d` so that each row of `$c` is multiplied by the corresponding element of `$d` or expressed in slices

```
    $result->slice("($i),($j)") = $c->slice("($i),($j)") *
$d->slice("($j)") $i=0..2, $j=0..1
```

How do we achieve that by threading?

```
$result = $c*$d
```

is not the right way.

Why? Well, the extradims don't match, `[ 3,2 ]` does not match `[ 2 ]` since 2 is not equal to 3. Just to see how PDL checks this let us actually execute the command. The slightly obscure error message is something like this

```
PDL: PDL::Ops::mult(a,b,c): Parameter 'b'
PDL: Mismatched implicit thread dimension 0: should be 3, is 2
Caught at file (eval 344), line 4, pkg main
```

This is PDL's way to tell you that the extra dimensions don't match.

So how do we do it? We use one of the dimension manipulation methods again. This time `dummy` comes in handy. We want to multiply each element in the nth row of `$c` with the nth element of `$d` . So we have to repeat each element of `$d` as many times as there are elements in each row of `$c` . This is exactly what we can achieve by inserting a dummy dimension of size $c-> **getdim** (0) as dimension 0 of `$d` :

```
perldl> print $d->dummy(0,$c->getdim(0))->pdim("New dims");
New dims Double D [3,2]
[
 [3 3 3]
 [4 4 4]
]
```

Using this trick we have a our threaded multiplication do what we want. And now the extra dimensions **match** (!):

```
$result = $c * $d->dummy(0,$c->getdim(0));
print $result;
```

Using our symbolic way of writing down the slices that are paired in a elementar y operation we can see that we achieve what we wanted

```
$c $d->dummy(0,$c->getdim(0)) $result
"($i),(j)" "($i),($j)" "($i),($j)"
```

But hang on, we want to verify (somewhat formally) that the right subslices of the original `$d` are used in each elementary operation. That is easily achieved by noting that the slice "($i),($j)" of the dummied `$d` is equivalent to the subslice "($j)" of the original 1D piddle `$d` . So we finally arrive at

```
$c $d $result
"($i),(j)" "($j)" "($i),($j)"
```

While this kind of analysis seems probably not justified when dealing with such a simple example it comes in very handy when looking at more complex threaded code.

But before we try our understanding on such an example we look once again at the way extra dims have to match in a thread loop. In the previous example, we had to find out about the size of `$c` 's first dimension (using `getdim(0)`) to make a dummy dimension that would fit `$c` 's extradims in the threaded multiplication. Since similar situations occur very often when writing threaded PDL code the matching rules for extra dimensions allow a dimension size of 1 to match any other dimension size: it is the **elastic** dimension size in a sense that it grows in a thread loop as required. As in the thread loop the corresponding extra dimension is marched through all its positions (e.g. `slice(":,($i)") $i=0..n-1`) the elastic dimension just uses its one and only position 0 repeatedly (`slice(":,(0)") $i=0..n-1`). Therefore, an equivalent and more concise way to write the threaded multiplicat ions makes use of this and the fact that a dummy dimension of size 1 is created by default if the second argument is omitted (see `help dummy`)

```
    print $c->pdim('c') * $d->dummy(0)->pdim('dd');
    $A [l,m] $B [n,l] $AB [n,m]
    $AB = inner $A->dummy(1), $B->xchg(0,1)
    $A->dummy(1) $B->xchg(0,1) $AB
    (l)[1,m] (l)[n] ()[n,m]
    ":,(0),($j)" ":,($i)" "($i)($j)"
```

Going back to the original piddles $A and $B we see that the slice expressions change to

```
    $A $B $AB
    ":,($j)" "($i),:" "($i),($j)"
```

and that means

```
    $AB->slice("($i),($j)") = inner $A->slice(":,($j)"),
$B->slice("($i),:") $i=0..n-1, $j=0..m-1
```

and that is exactly the definition of the matrix product as we explained above! Our bit of formalism has sort of "proved" it. You see that the slice and dimension matching formalism we developed can really be helpful when you try to verify that your complicated threading expression does what you want it to do. However, as you get more experience with threading we strongly suspect that you don't need this any more; you will rather develop a much better "feeling" how to write down the right combination of dimension manipulations to achieve the desired result in a thread loop.