PDL — Scientific Programming in Perl

Karl Glazebrook Christian Söller Tuomas J. Lukka
Marc Lehmann Jarle Brinchmann Doug Hunt
John Cerney Robin Williams Tim Pickering

October 16, 2001

Contents

Chapter 0

PDL — Year Zero

> "Why is it that we entertain the belief that for every purpose odd numbers are the most effectual?" - Pliny the Elder.

The PDL project began in February 1996, when I decided to experiment with writing my own 'Data Language'. I am an astronomer. My day job involves a lot of analysis of digital data accumualated on many nights observing on telescopes around the world. Such data might for example be images containing millions of pixels and thousands of images of distant stars and galaxies. Or more abstrusely, many hundreds of digital spectral revealing the secrets of the composition and propertues of these distant objects.

Obviously many astronomers before have dealt with these problems, and a large amount of software has been constructed to facilitate their analysis. However, like many of my colleagues, I was constantly frustrated by the lack of generality and flexibility of these programs and the difficulty of doing anything out of the ordinary quickly and easily. What I wanted had a name: 'Data Language', i.e. a language which allowed the manipulation of large amounts of data with simple arithmetic expressions. In fact some commericial software worked like this, and I was impressed with the capabilities but not with the price tag. And I thought I could do better.

As a fairly computer literate astronomer (read 'nerd' or 'geek' according to your local argot) I was very familiar with 'Perl', a computer language which now seems to fill the shelves of many bookstores around the world. I was impressed by it's power and flexibility, and especially it's ease of use. I had even explored the depths of it's internals and written an interface to allow graphics (the PGPLOT module[1]). The ease with which I could then create charts and graphs, for my papers, was refreshing.

[1]The PGPLOT module for perl is an interface to the pgplot graphics library (written in C and Fortran) created by Tim Pearson of Caltech. More information about this library can be obtained from: http://astro.caltech.edu/~tjp/pgplot/

Version 5 of Perl had just been released, and I was fascinated by the new features available. Especially the support of arbitrary data structures (or 'objects' in modern parlance) and the ability to 'overload' operators — i.e. make mathematical symbols like `+-*/` do whatever you felt like. It seemed to me it ought to be possible to write an extension to Perl where I could play with my data in a general way: for example using the maths operators manipulate whole images at once.

Well one slow night at an observatory I just thought I would try a little experiment. In a bored moment I fired up a text editor and started to create a file called 'PDL.xs' — a Perl extension module to manipulate data vectors. A few hours later I actually had something half decent working, where I could add two images in the Perl language, **fast**! This was something I could not let rest, and it probably cost me one or two scientific papers worth of productivity. A few weeks later the Perl Data Language version 1.0 was born. It was a pretty bare infant: very little was there apart from the basic arithmetic operators. But encouraged I made it available on the Internet to see what people thought.

Well people were fairly critical — among the most vocal were Tuomas Lukka and Christian Soeller. Unfortunately for them they were both Perl enthusiasts too and soon found themselves improving my code to implement all the features they thought PDL ought to have and I had heinously neglected. PDL is a prime example of that modern phenomenon of authoring large free software packages via the Internet. Large numbers of people, most of whom have never met, have made contributions ranging for core functionality to large modules to the smallest of bug patches. PDL version 2.0 is now here (though it should perhaps have been called version 10 to reflect the amount of growth in size and functionality) and the phenomenon continues.

I firmly believe that PDL is a great tool for tackling general problems of data analysis. It is powerful, fast, easy to add too and freely available to anyone. I wish I had had it when I was a graduate student! I hope you too will find it of immense value, I hope it will save you from heaps of time and frustration in solving complex problems. Of course it can't do everything, but it provides the framework, the hammers and the nails for building solutions without having to reinvent wheels or levers.

```
- Karl Glazebrook, Sydney, Australia. 4/March/1999
```

0.1 Who this book is for.

This book is for anybody who has to work on large volumes of data, for mathematical analysis or visualisation. You could be a scientist or an engineer working or research or design problems, a businessman trying to make sense of vast numbers of stock indices, a statistician crunching numbers and trying to figure out trends or a web site manager monitoring page accesses.

You know the benefits of being able to work in a very high-level language such as Perl (or TCL or Python), you can see how much easier it would be if you could take the same approach to your data analysis rather than griping around in low-level C or FORTRAN. You may have had experience with the high-level approach in commercial packages and wish for something similar which is free, public domain and Open Source so you can share your code with colleagues.

We are not assuming that you are a Perl expert, rather we hope you will pick up enough of a smattering of Perl in the early chapters to try out (and hopefully be impressed by!) the tutorials and examples. We hope this will inspire you to go out and learn more about Perl if you don't know it already. (We mention some books below).

If you are a Perl aficionado and you work on these types of problems, then PDL is definitely for you! Read on...

0.2 The case for a high-level approach.

We've all been there. You know how you want to analyse your data. You need to Fourier transform it, take the square root, multiply by a high-pass filter and sum up all the high frequence modes. But it's two in the morning and you are staring at the guts of your C or FORTRAN program trying to figure out why your program keeps crashing with array overflow errors. You know these problems have been solved individually innumerable times in the past, carefully written subroutines are available to do it. Why should it be so difficult?

The reason is though subroutines are available low-level languages still force a lot of complexity on you. You must manage memory yourself, declare variables however trivial, call subroutines with a whole bunch of arguments in case just one of them is needed, etc. And you must be able to pull together seperate subroutine libraries to do file input/output, user interaction, data processing and graphics.

Whereas all you really want to do is tell the computer things like 'read this', 'Fourier transform that', and 'Plot this', and have it be smart enough to do the right thing. What you are wishing for is in effect a high-level language, in this case it is called 'English'.

While natural language understanding is still quite a long way off, high-level computer languages are currently proliferating. Examples include Perl, TCL, JAVAscriptm, Visual Basic, Python, and many more. Such systems have also been developed for data processing. Worthy of note are commericial software such as IDL® ('Image Data Language' from Research Systems Inc.[2]), MATLAB® (from The Mathworks, Inc.[3]) and the public domain program Octave[4]. These implement special-purpose high-level languages where data is

[2] *www.rsinc.com*
[3] *www.mathworks.com*
[4] *www.octave.org*

handled in large chunks, via 'vector operations'.

What does this mean in practice? It means if you say:

$$C = A + B$$

then the operation is performed even if A and B are large arrays containing many millions of numbers. Further you can say something like:

$$D = FFT(C)$$

(to apply a Fast Fourier Transform) and get what you want. No messing about. These data analysis languages also implement nice graphics layers, as well as a large suite of mathematical algorithms.

Having used these systems ourselves the authors of PDL can attest to the superiority of that approach in terms of plain getting things done. We of course believe that PDL is now better than all those systems, for quite a few reasons, and that your life will be easier if you get it and use it.

0.3 The case for a free Data Language.

The free software community has taken off to an extraordinary extent in the few years. This has been most vivid in the success of the Linux, a free UNIX-like Operating System. Sometimes this movement is also described as 'Open Source' rather than 'free,' and the term 'free' is often used to mean freedom of use rather than freedom from price. Athough much of the code is indeed free/public domain money is made out of the sale of packaged distributions, support, books, etc. Nevertheless the software is usually available at minimal cost.

One key point is that the source code is available, so that however the software is obtained one has the ability to take it and in principle be able to change it to do whatever is required with it.

How is this relevant to data languages? The authors of PDL are all scientists. We write, obviously, as scientists but believe our ideas are directly relevant to all users of PDL. The scientific community has for hundreds of years believed in the free exchange of ideas. It has been traditional to publish *full details* about how research is done openly in journals. This is very close in spirit to the ideas behind the free software. These days much of what scientists do involves software, in fact large software packages to facilitate certain kinds of analysis are often the subject of major papers themselves with the software being freely available on the Internet. Such software is commonly written in C or FORTRAN to allow general use.

Why aren't they working at a higher level? As we explained above this would allow faster creation and make the software more portable and more easily

customisable. Well in our view one of the reasons this has not happened is because of the lack of a suitable free high-level data-centric language, with powerful enough facilities.

This is not just a minor point, it is critical. Even if software is not published and is for internal use among a team of researchers, in the modern world the team is often distributed among dozens of individuals across many instititutes and nations. The only way to ensure that all will be able to use software is if it is freely available. All the PDL authors have had direct experience with this problem in the past. We have often been hindered in sharing our code by collaborators having lack of access to software.

Moreover scientific work often involves extensive innovations and modifications to old ways of doing things. For software as well as being freely available it is critical to have access to the source code to permit easy customisation.

Finally there *is* also the issue of cost. Equivalent commercial packages cost several thousand dollars per workstation. We are not anti-commericial, these packages are very powerful and useful. However we certainly think there should be something like PDL that *anybody* can use and develop for free. Science is a worldwide activity and we like to think that anybody with a PC could use PDL to do research and analysis.

In our view PDL — a free, public domain, Open Source, data language — meets a great need. Today it is openly developed by a group of several dozen people collaborating via the Internet. Anybody with time, expertise or dedication can contribute to improving PDL.

0.4 So why Perl?

So we chose Perl as our implementation language. Our basic data language extensions could have been built around quite a few high-level languages so why did we choose Perl? [5]

1. We need a high-level language which looks after messy details for the user. This of course is why we don't want to use C or FORTRAN.

2. The language should be a commonly used and widely available on many platforms and with a good chance that you already use it for something else. Like the reader, the authors get tired of constantly have to learn new languages.

3. For the system to be fast and interactive the language should be able to run in an interpreted mode, i.e. commands typed can be instantly executed without having to mess around with compiling and linking. Most high-level languages offer this.

[5] Of course the real reason we chose Perl was because we were using it already and liked it a lot. These 'reasons' are really 'compelling rationalisations'!

4. The language must be Open Source (i.e. free, in the public domain and with the source code freely available and redistributable) as we wish our data language to be Open Source too. Why? So people can use it without restrictions, share their code, make improvements to the core language as well as extensions.

5. The language must offer a full suite of modern features. Users of PDL don't just need access to numerical and graphics features. They also want quick and convenient access to databases, network connectivity,, the World Wide Web, Object-Oriented and modular programming, graphical user interfaces, multi-process and multi-processor interactions, text handling, the list could go on for several more sentences. In fact none of the data languages mentioned above have all these features, in particular the commercial systems are hampered in their access to these features by their propritary nature and specialist syntax. We think it is easier to add numerical features to a robust language which has all these other features than to do it the other way around.

6. The language must have a clean and well-documented way of incorporating new subroutines, in low-level languages such as C and Fortran, in to the core. First this lets us implement PDL, secondly it allows diverse groups of people to create their own PDL modules and include compiled code with their own specialist subroutines.

7. The language must be very easy to use, with a reasonably familiar syntax to new users. To some extent this item and the previous one are contradictory. For example the Python language, which is admirable for it's sophisticated and clean Object-Oriented model, meets all the above requirements. Indeed their is already a numerical extension — NumPy[6]. However in our view the syntax is a bit too strange for new users. We prefer a language where simple code can still achieve useful results and which grows with the user. We recognise of course that much of this is just a matter of preference. To us NumPy looks really good, if you are into Python this is what you probably want to use.

8. Finally, and perhaps the most importantly, the language must be reasonably wide-spread and well-known, so people will have other reasons to want to use it apart from PDL. This is why we are not interested in specialist systems, even if they are free, such as Octave or RLAB[7] fine though they may be. Implementation in a true-general purpose full-featured language gives access to a wealth of useful features.

Perl, of course, fills all these constraints most admirably. Perhaps the runner-up would be TCL, though the lack of a consistent object-oriented framework is a problem for TCL. Of course we just said Python was *too* object-oriented, this

[6] *www.pfdubois.com/numpy/*
[7] *rlab.sourceforge.net*

is not a contradiction — in our view Perl gets it just right! Perl also has the singular advantage of being widespeadly used and having a huge collection of well organised modules, publically distributed worldwide on the CPAN network of Internet sites. As scientists who already used Perl a lot for day-to-day programming tasks PDL means we can do just about everything in Perl. Such integration is extremely productive.

0.5 What this book is about.

OK enough advocacy, you are still reading to let us get on with the task in hand. So what can the reader expect to get from this book?

This book is intended to be a complete introduction to PDL. We believe the best way to learn something useful is to learn by doing. So we kickstart the book with some examples of real use of PDL to rapidly show the reader what is is all about.

Then we go through the features of PDL systematically, showing how to use us and drawing on example problems from a range of scientific disciplines to give a sense of how real problems can be solved with PDL. We look at PDL graphics capabilties and how they can be used to visualise problems.

The further in you go the more technical the book will become and we will look at the feature set and the internals and show how to use advanced features such as modules, dataflow and Object-Oriented Programming.

Finally the book concludes with a demonstration of the power of PDL: we show how clever use of PDL can achieve amazing results in only a few lines of code. Deconstructing these is used as a tool to show how better use can be made of PDL.

0.6 What this book is NOT about.

This book is not about teaching Perl, although we hope that even if you don't know anything you will learn pidgin Perl as you read through the first few chapters. Perl is a pretty good 'learn as you go along' language. For a more formal introduction to Perl the following books are recommened:

> 'Learning Perl', *by Randal Schwartz & Tom Phoenix. Published by O'Reilly and Associates, 2001.*

For more advanced usage of Perl we recommend:

> 'Programming Perl', *by Larry Wall, Tom Christiansen & Jon Orwant. Published by O'Reilly and Associates, 2000.*

This book is not about building Graphical User Interfaces using widgets, though Perl is quite adept at this and we will show at least one example of combining PDL and a GUI done in perl/gTk. For widgets perhaps a book to try is:

> 'Learning Perl/Tk', *by Nancy Walsh. Published by O'Reilly and Associates, 1999.*

This book is not about algorithms for analysing data, though PDL is full of them. For a deep mathematical discussion of fitting, Fourier transforming, sorting, inverting and innumerable other useful and hideous things to do to your data the eternal best book is:

> 'Numerical Recipes in C : The Art of Scientific Computing' (and similarly 'Numerical Recipes in FORTRAN : The Art of Scientific Computing'), *by William H. Press, Saul A. Teukolsky, William T. Vetterling, Brian P. Flannery. Published by Cambridge University Press, 1993.*

This book is not about Perl advocacy. You have got some of that here in the introduction! For the rest we will let the language speak for itself.

0.7 Conventions used in this book

This book assumes at least the following versions of software:

1. PDL version 2.2.2
2. perl version 5.6.0

The following typographic conventions are used in this book:

`Fixed width font`
: Code examples.

Italic font
: Is used for filenames. Also URLs and electronic mail addresses.

Chapter 1

A Whirlwind tour through PDL

Or 'PDL for the Impatient'...

> "Maybe there are a few civilizations out there that have decided to stay home, piddle around and send out some radio waves once in a while."
>
> Annette Foglino, Space: Is Anyone Out There? Most astronomers say yes, Life, 1 Jul 1989.

It can be very frustrating to read an introductory book which takes a long time teaching you the very basis of a topic, in a 'Janet and John' style. While you wish to learn, you are anxious to see something a bit more exciting and interesting to see what the language can do.

Fortunately our task in this book on PDL is made very much easier by the high-level of the language. We can take a tour through PDL, looking at the advanced features it offers without getting involved in complexity.

The aim of this chapter is to cover a breadth of PDL features rather than any in depth, to give the reader a flavour of what he or she can do using the language and a useful reference for getting started doing real work. Later chapters will focus on looking at the features introduced here, in more depth.

1.1 Alright let's *do* something.

We'll assume PDL is correctly installed and set up on your computer system (see Appendix A for details of obtaining and installing PDL).

For interactive use PDL comes with a program called `perldl`. This allows you to type raw PDL (and perl) commands and see the result right away. It also allows command line recall and editting (via the arrow keys) on most systems. So we begin by running the `perldl` program from the system command line. On a Mac/UNIX/Linux system we would simply type `perldl` in a 'terminal window'. On a Windows system we would type `perldl` in a command prompt window. If PDL is installed correctly this is all that is required to bring up `perldl`.

```
 myhost% perldl
perlDL shell v1.30
 PDL comes with ABSOLUTELY NO WARRANTY. For details, see the file
 'COPYING' in the PDL distribution. This is free software and you
 are welcome to redistribute it under certain conditions, see
 the same file for details.
ReadLines enabled
Reading /home/kgb/.perldlrc...
Found docs database /usr/lib/perl5/site_perl/5.6.0/i386-linux/PDL/pdldoc.db
Type 'help' for online help
Type 'demo' for online demos
Loaded PDL v2.2.1
Reading local.perldlrc ...
perldl>
```

We get a whole bunch of informational messages about what it is loading for startup and the help system. Note; the startup is *completely* configurable, an advanced user can completely customize which PDL modules are loaded. We are left with the `perldl>` prompt at which we can type commands. This kind of interactive program is called a 'shell'.

Let's create something, and display it:

```
perldl> use PDL::Graphics::PGPLOT
perldl> imag (sin(rvals(200,200)+1))
Displaying 200 x 200 image from -1 to 0.999999940395355 ...
```

The result should look like Fig. 1.1 — a two dimensional `sin` function. `rvals()` is a handy PDL function for creating an image whose pixel values are the radial distance from the central pixel of the image. With these arguments it creates a 200 by 200 'radial' image.[1] `sin()` is the mathematical sine function, this already exists in perl but in the case of PDL is applied to all 40000 pixels at once, a topic we will come back to. The `imag()` function displays the image. You will see the syntax of perl/PDL is algebraic — by which we mean it is very similar to `C` and `FORTRAN` in how expressions are constructed. (In fact much more like `C` than `FORTRAN`). It is interesting to reflect on how much `C` code would be required to

[1] Try '`imag(rvals(200,200))`' and you will see better what we mean!

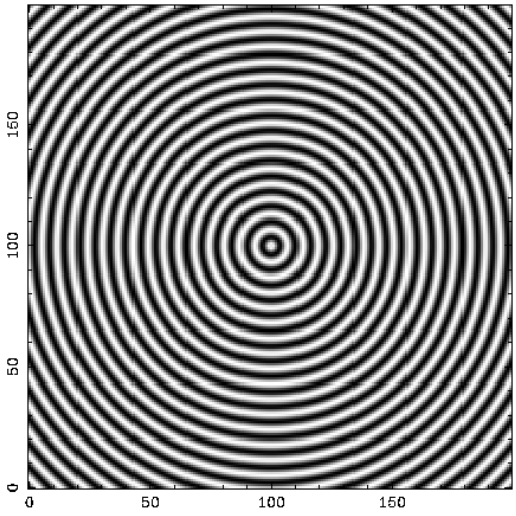


Figure 1.1: A two dimensional `sin` function

generate the same display, even given the existence of some convenient graphics library.

That's all fine. But what if we wanted to achieve the same results in a standalone perl script? Well it is pretty simple:

```
use PDL;
imag (sin(rvals(200,200)+1));
```

That's it. This is a complete perl/PDL program. One could run it by typing `perl filename`.[2]

Two comments:

1. The statements are all terminated by the '`;`' character. Perl is like `C` in this regard. When entering code at the `perldl` command line the final '`;`' may be omitted if you wish, note you can also use it to put multiple statements on one line. In our examples from now on we'll often omit the `perldl>` prompt for clarity.

2. The directive `use PDL` tells Perl to load the `PDL` module, which makes available all the standard PDL extensions. (Advanced users will be inter-

[2]In fact there are many ways of running it, most systems allows it to be setup so you can just type *filename*. See your local Perl documentation — the the perlrun manual page.

ested in knowing there are other ways of starting PDL which allows one to select which bits of it you want).

1.2 Whirling through the Whirlpool

Enough about the mechanics of using PDL, let's look at some real data! To work through these examples exactly you should have access to the *PDL Book example data set* available on the Internet at the locations mentioned in Appendix A.

For much of this chapter we'll be playing with an image of the famous spiral galaxy discovered by Charles Messier, known to astronomrs as M51 and commonly as the Whirlpool Galaxy. This is a 'nearby' galaxy, a mere 25 million light years from Earth. The image file is stored in the 'FITS' format, a common astronomical format, which is one of the many formats standard PDL can read. (FITS stores more shades of grey than GIF or JPEG, PDL can also read these formats).

```
$a = rfits("m51_raw.fits");
```

This looks pretty simple. As you can probably guess by now `rfits` is the PDL function to read a FITS file. This is stored in the perl variable `$a`.

This is an important PDL concept: PDL stores it's data arrays in simple perl variables ($a, `$x`, `$y`, `$MyData`, etc.) PDL data arrays are special arrays which use a more efficient, compact storage than standard perl arrays (`@a, @x, ...`) and are much faster to access for numerical computations. To avoid confusion it is convenient to introduce a special name for them, we call them *piddles* (short for 'PDL variables') to distinguish them from ordinary Perl 'arrays', which are in fact really lists. We'll say more about this later.

Before we start seriously playing around with M51 it is worth noting that we can also say:

```
$a = rfits "m51_raw.fits";
```

Note we have now left off the brackets on the `rfits` function. Perl is rather simpler than C and allows one to omit the brackets on a function all together. It assumes all the items in a list are function arguments and can be pretty convenient. If you are calling more than one function it is however better to use some brackets so the meaning is clear. For the rules on this 'list operator' syntax see the Perl syntax documentation. From now on we'll mostly use the list operator syntax for conciseness

Let's look at M51:

```
imag $a;
```

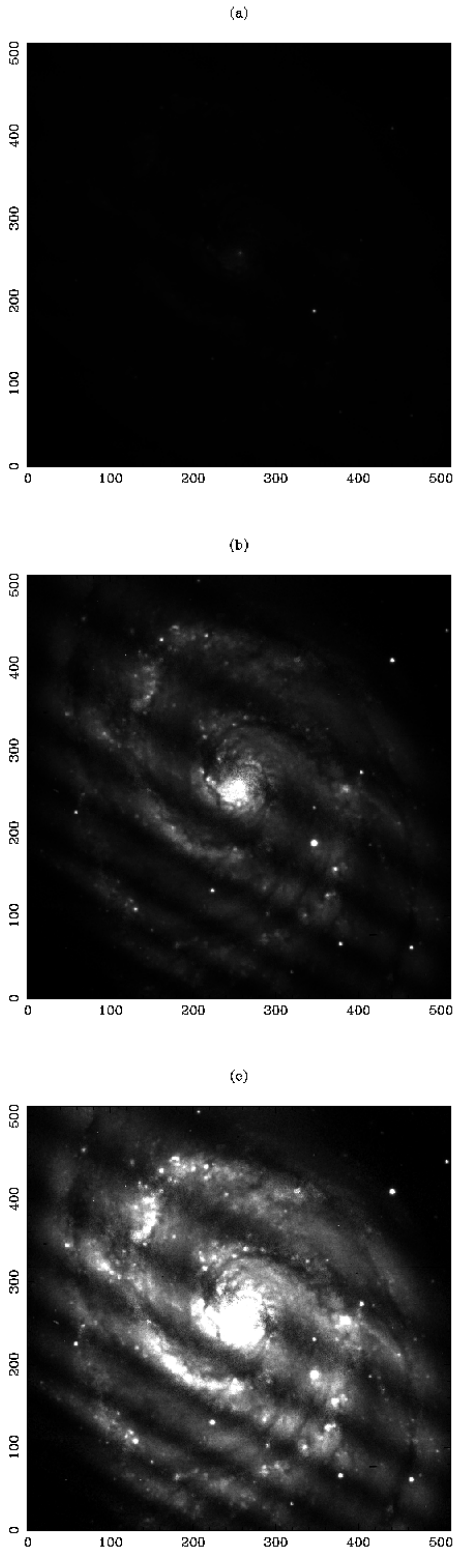


Figure 1.2: The raw image `m51_raw.fits` from the *PDL Book example data set* shown with progressively greater contrast using the `imag` command.

An image like Figure 1.2(a) should be seen. A couple of bright spots but where is the galaxy? It's the faint blob in the middle: by default the display range is autoscaled linearly from the faintest to the brightest pixel, and only the bright star slightly to the bottom right of the center can be seen without contrast enhancement. We can easily change that by specifying the black/white data values (Note: `#` starts a Perl comment and can be ignored — i.e. no need to type the stuff after it!):

```
imag $a,0,1000; # More contrast
imag $a,0,300;  # Even more contrast
```

You can see that `imag` takes additional arguments to specify the display range. In fact `imag` takes quite a few arguments, many of them optional. By typing '`help imag`' at the `perldl` prompt we can find out all about the function.

Anyway the results of this exercise are shown in Figures 1.2(b) and 1.2(c). Well. It is certainly a spiral galaxy with a few foreground stars thrown in for good measure. But what is that horrible stripey pattern running from bottom right to top left? That certainly is not part of the galaxy? Well no. What we have here is the uneven senistivity of the detector used to record the image, a common artifact in digital imaging. (Though it is rarely as bad as this! [3]) We can correct for this using an image of a uniformly illuminated screen, what is commonly known as a 'flatfield'.

```
$flat = rfits "m51_flatfield.fits";
imag $flat;
```

This is shown in Fig. 1.3. Because the image is of a uniform field, the actual image reflects the detector sensitivity. To correct our M51 image, we merely have to divide the image by the flatfield:

```
$gal = $a / $flat;
imag $gal,0,300;
wfits $gal, 'fixed_gal.fits'; # Save our work as a FITS file
```

Well that's a lot better (Fig. 1.4). But think what we have just done. Both `$a` and `$flat` are *images*, with 512 pixels by 512 pixels. **The divide operator '`/`' has been applied over all 262144 data values in the piddles `$a` and `$flat`.** And it was pretty fast too — these are what are known as *vectorised* operations. In PDL each of these is implemented by heavily optimised C code, which is what makes PDL very efficient for procession of large chunks of data. If you did the same operation using normal perl arrays rather than piddles it would be about ten to twenty times slower (and use ten times more memory). In fact we can do whatever arithmetic operations we like on image piddles:

[3] Alright. KGB confesses: he faked this up.

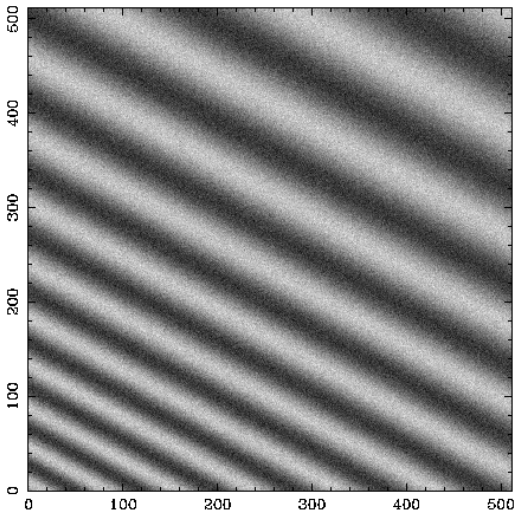


Figure 1.3: The 'flatfield' image showing the detector sensitivity of the raw data

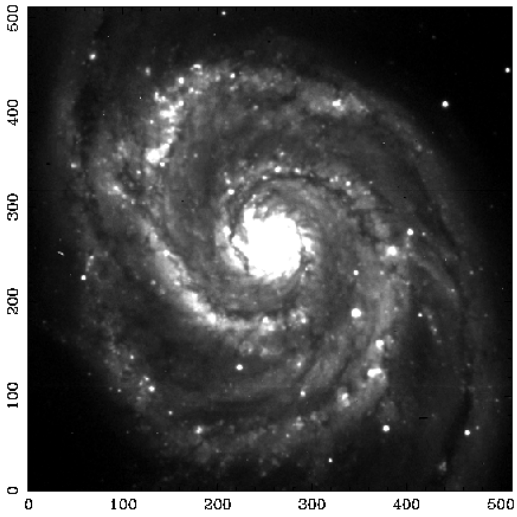


Figure 1.4: The M51 image corrected for the flatfield

```
$funny = log(($gal/300)**2 - $gal/100  + 4);
imag $funny; # Surprise!
```

Or on 1-D line piddles. On on 3-D cubic piddles. In fact piddles can support an infinite number of dimensions (though your computers memory won't).

This the key to PDL: the ability to process large chunks of data at once.

1.2.1 Measuring the brightness of M51

How might we extract some useful scientific information out of this image? A simple quanitity an astronomer might want to know is how the brightness of the the 'disk' of the galaxy (the outer region which contains the spiral arms) compares with the 'bulge' (the compact inner nucleus). Well let's find out the total sum of all the light in the image:

```
perldl> print sum($gal);
17916010
```

`sum` just sums up all the data values in all the pixels in the image — in this case the answer is 17916010. If the image is linear (which it is) and if it was calibrated (i.e. we knew the relation between data numbers and brightness units) we could work out the total brightness. Let's turn it round — we know that M51 has a luminosity of about 10^{36} Watts, so we can work out what one data value corresponds to in physical units:

```
perldl> p 10**36/sum($gal)
5.58159992096455e+28
```

This is also about 200 solar luminosities, (Note we have switched to using '`p`' as a shorthand for '`print`' — this only works in the `perldl`' shell.) which gives 4 billion solar luminosities for the whole galaxy.

OK we do not need PDL for this simple arithmetic, let's get back to computations that involbe the whole image. How can we get the sum of a piece of an image, e.g. near the centre? Well in PDL there is more than one way to do it (Perl aficionados call this phenomenon TIMTOWTDI). In this case, because we really want the brightness in a circular aperture, we'll use the `rvals` function:

```
$r = rvals $gal;
imag $r;
```

Remember `rvals`? It replaces all the pixels in an image with it's distance from the centre (Figure 1.5(a)). We can turn this into a *mask* with a simple operation like:

```
$mask = $r<50;
imag $mask;
```

The Perl *less than operator* is applied to all pixels in the image. You can see the result (Figure 1.5(b)) is an image which is 0 on the outskirts and 1 in the area of the nucleus. We can then simply use the mask image to isolate in a simple way the bulge and disk components (Figures 1.5(c) and 1.5(d)): it very easy to find the brightness of both pieces of the M51 galaxy:

```
perldl> $bulge = $mask * $gal
perldl> imag $bulge,0,300
Displaying 512 x 512 image from 0 to 300 ...
perldl> print sum $bulge;
3011125

perldl> $disk = $gal * (1-$mask)
perldl> imag $disk,0,300
Displaying 512 x 512 image from 0 to 300 ...
perldl> print sum $disk
14904884
```

You can see that the disk is about 5 times brighter than the bulge in total, despite it's more diffuse appearance. This is typical for spiral galaxies. We might ask a different question: how does the average *surface brightness*, the brightness per unit area on the sky, compare between bulge and disk? This is again quite straight forward:

```
print sum($bulge)/sum($mask);
print sum($disk)/sum(1-$mask);
```

We work out the area by simply summing up the 0,1 pixels in the mask image. The answer is the bulge has about 7 times the surface brightness than the disk — something we might have guessed from looking at Figure 1.4 — which tells astronomers it's stellar density is much higher.

Of course PDL being so powerful, we could have figured this out in one line:

```
perldl> print average($gal->where(rvals($gal)<50))
  / average($gal->where(rvals($gal)>=50))
6.56590509414673
```

We'll say more about `where()` and it's friends in Chapter 3.

1.2.2 Twinkle, twinkle, little star.

Let's look at something else, we'll zoom in on a small piece of the image:

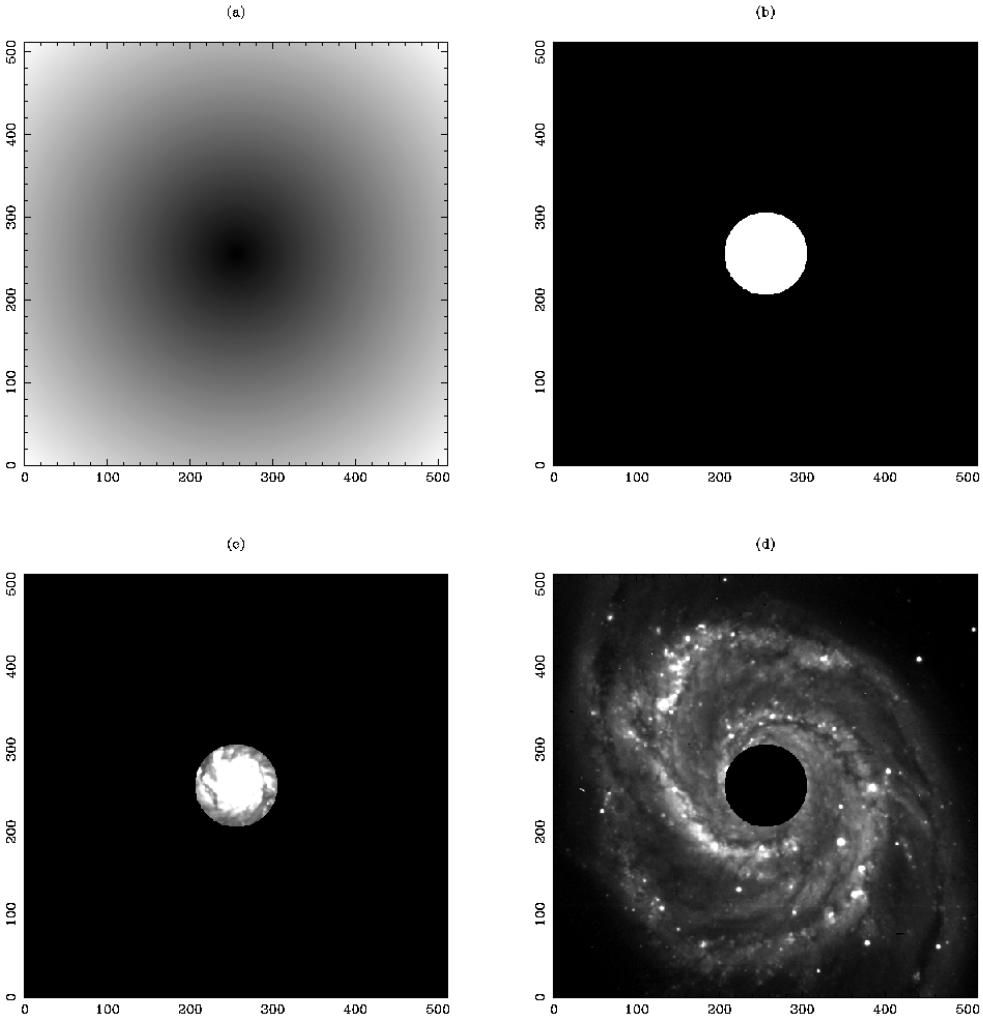


Figure 1.5: Using `rvals` to generate a mask image to isolate the galaxy bulge and disk

```
$section = $gal(337:357,178:198);
imag $section; # It's the bright star
```

Here we are introducing something new - we can see that PDL supports *extensions* to the Perl syntax. We can say `$var(a:b,c:d...)` to specify *multidimensional slices*. In this case we have produced a sub-image ranging from pixel 327 to 367 along the first dimension, and 168 through 208 along the second. We'll talk some more about `slicing and dicing` in Chapter 3. This sub-image happens to contain a bright star.

At this point you will probably be able to work out for yourself the amount of light coming from this star, compared to the whole galaxy.[4] But let's look at something more involved: the radial profile of the star. Since stars are a long way away they are almost point sources, but our camera will blur them out into little disks, and for our analysis we might want an exact figure for this blurring.

We want to plot all the brightness of all the pixels in this section, against the distance from the centre. (We've chosen the section to be conveniently centred on the star, you could think if you want about how you might determine the centroid automatically using the `xvals` and `yvals` functiona). Well it is simple enough to get the distance from the centre:

```
$r = rvals $section;
```

But to produce a one-dimensional plot of one against the other we need to reduce the 2D data arrays to one dimension. (i.e our 21 by 21 image section becomes a 441 element vector). This can be done using the PDL `clump` function, which 'clumps' together an arbitrary number of dimensions:

```
$rr  = clump($r,2); # Clump first two dimensions
$sec = clump($section,2);

points $rr, $sec;  # Radial plot
```

You should see a nice graph with points like those in Figure 1.6 showing the drop-off from the bright centre of the star. The blurring is usually measured by the 'Full Width Half Maximum' (FWHM) — or in plain terms how fat the profile is across when it drops by half. Looking at Figure 1.6 it looks like this is about 2−3 pixels — pretty compact!

Well we don't just want a guess — let's fit the profile with a function. These blurring functions are usually represented by the 'Gaussian' function. PDL comes with a whole variety of general purpose and special purpose fitting functions which people have written for their own purposes (and so will you we hope!). Fitting Gaussians is something that happens rather a lot and there is surpisingly enough a special function for this very purpose. (One could use

[4] Answer: about 2%

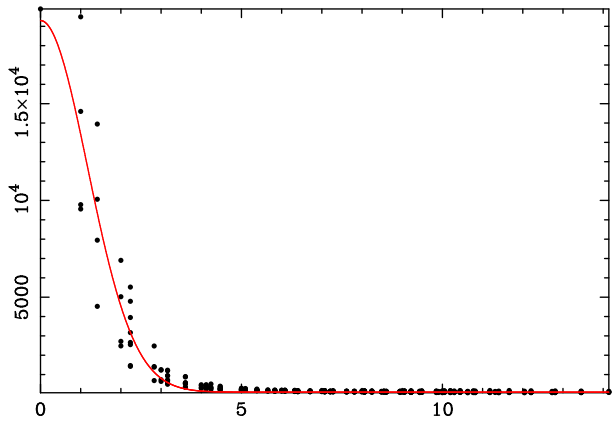


Figure 1.6: Radial light profile of the bright star with fitted curve

more general fitting packages like `PDL::Fit::LM` or `PDL::Opt::Simplex` but that would require more care).

```
use PDL::Fit::Gaussian;
```

This loads in the module to do this. PDL, like Perl, is modular. We don't load all the available modules by default just a convenient subset. How can we find useful PDL functions and modules? Well `help` tells us more about what we already know, to find out about what we don't know use `apropos`:

```
perldl> apropos gaussian
PDL::Fit::Gaussian ...
                Module: routines for fitting gaussians
PDL::Gaussian   Module: Gaussian distributions.
fitgauss1d      Fit 1D Gassian to data piddle
fitgauss1dr     Fit 1D Gassian to radial data piddle
gefa            Factor a matrix using Gaussian elimination.
grandom         Constructor which returns piddle of Gaussian random numbers
ndtri           The value for which the area under the Gaussian probability density functio
                infinity) is equal to the argument (cf erfi). Works inplace.
```

This tells us a whole lot about various functions and modules to do with Gaussians. Note that we can abbreviate `help` and `apropos` with `?` and `??` when using the `perldl` shell.

Let's fit a Gaussian:

```
($peak, $fwhm, $background) = fitgauss1dr($rr, $sec);
print  $peak, $fwhm, $background;
```

`fitgauss1dr` is a function in the module PDL::Fit::Gaussian which fits a Gaussian constrained to be radial (i.e. whose peak is at the origin). You can see that, unlike C and FORTRAN, Perl functions can return more than one result value. This is pretty convenient. You can see the FWHM is more like 2.75 pixels. Let's generate a fitted curve with this functional form.

```
$rrr = sequence(2000)/100  # Generate radial values 0,0.01,0,02..20

# Generate gaussian with given FWHM

$fit = $peak * exp(-2.772 * ($rrr/$fwhm)**2) + $background;
```

Note the use of a new function, `sequence(N)`, which generates a new piddle with values ranging 0..N. We are simply using this to generate the horizontal axis values for the plot. Now let's overlay it on the previous plot.

```
hold; # This command stops new plots starting new pages
line $rrr, $fit, {Colour=>2} ; # Line plot
```

The last `line` command shows the PDL syntax for optional function arguments. This is based on the Perl's built in hash syntax. We'll say more about this later in Chapter 4. The result should look a lot like Figure 1.6 — except unlike this book you should have some colour on your screen!. Not too bad. We could perhaps do a bit better by exactly centroiding the image but it will do for now.

Let's make a *simulation* of the 2D stellar image. This is equally easy:

```
$fit2d = $peak * exp(-2.772 * ($r/$fwhm)**2);
release; # Back to new page for new plots;
imag $fit2d;
wfits $fit2d, 'fake_star.fits'; # Save our work
```

But that plot (Figure 1.7(a)) is a bit boring. So far we have been using simple 2D graphics from the `PDL::Graphics::PGPLOT` library. In fact PDL has more than one graphics library (some see this as a flaw, some as a feature!). Using the `PDL::Graphics::TriD` library which does OpenGL graphics we can look at our simulated star in 3D (Figure 1.7(b)):

```
use PDL::Graphics::TriD; # Load the 3D graphics module
imag3d [$fit2d];
```

If you do this on your computer you should be able to look at the graphic from different sides by simply dragging in the plot window with the mouse! You can also zoom in and out with the right mouse button. Note that `imag3d` has it's a rather different syntax for processing it's arguments — for very good reasons — we'll explore 3D graphics further in Chapter 4.

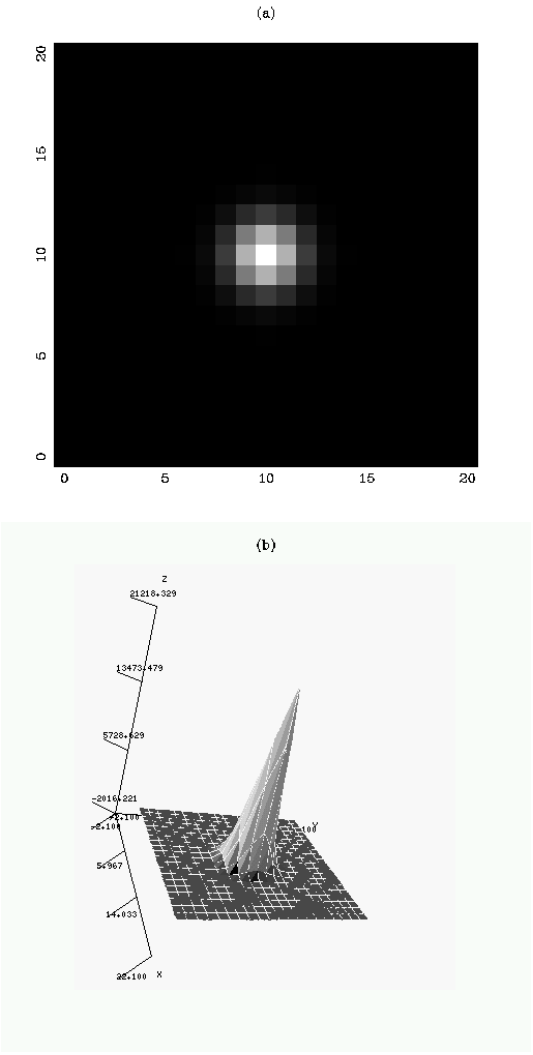


Figure 1.7: Two different views of the 2D simulated Point Spread Function

Finally here's something interesting. Let's take our fake star and place it elsewhere on the galaxy image.

```
$newsection = $gal(50:70,70:90);
$newsection +=  $fit2d;
imag $gal,0,300;
```

We have a bright new star where none existed before! The C-style `+=` increment operator is worth noting — it actually modifies the contents of `$newsection` in-place. And because `$newsection` is a *slice* of `$gal` the change also affects `$gal`. This is an important property of slices — any change to the slice affects the *parent*. This kind of parent/child relationship is a powerful property of many PDL functions, not just slicing. What's more in many cases it leads to memory efficiency, when this kind of linear slice is stored we only store the start/stop/step and not a new copy of the actual data.

Of course sometimes we DO want a new copy of the actual data, for example if we plan to do something evil to it. To do this we could use the alternative form:

```
$newsection = $newsection +  $fit2d
```

Now a new version of `$newsection` is created which has nothing to do with the original `$gal`. In fact there is more than one way to do this as we will see in later chapters.

Just to amuse ourselves, lets write a short script to cover M51 with dozens of fake stars of random brightnesses:

```
use PDL;
use PDL::Graphics::PGPLOT;

srand(42); # Set the random number seed

$gal  = rfits "fixed_gal.fits";
$star = rfits "fake_star.fits";

sub addstar {
   ($x,$y) = @_;
   $xx = $x+20; $yy = $y+20;
   # Note use of slice on the LHS!
   $gal($x:$xx,$y:$yy) += $star * rand(2);
}


for (1..100) {
   $x1 = int(rand(470)+10);
```

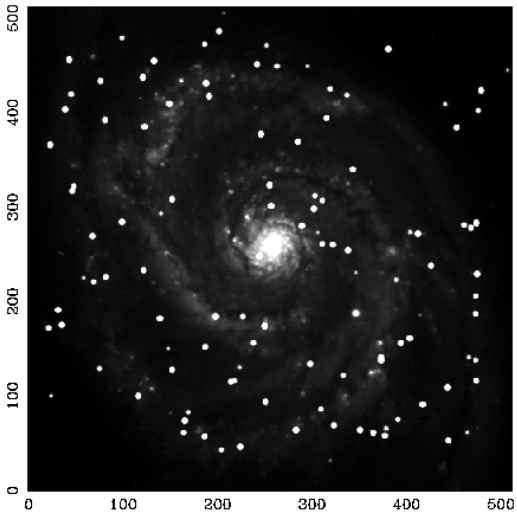


Figure 1.8: M51 covered in fake stars

```
    $y1 = int(rand(470)+10);
    addstar($x1,$y1);
}

imag $gal,0,1000;
```

This ought to give the casual reader some flavour of the Perl syntax — quite simple and quite like C except that the entities being manipulated here are entire arrays of data, not single numbers. The result is shown, for amusement, in Fig. 1.8 and takes virtually no time to compute.

1.2.3 Getting Complex with M51

To conclude this frantic whirl through the possibilities of PDL, let's look at a moderately complex (sic) example. We'll take M51 and try to enhance it to reveal the large-scale structure, and then subtract this to reveal small-scale structure.

Just to show off we'll use a method based on the Fourier transform — don't worry if you don't know much about these, all you need to know is that the Fourier transform turns the image into an 'inverse' image, with complex numbers, where each pixel represents the strength of wavelengths of different scales in the image. Let's do it:

```
use PDL::FFT; # Load Fast Fourier Transform package

$gal = rfits "fixed_gal.fits";
```

Now `$gal` contains real values, to do the Fourier transform it has to have complex values. We create a variable `$imag` to hold the imaginary component and set to zero.[5]

```
$imag = $gal * 0;       # Create imaginary component, equal to zero

fftnd $gal, $imag;      # Perform Fourier transform
```

`fftnd` performs a Fast Fourier Transform, in-place, on arbitrary-dimensioned data (i.e. it is 'N-dimensional'). You can display `$gal` after the FFT but you won't see much. If at this point we ran `ifftnd` to invert it we would get the original `$gal` back.

If we want to enhance the large-scale structure we want to make a filter to only let through low-frequencies:

```
$tmp = rvals($gal)<10;        # Radially-symmetric filter function
$filter = kernctr $tmp, $tmp; # Shift origin to 0,0
imag $filter;
```

You can see from the image that `$filter` is zero everywhere except near the origin (0,0) (and the 3 reflected corners). As a result it only let's through low-frequency wavelengths. So we multiply by the filter and FFT back to see the result:

```
($gal2, $imag2) = cmul $gal, $imag, $filter, 0; # Complex multiplication
ifftnd $gal2, $imag2;
imag $gal2,0,300;
```

Well that looks quite a bit different! (Figure 1.9(a)). Just about all the high-frequency information has vanished. To see the high-frequency information we can just subtract our filtered image from the original (Figure 1.9(b) — double yuk!):

```
$orig = rfits "fixed_gal.fits";
imag $orig-$gal2,0,100;
```

1.3 Roundoff

Well that is probably enough abuse of Messier 51. We have demonstrated the ease of simple and complex data processing with PDL and how PDL fit's neatly

[5] For reasons of efficiency complex numbers are represented in PDL by seperate real and imaginary arrays — more about this in Chapter 2.

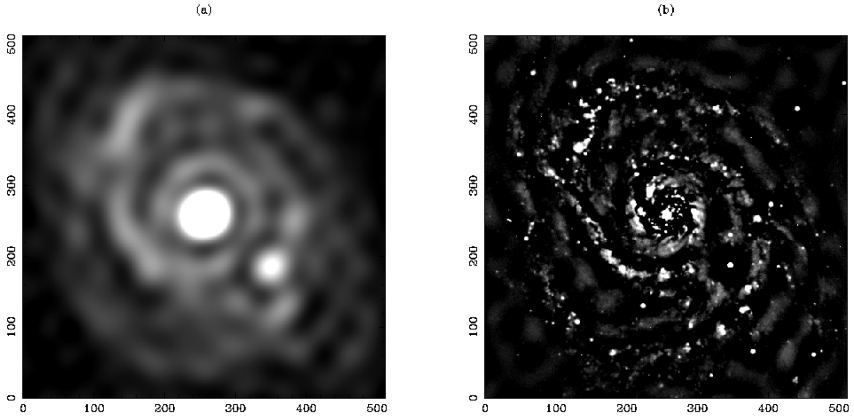


Figure 1.9: Fourier filtered smoothed image and contrast enhanced image with the smoothed image subtracted.

in to the Perl syntax as well as extending it. You have come across basic arithmetical operations and a scattering of useful functions — and learned how to find more. You certainly ought now to have a good feel of what PDL is all about. In the next chapter we'll take a more comprehensive look at the basic parts of PDL that all keen PDL users should know.

Chapter 2

The Basics of PDL

In Chapter 1 we had a whirlwind tour of the sort of things you can do with PDL. If you are still reading this you must now be convinced that PDL is worth using and learning in more depth. We will now focus on a more measured introduction to the basic concepts and features of the language.

2.1 Basic manipulations of piddles

Perhaps the best place to start with PDL is how the data arrays are represented to Perl. We use Perl objects, which makes it extremely powerful, but we will need to explain how piddles relate to everyday Perl variables.

2.1.1 What is a piddle?

PDL uses Perl 'objects' to hold piddle data. An 'object' is like a user-defined data-type and is a very powerful feature of Perl, PDL creates it's own class of 'PDL' objects to store piddles. These look like ordinary Perl variables such as `$x`, `$Foo`, `$MyData`, etc.

Most of the time you can forget about the fact that piddles are objects and treat them like ordinary variables:

```
$x = rfits 'file.fits';
$y = rvals($x);
$z = $x/$y;
print sqrt($x+$y+$z);
```

The only time the distinction becomes important is when creating piddles and using the '`=`' operator.

2.1.2 Creating piddles

Creation of new piddles has to be done via a PDL function. Otherwise Perl doesn't know the variable has PDL magic.

The simplest method for small piddles is to use the `pdl` function which can accept numbers and lists of numbers, as well as other piddles:

```
$a = pdl (1,2,3,4,5);                  # Create 1D piddle
$a = pdl ([1,2,3,4,5], [2,3,4,5,6]);   # Create 2D piddle
$a = pdl 42;                           # Create 0D (scalar) piddle
```

The numbers are specified using Perl's '`[]`' list reference syntax. See the help on `pdl` for more details. It is not normally necessary to worry about converting single numbers (Perl 'scalars') to piddles — all PDL functions will do this on-the-fly. So you can say:

```
$a = pdl (1,2,3,4,5);
$b = $a + 2;
```

without having to care whether '2' is a piddle or not. The result, `$b`, is a piddle of course.

Of course if you are a heavy-duty PDL user your piddles will be BIG. Thus you will need to use a PDL I/O function to read your data from a file in some format. For example in Chapter 1 we used `rfits`:

```
$a = rfits "fixed_gal.fits";
```

`rfits` is a PDL function which reads the data and returns it as a piddle. Another useful function is `rcols` which reads columns of data from a text file:

```
($a, $b) = rcols 'text.dat';
```

It is actually surprisingly easy to write these I/O functions — most of the code for `rfits` is straight forward Perl. PDL comes with a plethora of I/O functions for reading free-formatted data, raw binary formats and specially defined formats like FITS and GIFs.

Another way of creating BIG piddles is using functions such as `zeroes`, This creates piddles, filled with zeroes, of whatever dimensions you specify:

```
$a = zeroes 10;      # 10 element 1D piddle
$a = zeroes 100,100; # 100x100 2D piddle
$a = zeroes 3,4,5,6  # 3x4x5x6 4D piddle
```

Or you can use another piddle as a template for the dimensionaligy:

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

Other creation functions like `random` and `rvals` work in this way. Table ??? lists the Top Ten ways of creating piddles.

Table???.

2.1.3 Elementary arithmetic.

The following binary perl operators act on piddles:

```
+ * - / ** > < <= >= == !=  << >> | & ^  ! % <=>
```

these behave identically to their ordinary Perl equivalents, except they act element-by-element on the whole piddle. e.g.:

```
perldl>   $a = pdl (3,2,1,0,-1,-2,-3);
perldl>   $b= $a * 2; print $b;
[6 4 2 0 -2 -4 -6]

perldl>   print $b % 3;
[0 1 2 0 -2 -1 0]

perldl>   print !$b;
[0 0 0 1 0 0 0]

perldl>   print $a <=> $b;
[-1 -1 -1 0 1 1 1]
```

Additionally the following perl operators, which normally modify the variables in-place (like C), also act in-place on piddles:

```
++  --  +=  -=  *=  /=  |=  &=  ^=  %=

perldl> $a = pdl (3,2,1,0,-1,-2,-3);

perldl> $a++; print $a;
[4 3 2 1 0 -1 -2]

perldl> $a *= 3; print $a
[12 9 6 3 0 -3 -6]

perldl> $a |= 3; print $a
[15 11 7 3 3 -1 -5]
```

The operators:

```
~  x  .=
```

have been redefined in PDL to be more useful. The '~' operator produces an matrix transpose in PDL (more useful than the perl XOR meaning, which can be produced other ways). 'x' — also usually a string operator — implements matrix multiplication. The '.=' operator is used for the PDL assignment operator. (See below)

The standard Perl mathematical functions:

```
sqrt sin cos atan2 abs log exp
```

also act element-by-element on piddles.

```
perldl> print sin pdl (3,2,1,0,-1,-2,-3);
[0.14112001 0.90929743 0.84147098 0 -0.84147098 -0.90929743 -0.14112001]
```

PDL does of course come with a whole grab-bag of mathematical functions and algorithms which go far beyond what is available in vanilla Perl. You'll come across some more of these later in this book, and there is a big list in Appendix ??.

2.1.4 In-place arithmetic

It is pretty handy in PDL to be able to say `$a++` instead of `$b = $a+1`, the latter generates a second piddle and lots of memory will be consumed if `$a` is large. `$a++` in PDL is known as an operation. The operators such as '`+=`', '`*=`', '`/=`', etc. allow in-place versions of all the standard arithmetic operations to be performed.

But what about *functions* like `sin`, `sqrt`, etc. which we would also like to do in-place? Well as well as saying:

```
$b = sin($a);
```

with piddles it is also possible to say:

```
sin(inplace($a));
```

The `inplace` function sets a special flag on the piddle that the next operation is to be done in-place, if the operation supports it. All the standard mathematical functions defined by Perl (Section???) can be done in-place.

2.1.5 Using = and .=

Piddles are pretty convenient to use. But one thing to beware of:

```
# $a is a piddle
```

```
$b = $a; # Does NOT copy the piddle data
$b++;    # Changes $a !!!
```

This is because piddles are represented by Perl objects, `$a` just holds a *reference* to the piddle data (kind of like a pointer in C). Saying `$b = $a` just copies the reference.

A similar effect happens in a subroutine call:

```
$a = pdl (1,2,3);
foo($a);
print $a; # Answer: [2,3,4]

sub foo {
  my $x = shift;
  $x++;
}
```

The subroutine argument is *passed by reference* — any changes to `$x` change `$a`. This behaviour is actually a good thing for PDL: because we expect piddles to be LARGE we do not want a lot of unnecessary copying.

So how do we get a real copy? The recommended way is to use the `pdl` function:[1]

```
$b = pdl($a);
```

Once we have element-by-element arithmetic it turns out that as well as simple copying we need a special assignment operator and '`.=`' plays this role. Since PDL does not use strings we have hijacked this to provide a convenient assigment operator. For example:

```
perldl> $b = pdl (1,2,3);
perldl> $b .= 2
perldl> print $b
[2 2 2]
```

The assignment operates element-by element — just like the arithmetic operators. You can see that this is the same as `$b *= 0; $b += 2;` — only more succinctly and much faster. The assignment operator, like the arithemetic and other operators, also partakes in sophisticated PDL features such as *threading* and *dataflow* which we will talk about in Chapter XXX.

2.2 Piddles are NOT Perl 'arrays'

It is now time to answer a question which has probably been nagging at the back of your mind for a while.

[1] Object methods such as `$a->copy` are also available, more on this later

Why bother with piddles? Why not just use normal Perl 'arrays'?

By Perl 'arrays' we of couse mean entities like `@x` and `@Data` z which one would normally create and manipulate like this:

```
@x = (1,2,3);
push @x, 42;
$y = pop @x;
```

So why don't we just use Perl 'arrays'? Several very good reasons:

1. It is impossible to manipulate Perl 'arrays' arithmetically as one would like. i.e.:

   ```
   @y = @x * 2;  # Wrong!
   ```

   can not be made to operate element by element.

2. Perl 'arrays' are really what are known in computer science as 'lists' (and are represented internally by a list data structure). In fact if the PDL-Porters had their evil way they would ban the term 'array' from all of the standard Perl documentation and books. This is why the term 'piddle' was invented for use in PDL for what we think really are 'true arrays'.

3. Perl lists are intrinsically one-dimensional. You can have 'lists of lists' but this is not the same thing as true multi-dimensional arrays. Honest.

4. Perl lists consume a lot of memory. At least 20 bytes per number, of which only a few are for the actual value. This is because Perl lists are flexible, and can contains text strings as well as numbers. This flexibility requires an internal complex data structure which contains extra information such as a place holder for the number, a place holder for the text and pointers forward and back along the list.

5. Perl lists are scattered about memory. The list data structure means consecutive numbers are not stored in a neat block of consecutive memory addresses as C and FORTRAN programmers are used to. This makes it difficult to pass the arrays to low-level C and FORTRAN routines for processing — the numbers must be collected together -a process known as 'packing' — processed and unpacked back into lists. If you have 'lists of lists' then it get's even worse.

6. Perl lists do not support the range of datatypes that piddles do (byte arrays, integer arrays, single precision, double precision, etc.)

That is why PDL does not use Perl lists. Just to be clear from now on we'll always refer to PDL numeric data arrays as 'piddles' and Perl-style number/text arrays as 'lists'.

2.2.1 The benefits of piddles.

So what we have done is essentially implemented a new kind of data structure, the piddle, in Perl to hold large chunks of numerical data which we can process all at once.

The piddle data is held in a large chunk of consecutive memory addresses, which means it can be passed easily to C or FORTRAN code as if it was a C or FORTRAN array.

We have written a lot of C code to do all the obvious stuff, such as element-by-element arithmetic, for multiple datatypes, and this is all built into PDL. Thus when you say:

```
# $a,$b and $c are piddles

$b = $a+1;
$a++;
$c = 1+$a**2/($b-2);
```

you have it done on all elements, even if `$a`, `$b`, `$c` contain millions of them. Furthermore this arithmetic proceeds at C/FORTRAN speed.[2] `$b=$a+1` gets interpreted one, but might get executed millions of times in a compiled C loop. In short PDL is a *vectorised* language.

2.2.2 Lists and Hashes of piddles

Of course all this is not to say that Perl lists do not have tbeir uses. Since piddles are singular objects we can make lists of them:

```
@mylist = (rfits("my.fits"), zeroes(3,3), pdl(32));
```

This can be very convenient for many kinds of processing. For example one common PDL idiom is instead of:

```
($x,$y,$z) = rcols 'datafile'; # Read columns of data
```

one can say:

```
@x =  rcols 'datafile'; # Read columns of data
```

Then one can manipulate the individual piddles as `$x[0]`, `$x[1]` etc. (Each of which can have completely different dimensions).

[2]This is about 7 million numbers per second on Karl's state-of-the-art laptop. Which by the time you are reading this is probably a rather daggy laptop.

The other major data struture in standard Perl is the *hash* (also known as an *associative array*. These can also contain many piddles, this time indexed by a text string rather than position in a list. For example a face recognition program might want to store it's images this way:

```
%face = ();
$face{"Jim"} = rpic('jim.jpg');
$face{"Fred"} = rpic('fred.jpg');
$face("Default"} = zeroes(256,256);
```

So you can see that the combination of piddles, lists and hashes can be quite useful, albeit a tad confusing at times!

2.3 PDL Datatypes

Standard Perl only offers long integers and double-precision floating point as datatypes. This is insufficient for efficient numerical processing — both speed and memory consumption usually requires a number representation be only as accurate as required to represent the data an no more. For example if you are dealing with standard image formats like GIF or JPEG you might want only an 8-bit representation for each colour — in this case you then want to represent each number with a single byte.

For this reason PDL provides a whole bunch of datatypes. We can have piddles containing any of these:

Datatype	PDL function	Bytes/ element	Numerical range
Byte	byte()	1	0–255
unsigned short integer	ushort()	2	0–65535
short integer	short()	2	-32768 – 32767
long integer	long()	4	-2147483648 – 2147483647
IEEE single-precision floating point	float()	4	XXXX
IEEE double-precision floating point	double()	8	XXXX

Each of these corresponds to a C primitive datatype used in the low-level PDL code. The philosophy in PDL is that the types are machine independent — this means that a long integer is always 4 bytes in PDL across all types of computer/OS. Definitions are set up apporopriately at the C level to handle this.

2.3.1 Type conversion and creation

The functions listed in Table ??? provide type conversion:

```
# If $a is any piddle

$b = float $a;  # $b is a 'float' piddle
$b = byte $a;   # $b is a 'byte' piddle
```

and piddle creation with a specified type:

```
$b = ushort (1,2,3);           # $b is a 1-D 'ushort' piddle
$b = double ([1,2,3],[4,5,6]); # $b is a 2-D 'double' piddle
$b = long 42;                  # $b is a 0-D (scalar) 'long'  piddle
```

These use the same '`[]`' list-reference syntax as the `pdl` function. (Note that the `pdl` function defaults to double-precision just like Perl.)

Finally the type functions also function without arguments as tokens when creating arbitrary size piddles. For example to create a 512x512x512 byte data cube filled with zeroes:

```
$a = zeroes(byte, 512,512,512); # Note the "," after "byte"!
```

This array consumes 128Mb of memory on Karl's laptop. If the `byte` token was omitted PDL would try to create the piddle as double precision — being 8 times bigger this would be far too much for the laptop's memory! [3]

2.3.2 Complex types

For a time a long debate reigned over where PDL should support types of complex numbers as well. One issue was that standard C, on which PDL is built, does not have a primitive complex number type. Because of this C routines in numerical libraries which handle complex arrays take the real and imaginary components as seperate arguments. As PDL needs to use these routines and work efficiently we take the same approach — PDL functions which deal with complex numbers use seperate real and imaginary arrays.

A good example of this is the PDL::FFT module for dealing with Fourier Transforms. One says:

```
fft $real, $imag;
```

where the arguments are piddles. The PDL::FFT module also provides `cmul` and `cdiv` functions for complex arithmetic.

PDL::Complex module ??? XXXX

[3] PDL has the power to make your computer go all weak at the knees, with minimal typing. But if you really need to have everything in memory you can memory map your hard disk directly with `mapfraw` in PDL::Io::FastRaw.

2.4 PDL projection operators

We've now encountered the basics of PDL — element by element arithmetic. It is time to look at more advanced features which involve combining elements together inside a piddle. Perhaps the next most common class of operations after arithmetic are those known as *projection operations* which involve combing elements along axes.

One example of such an operator is summation. The `sumover` function implements this. To sum along rows:

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

[
 [0 1]
 [2 3]
 [4 5]
]

perldl> $b = sumover $a; # $b is now summed along axis 0
perldl> print $b;
[1 5 9]
```

`sumover` is paired with the related `sum` function which sums over all elements to form a scalar:

```
perldl> print sum $a
15
perldl> print sum $b  # Should be the same of course!
15
```

The rest of the projection operators follow this basic pattern, for example `maximum` and `max`:

```
perldl> print maximum $a
[1 3 5]
perldl> print max $a
5
```

One can think of this as `maximum` projecting from `N` to `N-1` dimensions and `max` projecting from `N` to `0` dimensions Here is a complete Table of those in standard PDL (of course you are free to define your own):

Operation	Dimensions Projection	
	$N \to N-1$	$N \to 0$
Pixel sum	sumover	sum
Pixel average	average	avg
Pixel minimum	minimum	min
Pixel maximum	maximum	max
Pixel median	medover	median
Pixel 'odd median'	oddmedover	oddmedian

2.4.1 What about projecting along other dimensions?

You might now be asking yourself: what about summing along columns as well as rows? Or more generally projecting along a different axis to the first? Are there special functions for this or are there special options to the projection functions?

The answer is **NO**. It works much more elegantly than that!

PDL comes with special dimensions manipulation functions. We simply use `sumover` (or whatever) with a dimension operation which brings the required dimension to the front. For example to sum along columns:

```
$a = random(10,3);
$b = sumover(mv($a,1,0)); # $b has dims 10.
```

This uses the `mv` function to transform `$a` from 10 by 3 to a 3 by 10 piddle. This is done 'virtually' — i.e. no more memory is used. This is an extremely powerful feature of PDL — functions like `sumover` only need to be written to handle the dimensions they care about, PDL with it's dimension operators can handle the rest.

2.5 'Threading' over extra dimensions.

The projection operators are written in PDL as functions of 1-D vectors — they project along the vector.

You will notice:

```
$a = sumover random(10);        # $a is 0-D: 1     element
$a = sumover random(10,9);      # $a is 1-D: 9     elements
$a = sumover random(10,9,8);    # $a is 2-D: 9x8   elements
$a = sumover random(10,9,8,7);  # $a is 3-D: 9x8x7 elements
```

i.e. even though `sumover` is written to handle 1-D piddles if supplied with extra dimensions `sumover` will automatically loop over the extras. Moreover this iteration happens at the compiled C level, so it is extremely very fast.

This feature of implicit looping over extra dimensions is known as 'threading over the dimensions'[4]. Here is another example:

```
$a = random (10,20,30);
$b = random (10,20);

$c = $a + $b;  # $c is 10x20x30
```

Yes the '+' function is *threaded*. This means you can do incredibly useful things like add images to datacubes and have the operation repeat for each image slice in the cube. With no special syntax.

You have of course seen before:

```
$a = random (10);
$b = $a + 2;
```

But perhaps you now realise this is again a case of threading. Here the 0-D piddle '2' is threaded over the 1-D vector `$a`. The previous example is just a higher-dimensional analogy.

We will look in more detail at the rules regarding threading in Chapter 4. For now we note that *implicit threading* works pretty much as you might expect:

1. Extra dimensions of size one are implicitly added on the end of all piddles to match the piddle with the biggest number of dimensions.
2. Each dimension threaded over must either be of size one or of a size matching the biggest piddle.
3. Dimensions of size one are continuously repeated during the thread loop.

Thus in:

```
sequence(10,20,3) + sequence(10) + sequence(10,20);
```

This becomes:

```
sequence(10,20,3) + sequence(10,1,1) + sequence(10,20,1);
```

and the result is *as if* the data was repeated along the unit dimensions, i.e. like:

```
sequence(10,20,3) + sequence(10,20,3) + sequence(10,20,3);
```

[4] No one to one relation to threads of execution on parallel computers though some versions of PDL will actually implement threading this way!

We'll restate those rules more formally in Chapter XXX — and also look at a way to make your own rules called *explicit threading*. But the easiest way to find out how threading works is to experiment.

Before we get too deeply involved in threading and how to write threaded functions we should look in more detail at the other half of the game: the variety of operations available in PDL for manipluating piddle dimensions. Chapter 3 will go into this in detail.

2.6 Example Problem

WHAT???

Chapter 3

Slicing, Dicing and Threading with PDL dimensions

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.

3.1 Finding piddle dimensions.

As we mentioned in Chapter 2 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);
```

3.2 The `slice` function — regular subsets along axes

We saw back in Chapter 1 how to extract a rectangular subset of a piddle:

```
$section = slice $gal, "337:357,178:198";
```

The piddle `$gal` was a 2D image, we used the `slice` function to extract a contiguous subset ranging from pixel 327 to 367 along the first dimension, and 168 through 208 along the second.

`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, there is no special syntax like some languages have. This is because there is no extra room in Perl for extra special purpose syntax (though there may be in future versions). But we do turn this to our advantage: in PDL we use other named functions to extract other kinds of slices. And if the user is unhappy with the syntax of `slice` they are able to write their own — `slice` is not special. We do lose something in conciseness but gain quite a bit in generaility.

3.2.1 The basic slicing specification.

The second argument 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  = slice $data, '0:20';
```

```
$data = zeroes(100,100,20);
$sec  = slice $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  = slice $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(100);  # 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 24 25 26 27
28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52
53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77
78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99]

perldl> print slice $x,'40:80:2'
[40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80]
```

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 slice $a,':,5:10,:'
10 6 3
perldl> print dims slice $a,',5:10,'
10 6 3
```

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

```
$z = zeroes 100,200;
$col = slice $z, "42,:";  # Column 42 (Dims = 1x200)
$row = slice $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 = slice $data,"$x1:$x2";
```

3.2.2 Modifying slices.

Here's the biggy:

```
perldl>  $x = sequence(25);
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 24]

perldl> $slice = slice($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*, which we will explore further in Chapter XXX. From now on we'll describe anything with this property

One thing we can't (yet) do though is this:

```
slice($x,'40:80:2') .= 42;  # Syntax error!
```

We have to use a temporary piddle to hold the slice before we can change it:

```
$tmp = slice($x,'40:80:2'); $tmp .= 42;
```

`This is due to a current Perl syntax limitation, which is expected to be removed in the near future.`

3.2.3 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 = 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.

3.2.4 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 in this chapter.

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

```
$z = zeroes 100,30;
$col = slice $z, "42,:";    # Column 42 - 2D (Dims = 1x200)
$col = slice $z, "(42),:";  # Column 42 - 1D (Dims = 200)
```

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

```
$col2 = slice $col, "*,:,*"; # Dims now = 1x200x1
```

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

```
$t = slice $z,":,*3,:"; # Dims now 100x3x30
```

This sort of thing is very useful for advanced threading trickery (see *PDL threading and the signature*).

3.2.5 PDL's Method notation

At this point we would like to introduce a new notation for calling `slice` and it's friends. This is bevause 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
$t = slice $z,":,*3,:";    # Function call

$t = $z->slice(":,*3,:");  # Method call
```

The method `$z->slice(..)` notation is very common for slices, as it makes more visual sense to have the slice specifier after the variable.

3.3 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 [4,7,9];
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);
```

3.4 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 C<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.

3.5 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; 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 `clump(-1)` to clump *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.

3.6 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'[1] 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]
]
```

3.7 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:

[1] No relation to the Dungeon Dimensions in Terry Pratchett's 'Discworld'!

```
perldl> print $a = sequence(100)/100;
[0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 0.11 0.12 0.13 0.14 0.15
0.16 0.17 0.18 0.19 0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27 0.28 0.29 0.3 0.31
0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43 0.44 0.45 0.46 0.47
0.48 0.49 0.5 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59 0.6 0.61 0.62 0.63
0.64 0.65 0.66 0.67 0.68 0.69 0.7 0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79
0.8 0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88 0.89 0.9 0.91 0.92 0.93 0.94 0.95
0.96 0.97 0.98 0.99]
```

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<0.2 | $a>0.9;
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 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<0.2 | $a>0.9);    print $idx;
[0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 91 92 93 94 95 96 97 98 99]
```

i.e. elements 0..19 and 91..99 in the original piddle are the ones we want. We can select these using the `index` function:

```
perldl> print $a->index($idx);
[0 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1 0.11 0.12 0.13 0.14 0.15
0.16 0.17 0.18 0.19 0.91 0.92 0.93 0.94 0.95 0.96 0.97 0.98 0.99]
```

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> $tmp = $a->index($idx);
perldl> $tmp .= 0;  # Set indexed values to zero
perldl> print $a;

[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.2 0.21 0.22 0.23 0.24 0.25 0.26 0.27
0.28 0.29 0.3 0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4 0.41 0.42 0.43
0.44 0.45 0.46 0.47 0.48 0.49 0.5 0.51 0.52 0.53 0.54 0.55 0.56 0.57 0.58 0.59
0.6 0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69 0.7 0.71 0.72 0.73 0.74 0.75
0.76 0.77 0.78 0.79 0.8 0.81 0.82 0.83 0.84 0.85 0.86 0.87 0.88 0.89 0.9 0 0 0
0 0 0 0 0 0]
```

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

```
$a = sequence(100)/100;
$tmp = $a->where($a<0.2 | $a>0.9);
$tmp .= 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);
$tmp = $a->index(pdl(0,9));
$tmp++;
```

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> $tmp = $a->index(pdl(0,9));
perldl> $tmp .= 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;
```

3.8 Example Problem

WHAT???

3.9 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?

3.9.1 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.

3.9.2 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 'bead.tif';    # image stored in the TIFF 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]
```

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": "").
      $this->info("%T %D\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[a] (see also Chapter XX on object oriented programming with PDL):

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

and we made the function return the piddle argument so that it can be seamlessly integrated into method invocation chains:

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

[a] If you want to know more about the `info` method check Appendix XX or try `help info` from within the perldl shell

Box 3.9.1: A small utility routine

So let's dissect what has happened. If you look at the documentation of `maximum`[2] 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[3]. And, most imporatantly, this

[2] remember, quickest way: at the perldl shell prompt type `? maximum`

[3] An N-dimensional piddle can be viewed as an (N-1)-dimensional array of one-dimensional subslices.

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 dimensions, 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.

3.9.3 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.

3.9.4 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[4]:

```
print $im3d->pdim('Step 1')->xchg(1,0)->pdim('Step 2')
     ->maximum->pdim('Step 3');
```

results in

```
Step 1       Short D [5,10,3]
Step 2       Short D [10,5,3]
Step 3       Short D [5,3]

[
 [ 45  46  47  48  49]
 [ 95  96  97  98  99]
 [145 146 147 148 149]
]
```

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

[4] `mv` would have been another possible choice: `$im3d->mv(1,0)->maximum`

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.

3.9.5 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`[5]. Similar concepts are also implemented in APL and J[6]. What we think distinguishes PDL from these previous languages is the consistent support of threading throughout PDL, the tight integration with the PLD 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[7]. Note that

[5] Yorick referenceXXX

[6] XXXrefs for both, although well hidden by a wealth of terminology and notation very different from that of most other conventional computer languages

[7] maybe you would like to write such a routine as an exercise to entertain yourself, well, maybe not

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`[8] supplies us with just the functionality we need. 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[9]

```
Benchmark: running PDL thread, Perl loops,
           each for at least 3 CPU seconds...
PDL thread:  3 wallclock secs ( 3.16 usr +  0.02 sys =  3.18 CPU)
             @ 1788.36/s (n=5687)
Perl loops:  6 wallclock secs ( 5.09 usr +  0.13 sys =  5.22 CPU)
             @  2.49/s (n=13)
```

That proofs our point: while the example using threading is executed at a rate of nearly 1800 per second using explicit loops has brought down the speed to less than 3/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

[8] for more details see `perldoc Benchmark`

[9] exact numbers obviously depend on the actual computer used

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[10]. 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 parallelized! The keen reader has probably already observed that those internal automatic loops of elementary operations over subslices do not have to be performed sequentially. On a machine with several CPUs multithreading can be used to run several iterations concurrently. In general, when using threading, interdependencies between the iterations of the thread loop are minimal making for very efficient parallel execution (ref XXXXXmultithreadingXXX ref). This revelation also explains the name *PDL threading* being thereby related but not identical to the use of the terms *thread* and *multithreading* in computer science.

The use of multithreading in PDL thread loops is not only a theoretical possibility. PDL v2 has some support for multithreaded execution of such loops[11]. However, the interface is currently somewhat crude and mainly intended to be a proof of principle. Let's illustrate this by performing our above example yet again but this time ask PDL to benchmark a multithreaded thread loop versus singlethreaded execution[12]:

```
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);
```

[10] mymax only deals with 2D input

[11] using the POSIX pthread software interface. See XXX

[12] multithreading will only be used if your hardware platform supports this and PDL is set up accordingly, check Appendix XX

```
                                        $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 SGI workstation with 4 R10000 CPUs this script yields the following output:

XXXX fill in

3.9.6 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
```

An 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]`. Nearly too simple to go on about much longer.

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:

@deq 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:

@deq 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\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 'PDL.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
}
```

3.9.7 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@startfoot `sig` will only know about functions that were created using `PDL::PP` or if we explicitly specified the signature in the PDL documentation (@@chap_advsci)@endfoot 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.

3.9.8 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 implemented 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 intuitively 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 barfed: PDL: overloaded operator: Parameter 'b':
Mismatched Implicit thread dimension 0: should be 3, is 2
at [...]
```

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<$c-¿*getdim*(0)> as dimension 0 of `$d`:

```
print $d->dummy(0,$c->getdim(0))->pdim("New dims");
```

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 elementary 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 multiplications 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.

end current text

```
1 -- elastic dim size
centroid example
matmult
```

output args -> creation rules otherwise as any other argument in threadloop

vector transformation (matrix x vector)

matrix multiplication

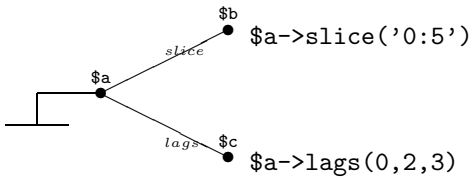
fwhm spread

3.9.9 Promotion: When a piddle has less dimensions than required

Promotion: why and how

3.9.10 Calling conventions and tight loops

3.9.11 Manipulating indices — generalised slicing



determine a shift between 2 images ->lags->lags->inner2d

3.9.12 Memory and speed implications — Vaffines

Implementation matters

3.9.13 writing your own threading aware function — at the perl level!

maybe in advsci chapter on own modules

3.9.14 A worked example

```
# code to generate surface

# code to show surface and normals

# calculate smooth normals
```

```
sub smoothn {
  my ($this,$p) = @_;
  # coords of parallel sides (left and right via 'lags')
  my $trip = $p->lags(1,1,2)->slice(':,:,:,1:-1') -
               $p->lags(1,1,2)->slice(':,:,:,0:-2');
  # coords of diagonals with dim 2 having original and reflected diags
  my $tmp;
  my $trid = ($p->slice(':,0:-2,1:-1')-$p->slice(':,1:-1,0:-2'))
                   ->dummy(2,2);
  # $ortho is a (3D,x-1,left/right triangle,y-1) array that enumerates
  # all triangles
  my $ortho = $trip->crossp($trid);
  $ortho->norm($ortho); # normalise inplace

  # now add to vertices to smooth
  my $aver = ref($p)->zeroes($p->dims);
  # step 1, upper right tri0, upper left tri1
  ($tmp=$aver->lags(1,1,2)->slice(':,:,:,1:-1')) += $ortho;
  # step 2, lower right tri0, lower left tri1
  ($tmp=$aver->lags(1,1,2)->slice(':,:,:,0:-2')) += $ortho;
  # step 3, upper left tri0
  ($tmp=$aver->slice(':,0:-2,1:-1')) += $ortho->slice(':,:,(0)');
  # step 4, lower right tri1
  ($tmp=$aver->slice(':,1:-1,0:-2')) += $ortho->slice(':,:,(1)');
  $aver->norm($aver);
  return $aver;
}

# code to show smoothed surface
```

3.9.15 explicit threading for complex cases

Chapter 4

Graphics with PDL

4.1 Introduction

Some words of wisdom here

4.2 2-dimensional graphics using PGPLOT

A central requirement of any data analysis package is the possibility of visualisation of data. PDL deals with this in a slightly different manner than some other packages in that no built-in graphics library is used, instead it uses other freely available external packages. In this chapter we will focus on the main 2D plotting package, PGPLOT. The 3D interface is detailed in the next chapter.

Here we will cover the use of the `PDL::Graphics::PGPLOT` package which uses the freely available PGPLOT subroutine package written by Tim Pearson[1]. This is a very powerful package and PDL::Graphics::PGPLOT does not provide easy access to everything in the PGPLOT package, although it hopefully does most of what you will need.

For advanced use you might have to use some PGPLOT commands directly, see section 4.9 for a discussion of this. But even if you don't you are recommended to at least keep a copy of the PGPLOT documentation lying around. It is available from `http://www.astro.caltech.edu/~tjp/pgplot`.

The goals of the chapter is to familiarise the reader with the PDL interface to PGPLOT and show how complicated datasets can be easily manipulated and displayed. The focus will be on interactive use to facilitate learning, but at the end we will turn to an object-oriented interface that might be more suited for scripts.

To use PDL::Graphics::PGPLOT it is necessary to have the PGPLOT package installed, and in addition have the Perl PGPLOT module (written by Karl Glazebrook and available through CPAN) installed and working. In the following we will assume that you have this all set up.

4.3 Introducing `PDL::Graphics::PGPLOT`

2-dimensional graphics in PDL is normally performed by the `PDL::Graphics::PGPLOT` module. The `PDL::Graphics::PGPLOT` package must be `use`'ed to give access to the commands. This introduction will be based on interactivity and use of `perldl` (see also The Whirlwind tour in chapter 1):

```
perldl> use PDL::Graphics::PGPLOT;
```

That is what you need to get running. We will now play around with a couple of commands before we turn to a systematic overview in the next two sections. We will concentrate on the `line` and `points` commands which draws continuous lines and individual plotting symbols respectively. The final result should look similar to Figure 4.1.

The first step is to start `perldl` and use the `PDL::Graphics::PGPLOT` package (some output is suppressed)

[1] PGPLOT can be obtained from `http://www.astro.caltech.edu/~tjp/pgplot/`

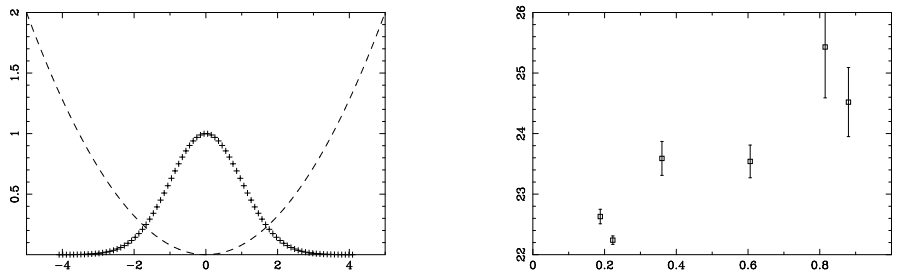


Figure 4.1: Our first try at using PDL::Graphics::PGPLOT.

```
> perldl
Type 'help' for online help
Type 'demo' for online demos
Loaded PDL v2.1
perldl> use PDL::Graphics::PGPLOT
```

Now we need to open a graphics device — there are quite a few that are supported by PGPLOT, here we will use a normal plot window that can be re-used

```
perldl> dev('/xs')
```

You should now have a large plot window on your screen, if you had some problems try to do `dev('?')` which will give you a list of available devices and allow you to choose one.

We first need to define a variable to have something to plot. The first plan is to simply plot a parabola and a Gaussian (bell) function as in the left panel in Figure 4.1, so we need an x-variable that is both positive and negative.

```
perldl> $x=zeroes(100)->xlinvals(-5, 5)
```

This creates a 100 element piddle starting at -5 and ending at 5. We can then very easily draw a parabola

```
perldl> line $x, $x*$x/12.5, {LINESTYLE=>'DASHED', Colour=>red}
```

which should draw a nice parabola with a dashed red line. As should be clear the `line` command draws a line and takes the x and y coordinates of the points on the line as arguments and options to the command are given as an anonymous hash.

We now want to plot a Gaussian on top of this, but if we were just to issue another plot command it would by default erase the screen, so instead we call the `hold` function to stop that from happening

```
perldl> hold
```

We can then continue plotting, now using symbols instead of a line

```
perldl> points $x, exp(-$x*$x/2),  {Symbol => 'Plus'};
```

Again, note that the function `points` function plots symbols instead of lines. PGPLOT has a large array of symbols, normally accessed using numbers, but the most common have text aliases defined.

The only thing left for us now is to ensure that the next plot will start afresh. Since we issued the `hold` command all subsequent plots will overplot the existing ones and since we do not want that anymore, we therefore have to `release` the device to the next set of plot commands:

```
perldl> release;
```

As a second example we will show how you can create plots with error bars. We will just carry on so the previous plot will be erased (enjoy it while you can). We first have to define some variables for the plot, so we need the $x\&y$ variables and the error on y.

```
perldl> $x = pdl(0.88, 0.223, 0.815, 0.606, 0.188, 0.360)
perldl> $y = pdl(24.52, 22.24, 25.43, 23.54, 22.63, 23.59)
perldl> $dy = pdl(0.57, 0.07, 0.84, 0.27, 0.12, 0.28)
```

In the previous example we let PGPLOT decide on the plotting ranges we were going to use, but now we want some more control over it. To do so we set it up using the `env` command

```
perldl> env(0, 1, 22, 26)
```

which sets the X-axis to go from 0 to 1 and the Y-axis from 22 to 26.

That is really all that is needed before plotting the error bars:

```
perldl> errb $x, $v, $dv, Symbol => 'Square';
```

Voilá. It almost looks like science. Of course in real life error-bars might not be symmetric (although you often wish they were), and we will explain how to do this later when we discuss `errb` in more detail below.

4.4 An overview of 2D plotting commands

Before we proceed to an overview of all commands in `PDL::Graphics::PGPLOT` it is necessary to define a couple of terms: The first is the concept of **device** — this is what the plotting commands work on, often this will be a *screen* device which shows resulting output on the screen in a window, but it can also be output to a file in some sort of format. Then inside each device there is a *plotting area* within which plotting commands gives a noticeable result.

Another important concept is *holding* of plots. When a plot is held, any subsequent plot commands will plot on top of the existing plot. To explicitly hold a plot you issue the command `hold` and to release it again you use `release`.

Finally most commands described in the following take a set of *options*. These are values that can be set to modify the default behaviour of the plotting routine and are very useful so we will first discuss the standard options and how options are specified.

4.4.1 Options in plot commands

As mentioned above and seen in the brief introduction to the PGPLOT interface earlier, we use options to modify the behaviour of plot commands. Below we will often see examples of *specific* options, those that are only recognised by a particular plot command. However in addition there are *general* options that are recognised by many or all plot commands. These are normally the options you use most so it is important to know these.

But first, how do you specify an option? If you read through the walk-through above you have probably already realised that they are set as keys in a hash:

```
line $x, $y, {Colour => 3}
```

However due to the way they are implemented in the code (using the `PDL::Options` package) the hash is more flexible than normal Perl hashes. Firstly the options are case-insensitive and secondly some have synonyms defined so that for instance `Color` and `Colour` are both accepted to avoid bad feelings on one side of the Atlantic. Finally most, if not all, options can be shortened so that `Lines` will be interpreted as `LineStyle`. This is mostly useful when working on the `perldl` command line however as it is error-prone in scripts (imagine that someone later implemented a `Lines` option which did something totally different, like draw 10 parallell lines, yeah, quite likely).

The following listing of standard options is based on the on-line documentation which you can access yourself inside `perldl` as

```
perldl> help PDL::Graphics::PGPLOT::Window
```

or on a command line using the `pdldoc` command

```
odin> pldoc PDL::Graphics::PGPLOT::Window
```

It is not envisaged that the standard option set will be significantly expanded from that listed here, but the on-line documentation should reflect any changes if they take place.

`Arrow` This options allows you to set the arrow shape, and optionally size for arrows for the vect routine. The arrow shape is specified as a hash with the key `FS` to set fill style, `Angle` to set the opening angle of the arrow head, `Vent` to set how much of the arrow head is cut out and `Size` to set the arrowsize.

The following

```
perldl>  $opt = {Arrow => {FS=>1, Angle=>60, Vent=>0.3, Size=>5}};
```

will set up an options hash for a broad arrow of five times the normal size.

Alternatively the arrow can be specified as a set of numbers corresponding to an extention to the syntax for the PGPLOT command `pgsah`. The equivalent to the above is

```
perldl> $opt = {Arrow => pdl([1, 60, 0.3, 5])};
```

For the latter the arguments must be in the given order, and if any are not given the default values of 1, 45, 0.3 and 1.0 respectively will be used.

`Arrowsize` The arrowsize can be specified separately using this option to the options hash. It is useful if an arrowstyle has been set up and one wants to plot the same arrow with several sizes. Please note that it is **not** possible to set arrowsize and character size in the same call to a plotting function. This should not be a problem in most cases.

```
perldl> $opt = {ARROWSIZE => 2.5};
```

`Axis` Set the axis type (see the `env` command below in section 4.4.4) It can either be specified as a number, or by a name as in the following table

Name	Numerical	Explanation
`Empty`	-2	draw no box, axes or labels
`Box`	-1	draw box only
`Normal`	0	draw box and label it with coordinates
`Axes`	1	same as `Normal`, but also draw ($X = 0$,$Y = 0$) axes
`Grid`	2	same as `Axes`, but also draw grid lines
`LogX`	10	draw box and label X-axis logarithmically
`LogY`	20	draw box and label Y-axis logarithmically
`LogXY`	30	draw box and label both axes logarithmically

The reason why this command is accepted by most commands is that when a command is called before a plot area is set up it will implicitly call `env` which interprets this option.

`AxisColour` Set the axis colour using the same syntax as for the `Colour` option below.

`Border` Normally the plot limits are chosen so that the plotted points just fit inside the plot area; with this option you can increase (or decrease) the limits by either a relative (ie a fraction of the original axis width) or an absolute amount. Either specify a hash array, where the keys are `Type` (set to '`Relative`' or '`Absolute`') and `Value` (the amount to change the limits by), or set to 1, which is equivalent to

```
Border => { Type => 'Rel', Value => 0.05 }
```

`Charsize` Set the character/symbol size as a multiple of the standard size.

```
$opt = {Charsize => 1.5}
```

`Colour` Set the colour to be used for the subsequent plotting — it has `color` as a synonym.. This can be specified as a number, and the most used colours can also be specified with name, according to the following table:

0	White	1	Black	2	Red	3	Green	4	Blue
5	Cyan	6	Magenta	7	Yellow	8	Orange	14	Darkgray
16	Lightgray								

However there is a much more flexible mechanism to deal with colour. The colour can be set as a 3 or 4 element anonymous array (or piddle) which gives the RGB colours. If the array has four elements the first element is taken to be the colour index to change. For normal work you might

want to simply use a 3 element array with R, G and B values and let the package deal with the details. The R,G and B values go from 0 to 1.

In addition the package will also try to interpret non-recognised colour names using the default X11 lookup table, normally using the `rgb.txt` that came with PGPLOT.

For more details on the handling of colour it is best that the user consults the PGPLOT documentation. Further details on the handling of colour can be found in the documentation for the internal routine `_set_colour`.

`Filltype` Set the fill type to be used by `poly`, `circle`, `ellipse` and `rectangle`. The fill can either be specified using numbers or name, according to the following table, where the recognised name is shown in capitals - it is case-insensitive, but the whole name must be specified.

1	Solid	2	Outline	3	Hatched	4	CrossHatched

```
$opt = {Filltype => 'Solid'};
```

(see below for an example of hatched fill)

`Font` Set the character font. This can either be specified as a number following the PGPLOT numbering or name as follows (name in capitals):

1	Normal	2	Roman	3	Italic	4	Script

Note that in a string, the font can be changed using the escape sequences `\fn`, `\fr`, `\fi` and `\fs` respectively. See the documentation in section 4.4.9 for more information regarding escape sequences.

```
$opt = {Font => 'Roman'};
```

gives the same result as

```
$opt = {Font => 2};
```

`Hatching` Set the hatching to be used if either fillstyle 3 or 4 is selected (see above) The specification is similar to the one for specifying arrows. The arguments for the hatching is either given using a hash with the key `Angle` to set the angle that the hatch lines will make with the horizontal, `Separation` to set the spacing of the hatch lines in units of 1% of $min(\text{height}, \text{width})$ of the view surface, and `Phase` to set the offset the hatching. Alternatively this can be specified as a 1x3 piddle `$hatch=pdl[$angle, $sep, $phase]`.

```
$opt = {Filltype => 'Hatched',
        Hatching => {Angle=>30, Separation=>4}};
```

Can also be specified as

```
$opt = {Fill=> 'Hatched', Hatch => pdl [30,4,0.0]};
```

For another example of hatching, see the command `poly` in section 4.4.5 below.

`Justify` A boolean value which, if true, causes both axes to drawn to the same scale. If you want more information about this option you are advised to consule the PGPLOT documenation for the `pgenv` command.

`Linestyle` Set the line style. This can either be specified as a number following the PGPLOT numbering or as a name as shown in the following table.

1	Solid	2	Dashed
3	Dot-Dash	4	Dotted
5	Dash-Dot-Dot		

Thus the following two specifications both specify the line to be dotted:

```
$opt = {Linestyle => 4};
$varopt = {Linestyle => 'Dotted'};
```

The names are not case sensitive, but the full name is required.

`Linewidth` Set the line width. It is specified as a integer multiple of 0.13 mm.

```
$opt = {Linewidth => 10}; # A rather fat line
```

`PlotPosition` The position of the plot on the page relative to the view surface in normalised coordinates as an anonymous array. The array should contain the lower and upper X-limits and then the lower and upper Y-limits. To place two plots above each other with no space between them you could do

```
$win->env(0, 1, 0, 1, {PlotPosition => [0.1, 0.5, 0.1, 0.5]});
$win->env(5, 9, 0, 8, {PlotPosition => [0.1, 0.5, 0.5, 0.9]});
```

`Symbol` The plot symbol to use, with the default being 17 which gives a small filled circle. This is an option for `points` and `errb` at the moment, but could be used for others too. It is either given a piddle with the same number of elements as the plot variable, a name (or number) specifying the symbol to use according to the following (recognised name in capital letters):

0	Square	1	Dot	2	Plus	3	Asterisk
4	Circle	5	Cross	7	Triangle	8	Earth
9	Sun	11	Diamond	12	Star	17	Default

PGPLOT has support for a much larger number of symbols. The reader is advised to consult the PGPLOT documentation for further information or write a short program that loops through all symbols. Note however that there are a *lot*. For instance symbol 2830 is a cyrillic character — the system used is the Hershey system for symbols. In addition you draw regular polygons with n-sides by setting the symbol to $-n$ although -1 and -2 draws a dot with the diameter set to the current linewidth.

`Title` The title on top of the plot box.

`XTitle` The title for the X-axis of the plot.

`YTitle` The title along the Y-axis.

4.4.2 Hard-copies and plot options

The default options for screen display are not ideal for hard-copies (typically PostScript). Thus there is a separate set of options for certain properties when the output device is a hard-copy one. Here we will quickly summarize these

`HardLW` The line width used on hard-copy devices. The default is 4.

`HardCH` The character size used on hard-copy devices. The default is 1.4.

`HardFont` The default font used on hard-copy devices. It defaults to 2.

`HardAxisColour` The default colour to draw the axis with on a hard-copy device. This is particularly important since light green (default screen colour) is not very visible on paper. The default is 1 (black). The setting of colours work as with `Colour`

`HardColour` The default plot colour on hard-copy devices, it defaults to 1 (black).

These options should be set either in the call to `dev` (see section 4.4.4 below) or redefined using the method outlined in the next section.

4.4.3 Setting default values for options

You might not be happy with the default settings for the various options and want to set a different value permanently instead of specifying it with every call to `dev`, `env` or some other command. There is some support for this, but it is limited in that it is not case-insensitive nor does it have synonyms (except for colour/color) so the options *must* be written as above. (You will be notified if you did something wrong).

That said it is fairly easy to use. You would normally set this in your `.perldlrc` file (see XXXX). The relevant function is `set_pgplot_options` which takes a hash as argument with the options and their values, as in the following example:

```
use PDL::Graphics::PGPLOTOptions ('set_pgplot_options');
set_pgplot_options('Device' => '/xs', 'LineWidth' => 10);
```

Note that some settings might affect more than you like. In particular the `LineWidth` and `LineStyle` options will also affect the axis and axis labels drawn. Unless you are after the blurred, completely unreadable look that is not what you want. Trust me. However character size, device default plot symbol, border and other options can be conveniently be specified in this way.

4.4.4 Setting up the plot area

The first step for the budding plot maker is to set up the drawing area. This involves selecting what device you want to create the plots on and then setting the region you want to plot in[2].

The destination for your plot commands is set with the `dev` command, and with different arguments to `dev` you can send plots to various output devices such as

GIF files — `dev('giffile.gif/gif')`

Postscript files — `dev('filename.ps/ps')`

X-windows plotting windows — `dev('/xs')`

If you wish to have several plotting panels per page you can specify the number in the x and y directions as further arguments to `dev` so that to get four panels you would write `dev('/xs', 2, 2)`.

For more detailed control over the created device, you can specify various options. The main four options you might use are

`Aspect` The aspect ratio of a newly created output device. If your device is a graphics window under a window system, this might or might not be

[2]Note that it is actually not necessary to do this manually as PDL::Graphics::PGPLOT will do this automatically for you as in the previous section, but very often you would want to modify this.

applied when the window is created, but it should be updated as soon as you plot to it. The default value is 0.618, ie. the golden ratio.

`WindowWidth` The width of the created output window. The width is specified in units of inches, which is reasonably easy to deal with when printing out, but if your device is a graphics window it is all a bit more unclear since different setups might have different ideas of what an inch corresponds to in pixels.

`WindowXSize` The X-size of the plot window, specified as `WindowWidth` and combined with `Aspect` if `WindowYSize` is not set.

`WindowYSize` As above but for the Y-size.

`NX` **and** `NY` These two options set the number of panels in the X and Y direction respectively and are alternatives to specifying the numbers of panels directly in the call to `dev` as `dev(<device>, <nx>, <ny>)`.

The options are specified in an anonymous hash so that

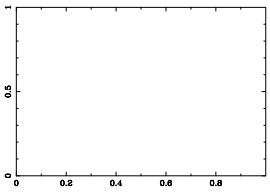
```
perldl> dev('/xs', {NX => 4, NY => 2})
```

will create a plot window with four panels in the X-direction and 2 in the Y-direction, with a default aspect ration and size. Alternatively the same window could have a specified width and aspect ratio by specifying those options as

```
perldl> dev('/xs', {NX => 4, NY => 2, Aspect => 1, WindowWidth => 5})
```

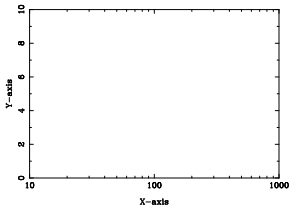
However `dev` does not actually draw anything for you, it merely selects the output device. To set up a plot you either call a plot command directly, or if you want more control over the axis ranges you use the command `env`. This useful command takes the upper and lower limits in X and Y as input

```
env(0, 1, 0, 1);
```



sets up a plotting area with both axes going from 0 to 1. If a logarithmic axis is desired this can be achieved by passing an option to the `env` command, we can also use this to set the axis labels

```
env(1, 1000, 0, 1, {Axis => 'LOGX',
  Xtitle => 'X-axis', Ytitle => 'Y-axis'})
```



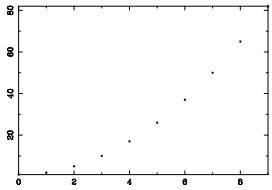
further info on the `Axis` option can be found in section 4.4.1.

It is important to realise that when you call `env` explicitly it automatically holds the plot for you, so subsequent plot commands will plot on top of the plotting area, and if you want to make a new plot you need either to call `env` again or call `release` explicitly.

4.4.5 Drawing lines and plotting points

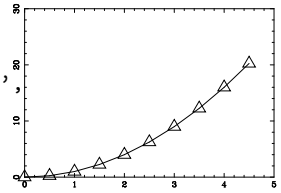
The most important commands in the graphics package are probably the line drawing and point plotting commands `line` and `points`. The most basic command is `points` which plots particular symbols at given x&y values:

```
perldl> $x = sequence(10)
perldl> $y = $x*$x + 1
perldl> points $x, $y
```



The action of the `points` command can be modified by adding options. The most important is `Symbol` which changes the plot symbol and `Charsize` which changes the size of plot symbols; in addition the `Plotline` option is a toggle which if set causes a line to be drawn through the plots:

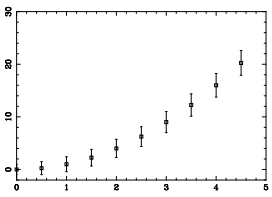
```
perldl> points $x, $y, {Symbol => 'Triangle',
   Plotline => 1, Charsize => 5}
```



The string `Triangle` is equivalent to symbol number 7 and in general symbols will have to be accessed using the numerical system, but there are textual equivalents for many commonly used symbols (see section 4.4.1). The `points` command does also accept a piddle as the symbol value, in which case it should have the same length as $x and $y and each point will be plotted with the corresponding symbol value.

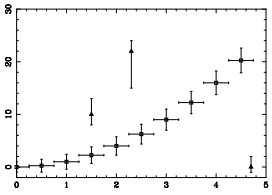
Plotting error-bars Closely related to `points` is the routine for plotting symbols with error-bars, `errb`. This can be called in a variety of ways to allow for various ways of giving errorbars and whether horisontal or vertical errorbars are required. A typical call is

```
perldl> env(0, 5, -2, 30)
perldl> $x=sequence(10)/2.0; $y=$x*$x
perldl> $dy = sqrt($x+1);
perldl> errb $x, $y, $dy,
        {Symbol => 'Square'}
```



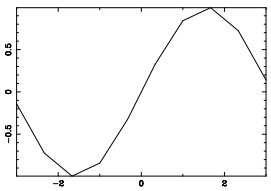
which plots squares with symmetrical vertical error-bars. To get error bars in the horisontal direction one give these before the y-errors. Likewise it is possible to get asymmetric error-bars by giving the upper and lower limits of the error bars separately for the X and Y variables as in the following example

```
perldl> $x2 = pdl(1.5, 2.3, 4.7)
perldl> $y2 = pdl(10, 22, 0)
perldl> $dx = $x2->zeroes(); # No X-errors
perldl> $y_u= pdl(12,29,1)-$y2
perldl> $y_l= $y2 - pdl(7, 20, -2)
perldl> errb $x2, $y2, $dx, $dx,
       $y_l, $y_u, {Symbol => 'Triangle'}
```



Drawing lines We saw above that we could draw lines between points by setting the `PlotLine` option to `points`, however there are much better ways to draw lines. The basic line-drawing command is `line` which draws a straight line between each point.

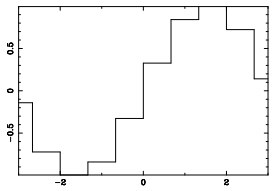
```
perldl> $x = zeroes(10)->xlinvals(-3, 3)
perldl> line $x, sin($x)
```



The style, width and colour of the line can be changed with the options `LineStyle`, `LineWidth` and `Colour`/`Color` respectively as outlined in section 4.4.1 above.

Plotting histograms A very similar command is `bin` which is useful for plotting histograms. This command draws horisontal lines between x_i and x_{i+1} with the value y_i.

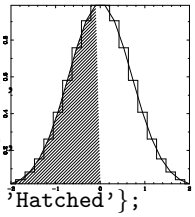
```
perldl> $x = zeroes(10)->xlinvals(-3, 3)
perldl> bin $x, sin($x)
```



By default the routine assumes that the X-values are the start points of the bin, if instead your values are for the centers of the bins, you need to set the option `Centre/Center` to a true value. In addition the appearance of the lines can be modified using the same options as for the `line` command.

Drawing polygons Finally the `poly` command is like `line` but fills the polygon defined by $x and $y with the chosen fillstyle (defaults to solid fill). If you display this you should consider putting `FillStyle => 'Outline'` in your `.perldlrc` file as explained in section 4.4.3, or you can set it explicitly as in the following example:

```
perldl> $x=zeroes(20)->xlinvals(-2,2);
perldl> $y=exp(-$x*$x);
perldl> $xpoly = append($x->where($x <= 0),
              pdl(0));
perldl> $ypoly = append($y->where($x <= 0),
              pdl(0));
perldl> poly $xpoly, $ypoly, {FillType => 'Hatched'};
```

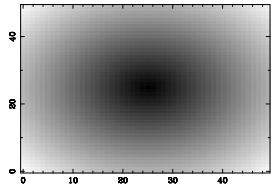


In this example it is worth noting the added complications to ensure that the polygon is closed. In addition we have used the option `FillType` to change the style of fill used. This can be finely adjusted if necessary, for further examples see the PDL::Graphics::PGPLOT help file and the discussion of `FillType` in section 4.4.1.

4.4.6 Displaying images

PGPLOT was originally designed for astronomy and as such it has good support for the display of 2D-data. In PDL this support has been simplified and there is now only one command for image display, `imag`, which internally chooses between different PGPLOT display commands. The simplest use of `imag` is to let it act on a 2D piddle so

```
perldl> $a = rvals(50,50,
        {Center => [ 25, 25]});
perldl> imag $a;
```



should produce a symmetric shape in your graphics window.

However, most likely you will find that the shape is not circularly symmetric because the aspect ratio of your graphics window is different from 1. How then can we correct this? The easiest solution is probably to make sure that your graphics device has aspect ratio 1 by giving the `Aspect` option to the `dev` command (see 4.4.4).

That isn't always an option though, and an alternative approach is to use the option `Pix` to the `imag` command. This lets you adjust the aspect ratio of the image pixels. You can in addition specify the number of image pixels per screen unit with the option `Pitch` so that to display the previous image with square pixels and 2 image pixels per screen pixel you use

```
perldl> imag $a, { Pix => 1,  Pitch => 2 }
```

You can also use `Unit` to specify the unit used for scaling and `Scale` for the reciprocal of `Pitch`, see the PDL::Graphics::PGPLOT documentation for details. The `Pix` option does only adjust the coordinate ranges and this might not always be what you require. In such situations a solution might be to create a square plot window directly as mentioned earlier.

In addition you might want to specify a stretch of the gray-scale of the image. This can be obtained first by specifying the max and min values of the displayed image (everything above is set to the max value and everything below to the min value). This is set with the `Min` and `Max` options. Additionally it is possible to adjust the image transfer function using the option `ITF`. Allowed values are `Linear`, `Log` and `Sqrt`.

You can also add a colour bar (colour wedge in PGPLOT parlour) to the image display. This is accomplished either using the `draw_wedge` (see below) command directly or by setting the `DrawWedge` option to true in your call to `imag`. If you want to pass options to the `draw_wedge` command, you can do that with the `Wedge` option. See below for further details.

Transforms Finally a very useful feature of PGPLOT that is relevant both to images and also the contour plots (see below) is the concept of a transform "matrix". This is a 6 element vector, T_i which maps input pixels into display

pixels so that pixel i, j is mapped to

$$X_{ij} = T_0 + T_1 i + T_2 j \tag{4.1}$$
$$Y_{ij} = T_3 + T_4 i + T_5 j. \tag{4.2}$$

It is always simplest to refer to this equation the first few times one sets up a transform vector.You use this whenever your pixel positions in the real world were different from that represented by your input image array.

We have put all this together in Example 4.4.6. Here we show the use of the `ITF` and and `Transform` options. Note that using `Transform` in conjunction with `Pix` is going to lead to unwanted results!

Colour bar/wedge It is often desireable to annotate an image with a colour wedge showing the range of values in the image. This is accomplished with the `draw_wedge` function in PDL::Graphics::PGPLOT (but you can avoid calling this directly by setting the `DrawWedge` option in your call to `imag`, see above). This function should normally give a decent result without the user setting any options except the `Label` option which sets the annotation, but occasionally it is necessary to change its behaviour and that is done by setting the following options:

`Side` What side the wedge will appear on, the default is the right side and it is specified as a single character, 'B' for bottom, 'L', 'T' and 'R' for left, top and right respectively.

`Displacement` The distance away from the axis. Default=2.

`Width` The width of the wedge. Default=3

`Foreground` The value to set the foreground colour to. This can be referred to as `Fg` as well. The default is the max value used by `imag` when drawing the image.

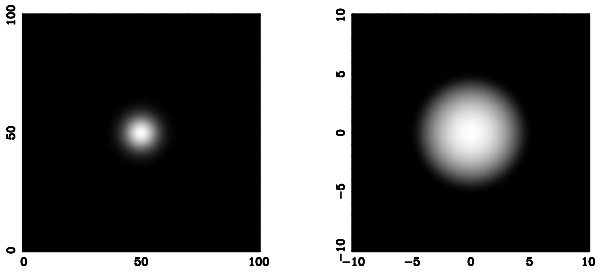
`Background` The value to set the background colour to. This can be referred to as `Bg` as well. The default is the min value used by `imag` when drawing the image.

`Label` The label used to annotate the wedge.

Note that you will sometimes need to directly set the plot size to avoid clipping in the display. A full example that shows the use of `draw_wedge` can be seen in Figure 4.3 where we display a galaxy and display a look-up table next too it.

4.4.7 Contour plots and vector fields

Contour plots are very similar to image displays and display lines at particular levels of the image. The function to create contour plots is `cont` which at the

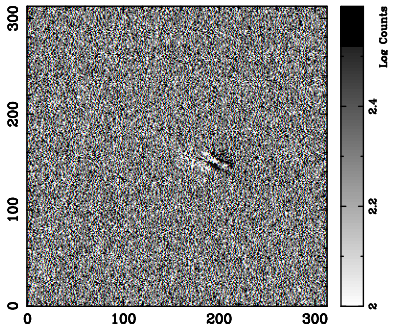



```
use PDL;
use PDL::Graphics::PGPLOT;

# Create two plot areas in the X-directions
dev('/xs', 2, 1);
# Create a Gaussian around the center of the image
$a=rvals(101, 101, {Center => [50, 50]});
$y = exp(-$a*$a/50.);

# Display with a linear transfer function
imag $y;
# This transform vector maps the extreme points to ± 10
my $tr = pdl(-10, 1.0/5.0, 0, -10, 0, 1.0/5.0);
# Finally display the image with the transform and
# a logarithmic transfer function.
imag $y, {Transform => $tr, ITF => 'Log'};
```

Figure 4.2: Contrasting two different ways of displaying the same image. On the left is the default display of a Gaussian, whereas on this right is the result when mapping the pixels to a range from -10 to 10 with a logarithmic transfer function.

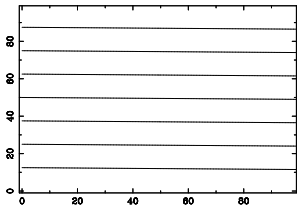



```
dev '/xs', {WindowWidth => 6, Aspect => 1};
$im = rfits('Frei/n4013_lJ.fits');
$im += abs(min($im)-1);
$im = log10($im);
imag($im, {PlotPosition => [0.1, 0.85, 0.175, 0.925],
            Min => 2.6, Max => 2.0 });
draw_wedge({Wedge => {Width => 4, Label => 'Log Counts',
                      Displacement => 1}});
```

Figure 4.3: An example of the use of the `draw_wedge` command.

simplest level only takes a 2D array as its argument.

```
$a = sequence(100,100); cont $a;
```



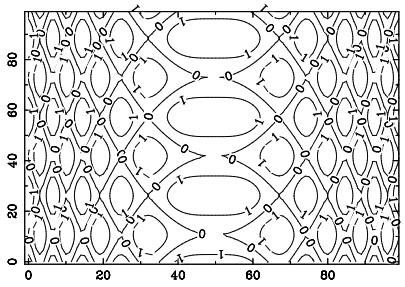
That might be all you need, but most likely you would like to specify contour levels, label contours and maybe draw them in different colours.

You use the option `Contours` to give the wanted contour levels as a piddle and `Labels` to give an anonymous array of strings for labels as shown in the example below

```
use PDL;
use PDL::Graphics::PGPLOT;

dev('/xs');
$y = ylinvals(zeroes(100,100), -5, 5);
$x = xlinvals(zeroes(100,100), -5, 5);
$z = cos($x**2)+sin($y*2);

cont $z, {Contours => pdl(-1, 0, 1),
          Labels => ['-1', '0', '1']};
```



In addition it is possible to colour the labels differently from the contour lines (`LabelColour`), to specify the number of contours instead of their values (`NContours`) and to draw negative contours as dashed lines and positive as solid lines by setting the option `Follow` to a value > 0.

Overlaying a contour plot on top of an image is as easy as displaying the image, call `hold` and display the contour plot. The reader might want to try a colour version of the example above ($z as in the example):

```
perldl> ctab('Fire');
perldl> imag $z;  hold;
perldl> cont $z, {Contours => pdl(-1,0,1)};
```

The final 2D plot command we will deal with here is the command for plotting a vector field, `vect`. This command takes two arrays as arguments. The first gives the horisontal component and the second the vertical component of the vector field. The length of the vectors can be set using the `SCALE` option and the position relative to the pixel centers with the option `POS`.

What is important to note with a command like `vect` is that you can use the

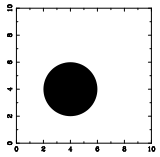
`Transform` option to map a smaller vector array to a larger image. This is often useful because a vector field with 256×256 arrows on top of a similarly sized image will quickly be unreadable. The result of using this technique is shown in Figure 4.4 together with the code that produced the plot.

4.4.8 Drawing simple shapes

In addition to the simple commands described above, there are a few convenient commands for drawing simple shapes such as circles, ellipses and rectangles. These are fairly straightforward commands with similar options and invocations so I will go through them fairly quickly. A common issue with these commands as with the `poly` command is that they draw filled shapes, if you want outlined shapes to be drawn you have to set the `Filltype` option to `Outline`.

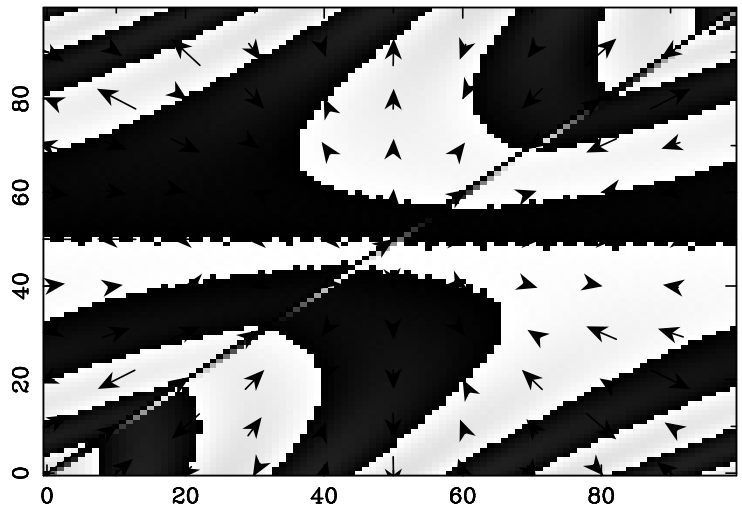
The circle command is probably the simplest, it draws a circle (which may or may not look like a circle depending on the aspect ratio of your display — see section 4.4.4 above). The user specifies the radius and the X&Y position of the centre

```
perldl> dev '/xs', {Aspect => 1, WindowWidth => 5}
perldl> env 0, 10, 0, 10
perldl> $radius=2; ($x, $y) = (4, 4)
perldl> circle $x, $y, $radius, {LineWidth => 3}
```



The radius and position of the centre can also be specified using the `Radius`, `XCenter` and `YCenter` options instead of as arguments — use whichever seems more natural.

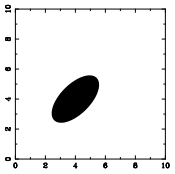
The `ellipse` function is like the `circle` function but it requires the user to specify the minor and major axis and the angle between the major axis and the horisontal. For ease of use it is probably better to specify these as options, but if you remember the order you can also give them directly as arguments to the function (x-position, y-position, major axis, minor axis, angle)




```
perldl> $x = xlinvals(zeroes(100,100), -5, 5)
perldl> $y = ylinvals(zeroes(100,100), -5, 5)
perldl> $z = sin($x*$y/2)
perldl> imag $z; hold;
Show the partial derivatives wrt. x & y as vectors
perldl> $xcomp = $x*cos($x*$y/2)/2
perldl> $ycomp = $y*cos($x*$y/2)/2
We want to show only every sixth vector for clarity
perldl> $s = '0:-1:10,0:-1:10';
Finally we need to map the final 10x10 array to the 100x100 image
perldl> $tr = pdl(0,10,0,0,0,10)
perldl> vect $xcomp->slice($s), $ycomp->slice($s), {Transform=>$tr}
```

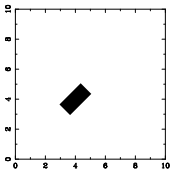
Figure 4.4: An example of the `vect` command

```
perldl> dev '/xs', {Aspect => 1, WindowWidth => 5}
perldl> env 0, 10, 0, 10
perldl> ellipse 4, 4, {MajorAxis => 2,
        MinorAxis => 1, Theta => atan2(1,1)}
```



And finally the `rectangle` command draws rectangles where you can give the position of the centre, the length of the sides and the angle with the horisontal. The operation is very similar to the `ellipse` command with the length of the sides of the rectangle taking place of the major and minor axis.

```
perldl> dev '/xs', {Aspect => 1, WindowWidth => 5}
perldl> env 0, 10, 0, 10
perldl> rectangle 4, 4, {XSide => 2,
        YSide => 1, Angle => atan2(1,1)}
```



Note that `Angle` and `Theta` are synonyms.

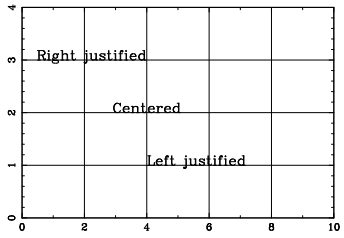
In addition you can set the sides to be similar by setting the `Side` option to the length you require. The lengths are all specified in data-coordinates (which is why you should do a plot or call `env` before using any of these commands).

For other shapes or when these are not sufficiently flexible you should use the `poly` command which is called by both `rectangle` and `ellipse`.

4.4.9 Text and legends

The main command for drawing text on the plotting surface is the `text` command which at its basic level just draws a string from the given x&y position:

```
perldl> dev '/xs'
perldl> env 0,10,0,10, {Axis => 'GRID'}
perldl> text 'Left justified', 4, 1
perldl> text 'Centered', 4, 2,
        {Justification => 0.5}
perldl> text 'Right justfied', 4, 3,
        {Justification => 1.0}
```



Here I have included grid-lines to show the effect of the different justifications. Note that `Justify` is a synonym for `Justification`, and that you need to give numerical values for the position. Normally the text background is transparent as shown here, but you can also set an opaque background by setting the `BackgroundColour` option to a colour name or value (see also the next section).

In addition to the justification option one can also change the angle of the text using the `Angle` option and specify the text and/or x&y as options (the best advice is to either do all or none).

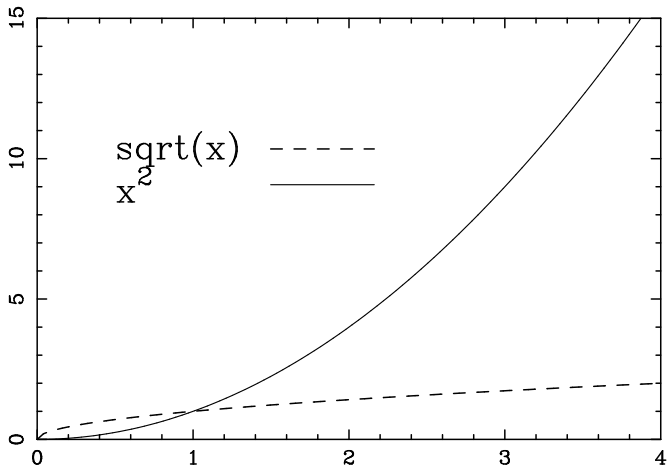
```
perldl> text {XPos => 1, YPos=> 4, Angle => 25, Text => 'Tilted'}
```

Non-alphanumeric symbols PGPLOT has extensive support for non-alphanumeric characters in text strings and also offers reasonable control over the display of superscripts, subscripts etc. This is all achieved using *escape sequences*. In PGPLOT these are all signaled by the character `\`. Thus `\u` starts a superscript or ends a subscript — it signals a shift "up". Likewise `\d` starts a subscript or ends a superscript. See the example for `legend` in Figure 4.5 for an example. Here is an incomplete list of the most useful ones, but there are many more. Consult the PGPLOT documentation for a full list.

`\u`	Go up	`\d`	Go down
`\x`	$\times$	`\.`	$\cdot$
`\A`	Å	`\ga`	α
`\gb`	β	`\gg`	γ
`\m`n	Graph marker n	`\(nnnn)`	Hershey character `nnnn`
`\(0562)`	$\mathcal{L}$	`\(0683)`	∂
`\(2281)`	$\odot$	`\(0766)`	∞
`\(2233)`	$\pm$	`\(2266)`	∇

The only additional text-related function in the PDL::Graphics::PGPLOT interface is the `legend` command which draws a legend in the plot window. This is a more complex routine which can be a time-saver as soon as you have learned how to use it. It takes the same arguments as the `text` command with the exception that the text argument is an anonymous array of labels for the legend, and that a fourth argument is accepted which specifies the width of the box

in which the legend will be drawn. If this is not set or it is set to the string `Automatic` it will be adjusted to contain the legend with the default font-size (or that set by the user via the `CharSize` option).




```
perldl> $x = sequence(10); $y1 = sqrt($x); $y2 = $x**2;
perldl> line $x, $y1, {LineStyle => 'Dashed', Colour => 'Red'}
perldl> line $x, $y2, {LineWidth => 3, Colour => 'Blue'}
perldl> legend ['sqrt(x)', 'x\u2'], 0.5, 10,
        {LineStyle => ['Dashed', undef],
          LineWidth => [undef, 3],
          Colour => ['Red', 'Blue'], Width => 1.0}
```

Figure 4.5: An example of the legend command

The idea of the `legend` command is that you give the line-styles, line-widths, colours or symbols you want to illustrate as anonymous arrays to the `LineStyle`, `LineWidth`, `Colour` and `Symbol` options. Not very clear? Well, maybe an example will help.

Figure 4.5 shows an example of `legend` in use. Two lines are drawn, a red dashed line and a blue thick line. To annotate this plot using `legend` you give the text annotations as an (anonymous) array of strings, the X&Y position of the legend box and an anonymous hash containing information about the legends to draw as shown in the example. The options used to specify a particular draw

style are the same as the ones used in the call to `line` and will undergo the same translations — note however that you can specify a value of `undef` which requests that the current default for the linestyle/linewidth/colour etc. is used. The `Width` option is used to set the width of the legend box and is given in data coordinates. The idea is that you will create the plot, see where you want the legends to go and then set the X&Y and width to the appropriate settings and redoing the plot, possibly using the replay mechanism (see section 4.7).

The legend command has several options the main of which are illustrated in Figure 4.5. The remaining options are useful for tweaking the appearance, and a full list is as follows

Text The text, this is an alternative to specifying it as the first argument to the function.

XPos The X-position of the text, again as an alternative to specifying it as the second argument.

YPos The Y-position of the text, again as an alternative to specifying it as the third argument.

Width The width of the (invisible) box the legend is drawn inside. This can also be specified as the fourth argument to the `legend` command. If this is set to the string `Automatic` the width is calculated from the character size used.

Height This can be used as an alternative constraint on size, giving the height of the legend box. If both `Width` and `Height` are specified the smallest size is used (characters are not compressed or stretched to fit).

TextFraction The fraction of the box set aside for text. The default is 0.5 which usually is ok. Note that this option used to be called `Fraction`, which still is available as a synonym.

TextShift This option allows for fine control of the spacing between the text and the start of the line/symbol. It is given in fractions of the total width of thelegend box. The default value is 0.1.

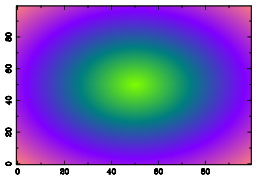
VertSpace By default the text lines are separated by one character height (in the sense that if the separation were 0 then they would lie on top of each other). The `VertSpace` option allows you to increase (or decrease) this gap in units of the character height; a value of 0.5 would add half a character height to the gap between lines, and -0.5 would remove the same distance. The default value is 0. This option has `VSpace` as a synonym (more natural for the TeX-heads out there).

4.5 Using colour

PGPLOT has a two disjoint sets of colours. One set determines the colour table used when displaying images and is initialised to a grayscale, and the other is a set of 15 colours used to colour all other plotting objects. The latter set is accessible through the `Colour` option described in section 4.4.1 above. Here we will concentrate on accessing the lookup-table for image display.

The command used to change the colour table is `ctab`, which in its generic form takes six arguments specifying the intensity levels, red, green and blue colour components, contrast and brightness levels. The contrast and brightness are optional so that we can say

```
perldl> $int = pdl([0, 0.33, 0.66, 1.0])
perldl> $r = pdl([0.5, 0, 0.5, 1])
perldl> $b = pdl([0.0, 0.5, 1.0, 0.5])
perldl> $g = pdl([1.0, 0.5, 0.0, 0.5])
perldl> ctab($int, $r, $g, $b);
perldl> $a = rvals(100, 100)
perldl> imag $a
```



which should display a circularly symmetric figure with green in the centre, going through blue to red-ish where $a is at a maximum.

It is however normally sufficient to use the colour tables made available by `PDL::Graphics::LUT`. This package makes available a large number of standard colour tables which can be accessed using the following commands

`lut_names` This returns a perl list of the available colour tables.

`lut_ramps` As above, but returns a list of the names of the available intensity ramps.

`lut_data` And finally the data in the tables can be accessed with this function which takes as arguments the name of the colour table, and optionally a scalar determining if the colour table is to be reversed and the name of an intensity ramp (default is a linear intensity ramp). The function returns four piddles with intensity and RGB values which can immediately be passed to `ctab`.

Note that these commands do not set the colour table for you, you will still need to call `ctab` to do that.

Thus to set one of the colour tables in the `PDL::Graphics::LUT` package, you do

```
  perldl> use PDL::Graphics::LUT;
  perldl> print "Available tables: ".join(', ', lut_names());
Available tables: aips0, backgr, bgyrw, blue, blulut, color, ....
```

```
<.. Output deleted ..>
  perldl> imag rvals(100,100);
  perldl> ctab(lut_data('rainbow1'));
```

which should give you a colour table that goes from black through green, blue and yellow to red.

4.6 Threading in PDL::Graphics::PGPLOT

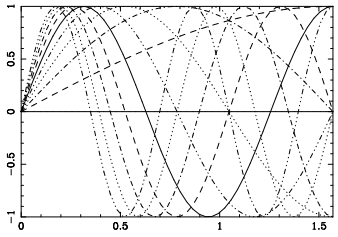
The plot commands do not always lend themselves to easy threading (see chapter 3 for a detailed discussion about threading) because it can sometimes be difficult to know what the user intends to do when (say) an array of images is passed to the `imag` command. Are they to be displayed in several plot panels, are they to be plotted on top of each other, seamlessly plotted next to each other? But even more complex is the question of treatment of options and how to deal with these if there are less options than for instance, lines to draw (a common occurence if you wanted to draw a *lot* of lines).

That said the PDL::Graphics::PGPLOT interface does have limited support for threading in the `line` and `points` functions. These call the `tline` and `tpoints` internally, and work just like `line` and `points` except that they expect the input y-piddle to be 2D, with each line in the array plotted against the x-piddle.

The way the options are treated is the most interesting. To set options for a set of lines, give an anonymous array as argument to that option with a value for each line. If you give more options than there are lines, the surplus is ignored. However if you give less, the options are repeated from the start. Although possibly a bit confusing this is very powerful because you can get a large number of combinations of colour and linestyle. For instance if you give 4 colours and 5 linestyles, you get a total of 20 distinct combinations and should you give 3 linewidths as well you will suddenly have 80 different styles to work with with very little typing. Note however that you need to make sure that the numbers you give are relatively prime — otherwise you will get much less possibility, just think of the situation where you have 4 linestyles and 4 colours, they will just loop in harmony and result in only 4 combinations.

Anyway, let us see how it all works in practice by creating a plot of sine curves with different frequencies. This is a simple example where we want to colour all even frequencies with red and all odd with blue and vary the line-styles as well.

```
perldl> $pi=4*atan2(1,1);
perldl> $x=zeroes(50)->xlinvals(0, $pi)
perldl> $freq = sequence(10)
perldl> $y = sin($freq*transpose($x))
perldl> line $x, $y,
       {Colour => ['Red', 'Blue'],
         Linestyle=>[0,1,2,3,4,5]}
```



Small problem: If no options are given, the plot is not properly plotted

4.7 Recording and playing back plot commands

Have you ever created a good-looking plot on the command line of an interactive data program, be it PDL, IDL, Matlab, Octave or any other package, and wished that you could make a quick Postscript copy of it only to find that you need to redo all the commands? I certainly have.

In the newer versions of PDL this is thankfully not the case anymore. These have a recording facility built in. However this is not enabled by default (for reasons described later in this section), you need to turn it on yourself. The way to do this is to set the `$PDL::Graphics::PGPLOT::RECORDING` variable to a true value.

```
perldl> $PDL::Graphics::PGPLOT::RECORDING = 1
```

You can turn this on automatically in the `perldl` shell if you put this command in your `/.perldlrc` file. Alternatively you can turn on recording for each plot device independently by setting the `Recording` option to true when starting a device:

```
perldl> dev '/xs', {Recording => 1}
```

Note that if you set the variable it must be set *after* you have `use`'d the PDL::Graphics::PGPLOT because this package sets the variable when it initialises to its default value of zero.

In the following I will focus my attention on using the recording and playback functions in the `perldl` shell as I envisage that it will be most useful there. There are a couple of potential uses in scripts as well which I will get back to below, but this is not well thought through yet.

Before we continue it should also be added that the recording facility is somewhat experimental. In particular it doesn't deal very well with multi-panel

plotting where you jump back and forth between panels. If you want to do that, make sure you specify the `Panel` option for every call.

It is very easy to use the recording facilities with a few less obvious aspects. An example should go a long way to get you to understand the basics. First we set up a simple plot using the commands we learned above:

```
perldl> use PDL::Graphics::PGPLOT
perldl> $PDL::Graphics::PGPLOT::RECORDING = 1
perldl> $x = sequence(10)
perldl> $y = random(10)
perldl> dev '/xs'
perldl> env(-1, 11, -0.5, 1.5, {Xtitle => 'Number'})
perldl> points $x, $y, {Symbol => 'Plus'}
```

which should give you a scatter plot on screen. Now after constructing this fantastic piece of scientific illumination you decided to make a Postscript version of it, but you are loathe to use the up key to execute the commands again so you decide to use the recording facilities.

```
perldl> $s = retrieve_state()
perldl> dev 'replay_ex.ps/ps'
perldl> replay $s
```

That is all. These commands should now have created a file called `replay_ex.ps` in the present directory.

The `retrieve_state` commands retrieves the current state of the plot device and returns a variable to hold this in. This state contains references to the data plotted and plot commands executed and can be replayed, or re-executed, at a later stage using the `replay` command. You can also turn on and off recording temporarily with the `turn_off_recording` and `turn_on_recording` commands.

This suffices for most situations and should work for any complexity of plot constructed. There are however a few rules that needs to be observed and possible pitfalls.

- If you turn on recording globally using `$PDL::Graphics::PGPLOT::RECORDING`, you must set the variable *before* opening a plot device because the value of the variable is only checked then. If you forget, you can of course always turn it on with the `turn_on_recording` function.
- The state is cleared whenever the plot window is erased, or if the user executes the `clear_state` command. In particular this occurs when you change plotting device (although if you use several windows they will each have their own state; see also the following section), so use the `retrieve_state` command *before* you change device!

- The state contains references to the data plotted. This does not use memory (at least not appreciably!), but it does mean that an extra reference to the data is kept and the memory to the data might not be freed when you expect it to. This can be problematic if you make a lot of image displays. The best ways to avoid this problem in the `perlld` shell is to call the `clear` on the state:

  ```
  perldl> $s->clear()
  ```

  or to re-use the variable next time you call `retrieve_state`. Note that this should only be a problem if you explicitly call `retrieve_state`.

- Finally since only references to the data are held, make sure you do not modify them before calling `replay` or you might end up with a rather different looking plot!

What we covered now is the basic use of the recording facility, which hopefully will come in handy rather often (which is why I recommend enabling it permanently in the `perldl` shell as outlined above). However there are slightly less common uses of the facility that might come in handy:

Redoing a plot with slightly different data The fact that the recording state contains references to the data enables a somewhat tricky but potentially very useful trick to be executed: Redoing the plot with adjusted data. Sometimes you make a complex plot only to discover that you had made an error with your data and you need to redo it. This is where you can use the recording functions: Retrieve the state, make adjustments to the data making sure not to break the link (see chapter XXXX) and run `replay`.

However, although this sounds quite easy it has a few subtleties that can give surprising results at times. It might therefore be a good idea to look at a few, very similar and very basic, examples and compare their effects. So let us first of all open a plot device:

```
perldl> dev '/xs', {Recording => 1}
```

NOTE: What I describe here is not well tested and is probably buggy. This needs to be sorted out before finishing — at least I have had a few weird results when tryin this out.

We are going to use our age-old example of plotting a parabola, and replaying it with various parameter sets. Let us therefore define a couple of variables and plot this, first letting PDL decide on the plot limits:

```
perldl> $x = sequence(10); $y = $x*$x
perldl> line $x, $y;
perldl> $s = retrieve_state()
```

The whole point of this problem is to change the variables, so let us add 3 to the x-values and replay the command:

```
perldl> $x += 3
perldl> replay $s
```

This should give you a part of a parabola from $x = 3$ to $x = 12$, but now defined by the equation $y = (x - 3)^2$. Also the limits of the plot window should have adjusted themselves to the new x values. Note that the y values are unchanged. This may or maynot be what you expect but it is the correct behaviour at the moment — see XXXX about dataflows.

In the previous example the limits in the plot window adjusted to the new values for x & y because the `line` command sets the plot limits if the plot is not held (such as with an explicit call to `env`). But what happens if we redo the example with our own chosen limits?

```
perldl> $x = sequence(10); $y = $x*$x
perldl> env (0, 9, 0, 81)
perldl> line $x, $y;
perldl> $s = retrieve_state()
perldl> $x += 3; replay $s
```

The result now should be as shown in Figure xxx which has the same plot limits as before, but a shifted parabola. This is because the state now remembers the explicit `env` statement that you had made and uses that to set the limits.

Finally you must remember that the reference is not to a variable name, but to a piddle which exists separately from the variable. Thus you cannot change your data at a whim, so the following change will change the data back to where we started

```
perldl> $x -= 3; replay $s
```

But the following will *not* plot a parabola starting at $x = 5$:

```
perldl> $x = sequence(10)+5.0; replay $s
```

The reason for this is that the reference kept in the state object is to the actual *data* in the previous x-object and not to the variable name.

However sometimes you want to give a entirely new dataset to the plot. Say you wanted to plot a sine curve instead of a parabola. Is there any way to do that? The answer is yes, but it looks rather ugly, so you might want to consider whether this is something you want to do

```
perldl> $x = sequence(10); $y = $x*$x
perldl> line $x, $y; $s=retrieve_state()
```

```
# Now let us transfor this to a sine plot
perldl> $y -= $y; $y += sin($x)
perldl> replay $s
```

And voilá! a sine curve does step forth. Not exactly elegant, but this trick allows you to replace any variable used in a complex plot with a totally different content.

Using recording in scripts In general the recording facility is of rather limited use in scripts because you can just as easily encapsulate your plot commands in a subroutine and just call the subroutine when need be. At present the only saving is probably in typing, but if the facility is extended to saving and restoring plot commands the situation would change.

4.8 The object oriented approach

Assume for once that you are developing a simulation. When you are testing the code (all written in PDL of course) you have to keep track of how some data changes at every time-step, but at the same time you want to look at time-averages. If you were to use what we discussed above you would probably want to display the time-steps in one panel and the time-averages in another panel in a plot window. The problem with this is of course that one panel is updated a lot more often than the other so you have to waste a lot of time re-plotting the time-average.

Clearly there are two possible ways to improve this: a) have a method which allows you to plot to a given panel when you want and b) have to plot windows. It is possible to use the first approach by giving the `Panel` option to the plot commands:

```
dev('/xs');
for (my $i=0; $i<$n; $i++)
   $integrand = func($x, $i);
   points $x, $integrand, {Panel => 2};
   $sum += $integrand;

points $x, $sum/$n, {Panel => 1};
```

So that this hypothetical code-bit would keep plotting in panel 2, updating the plot there until the loop is over at which point panel 1 is updated.

This can be practical, but it is rather limited given the requirement of giving the panel number every time. Instead an alternative approach would be to create several plot windows, and for this you really ought to use an object oriented

approach[3]. In this approach every plot device is a separate object and you call every plot command via this object. So the previous example would be

```
my $opt = {Device => '/xs', WindowWidth => 7, Aspect => 1};
my $integrand_window = PDL::Graphics::PGPLOT::Window->new($opt);
my $integral_window = PDL::Graphics::PGPLOT::Window->new($opt);

for (my $i=0; $i<$n; $i++)
   $integrand = func($x, $i);
   $integrand_window->points($x, $integrand);
   $sum += $integrand;

$integral_window->points($x, $sum/$n);
```

4.8.1 Why use the OO interface

So, you may say, what is the point with the OO interface except appeasing the OO fanatics around? It seems to require more typing and I can see no significant advantage.

In many situations these are valid arguments, if you are just plotting data on the command line in `perldl`, for instance, or do not need multiple plot windows. And at some level the OO interface is primarily a convenience for the programmer, and it is in fact how the PDL::Graphics::PGPLOT package is implemented. That said though there are some (possibly strong) arguments for using the OO interface:

- You do not pollute your namespace, which means that you are free to define routines that are called `line`, `points` and so on. This is the main reason why I use this interface personally when doing simple plots in programs.
- It is a *lot* easier to deal with multiple plot windows when using the OO interface, in fact I would personally discourage people from having multiple plot windows without using the OO interface.

Eventually an argument in favour of the OO interface will hopefully be that it would enable an easier mix of different plotting packages so that they can all be accessed in a similar way, but we are not there yet.

4.8.2 Usage of the OO interface

To use the OO interface one needs to create a new plot object and then call the plot routines through this object. If you want several windows, you just

[3]It is possible to have several plot windows without using the OO approach, although more cumbersome, see the on-line documentation for details.

create more objects and switching between these should be straightforward as you should be able to see in the following examples.

Note that since the OO interface is less suited to use on the command line, I have opted to show the examples as small code-bits but they should all be possible to execute from the `perldl` command line. In addition this section will merely give several examples of use of the OO interface and no discuss (again) the different commands since they are the same as we went through above, it is just a different way of calling them.

Opening a plot object and plotting a simple plot To create a plot object we first need to `use` the `PDL::Graphics2D` package — this is merely a shortcut for the true `PDL::Graphics::PGPLOT::Window` package, but why type more when it doesn't gain you anything? Then we create the object using the standard Perl notation `PDL::Graphics2D->new()`:

```
use PDL;

# Note that we could also access this as
# PDL::Graphics::PGPLOT::Window, but since this is
# shorter I advocate its use.
use PDL::Graphics2D;

# Now create a plot window
my $winopt = {Device => '/xs', WindowWidth => 7, Aspect => 1};
my $w = PDL::Graphics2D->new($winopt);

# Create a simple plot
$x = sequence(10);
$w->points($x, $x*$x, {Symbol => 'Triangle'};
```

Note how we use the window object (`$w`) when calling the `points` routine — since we didn't `use` the `pgpkg` package there isn't any function called `points` in our namespace and we use the window object to get hold of it. The structure is of course very similar to what we did in section 4.4.5 above and there really is little practical difference between the two interfaces when plotting to only one window.

Therefore let us up the stakes somewhat and try a more practical example. In many situations you might have one plot where each point in the plot has many values associated to it (ie. your plot is a slice in a multidimensional space). When you examine such data you often would like to click on a point on your plot and bring up associated data for that point in a different display — this is an obvious situation for the OO interface.

The logic for this project is easy: We first create two windows

```
use PDL;
```

```
use PDL::Graphics2D;

# Create two identical windows
my $winopt = {Device => '/xs', WindowWidth => 7, Aspect => 1};
my $data = PDL::Graphics2D->new($winopt);
my $associated = PDL::Graphics2D->new($winopt);
```

Note that it is a good idea to name your variables containing the window objects with sensible names for later use.

The next step is to plot data (well, in this example I will merely create them):

```
my $x = sequence(10);
my $y = $x**2;
# Plot points using standard symbol
$data->points($x, $y);
```

which should draw a nice parabola on your screen. Now the user (that is you, reader) has to click on (or near) a point to select it — we will then use the X-value of that point to set the period of sine curve. (No not at all an artificial example . . . But hang in there, it will get better).

```
print "Dear user, please click on (or close to) a point
n";
my ($xin, $yin) = $data->cursor();
# closest will now contain the index of the point closest to
#  where the user clicked.
my $closest = minimum_ind(abs($x-$xin) + abs($y-$yin));

my $y_associated = sin($x->at($closest)*$x);
$associated->line($x, $y_associated);
```

That should now give you a sine wave in the second window with a frequency dependent on where along the X-axis you clicked. Of course it would be a lot easier to use $xin, but that wasn't what we tried to do after all.

This is of course a very simplified example, but it does provide a framework for a more comprehensive data explorer. From astronomy a typical example would be to plot scatter-plots for two variables and bringing up images of the objects by clicking at their data in the plot window. In other situations the data might be financial data for a set of companies and clicking on the points would bring up a comprehensive summary of that company. But this is of course where your imagination is supposed to run amok.

The bottom line is that whatever your requirements are, the OO approach is probably better when you need more than one plot window, but when you only use one window, and particularly on the `perldl` command line.

4.9 Using PGPLOT commands directly

The Perl module PGPLOT[4] contains interfaces to all PGPLOT functions. The majority of these functions have alternative interfaces in the PDL package, but there might be situations when you need to use these functions directly. And in addition if you are used to using PGPLOT from before you might prefer the interface, although it is rather inconvenient when dealing with PDL.

Full documentation for the PGPLOT functions can be found at Tim Pearson's WWW page: `http://astro.caltech.edu/~tjp/pgplot/`. This is not the place to discuss the details of PGPLOT, but it is interesting to learn how to access these routines from PDL with piddles as arguments.

Typical PGPLOT drawing functions take as arguments the number of points and references to perl arrays to give x&y coordinates, thus

```
@x = (1,2,3); @y = (3,-1,7);
pgpoint(3, \@x, \@y, 4);
```

will plot three points with the x&y values indicates and using plotting symbol 4 (circle).

The complication for PDL users is that piddles are not perl arrays and hence have to be converted to array references before they can be passed to a PGPLOT function. This is achieved with the `get_dataref` command which returns a reference to the data in a piddle. Thus the example above would be written

```
$x = pdl(1,2,3); $y = pdl(3,-1,7);
pgpoint($x->nelem, $x->get_dataref, $y->get_dataref, 4);
```

in PDL.

In general you should use the provided wrapper routines for readability, but feel free to combine the two if you prefer. You should be able to pick'n'mix functions from the PDL interface and from PGPLOT directly, although a few subtle bugs might creep in (in particular the handling of several plot windows).

There are several situations where direct access to PGPLOT might be necessary. Although hopefully they are not very common, it can be useful to look at a few to see what the PDL::Graphics::PGPLOT module doesn't do. Since it is possible to mix PGPLOT commands with the PDL::Graphics::PGPLOT commands this is not a major problem though, although it might require you to learn some PGPLOT. So to turn to some examples, I have decided to list a few simple problems:

- Drawing several plot boxes on top of each other to get differently shaded grids. This is done in one of the demonstration programs that come with

[4] Written and maintained by Karl Glazebrook, and can be downloaded at any CPAN mirror (it is a prerequisite for the PDL package).

PGPLOT and can't be easily done in PDL::Graphics::PGPLOT without some playing around with the `PlotPosition` option. It is a lot easier to call `pgbox` directly.

- Complex contour plots — in particular non-rectangular. At present there is no support for non-rectangular contour plots in PDL::Graphics::PGPLOT, and neither is any support planned for the near future. You are advised to read the PGPLOT documentation for `pgconx` and have a look at demo #3 in the PGPLOT distribution for an example.

The bottom line is that as your plots get more and more complex you might end up in a situation where you need the finer control offered by the PGPLOT package, but for day-to-day use it is hoped that PDL::Graphics::PGPLOT will address most people's needs. And if doesn't we should be told!

Figure 4.6: A 3D surface graph plotted using `gnuplot`, using the commands `set isosamples 30; splot [0:7] [0:7] sin(x)*sin(y)`.

Figure 4.7: The same 3D surface, plotted using the `PDL::TriD` module. The two different images were obtained literally by grabbing the image in the window opened by PDL and dragging it with the mouse to rotate.

4.10 3D Graphics with OpenGL

4.10.1 Introduction

There are lots of programs that let you plot so-called 3D surface graphs, such as the one shown in Fig.4.6. However, from the beginning, PDL's 3D graphics have had something different that we feel is really useful: motion, or as we call it "twiddling". Dragging the 3D image with the mouse rotates the image, at the speed allowed by your display hardware. This turned out to be quite useful for displaying functions: the human eye is able to grasp the presented 3D surface much better when it moves, especially in response to the mouse.

Let's start with plotting the surface we showed using `gnuplot` in the beginning:

```
use PDL::Graphics::TriD;
$x = xlinvals(zeroes(30), 0, 7);
imag3d [ sin($x) * sin($x->dummy(0)) ];
```

This should produce a new window with the image seen in Fig.4.7. Notice that your console window is now frozen: it is waiting for you to twiddle in the graphics window using the mouse and to press `q` in that window once you're done.

If the above commands produce an error instead of a new window, it might be that your PDL wasn't compiled with the option to include the 3D library — tough luck (XXX).

That above expression is a bit more difficult than the `gnuplot` version, and there's a simple reason for that: `gnuplot` is primarily meant for plotting functions; PDL is meant for handling and plotting numerical data. So to plot a function, we have to create the data for the function first which is a bit more difficult.

Now let's go through that part by part. The first line simply tells Perl to load the `PDL::Graphics::TriD` module. The name comes from the fact that you can't have parts of module names starting with numbers, unfortunately. The second line

```
$x = xlinvals(zeroes(30), 0, 7);
```

creates a one-dimensional piddle with 30 elements that has linear values from 0 to 7:

```
perldl> p $x
[0 0.24137931 0.48275862 0.72413793 0.96551724.....
```

The `xlinvals` and the corresponding `ylinvals` and `zlinvals` are useful for exactly this purpose: creating piddles of equally spaced values. The final line,

```
imag3d [ sin($x) * sin($x->dummy(0)) ];
```

is what draws the actual image. The expression inside, uses the variable `$x` for both the X and Y coordinates, via a clever use of the `dummy` operation. (XXX explanation?) This results in a 2-dimensional piddle with the values for the Z coordinate. So far you've already seen all this. And the final part, `imag3d [`*vals*`]` is the call that creates the 3D plot and opens the new window for it. The brackets around the parameter may be slightly surprising: the 2-D commands work well without those but there is a good reason for this, as you'll learn later on: otherwise there would be a bad ambiguity.

4.10.2 Parametric Graphics

We alluded in the introduction that allowing

```
imag3d $piddle;
```

could be ambiguous and should be written

```
imag3d [$piddle];
```

if `$piddle` is intended to be the Z axis values of a rectangular 2D plot. Now is the time to find out why. The simple truth is that

```
imag3d $piddle;
```

is in fact legal code - *iff* the first dimension of `$piddle` has exactly three elements. As you probably have already guessed, these three elements are X, Y and Z. So what you can do is pass `$piddle` with dimensions $(3, t, u)$ which is the same as a 2-dimensional (t, u) lattice with a 3-vector at each point. This piddle will then be interpreted parametrically: the mesh will be drawn as a function of t and u. (XXX Complicated)

Let's have an example: a curve that is not possible to plot with just Z axis values, say the surface of a torus, with colors coming from somewhere. First, set up the piddles and the parameter variables:

```
use PDL::Graphics::TriD;
$torus = zeroes(3, 60, 20);
$x = $torus->slice("(0)");
$y = $torus->slice("(1)");
$z = $torus->slice("(2)");
$t = xlinvals $x, 0, 6.28;
$u = ylinvals $x, 0, 6.28;
```

Note that the coordinate separation *can* be done in just one line:

```
($x, $y, $z) = map {$torus->slice("($_)")} 0..2;
```

Next, we color the torus. Let's put stripes on it:

```
$r = (1+sin(2*$t + $u))/2;
$g = (1+cos(2*$t + 2*$u))/2;
$b = (1+sin(2*$t + 3*$u))/2;
```

Then, we choose the outer and inner radii and put the coordinates into the slices. We'll let the torus lie in the XY plane so the parametric coordinates can be easily derived. (XXX diagram?)

```
$r_o = 3;
$r_i = 1;
$x .= ($r_o + $r_i * sin($u)) * sin($t) ;
$y .= ($r_o + $r_i * sin($u)) * cos($t);
$z .= $r_i * cos($u);
imag3d_ns $torus, [$r, $g, $b];
```

And there's our colorful torus! It looks a bit more like a barrel because TriD automatically scales the axes but there it is. Note how we use `imag3d_ns` to get the colors instead of the shaded version.

Now, there is more than one way to do it. If your data is not by default in the three-vector format (as ours wasn't above), it is probably easier to do

```
imag3d [$x, $y, $z], [$r, $g, $b];
```

which will produce the same results. Also, we could concatenate the RGB piddles to form

Now, since PDL does its best to make dimensions usable anywhere, we can easily plot several parametrics of the same parameters at once, if we pack all the surfaces into a piddle of dimensions $(3, n_t, n_u, ...)$ where the three periods in the end indicate the beginning of the extra parameters.

For example, we can plot a family of shrinking toruses by adding an extra dimension into `$torus`:

```
$cone = $torus->dummy(3, 4)->copy();
$fac = axisvals($cone, 3);
$cone *= $fac + 2;
($tmp = $cone->slice("2")) += 4 * $fac;
imag3d $cone;
```

And further, if we want to distort them, it's perfectly possible:

```
$x = $cone->slice("0");
($tmp = $cone->slice("2")) += 0.1 * $x ** 2;
imag3d $cone;
```

Any other kind of mutilation is also possible but we leave you to discovering the interesting things that are possible by yourself, because we have to move to something else that's important to cover: coordinate systems. So far, all the examples you've seen have happened in the Euclidean coordinate system where the coordinates are specified as measures X, Y and Z on three orthogonal axes.

Or actually this is not true: in fact, we have used two kinds of coordinates, the explicit X, Y and Z given in this section but in the preceding sections, only Z has been given and X and Y have been assumed by the system from the context.

Of course, since PDL tries to follow "simple things simple, complicated things possible", it is possible to override the default context.

4.10.3 Types of 3D Graphical Objects

So far, we've only been toying with surfaces. However, PDL can do much more. We can plot points; here's a picture of two samples from different (overlapping) probability distributions, plotted with different colors:

```
use PDL::Graphics::TriD;
$i = zeroes(8000);
$which = random($i) < 0.5;
$x = grandom($i) * (1 + $which);
$y = grandom($i) * (0.5 + $which);
$z = grandom($i) * (2 - $which);
$x += $which * $y; $y += $which * $z; # Make it oblique
points3d [$x, $y, $z], [$which, 0.5*(1-$which), 1-$which];
```

A lot of fun things can be done with points but we'll go into that later.

Then, there are — of course — lines. As a fun demo of lines, let's plot a number of flowlines moving in the Lorenz attractor. As you may know, the Lorenz attractor is described by

$$\frac{dx}{dt} = \sigma(y - x) \tag{4.3}$$

$$\frac{dy}{dt} = (r - z)x - y \tag{4.4}$$

$$\frac{dz}{dt} = (y - b)z \tag{4.5}$$

where $\sigma = 10$, $r = 28$ and $b = 8/3$. Because we're just doing this as a simple demo, we'll use the extremely unstable $d = \Delta$ method integration. We'll plot six trajectories that start close to each other.

```
   use PDL::Graphics::TriD;
   $n = 500;
   $nstart = 0;
   $nc = 6;
   $delta = 0.015;
   # $x = pdl(1, 1, 1, 1, 1);
   # $y = pdl(1, 1, 1, 1, 1);
   # $z = pdl(1, 1.01, 1.02, 1.03, 1.04);
   $xs = zeroes($n, $nc);
   $ys = zeroes($n, $nc);
   $zs = zeroes($n, $nc);
   $x = -23 * ones($nc);
   $y = -2 * ones($nc);
   $z = 20 * ones($nc) + 0.02 * xvals($nc);
   $sigma = 10; $r = 28; $b = 8.0/3.0;
   for(-$nstart..$n-1) {
    if($_ >= 0) {
($t = $xs->slice("($_)")) .= $x;
($t = $ys->slice("($_)")) .= $y;
($t = $zs->slice("($_)")) .= $z;
}
    $dx = $sigma * ($y - $x);
$dy = ($r - $z)*$x - $y;
$dz = $x*$y - $b * $z;
$x += $delta * $dx;
$y += $delta * $dy;
$z += $delta * $dz;
   }
   $col = yvals(1, $nc) / ($nc-1);
   $tim = xvals($n) / ($n-1);
   line3d [$xs, $ys, $zs], [$col, $tim , 1-$col];
```

Unfortunately, this plot has too much stuff going on so it's difficult to see where the functions diverge even though they have different colors at different times. This is an excellent time to change variables: let's get rid of X and plot the time step instead:

```
   line3d [$tim, $ys, $zs], [$col, $tim , 1-$col];
```

This yields a much clearer plot of the chaotic behaviour when the lines diverge with time (Fig.XXX).

In the latest versions of PDL it is possible to adjust the line width as well:

```
line3d [$tim, $ys, $zs], [$col, $tim , 1-$col], {LineWidth => 10}
```

gives the same plot but with much thicker lines.

The basic rectangular surface you already saw in the preceding sections. It also has an option to turn off the lines. XXX?

There is also a command `mesh3d` similar to the surface which just draws the surface as a wire mesh instead of a solid surface. On slow machines this can be of great help.

Finally, there are two commands for quickly painting strictly rectangular true-color images: `imagrgb` and `imagrgb3d`. This can be demonstrated by Tuomas J. Lukka's 4-liner:

```
use PDL; use PDL::Graphics::TriD;$a=zeroes 300,300;$r=$a->xlinvals(-1.5,
0.5);$i=$a->ylinvals(-1,1);$t=$r;$u=$i;for(1..30){$q=$r**2-$i**2+$t;$h=2
*$r*$i+$u;$d=$r**2+$i**2;$a=lclip($a,$_*($d>2.0)*($a==0));($r,$i)=map{$_
->clip(-5,5)}($q,$h);}imagrgb[$a/30];
```

This, as odd as it may sound, plots a grayscale Mandelbrot. If you work your way through the code, you'll see that it simply iterates the standard Mandelbrot iteration formula

$$z \leftarrow z^2 + C$$

where C is the original point. Then it uses `lclip` to keep the numbers in a reasonable range and colors the points according to the iteration when the point crossed the distance $\sqrt{2}$ from the origin. The piddle `$a` is two-dimensional so just like for coordinates, it is enclosed in an array ref. It is also possible to use

```
imag3gb [$r, $g, $b];
imag3gb $colors;
```

where the RGB piddles are two-dimensional and `$colors` has three dimensions, the first of which is of length three.

The command `imagrgb3d` does the same but allows the user to place the rectangle anywhere in 3-space. This is useful e.g. for putting an image underneath a plotted surface of the same function, as we shall see in the next section.

4.10.4 More than one Image

If you have used the PDL PGPLOT interface for plotting multiple graphs then TriD is not going to surprise you: the commands `hold3d` and `release3d` work

just like their PGPLOT counterparts. Before going further, however, let me remind you that for many plots, it is not necessary to explicitly plot several points, lines, surfaces or whatever: it can be easier just to use extra dimensions, like we used for the torus cone in the first section.

However, if you want to put objects of more than one type, or objects of more than one resolution on the same graph, then you do need to do so explicitly. As an example we'll use some fractal mountain code by Tuomas J. Lukka from the 3D Gallery. Unlike with the mandelbrot that has a well-known algorithm, this code we'd just better format clearly from the start (the parameters have also been slightly modified and the code has been modified to plot all the iterations on top of each other).

```
# XXX FIX - LOOKS BAD.
use PDL;
use PDL::Image2D;
use PDL::Graphics::TriD;
$k=ones(3,3) / 9;
$a=20;
$b=$a*(random(2,2)-0.5);
hold3d();
# Set the coordinate system: XXX hack!!! FIX TriD
line3d pdl([[0, 0, 0,], [0, 0, 10]]);
for(0..4) {
   if($_ != 0) {
    $c=$b->dummy(0,2)->clump(2)->xchg(0,1)->
          dummy(0,2)->clump(2)->xchg(0,1)->copy;
    $c+=$a*($c->random-0.5);
    $a/=1.5;
    $b=conv2d($c,$k);
   }
   imag3d[xlinvals($b, 0, 1), ylinvals($b, 0, 1),
    $b + 2.0*$_],{Lines => 0};
}
release3d();
```

Even laid out bare, this code is a mouthful with that big double `dummy-clump-xchg` thing in the middle. But in fact the function is really simple: the `dummy-clump-xchg` thing simply doubles the length of each dimension, copying each value to two consecutive locations. After doubling the resolution, we add some noise from the `random` function (the magnitude of the noise is diminished each time). Finally, we pull in `PDL::Image2D` for the `conv2d` routine that does 2-dimensional convolutions (optimized for small kernels like ours). We use a 5×5 kernel to smooth our data at each step by convolution.

That's the numerical part, now for the graphics part:

4.10.5 Animation

4.10.6 Putting it all together — cool hacks

Here's one where the original idea is by Robin Williams, done for the 3D Gallery. This gallery is available in the PDL distribution in the file `Demos/TriDGallery.pm`. The idea is to put interesting scripts that do a lot using just 4 lines of 72 characters. The crux of the idea is to use OpenGL points to perform volume-like rendering. Now, if you are really into volume rendering, we recommend taking a look at Karma, discussed in XXX: this is just a quick hack. However, the principles are interesting enough that we thought you might enjoy them.

Let's start with a function of three variables, whose zeroes are a sphere and an ellipsoid inside the sphere, with the Y axis slightly distorted to form a parabola with the Z axis:

```
sub f {
  my($x, $y, $z) = @_;
  $y = $y +  0.04 * $z**2;
  return (($x**2 + $y**2 + $z**2) - 100) *
         ((2*$x**2 + 4*$y**2 + 4*$z**2) - 100);
}
```

Note here that we can't use the `+=` operator for `$y` since below we use the same piddle for the three coordinates (with a simple `dummy` transformation).

Now, we want to picture approximately where the function crosses zero, but since there are two separate zero surfaces we can't just use an algorithm that finds a zero and creates an isosurface. Besides, an isosurface renderer wouldn't be able to show both the sphere and the ellipsoid simultaneously. So rather, let's first calculate the sign of the function in a $50 \times 50 \times 50$ lattice. The radius of the sphere is $\sqrt{100} = 10$ so we make the coordinate system slightly larger.

```
use PDL::Graphics::TriD;
$x = xlinvals(zeroes(float,50), -11, 11);
$f = f($x, $x->dummy(0), $x->dummy(0)->dummy(0));
$sign = byte($f>0);
```

Now that we have the sign, why don't we simply find the set of points where the sign has changed. It is simplest to do this over just one dimension:

```
$df = ($sign->slice("0:-2") != $sign->slice("1:-1"))
points3d [whichND($df)]
```

And indeed, we get a rotatable set of points in 3-space that are in the shape of a sphere with an ellipsoid inside, slightly distorted, just as ordered. This is not yet a good picture: there is a hole in the point set where the surface is

parallel to the X axis, naturally, since there there is no difference between the sigh between the points next to each other on X axis.

To do a more complete job, we need to compare the signs not only along X but other dimensions as well. This is possible due to the wonderful invention by Robin Williams:

```
$a = $sign;
foreach(1,2,4) {
   $t=($a->slice("0:-2")<<$_);
   $t+=$a->slice("1:-1");
   $a=$t->mv(0,2);
}
points3d [whichND(($a != 0) & ($a != 255))];
```

It's a bit cryptic but truly beautiful so bear with us while we go through it. The loop is executed thrice, once for each dimension. In the beginning, we know that all the values in `$a` are either 0 or 1. The first line of the loop takes a slice from `$a`, leaving the last element of dimension one out and shifts if by the loop index `$_`. The second line takes another slice, this time leaving out the first element and adds it to the first. Finally, the dimensions are rotated for the next invocation.

Choosing the shifts to be $1, 2, 4$ is the key: this way after the first round, the piddle contains values $0, 1, 2, 3$, after the second it contains 0...15 and after the third, 0..255. None of the shifts shift anything on top of each other so the plus operation could be replaced with a bitwise or.

So after the loop, we have a three-dimensional piddle with one index less in each dimension, and each value in that piddle contains in its 8 bits the 8 corners of a small cube. Finally, to find whether the function crosses zero at that cube, we simply check whether all the bits are equal, i.e. whether the number is 255 or 0 and if it isn't we know the function changes sign.

The image quality can be slightly improved by removing the Moire effect through randomization:

```
points3d [map {$_+$_->float->random} whichND(($a != 0) & ($a != 255))]
```

note how the `map` is necessary: the whole point is that `whichND` returns a *list* of piddles.

Now, to further improve image quality we could add different-color pixels but that would require alpha blending to the OpenGL parameters and this would get into complications we don't necessarily want here. So now we're going to KISS[5] this topic away and move to the next one.

[5]Keep It Simple, Stupid

4.10.7 The VRML backend

4.11 Pixel manipulations using the Gimp

Ever wondered how some algorithm would look if applied to a picture? Sure, you could load an image, apply your function and display the result, but there is a much more exciting way: the Gimp.

GIMP is an acronym for "GNU Image Manipulation Program".[6] As it's name implies, it's main purpose is manipulating and creating images of any kind – be it a photo of your niece, a gif animation or a computer-generated image. Since it is directed at end-users it is mainly used as an interactive application.

Of course(!) there is also a perl-interface, usually called *gimp-perl*.[7] Using that interface, you can write so-called *plug-ins*, programs that can add menu entries and functions to the gimp, allowing users to apply effects to their images.

While most perl extensions to gimp concentrate on combining built-in-functionality and other plug-ins to create effects, it is of course possible to directly manipulate raw pixel data. The ideal way to cope with large amounts of data (and images *are* large) in perl is PDL – and that's why PDL is used to represent any kind of raw data in gimp-perl.

4.11.1 The components of Gimp

A simple plug-in

4.11.2 Direct pixel access

Most of the complexity in dealing with the gimp is in getting and storing pixel data – once you have a piddle you can do anything with it.

There are two principal ways to access pixels. The first way is similar to other graphics libraries: You can get or set pixels, rows, columns or rectangles of pixel data. The second way is *iterating* over a region.

Pixel regions

The pixel iterator

4.11.3 Gimp Plug-Ins

The following perl program is a short but full-featured gimp-plug-in, complete with a dialog to ask for input parameters. You can use it for your own experi-

[6] The official GIMP webpage is `http://www.gimp.org/`

[7] The "official" and only homepage for gimp perl can be found at `http://gimp.pages.de/`

ments.

4.11.4 More Information

It is far beyond the scope of this book to dive deeper into gimp programming.

Chapter 6

Advanced Examples of Using PDL

This chapter presents three interesting case studies in which more complex PDL constructs are used. Starting out will be an example of PDL's use in plotting a weather map. Next is an in-depth discussion of object-oriented PDL in which complex numbers are added as a data type. A numerical simulation example rounds out this tour of hairy PDL examples.

[or some such intro paragraph]

6.1 A PDL weather map

The object of this example is to produce a contour plot of global temperatures overlaid upon a linear projection of a world map. Let's take a look at the finished product in fig. 6.1.

6.1.1 Drawing the map

The first difficulty is to draw the world map. This is actually quite a trick, as map databases are large and complex. A good mapping facility is generally an add-on costing hundreds of dollars in packages such as MATLAB. Fortunately, there exists a perl module which is an interface to the popular GMT [1] map plotting package. This perl extension, PDL::Graphics::PGPLOT::Map, uses PGPLOT to plot maps extracted from the GMT coastline database.

[1] GMT (Generic Mapping Tools), is a suite of UNIX tools for drawing maps and plotting data on them. Included is a scrupulously assembled set of world coastline databases at various resolutions. GMT was created by Paul Wessel and Walter H. F. Smith.

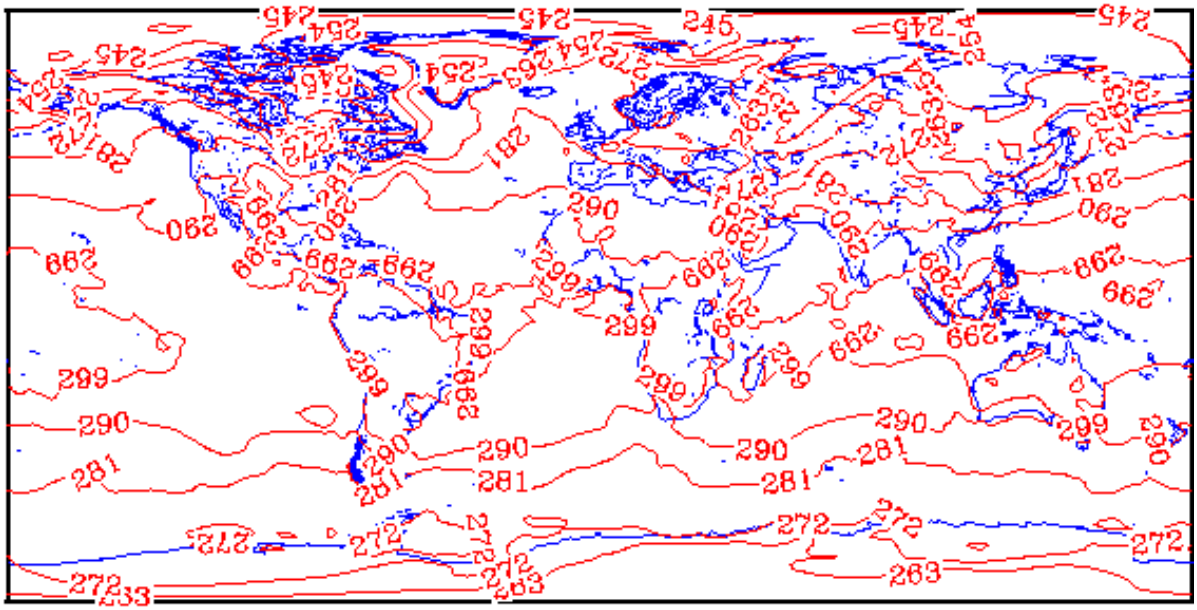


Figure 6.1: A contour plot of global temperatures, overlaying a world map.

```
worldmap ({WEST => $minlon, EAST => 180, SOUTH => -90, NORTH => 90,
   PROJECTION => 'LINEAR'});
```

The above command sets the boundaries of the map and requests a simple linear projection. A linear projection is convenient for plotting contours.

6.1.2 Getting the weather data

Next we will find something to plot. In this case, we choose National Center for Environmental Prediction 'AVN' model data[2]. The AVN model data consist of various atmospheric parameters (temperature, humidity, etc.) stored on latitude/longitude grid points and pressure levels (1000 millibars, 850 millibars, etc.) for the entire world. These data files are generated by complex weather models initialized using input data from weather balloons, airplane measurements, satellite measurements and other sources. They are stored in netCDF (network Common Data Format), a binary format used to store gridded data of all types in a system-independent, easy-to-access fashion. Here we open up an example file and grab a global surface temperature grid:

[2]These data are generally available in a compact yet obtuse binary format called 'GRIB' which is distributed via the Internet Data Distribution system (see `http://www.unidata.ucar.edu /projects/idd/index.html`) and then converted to the nicer netCDF format (see `http://www.unidata.ucar.edu /packages/netcdf/index.html`) via the Unidata Decoders (info at `http://www.unidata.ucar.edu /packages/decoders/index.html`).

```
my $infile = "avn_97.032.00_cdf";
my $nc = PDL::NetCDF->new($infile);
my $tsfc = $nc->get("T");
$tsfc = $tsfc->slice(":,:,(0),(0)");
```

We open this file using an extension to PDL called `PDL::NetCDF` which allows one to easily read and write PDL data objects to/from netCDF files. Then we grab a four dimensional grid of temperature data (longitude by latitude by pressure level by time) and slice out the surface data for a single time step into `$tsfc`.

6.1.3 Plotting the contour map

We must now determine how the gridded data we just read is oriented on the globe. Fortunately these netCDF data files have several variables included along with the actual weather data which tell where the grid starts and ends and how widely spaced the data are. These values are used:

- `$la1` — The latitude of the (0,0) point on the grid. The example grid starts at -90 latitude.
- `$lo1` — The longitude of the (0,0) point. This grid starts at -30 (30 degrees west) longitude.
- `$di` — The longitude increment, 1.25 degrees in this example.
- `$di` — The latitude increment, 1.25 degrees in this example.

Once these variables are read in, we rotate the surface temperature grid so that it starts at 180 degrees west longitude (the edge of the map), instead of 30 degrees west:

```
$tsfc = $tsfc->rotate(int(($lo1 - $minlon)/$di));
```

In the final lines of the script, we generate contour labels for the graph and plot the temperature contours:

```
my $ncont = 8;
my $range = $tsfc->max - $tsfc->min;
my $interval = int($range/$ncont);
my $contours = (sequence ($ncont) * $interval) + int($tsfc->min) + 1;
my @labels   = list $contours;

cont ($tsfc, {LABELS     => \@labels,
      LABELCOLOR => 3,
```

```
CHARSIZE   => 0.7,
LINEWIDTH  => 1,
CONTOURS   => $contours,
COLOR      => 2,
TRANSFORM  => pdl ([$minlon, $di, 0, $la1, 0, $dj])});
```

The contour labels are determined dynamically from the data in the temperature field. Only the number of contour levels (12) is hard-coded. A linear transform based on the grid specification variables read in earlier is used to line up the contour map with the coastlines. See `perldl> help cont` and the PGPLOT library documentation[3] for more information.

The entire script is shown in figure 6.2.

6.1.4 What happened?

This example used a convenient PDL interface to the popular GMT mapping package. It also used the PDL extension PDL::NetCDF to read in a large global grid of temperature data. Finally, some of the more advanced features of PDL::Graphics::PGPLOT were used to overlay a contour map of the data. All the software used in this example is freely available and open source. As is typical in the open source world, half the battle can be installing all the necessary software. Here is a list of necessary packages to get this example to work:

- PDL itself (of course)
- The pretty good plotting package (pgplot), available at `http://astro.caltech.edu / ~ tjp/pgplot/install.html`. The example uses a patch to pgplot which allows output in the popular (and unpatented) PNG format. These patches are available from `http://hea-www.harvard.edu / ~ rpete/pgplot-png/`.
- The netCDF data format library, available at `http://www.unidata.ucar.edu /packages/netcdf/index.html`.
- The PDL::NetCDF PDL extension, available on CPAN under `modules /by-module /PDL/DHUNT`.
- The PDL::Graphics::PGPLOT::Map package, also available on CPAN under `modules /by-module /PDL/DHUNT`.

[3] on the web at `http://astro.caltech.edu /~tjp/pgplot/subroutines.html#PGCONT`

```
#!/usr/bin/perl -w

#
## Create a weather map using PDL
#

use PDL;
use PDL::NetCDF;
use PDL::Graphics::PGPLOT;
use PDL::Graphics::PGPLOT::Map;

# Local path information for pgplot
$ENV{PGPLOT_FONT} = "/usr/lib/pgplot/grfont.dat";

# Plot to a postscript file
new_window "pdlmap.ps/VCPS";

# Plot a world map
my $minlon = -180;
worldmap ({WEST => $minlon, EAST => 180, SOUTH => -90, NORTH => 90,
           PROJECTION => 'LINEAR'});

# Input NetCDF file of 'AVN' model data.
my $infile = "avn_97.032.00_cdf";

# Open up this file using the PDL::NetCDF module
my $nc = PDL::NetCDF->new($infile);

# Read in global temperature fields (including all upper air levels)
my $tsfc = $nc->get("T");

# Slice out the surface data (get rid of the upper air levels)
$tsfc = $tsfc->slice(":,:,(0),(0)");  # get rid of dummy 3rd dimension

# Read in grid orientation information from the netCDF file
# This is done to match up the plotted contour data to the map.
my $la1= $nc->get('La1')->at(0);  # Latitude of first point (0,0)
my $lo1= $nc->get('Lo1')->at(0);  # Longitude of first point
my $di = $nc->get('Di')->at(0);   # Longitude increment
my $dj = $nc->get('Dj')->at(0);   # Latitude increment

# Rotate tsfc so it starts at minlon = -180
# (it currently starts at lon = $lo1, -30)
$tsfc = $tsfc->rotate(int(($lo1 - $minlon)/$di));

# Determine countour levels/labels
my $ncont = 8;
my $range = $tsfc->max - $tsfc->min;
my $interval = int($range/$ncont);
my $contours = (sequence ($ncont) * $interval) + int($tsfc->min) + 1;
my @labels   = list ($contours);

# Plot a contour map of the global surface temperatures.
cont ($tsfc, {LABELS     => \@labels,
              LABELCOLOR => 3,
              CHARSIZE   => 0.7,
              LINEWIDTH  => 1,
              CONTOURS   => $contours,
              COLOR      => 2,
              TRANSFORM  => pdl ([$minlon, $di, 0, $la1, 0, $dj])});

release;
```

Figure 6.2: Source code for the contour map

6.2 Objects using PDL - defining your own data types

PDL provides a very powerful datatype: Vectors and matrices[4] can be used to represent almost any real-world problem. However, often one wants to extend the information stored in a piddle or even the functionality that PDL provide.

Extending PDL is easy, you just have to put your new method into the `PDL` package and it can then be used just like any other PDL method:

```
sub PDL::ring_modulate {
   my $carrier = shift;
   my $signal  = shift;

   $carrier * $signal;
}
```

Now `ring_modulate` can be used just like any other pdl method:

```
$car->ring_modulate($sig)->clip(-1,1);
```

Attaching data is also very easy, as every piddle has an attached *attribute* or *header* hash[5] that you can fill with any data you like. The `PDL::Audio`-module for example attaches information like sampling rate and filetype to a piddle:

```
sub raudio {
   my $audio =  <... read some audio stream into a piddle ...>

   my %hdr = (
      rate     => $samplingrate,
      filetype => $filetype,
      channels => $audiochannels,
   );

   # attach the header (actaully a reference to it)
   $audio->sethdr(\%hdr);

   $audio;
}
```

This information can then be queried or changed by anybody who is interested:

[4] Of course, PDL objects can have any number of dimensions.

[5] Actually it can be any type, but if you expect to work well with others you should use a hash.

```
$srate = $audio->gethdr->{rate};
$audio->gethdr->{filetype} = FILE_AIFF;
```

If you want to attach this information not only to the piddle itself but also to all copies of it, you can set the *headercopy* flag, using `hdrcpy`:

```
# enable header-copying
$piddle->hdrcpy(1);

$otherpiddle = $piddle + 1;
# now both $piddle and $otherpiddle share the same header hash.
```

This is not as useful as it sounds, however, since all piddles created that way share the same hash, so one must be careful when changing parts the header as this change will be effective in many piddles.

Both methods of extending PDL have drawbacks: Putting another function into the already crowded PDL namespace can be difficult[6], and using verbose method names (e.g. `audio_phaseshift` instead of simply `shift`) is counterproductive: After all, *we* already know that our piddle represents some audio signal, and so should perl.

Similarly, attaching extra data to a piddle doesn't change its nature: All operations still treat it as a regular piddle.

Neither of the above methods lets us create piddles that "know" that they are audio files, or "know" that they should be multiplied like complex numbers instead as of vectors.

6.2.1 Implementing derived classes: The Complex module

While we are at complex numbers... I once needed complex arithmetic for PDL myself, but couldn't find good support for it, so I sat down and tried to write a module implementing complex numbers using regular piddles. This was not only a very useful way to learn more about PDL, after experimenting and a lot of help from others it also became quite natural to use, even more than I initially expected. As it turned out, subclassing PDL is not difficult, but it is not always obvious how to do it.

The most natural representation for complex numbers in PDL is a vector with two elements. I decided to put the real part into the first element and the imaginary part into the second, so $5 - 3i$ becomes the piddle `[ 5 -3 ]`.

The first thing we need is a way to create complex numbers. Instead of providing a specialised constructor as one would like for other objects[7], I opted for typecasts:

[6] "Hopefully nobody else called her function `shift`... Oops."

[7] e.g. for objects not representing numbers, like an audio signal or an image.

```
package PDL::Complex;

# cast from PDL to PDL::Complex
sub cplx($) {
   bless $_[0]->slice('');
}

*PDL::cplx = \&cplx;

# cast back
sub real($) {
   bless $_[0], PDL::;
}

# provide 'i'
$i = cplx pdl 0, 1;
sub i() { $i->copy }
```

What's going on here? The `cplx` function expects a piddle (it doesn't really matter wether it's already a complex number of type `PDL::Complex` or a regular piddle (type `PDL`)) and re-`bless`es it into the `PDL::Complex` namespace. This has the effect of changing it into a complex number.

The unusual `slice('')` creates a new piddle object that references the same data as the original one. It is this operation that really makes it a cast. Without it, the *original* piddle would be changed, and using `copy` would destroy the dataflow a PDL-user expects.

The line

```
*PDL::cplx = \&cplx;
```

is a bit tricky: It puts a copy of the `cplx` function into the PDL namespace, so that regular piddles can be converted to complex numbers. I decided that it makes sense to call `cplx` on complex numbers as well, as it is a no-op then and can be used safely to ensure that a pdl is of complex type, much like the standard `pdl` constructor also accepts a piddle as argument.

The same could be said about `real`, but since this is less often used on nirmal piddles it makes sense to provide only a version that works on complex numbers.

The next line of code contains the first call to `cplx`: It creates the complex constant i ($= 0 + 1i$) and puts it into the global variable `$i`. To make matters even easier a function with the same name is created. This allows users to write code like the following:

```
5+3*i
pdl(1,2,3) + pdl(3,4,5)*i
$somepiddle *= 1-2*i;
```

Which looks very natural and makes the casting operators less neccessary.

Constructing derived piddle objects

Apart from implementations for the complex arithmetic operators, there is another detail one must know when subclassing PDL: PDL functions need a way to create result values of the right type. While PDL knows how to create an object of type `PDL`, it has no idea how to create an `Icky::Yicky::Data`. Instead, it calls the `initialize` method of the class, e.g. `Icky::Yicky::Data->initialize`.

All standard PDL functions always generate a result with the same type as their first argument (as if they were method calls, which, in some sense they all are).

If the new datatype can be implemented using a plain PDL, you do not have to supply an initialize method; PDL's default initialize will return an empty piddle blessed into the correct class. This is adequate for simple classes like `PDL::Complex`.

For more complex cases you can use a special feature of PDL: When PDL sees that the object it got passed is not a piddle, but a hash, it tries to find the real piddle in `$object->{PDL}`, that is, you can create a fairly standard perl object which still acts as a PDL by writing a constructor like the following:

```
sub initialize {
   bless {
      mydata => ...,
      PDL    => PDL->null,
      datax  => ...,
   }, __PACKAGE__;
}
```

Overloading standard functions

Unless the derived objects act just like plain piddles[8], you also need to override the perl operators for the new type.

The first step is to implement these new operators as normal functions. Here are two examples, that implement `Csub` (addition)[9]. and `Csin` (the complex sine):[10]

```
pp_def 'Csub',
```

[8] which makes sense in a lot of cases where the derived class just adds new data or new methods but does not alter semantics.

[9] Of course `Csub` is the same as the normal subtraction operator. However, for reasons that will be explained shortly, the subtraction operator *must* be overloaded, so it doesn't hurt to implement it for completeness or symmetry, or whatever. `Csub` might also be slightly faster in case the compiler doesn't do loop unrolling

[10] If you are not interested in PP code you can safely skip this example

```
        Pars => 'a(m=2); b(m=2); [o]c(m=2)',
        Code => q^
           $GENERIC() ar = $a(m=>0), ai = $a(m=>1);
           $GENERIC() br = $b(m=>0), bi = $b(m=>1);
           $c(m=>0) = ar + br;
           $c(m=>1) = ai + bi;
        ^
;

pp_def 'Csin',
        Pars => 'a(m=2); [o]c(m=2)',
        Code => q^
           $GENERIC() ar = $a(m=>0), ai = $a(m=>1);
           double s, c;
           SINCOS (ar, s, c);
           $c(m=>0) = s * cosh (ai);
           $c(m=>1) = c * sinh (ai);
        ^
;
```

The obvious (but, as usual, imperfect) way to make '-' act like 'Csub' is to use the *overload* pragma:

```
use overload
   '-'   => sub {
      my ($a, $b, $swapped) = @_;
      $b = r2C $b unless UNIVERSAL::isa $_[1], PDL::Complex::;
      return $swapped ? Csub ($b, $a) : Csub ($a, $b);
   };
   'sin' => \&Csin,
```

Now you can see why I used `Csub` for this example and not `Cadd`: subtraction is not symmetric, so the implementation must take care to swap the operands when neccessary[11]. To support cases like `5 - $complex` and `$regularpiddle - $complex`, where one of the operands is a plain number or a regular piddle consisting of all real numbers, the operator also tries to convert the operand to the complex domain first (`r2C` maps each number r to the vector/complex number [r 0]).

This is also the reason why even operators like `Cadd` and `Csub` have to be overloaded – although their mathematical behaviour is identical to the standard '+' and '-' operators, the *type* of the result differs.

Fortunately, the sine function is much simpler, it is nothing more than a direct call to `Csin`.

[11] The manpage to the `overload` pragma explains this in greater detail.

Perl's idea of overloading

If the world were perfect we would be finished. But of course it is not: while the overloaded definitions seem to work in most cases[12], perl sometimes ignores our definitions and uses the regular PDL-operator. For example, when the first operand is a regular piddle, perl chooses the standard operation[13]. Sometimes, for example in `5*i+pdl(6)`, perl calls the standard operator for no obvious reason.

Fortunately, this can be fixed in the same generic way for each extension module, so there is hope that PDL will take care for this problem in the future. Until then, however, you also have to replace all binary operators in PDL, which is a bit tricky. Here is how it's done for addition:

```
{
  package PDL; # place definitions in the PDL namespace
  no warnings; # perl might complain about redefinitions

  sub cp($;@) {
    my $method;
    if (ref $_[1]
            && (ref $_[1] ne 'PDL')
            && defined ($method = overload::Method($_[1], '+')))
            { &$method($_[1], $_[0], !$_[2])}
    else { PDL::plus (@_)}
  }

  use overload
    '+' => \&cp,
  ;
}
```

For subtraction, multiplication and division you have to replace the '+' by the corresponding operator name and the call to `PDL::plus` by a call the `PDL::minus`, `PDL::mult` or `PDL::divide`. Just look at the source of the Complex module[14] for appropriate definitions.

However, once all this is in place, everything should work smoothly:

```
perldl> use PDL::Complex;
perldl> p 1 + i * r2C pdl 1,7
[
 [1 1]
```

[12] The Complex module was already used for some projects when Christian Sller pointed out that something must be badly broken somewhere.

[13] One could argue that this is not very helpful, but at least it sounds consistent.

[14] e.g. by `perldoc -m Complex`

```
 [1 7]
]

perldl> p sin 5 * i ** 0.5
[-6.5908487 -15.829068]
```

6.3 Numerical simulations in PDL

The majority of this book has described the capabilities of PDL as a fast, flexible language for data analysis. In the present section, I will describe some facilities and methods for data *generation*.

Low level languages such as Fortran or C are often used for numerical analysis, mainly for speed and because of the availability of numerical libraries. As data sets become larger, the speed advantage of a compiled program becomes less important — so long as array handling is done in a low-level fashion, as in PDL, there is little advantage in using a compiled language for the high-level part of the algorithm.

Using PDL has significant advantages over a traditional compiled language. The interpreted shell allows for an adaptable toolkit approach to the development of algorithms and for close integration between numerical algorithms and a flexible graphics front-end. The slicing methods intrinsic to PDL can also make the structure of the high-level algorithm clearer in the implementation.

My aim here is to introduce the use of PDL in numerical simulations — techniques specific to PDL, and methods available in the PDL distribution — rather than to provide a tutorial in numerical methods. As a result, the examples are chosen for clarity. Better algorithms for some problems are included in the PDL distribution, while the methods illustrated are easily extended using improved algorithms from elsewhere (e.g. [xxx]).

6.3.1 The wave equation

We now discuss a simple example, the numerical integration of a one-dimensional wave equation,

$$\frac{\partial^2 \phi}{\partial t^2} = -c^2 \frac{\partial^2 \phi}{\partial x^2}, \tag{6.1}$$

or for our present purpose as two separate differential equations

$$\frac{\partial \phi}{\partial t} = c\frac{\partial \rho}{\partial x}, \tag{6.2}$$

$$\frac{\partial \rho}{\partial t} = -c\frac{\partial \phi}{\partial x}. \tag{6.3}$$

The general solution to these equations consists of two waveforms with constant shape, one progressing to the left at speed c, the other to the right at the same speed. In the example below, we have placed the gas in a finite 'box', so the waves reflect off the ends repeatedly: the equation, in this case, models the behaviour of the air in a flute or the string on a guitar, when subject to a small initial impulse.

The code implements the first-order Lax method, a simple algorithm for this time-dependent problem. In effect, the code assumes that the gas within each

numerical grid cell is of uniform density. This leads to quite high numerical diffusion, which is added to further to maintain stability. More complex algorithms which approximate the data within the cells as linear, quadratic or higher power-laws maintain waves for longer without damping. However, the Lax method illustrates well the way in which these more complex algorithms can be coded in PDL.

The guiding principle is to avoid the generation of temporary arrays within the simulation loop. This means that some obscure preparatory work is needed before the loop is reached. Having got past this, the algorithm is clearly displayed once the simulation loop is reached.

In this code, I have used separate arrays for primitive data (the values used to calculate the fluxes) and conserved data (the values to which the fluxes are added). This distinction is not really necessary in the present example, but makes it easier to generalize the code to more complex systems.

The first statements in the code allocate all the array space which will be needed: the data array (allowing ghost cells for the implementation of boundary conditions), the flux array (which stores rate of transfer of material across the edges of the cells), and an array to accumulate the change in the data over a time step before it is added back to the data.

Next, various slices of these arrays are prepared. By retaining these slice PDLs, no calls of 'slice' need to be made within the loop. The names of these slices are chosen to reflect their contents: `$eleftp` contains the values of the data to the left of the current edge, `$crghtp` to the right of the current cell.

Various constants characterizing the simulation are then set up: the boundary conditions, the maximum wave speed (which limits the allowed timestep of the algorithm). The initial contents of the simulation grid are set in the primitive (or physical) array `$p`, and transferred to the conserved variable array `$u` by the subroutine `ptou()`. Here this is a rather pointless direct copy – the relation between primitive and conserved variables is less trivial in many more interesting examples.

Having laid out this ground work, we now reach the simulation loop. First, the primitive variables at the start of the timestep and the allowed size of this step are calculated. The timestep is a constant for this simple example, but often it will depend on the contents of the simulation grid. One common limiting factor for the timestep is the so-called 'Courant condition'. This stability limit may be interpreted as meaning that, in the physical problem, the waves carrying perturbations from any cell will not have propagated beyond its immediate neighbours within the timestep. Methods for determining the limits which apply to any specific problem are described in most textbooks on numerical analysis.

Next, the values in the 'ghost' cells beyond the edge of the simulation are set. The numerical algorithm uses data values on either side of each interface to calculate the flux across the interface, so to calculate the fluxes at the edge of the grid, extra data has to be 'made up' for fictitious cells outside it. Here, I

```
# 1D wave equation using Lax method

use PDL;
use PDL::Graphics::PGPLOT;

*plus = \&PDL::plus;
*minus = \&PDL::minus;

$ncell = 500;
$cour  = 0.3;
$nq = 2;

$pb  = zeroes($nq,$ncell+2)->float; # Primitive data with boundary cells
$u   = zeroes($nq,$ncell)->float;   # Conserved data
$f   = zeroes($nq,$ncell+1)->float; # Array for fluxes
$df  = zeroes($nq,$ncell)->float;   # Array for flux differences

$p     = $pb->slice(",1:-2");       # Primitive data in active grid
$leftb = $pb->slice(",0:0");        # Left boundary cell
$rghtb = $pb->slice(",-1:-1");      # Right boundary cell
$lefti = $p->slice(",0:0");         # Left inside cell
$rghti = $p->slice(",-1:-1");       # Right inside cell

$eleftp = $pb->slice(",0:-2");      # Edge left values
$erghtp = $pb->slice(",1:-1");      # Edge right values

$leftf  = $f->slice(",0:-2");       # Left flux
$rghtf  = $f->slice(",1:-1");       # Right flux

$bc    = pdl (1,-1);                # utility constant for boundary
$cwave = 1;

# set density
($t = $p->slice("(0)")) .= 1;
# and rate of change
($t = $p->slice("(1)")) .= 0;
# and initial perturbation
($t = $p->slice("(0),0:".($ncell/2-1))) .= 2;
ptou($p,$u);

$t = 0.;
$i = 0;
```

Table 6.1: Example numerical simulation: the one dimensional wave equation, using the Lax method. Part one: definitions.

```
# General first-order algorithm
while (1) {
  utop($u,$p);
  $dt = $cour/maxv($p);
  $leftb .= $bc*$lefti;         # Set boundary cells
  $rghtb .= $bc*$rghti;
  # Calculate fluxes
  flux($eleftp,$erghtp,$f);
  minus($leftf,$rghtf,$df,0);
  # Update array to new timestep
  $df *= dt;
  $u  += $df;
  $t  += $dt;
  $i++;
  if (!($i % 10)) {
    line $p->slice("(0)");
    print "Step $i Time $t step $dt total ",sum($p->slice("(0)")),"\n";
  }
}

# Problem-specific functions

# Lax method flux for wave equation.
sub flux {
  my ($lp,$rp,$f) = @_;
  plus($lp,$rp,$f->slice("1:0"),0);
  $f *= 0.5;
  # Add artificial viscosity
  $f += $lp; $f -= $rp;
}

sub utop { my ($u, $p) = @_; $p .= $u; }
sub ptou { my ($p, $u) = @_; $u .= $p; }
sub maxv { 1.; }
```

Table 6.2: Example numerical simulation: the one dimensional wave equation, using the Lax method. Part two: the algorithm.

have set the cells beyond the grid to have the same value of ϕ, but the opposite value of ρ: the effect of this is that waves are relected from the ends of the grid.

Then, the flux subroutine is called, using two different 'views' of the grid data: `$eleftp`, the data value in the cell at the left side of each interface, and `$erghtp`, the data at the right side. The flux function 'injects' the values of the flux into a pre-existing pdl, `$f`. In the present example, the flux is a simple function of the data and so we have coded the flux function in perl: often it will be far more efficient to do this using PP.

The reason for 'injecting' the flux data into `$f` is made clear in the subsequent lines. We could have let the flux function automatically generate a fresh PDL with the flux values. However, we have already prepared two different views of the original flux array at the beginning of the routine: `$leftf`, the flux through the left edge of each cell, and `rghtf`, the flux through the right edge. The difference between these fluxes is then calculated using `minus`: both `$df = $leftf-$rghtf` and `$df .= $leftf-$rghtf` would allocate a new array at each timestep, the second as an intermediate array which would then have to be copied into `$df`.

The flux difference is then multiplied by the timestep and added to the conserved value PDL to update the contents for the new tiemstep; the timestep and step counter are incremented to correspond to this evolved state. Finally, a plot of the data is made every 10 timesteps, to illustrate how the grid data is evolving.

Studying the development of the solution in this example, you will see that the total amount of gas within the grid remains constant to high accuracy. The algorithm was designed to achieve this, which is satisfying. The waves which move apart also have reasonably constant amplitude, which is good. However, the initially sharp shape of the solution rapidly decays to something rather smoother, where in the equations we are modelling the waveforms should remain sharp. All numerical algorithms of the form we have discussed have some of this broadening, called numerical diffusion: somewhat more complex higher order algorithms can nevertheless improve this performance dramatically.

6.3.2 Multiple dimensions and higher order

The simple algorithm defined in the previous subsection can be improved significantly by changing the algorithm to assume that there is a gradient of density of the conserved variables within the grid. The effect of this is to maintain sharp features in the grid data with far less artificial diffusion.

The version I will now show has also been generalized to two spatial dimensions – the further extension of this to even higher dimensional spaces is fairly simple, although the computational time requirements increase so rapidly that even going as far as three dimensions will slow things down dramatically.

I should emphasise again that the algorithm used here is not really of a quality to be used in practise. In addition, there are various parts of the algorithm

presented here which might better be transferred into PP for 'production' runs.

I have tried to keep the method as similar as possible to the first example. In Table 6.3, the first part is shown, in which various data arrays are defined: the dimensions of the computational grid are set in the array `@d`, a flag variable `$second` says whether the code should run with second order accuracy (in space and time). An additional conserved value is required to treat the two-dimensional wave equation. Additional arrays are used to save partial sums of fluxes and gradient values. The boundary constant becomes an array, with an element for each dimension.

In Table 6.4, many slices of the data piddles are defined. In particular, one slice is defined for each dimension of interest, so we can treat the several dimensions of the physical problem by looping over the elements in these arrays. Most if the slices are fairly obviously related to those used in the one-dimensional algorithm. I move the dimension of interest to be the first spatial dimension in the first line so that algorithm looks the same in each dimension: this move is unwound in the definition of the flux arrays `@leftf` and `$rghtf`, so the correct components of the conserved values are updated.

Numerous extra slices are also defined to help in the calculation of the gradient arrays for the second-order algorithm. The components of `$ga` contain the differences between physical values between cells neighbouring a given edge of the grid. The values on opposite edges of each grid cell are then combined in the elements of `$gb` to give an average gradient within the grid cell.

In Table 6.5, we show the hub of the algorithm. The first part is just the same as the first-order algorithm, iterated over dimensions to turn it into a multidimensional code. The is all that happens if `$second` is not set.

If `$second` is set, however, the first-order result is just used to update the physical values to provisional half time-step values, so a leapfrog scheme can then update the results with a correction which is second-order in time. The second order step is similar to the first order one, except that in the flux calculation, the primitive variables on either side of the grid edge are corrected for the gradient calculated within the grid cell.

The gradient is calculated from those at each edge of the grid cell using an averaging function. This could just be the arithmetic mean (in which case the gradient within the cell would just be a centred difference between the primitive values in the neighbours of the cell). A somewhat different, 'limited', value is actually defined in the function `av` (see Table 6.6: such limited averages have better properties in the presence of sharp discontinuities in the solution, preventing ringing by decreasing the order of the simulation just in the neighbourhood of the discontinuity.

In Table 6.6, gives a few additional functions for this algorithm. While `utop()`, `ptou()` and `maxv()` are unchanged, `flux()` has had to be altered to take an extra argument, the direction of the current sweep of integration, and to cater for the extra conserved variable. The awkwardness of this function and the new

```
# 2D wave equation using Lax method extended to second order

use PDL;
use PDL::Graphics::PGPLOT;

*plus = \&PDL::plus;
*minus = \&PDL::minus;

@d = (50,50);
$nd = $#d+1;
$cour  = 0.2;
$second = 1;
$nq = 3;
my ($inside) = ",1:-1" x $#d;
my ($inside2) = ",1:-2" x $#d;

$pb = zeroes($nq,map($_+2,@d[0..$#d]))->float;
                                          # Primitive data with boundary
$p  = $pb->slice(",1:-2"x$nd);            # Primitive data in active grid
$u  = zeroes($p);                         # Conserved data
$f  = zeroes($pb->slice(",1:-1"x$nd));    # Fluxes
$df = zeroes($p);                         # Flux differences
$dft = zeroes($p);                        # Flux differences
$ga = zeroes($f);                         # Difference array for grad
$gb = zeroes($pb);                        # Array of gradients

# utility constant for boundary
if ($nq == 2) {
  @bc = (pdl (1,-1));
} else {
  @bc = (pdl (1,-1,1),pdl(1,1,-1));
}

$cwave = 1;

# set density
($t = $p->slice("(0)"))  .= 1;
# and rate of change
($t = $p->slice("1:-1")) .= 0;
# and initial perturbation
($t = $p->slice("(0),0:".($d[0]/2-1).",0:".($d[1]/2-1))) .= 2;
ptou($p,$u);

$t = 0.;
$i = 0;
```

Table 6.3: The two dimensional wave equation, using the Lax method extended to second-order. Part One: defining constants and allocating array space.

```
# Ensure slice arrays are empty
$#leftb = $#rghtb = $#lefti = $#rghti = $#eleftp = $#erghtp =
  $#ft = $#leftf= $#rghtf = -1;
if ($second) {
  $#ga = $#leftga = $#rghtga = $#gt = $#gg = $#leftgbf = $#rghtgbf =
    $#leftgbb = $#rghtgbb = $#leftgbi = $#rghtgbi = -1;
}

foreach (1..$nd) {
  my ($tt) = $pb->mv($_,1)->slice(",$inside2");
  unshift @leftb, $tt->slice(",0:0");        # Left boundary cell
  unshift @rghtb, $tt->slice(",-1:-1");      # Right boundary cell
  unshift @lefti, $tt->slice(",1:1");        # Left inside cell
  unshift @rghti, $tt->slice(",-2:-2");      # Right inside cell
  unshift @eleftp, $tt->slice(",0:-2");      # Edge left values
  unshift @erghtp, $tt->slice(",1:-1");      # Edge right values

  $tt = $f->mv($_,1)->slice(",$inside");
  unshift @ft, $tt;
  unshift @leftf, $tt->slice(",0:-2")->mv(1,$_);        # Left flux
  unshift @rghtf, $tt->slice(",1:-1")->mv(1,$_);        # Right flux

  if ($second) {
    # Gradient slices for second order
    $tt = $ga->mv($_,1)->slice(",$inside");
    unshift @ga, $tt;
    unshift @leftga, $tt->slice(",0:-2");
    unshift @rghtga, $tt->slice(",1:-1");
    $tt = $gb->mv($_,1)->slice(",$inside2");
    unshift @gt, $tt;
    unshift @gg, $tt->slice(",1:-2");
    unshift @leftgbf, $tt->slice(",0:-2");
    unshift @rghtgbf, $tt->slice(",1:-1");
    unshift @leftgbb, $tt->slice(",(0)");
    unshift @rghtgbb, $tt->slice(",(-1)");
    unshift @leftgbi, $tt->slice(",(1)");
    unshift @rghtgbi, $tt->slice(",(-2)");
  }
}
```

Table 6.4: The two dimensional wave equation, using the Lax method extended to second-order. Part Two: sliced arrays.

```
# Second-order algorithm
while (1) {
  utop($u,$p);
  $dt = $cour/maxv($p);

  # First order stage to generate half-timestep values
  foreach (0..$#d) {
    $leftb[$_] .= $bc[$_]*$lefti[$_];              # Set boundary cells
    $rghtb[$_] .= $bc[$_]*$rghti[$_];
    # Calculate fluxes
    flux(1+$_,$eleftp[$_],$erghtp[$_],$ft[$_]);
    if ($_) {
      minus($leftf[$_],$rghtf[$_],$dft,0);
      $df += $dft;
    } else {
      minus($leftf[$_],$rghtf[$_],$df,0);
    }
  }

  if ($second) {
    # Second order stage
    # Generate half-timestep values
    $df *= 0.5*$dt;
    $df += $u;
    utop($df,$p);
    foreach (0..$#d) {
      $leftb[$_] .= $bc[$_]*$lefti[$_];     # Set boundary cells
      $rghtb[$_] .= $bc[$_]*$rghti[$_];
      $ga[$_]    .= $erghtp[$_];
      $ga[$_]    -= $eleftp[$_];
      av($leftga[$_],$rghtga[$_],$gg[$_]);
      $leftgbb[$_] .= $bc[$_]*$leftgbi[$_];
      $rghtgbb[$_] .= $bc[$_]*$rghtgbi[$_];
      flux(1+$_,$eleftp[$_]+0.5*$leftgbf[$_],$erghtp[$_]-0.5*$rghtgbf[$_],
   $ft[$_]);
      if ($_) {
        minus($leftf[$_],$rghtf[$_],$dft,0);
        $df += $dft;
      } else {
        minus($leftf[$_],$rghtf[$_],$df,0);
      }
    }
  }

  $df *= $dt;
  $u += $df;
  $t  += $dt;
  $i++;
  if ($i % 10) {
    if ($#d == 0) {
      line $p->slice("(0)," . ((",(10)") x $#d));
    } else {
      imag $p->slice("(0)");
    }
    print "Step $i Time $t step $dt total ",sum($p->slice("(0)")),"\n";
  }
```

```
# Problem-specific functions
sub flux {
  my ($dirn,$lp,$rp,$f) = @_;
  # Lax method flux for wave equation.

  plus($lp->slice("0"),$rp->slice("0"),$f->slice("$dirn"),0);
  plus($lp->slice("$dirn"),$rp->slice("$dirn"),$f->slice("0"),0);
  my ($t) = $f->slice(3-$dirn); $t .= 0;
  $f *= 0.5;
  # Add artificial viscosity
  ($t = $f->slice("0")) += $lp->slice("0");         $t -= $rp->slice("0");
  ($t = $f->slice("$dirn")) += $lp->slice("$dirn"); $t -= $rp->slice("$dirn");
}
sub utop { my ($u, $p) = @_; $p .= $u; }
sub ptou { my ($p, $u) = @_; $u .= $p; }
sub maxv { 1.; }

sub av {
  my ($a, $b, $ret) = @_;
  $a = $a->clump(-1); $b=$b->clump(-1); $ret=$ret->clump(-1);
  my ($prod) = $a*$b;
  $ret .= 0;
  my ($ta,$tb,$tret,$tprod) = where ($a,$b,$ret,$prod,$prod>0);
  $tret .= ($tprod*($ta+$tb)/($ta*$ta+$tb*$tb));
}
```

Table 6.6: The two dimensional wave equation, using the Lax method extended to second-order. Part Four: auxiliary functions

function `av()` (not to mention their slowness in practise!), indicate that they would best be implemented in PP.

6.3.3 Gas dynamics

The dynamics of a perfect gas is treated in the PDL::Hydro package in the PDL distribution. By putting `use PDL::Hydro;` at the head of the evolution codes above, I can load the `flux()`, `utop()`, `ptou()`, `maxv()` and `av()` routines provided by this package — the versions provided for the wave equation must be removed. The value of `$nq` needs to be increased (to provide space for density, velocity and pressure variables), and `$bc` must be changed to an appropriate value. Finally, it is better to change the initial conditions to having a step in pressure rather than density (`($t =$p->slice("(2),0:".($ncell/2-1))) .= 2;`) — otherwise, the initial grid contains are an equilibrium, and the values in the grid will just remain constant!

The `flux()` routine treats the interaction between neighbouring cells as a shock tube problem between uniform states, and calculates the flux across the interface between the cells by an exact method. This approach, based on a method introduced by Godunov, is used in many widely used gas- and hydrodynamics algorithms. By default, `PDL::Hydro` assumes the gas being studied is adiabatic, with a adiabatic constant of 5/3 appropriate for monatomic gas. A value of 7/5 is perhaps more typical for air, for instance, and this may be set by including `setadc(1.4)` at the start of the routine.

In a typical one-dimensional simulation (Figure XXX), the high pressure to the left of the grid drives a sharp shock to the right, and a rarefaction moves to the left (this rarefaction is smooth in the full solution). In between there remains a contact discontinuity between hot and cool gas. In reality, this contact discontinuity should remain sharp: its smooth form is again the result of the numerical viscosity introduced by this low order scheme.

6.3.4 Ordinary Differential Equations

– initial value problems by RK

– boundary value problems, via PDL::Func

A – Getting PDL

NAME

Getting/Downloading PDL

Description

The following is an overview of the ways to download PDL and get it up and running. Much information is available on the PDL web-site (pdl.perl.org), and in various other places. Where possible, we will provide pointers to the information, but not repeat the information here.

Download Options

There are primarily two methods to Download/Install the PDL package; Installing from pre-built executable files, or building from the source code. Installing from pre-built executables is usually faster and more convenient for most users. Building from source is more complicated, but gets you the greatest configuration flexibility (Add on packages, graphics options, etc)

Pre-Built Binary Formats

- RPM (Redhat Package Manager) File. This is a pre-built executable format that is very convenient for Linux Platforms.
- DEB (Debian) format for the Debian operating system.
- PPM (Perl Package Manager) File. This is a pre-build executable for Win32 Platforms. Note: The Win32 port is considered experimental at the time of this writing.

Other pre-built executables are being added all the time (BSD, Solaris, etc.) Check the download page at pdl.perl.org for the latest list.

Building from source.

Try one of the pre-built binaries first, if you can. You will get going quicker. Building from source is an option for those who want a custom configuration, or who want to examine the workings of the source code.

For most all Unix flavors, building from source is a matter of following the standard Perl package build procedure:

1.) **perl Makefile.PL**
2.) **make**
3.) **make test**
4.) **make install**

The INSTALL file in the source distribution describes the installation procedure in detail, along with some common problems and solutions.

For Win32 platforms, building from source is possible, but not recommended at the time of this writing. Again, using one of the pre-built executables is the way to go for this platform.

Download Locations:

The download page of the PDL Web-site (pdl.perl.org) contains the complete list of places you can get PDL. Additionally, CPAN (the Comprehensive Perl Archive Network, at http://www.perl.com/CPAN-local/modules/by-module/PDL/) contains the official release of the source code.

PDL Add-Ons and Dependencies

PDL can be configured to use external analysis and graphical libraries. Some of the more popular examples are PGPLOT (for 2D Plotting), OpenGL (for 3-D Plotting), Slatec (Fortran Analysis Library). The DEPENDENCIES file in the source distribution contains a complete list of possible add-ons for PDL.

Appendix B - PDL modules and functions index

List of PDL modules

PDL Core Routines

`PDL::Basic`
: Basic utility functions for PDL

`PDL::Core`
: fundamental PDL functionality

`PDL::Math`
: extended mathematical operations and special functions

`PDL::Primitive`
: primitive operations for pdl

`PDL::Slices`
: Stupid index tricks

PDL I/O functions

`PDL::IO::FastRaw`
: A simple, fast and convenient io format for PerlDL.

`PDL::IO::FlexRaw`
: A flexible binary i/o format for PerlDL.

`PDL::IO::Misc`
: misc IO routines for PDL

`PDL::IO::NDF`
: PDL Module for reading and writing Starlink

`PDL::IO::Pic`
: image I/O for PDL

`PDL::IO::Pnm`
: pnm format I/O for PDL

PDL Graphics functions

`PDL::Graphics::IIS`
: Display PDL images on IIS devices (saoimage/ximtool)

`PDL::Graphics::Karma`
: interface to Karma visualisation applications

`PDL::Graphics::LUT`
: provides access to a number of look-up tables

`PDL::Graphics::OpenGLQ`
: quick routines to plot lots of stuff from piddles.

`PDL::Graphics::PGPLOT`
: PGPLOT enhanced interface for PDL

`PDL::Graphics::TriD`
: PDL 3D interface

`PDL::Graphics::TriD::Rout`
: Helper routines for Three-dimensional graphics

`PDL::Graphics::TriD::Tk`
: A Tk widget interface to the PDL::Graphics::TriD.

PDL Library functions

`PDL::FFT`
: FFTs for PDL

`PDL::FFTW`
: PDL interface to the Fastest Fourier Transform in the West v2.x

`PDL::Fit::Gaussian`
: routines for fitting gaussians

`PDL::Fit::LM`
: Levenber-Marquardt fitting routine for PDL

`PDL::Image2D`
: Miscellaneous 2D image processing functions

`PDL::ImageND`
: useful image processing routines which work in N-dimensions

`PDL::ImageRGB`
: some utility functions for RGB image data handling

`PDL::Slatec`
: PDL interface to the slatec numerical programming library

PDL development related routines

`PDL::Dbg`
: functions to support debugging of PDL scripts

`PDL::Doc::Perldl`
: commands for accessing PDL doc database from 'perldl' shell

PDL routines of derived classes

`PDL::Char`
: PDL subclass which allows reading and writing of fixed-length character strings as byte PDLs

`PDL::Complex`
: handle complex numbers

`PDL::Func`
: useful functions

List of functions by category

PDL Core Routines

PDL::Basic
: allaxisvals, axisvals, hist, rvals, sequence, transpose, xlinvals, xvals, ylinvals, yvals, zlinvals, zvals

PDL::Core
: at, barf, byte, cat, convert, copy, Datatype_conversions, diagonal, dims, doflow, dog, double, dummy, float, flows, get_datatype, getdim, gethdr, getndims, hdrcpy, howbig, info, inplace, isempty, list, listindices, long, make_physical, mslice, nelem, new, new_from_specification, null, nullcreate, ones, pdl, reshape, set, sethdr, short, thread, thread1, thread2, thread3, thread_define, threadids, topdl, type, unwind, ushort, zeroes

PDL::Math
: acos, acosh, asin, asinh, atan, atanh, badmask, bessj0, bessj1, bessjn, bessy0, bessy1, bessyn, ceil, cosh, eigens, erf, erfc, erfi, floor, lgamma, ndtri, pow, rint, simq, sinh, squaretotri, svd, tan, tanh

PDL::Primitive
: all, and, andover, any, append, assgn, average, avg, axisvalues, band, bandover, bor, borover, clip, crossp, cumuprodover, cumusumover, fibonacci, grandom, hclip, histogram, histogram2d, indadd, inner, inner2, inner2d, inner2t, innerwt, interpol, interpolate, intover, lclip, matmult, max, maximum, maximum_ind, maximum_n_ind, median, medover, min, minimum,

minimum_ind, minimum_n_ind, minmax, minmaximum, norm, oddmedian, oddmedover, one2nd, or, orover, outer, prodover, qsort, qsorti, random, randsym, stats, sum, sumover, vsearch, where, which, which_both, whichND, whistogram, whistogram2d, wtstat, zcheck, zcover

PDL::Slices
clump, diagonalI, dice, dice_axis, identvaff, index, index2d, lags, mv, oneslice, reorder, rld, rle, rotate, slice, splitdim, unthread, using, xchg

PDL I/O functions

PDL::IO::FastRaw
mapfraw, maptextfraw, readfraw, writefraw

PDL::IO::FlexRaw
mapflex, readflex, writeflex

PDL::IO::Misc
bswap2, bswap4, bswap8, isbigendian, rasc, rcols, rcube, rdsa, rfits, rgrep, wcols, wfits

PDL::IO::NDF
propndfx, rndf, wndf

PDL::IO::Pic
rpic, rpiccan, rpiccan, wmpeg, wpic

PDL::IO::Pnm
pnminascii, pnminraw, pnmout, rpnm, wpnm

PDL Graphics functions

PDL::Graphics::IIS
iis, iiscirc, iiscur, saoimage, ximtool

PDL::Graphics::Karma
kcur, kim, koords, koverlay, kpvslice, krenzo, krgb, kshell, kslice_3d, kstarted, kview, xray

PDL::Graphics::LUT
lut_data, lut_names, lut_ramps

PDL::Graphics::OpenGLQ

PDL::Graphics::PGPLOT
bin, cont, ctab, dev, env, errb, hi2d, imag, imag1, line, points, poly, vect

PDL::Graphics::TriD
attract, buttonmotion, combcoords, contour_segments, GLinit, grabpic3d, hold3d, hold3d, imag3d, imagrgb, imagrgb3d, keeptwiddling3d, keeptwiddling3d, lattice3d, line3d, mesh3d, points3d, Populate, refresh, repulse, twiddle3d, vrmlcoordsvert

PDL::Graphics::TriD::Rout
attract, combcoords, contour_segments, repulse, vrmlcoordsvert

PDL::Graphics::TriD::Tk
buttonmotion, GLinit, Populate, refresh

PDL Library functions

PDL::FFT
Carg, Cbconj, Cbexp, Cbmul, Cbscale, Cconj, Cdiv, cdiv, Cexp, Cmod, Cmod2, Cmul, cmul, convmath, Cscale, fft, fftconvolve, fftnd, fftw, fftwconv, ifft, ifftnd, ifftw, infftw, inrfftw, irfftw, kernctr, load_wisdom, nfftw, nrfftw, realfft, realifft, rfftw, rfftwconv

PDL::FFTW
Carg, Cbconj, Cbexp, Cbmul, Cbscale, Cconj, Cdiv, Cexp, Cmod, Cmod2, Cmul, Cscale, fftw, fftwconv, ifftw, infftw, inrfftw, irfftw, load_wisdom, nfftw, nrfftw, rfftw, rfftwconv

PDL::Fit::Gaussian
fitgauss1d, fitgauss1dr

PDL::Fit::LM
lmfit, tlmfit

PDL::Image2D
bilin2d, cc8compt, centroid2d, conv2d, max2d_ind, med2d, patch2d, polyfill, polyfillv, rescale2d, rot2d

PDL::ImageND
circ_mean, circ_mean_p, convolve, ninterpol, rebin

PDL::ImageRGB
bytescl, cquant, interlrgb, rgbtogr

PDL::Slatec
chcm, chfd, chfe, chia, chic, chid, chim, chsp, eigsys, ezfftb, ezfftf, ezffti, geco, gedi, gefa, gesl, matinv, pcoef, poco, podi, polfit, polycoef, polyfit, polyvalue, pvalue, rs, svdc

PDL development related routines

PDL::Dbg
px, vars

PDL::Doc::Perldl
apropos, help, sig, usage

PDL routines of derived classes

PDL::Char
atstr, new, setstr, string

PDL::Complex
Cabs, Cabs2, Cacos, Cacosh, Casin, Casinh, Catan, Catanh, Ccmp, Ccos, Ccosh, Clog, Cp2r, cplx, Cpow, Cproj, Cr2p, Croots, Csin, Csinh, Csqrt, Ctan, Ctanh, i2C, r2C, rCpolynomial, re, re, real

PDL::Func
attributes, get, gradient, init, integrate, interpolate, routine, scheme, set, status

Alphabetical Listing of PDL Functions

abs Module: PDL::Ops

```
Signature: (a(); [o]b())
```

elementwise absolut value

```
$b = abs $a;
$a->inplace->abs;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary `abs` operator/function.

acos
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The usual trigonometric function.

acosh
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard hyperbolic function.

all Module: PDL::Primitive

Return true if all elements in piddle set

Useful in conditional expressions:

```
if (all $a>15) { print "all values are greater than 15\n" }
```

allaxisvals
Module: PDL::Basic

Generates a piddle with index values

```
$z = allaxisvals ($piddle);
```

`allaxisvals` produces an array with axis values along each dimension, adding an extra dimension at the start.

C<*allaxisvals*($piddle)-¿*slice*("($nth)")> will produce the same result as `axisvals($piddle,$nth)` (although with extra work and not inplace).

It's useful when all the values will be required, as in the example given of a generalized the `rvals` entry in the *rvals*| manpage.

and Module: PDL::Primitive

Return the logical and of all elements in a piddle

```
$x = and($data);
```

and2
Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

binary *and* of two piddles

```
$c = and2 $a, $b, 0;     # explicit call with trailing 0
$c = $a & $b;           # overloaded call
$a->inplace->and2($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `&` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

andover
Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via and to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the and along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = andover($b);

$spectrum = andover $image->xchg(0,1)
```

any Module: PDL::Primitive

Return true if any element in piddle set

Useful in conditional expressions:

```
if (any $a>15) { print "some values are greater than 15\n" }
```

append

Module: PDL::Primitive

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

append two piddles by concantening along their respective first dimensions

```
$a = ones(2,4,7);
$b = sequence 5;
$c = $a->append($b);  # size of $c is now (7,4,7) (a jumbo-piddle ;)
```

`append` appends two piddles along their first dims. Rest of the dimensions must be compatible in the threading sense. Resulting size of first dim is sum of sizes of the two argument piddles' first dims.

apropos

Module: PDL::Doc::Perldl

Regex search PDL documentation database

```
apropos 'text'

perldl> apropos 'pic'
rpic            Read images in many formats with automatic format detection.
rpiccan         Test which image formats can be read/written
wmpeg           Write an image sequence ((x,y,n) piddle) as an MPEG animation.
wpic            Write images in many formats with automatic format selection.
wpiccan         Test which image formats can be read/written
```

To find all the manuals that come with PDL, try

```
apropos 'manual:'
```

and to get quick info about PDL modules say

```
apropos 'module:'
```

You get more detailed info about a PDL function/module/manual with the `help` function

asin
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The usual trigonometric function.

asinh
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard hyperbolic function.

assgn
Module: PDL::Primitive

```
Signature: (a(); [o]b())
```

Plain numerical assignment. This is used to implement the ".=" operator

at Module: PDL::Core

Returns a single value inside a piddle as perl scalar.

```
$z = at($piddle, @position); $z=$piddle->at(@position);
```

`@position` is a coordinate list, of size equal to the number of dimensions in the piddle. Occasionally useful in a general context, quite useful too inside PDL internals.

```
perldl> $x = sequence 3,4
perldl> p $x->at(1,2)
7
```

atan
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The usual trigonometric function.

atan2

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

elementwise `atan2` of two piddles

```
$c = $a->atan2($b,0); # explicit function call
$c = atan2 $a, $b;    # overloaded use
$a->inplace->atan2($b,0);     # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `atan2` function. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

atanh

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard hyperbolic function.

atstr

Module: PDL::Char

Function to fetch one string value from a PDL::Char type PDL, given a position within the PDL. The input position of the string, not a character in the string. The length of the input string is the implied first dimension.

```
$char = PDL::Char->new( [['abc', 'def', 'ghi'], ['jkl', 'mno', 'pqr']] );
print $char->atstr(0,1);
# Prints:
# jkl
```

attract

Module: PDL::Graphics::TriD::Rout

```
Signature: (coords(nc,np);
               int from(nl);
               int to(nl);
               strength(nl);
               [o]vecs(nc,np);;
               double m;
               double ms;
      )
```

Attractive potential for molecule-like constructs.

`attract` is used to calculate an attractive force for many objects, of which some attract each other (in a way like molecular bonds). For use by the module the *PDL::Graphics::TriD::MathGraph|PDL::Graphics::TriD::MathGraph* manpage. For definition of the potential, see the actual function.

attributes

Module: PDL::Func

```
$obj->attributes;
PDL::Func->attributes;
```

Print out the flags for the attributes of a PDL::Func object.

Useful in case the documentation is just too opaque!

```
PDL::Func->attributes;
Flags  Attribute
 SGR    x
 SGR    y
 G      err
```

average

Module: PDL::Primitive

```
 Signature: (a(n); int+ [o]b())
```

Project via average to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the average along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = average($b);

$spectrum = average $image->xchg(0,1)
```

avg Module: PDL::Primitive

Return the average of all elements in a piddle

```
$x = avg($data);
```

axisvals

Module: PDL::Basic

Fills a piddle with index values on Nth dimension

```
$z = axisvals ($piddle, $nth);
```

This is the routine, for which the `xvals` entry in the *xvals|* manpage, the `yvals` entry in the *yvals|* manpage etc are mere shorthands. `axisvals` can be used to fill along any dimension.

Note the 'from specification' style (see the `zeroes` entry in the *zeroes|PDL::Core* manpage) is not available here, for obvious reasons.

axisvalues

Module: PDL::Primitive

```
Signature: ([o,nc]a(n))
```

Internal routine

`axisvalues` is the internal primitive that implements the `axisvals` entry in the *axisvals|PDL::Basic* manpage and alters its argument.

badmask

Module: PDL::Math

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

Clears all `infs` and `nans` in `$a` to the corresponding value in `$b`

band

Module: PDL::Primitive

Return the bitwise and of all elements in a piddle

```
$x = band($data);
```

bandover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via bitwise and to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the bitwise and along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = bandover($b);
```

```
$spectrum = bandover $image->xchg(0,1)
```

barf

Module: PDL::Core

Standard error reporting routine for PDL.

`barf()` is the routine PDL modules should call to report errors. This is because `barf()` will report the error as coming from the correct line in the module user's script rather than in the PDL module.

It does this magic by unwinding the stack frames until it reaches a package NOT beginning with `"PDL::"`. If you DO want it to report errors in some module PDL::Foo (e.g. when debugging PDL::Foo) then set the variable `$PDL::Foo::Debugging=1`.

Additionally if you set the variable `$PDL::Debugging=1` you will get a COMPLETE stack trace back up to the top level package.

Finally `barf()` will try and report usage information from the PDL documentation database if the error message is of the form 'Usage: func'.

Remember `barf()` is your friend. ∗Use∗ it!

At the perl level:

```
barf("User has too low an IQ!");
```

In C or XS code:

```
barf("You have made %d errors", count);
```

Note: this is one of the few functions ALWAYS exported by PDL::Core

bessj0

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard Bessel Functions

bessj1

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard Bessel Functions

bessjn

Module: PDL::Math

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

The standard Bessel Functions

bessy0
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard Bessel Functions

bessy1
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard Bessel Functions

bessyn
Module: PDL::Math

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

The standard Bessel Functions

bilin2d
Module: PDL::Image2D

```
Signature: (I(n,m); O(q,p))
```

Bilineary maps the first piddle in the second. The interpolated values are actually added to the second piddle which is supposed to be larger than the first one.

bin Module: PDL::Graphics::PGPLOT

Plot vector as histogram (e.g. `bin(hist($data))`)

```
Usage: bin ( [$x,] $data )
```

Options recognised:

```
CENTRE - if true, the x values denote the centre of the bin
         otherwise they give the lower-edge (in x) of the bin
CENTER - as CENTRE
```

The following standard options influence this command:

```
AXIS, BORDER, COLOUR, JUSTIFY, LINESTYLE, LINEWIDTH
```

bitnot
Module: PDL::Ops

```
Signature: (a(); [o]b())
```

unary bit negation

```
$b = ~ $a;
$a->inplace->bitnot;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary ~ operator/function.

bor Module: PDL::Primitive

Return the bitwise or of all elements in a piddle

```
$x = bor($data);
```

borover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via bitwise or to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the bitwise or along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = borover($b);
```

```
$spectrum = borover $image->xchg(0,1)
```

browse

Module: PDL::IO::Browser

```
Signature: (a(n,m))
```

browse

Module: PDL::IO::Browser

browse a 2D array using terminal cursor keys

```
browse $data
```

This uses the CURSES library to allow one to scroll around a PDL array using the cursor keys.

bswap2

Module: PDL::IO::Misc

```
Signature: (x(); )
```

Swaps pairs of bytes in argument *x()*

bswap4
Module: PDL::IO::Misc

```
Signature: (x(); )
```

Swaps quads of bytes in argument *x()*

bswap8
Module: PDL::IO::Misc

```
Signature: (x(); )
```

Swaps octets of bytes in argument *x()*

buttonmotion
Module: PDL::Graphics::TriD::Tk

Default bindings for mousemotion with buttons 1 and 3

byte
Module: PDL::Core

Convert to byte datatype — see 'Datatype_conversions'

bytescl
Module: PDL::ImageRGB

Scales a pdl into a specified data range (default 0−255)

```
$scale = $im->bytescl([$top])
```

By default $top=255, otherwise you have to give the desired top value as an argument to `bytescl`. Normally `bytescl` doesn't rescale data that fits already in the bounds 0..$top (it only does the type conversion if required). If you want to force it to rescale so that the max of the output is at $top and the min at 0 you give a negative $top value to indicate this.

Cabs
Module: PDL::Complex

```
Signature: (a(m=2); [o]c())
```

complex `abs()` (also known as *modulus*)

Cabs2
Module: PDL::Complex

```
Signature: (a(m=2); [o]c())
```

complex squared `abs()` (also known *squared modulus*)

Cacos
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

Cacosh
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

callext
Module: PDL::CallExt

Call a function in an external library using Perl dynamic loading

```
callext('file.so', 'foofunc', $x, $y); # pass piddles to foofunc()
```

The file must be compiled with dynamic loading options (see `callext_cc`). See the module docs `PDL::Callext` for a description of the API.

callext_cc
Module: PDL::CallExt

Compile external C code for dynamic loading

Usage:

```
% perl -MPDL::CallExt -e callext_cc file.c -o file.so
```

This works portably because when Perl has built in knowledge of how to do dynamic loading on the system on which it was installed. See the module docs `PDL::Callext` for a description of the API.

Carg
Module: PDL::FFTW

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

argument of a complex number.

```
$out_pdl_real = Carg($a_pdl_cplx);
```

Casin
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

Casinh
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

cat Module: PDL::Core

concatentate piddles to N+1 dimensional piddle

Takes a list of N piddles of same shape as argument, returns a single piddle of dimension N+1

```
perldl> $x = cat ones(3,3),zeroes(3,3),rvals(3,3); p $x
[
 [
  [1 1 1]
  [1 1 1]
  [1 1 1]
 ]
 [
  [0 0 0]
  [0 0 0]
  [0 0 0]
 ]
 [
  [1 1 1]
  [1 0 1]
  [1 1 1]
 ]
]
```

Catanh
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

Cbconj
Module: PDL::FFTW

```
Signature: (a(n))
```

Complex inplace conjugate.

```
Cbconj($a_pdl_cplx);
```

Cbexp
Module: PDL::FFTW

```
Signature: (a(n))
```

Complex inplace exponentation.

```
$out_pdl_cplx = Cexp($a_pdl_cplx);
```

Cbmul
Module: PDL::FFTW

```
Signature: (a(n); b(n))
```

Complex inplace multiplication.

```
Cbmul($a_pdl_cplx,$b_pdl_cplx);
```

Cbscale
Module: PDL::FFTW

```
Signature: (a(n); b())
```

Complex inplace multiplaction by real.

```
Cbscale($a_pdl_cplx,$b_pdl_real);
```

cc8compt
Module: PDL::Image2D

```
Signature: (a(m,n); [o]b(m,n))
```

Connected 8-component labeling of a binary image.

Connected 8-component labeling of 0,1 image — i.e. find seperate segmented objects and fill object pixels with object number

```
$segmented = cc8compt( $image > $threshold );
```

Ccmp
Module: PDL::Complex

```
Signature: (a(m=2); b(m=2); [o]c())
```

Complex comparison oeprator (spaceship). It orders by real first, then by imaginary.

Cconj
Module: PDL::FFTW

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

Complex conjugate.

```
$out_pdl_cplx = Cconj($a_pdl_cplx);
```

Ccos

Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

```
cos (a) = 1/2 * (exp (a*i) + exp (-a*i))
```

Ccosh

Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

```
cosh (a) = (exp (a) + exp (-a)) / 2
```

Cdiv

Module: PDL::FFTW

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

Complex division.

```
$out_pdl_cplx = Cdiv($a_pdl_cplx,$b_pdl_cplx);
```

cdiv

Module: PDL::FFT

```
Signature: (ar(); ai(); br(); bi(); [o]cr(); [o]ci())
```

Complex division

ceil

Module: PDL::Math

```
Signature: (a(); [o]b())
```

Round to integral values in floating-point format

centroid2d

Module: PDL::Image2D

```
Signature: (im(m,n); x(); y(); box(); [o]xcen(); [o]ycen())
```

Refine a list of object positions in 2D image by centroiding in a box

`$box` is the full-width of the box, i.e. the window is `+/- $box/2`.

Cexp

Module: PDL::FFTW

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

Complex exponentation.

```
$out_pdl_cplx = Cexp($a_pdl_cplx);
```

chcm

Module: PDL::Slatec

```
Signature: (x(n);f(n);d(n);int check();int [o]ismon(n);int [o]ierr())
```

Check the given piecewise cubic Hermite function for monotonicity.

The outout piddle `$ismon` indicates over which intervals the function is monotonic. Set `check` to 0 to skip checks on the input data.

For the data interval `[x(i),x(i+1)]`, the values of `ismon(i)` can be:

- -3 if function is probably decreasing
- -1 if function is strictly decreasing
- 0 if function is constant
- 1 if function is strictly increasing
- 2 if function is non-monotonic
- 3 if function is probably increasing

If `abs(ismon(i)) == 3`, the derivative values are near the boundary of the monotonicity region. A small increase produces non-monotonicity, whereas a decrease produces strict monotonicity.

The above applies to `i = 0 .. nelem($x)-1`. The last element of `$ismon` indicates whether the entire function is monotonic over $x.

Error status returned by `$ierr`:

- 0 if successful.
- -1 if `n < 2`.
- -3 if `$x` is not strictly increasing.

chfd

Module: PDL::Slatec

```
Signature: (x(n);f(n);d(n);int check();xe(ne);[o]fe(ne);[o]de(ne);int [o]ierr())
```

Interpolate function and derivative values.

Given a piecewise cubic Hermite function — such as from the `chim` entry in the *chim*| manpage — evaluate the function (`$fe`) and derivative (`$de`) at a set of points (`$xe`). If function values alone are required, use the `chfe` entry in the *chfe*| manpage. Set `check` to 0 to skip checks on the input data.

Error status returned by `$ierr`:

- 0 if successful.
- ¿0 if extrapolation was performed at `ierr` points (data still valid).
- -1 if C<*nelem*($x) ¡ 2>
- -3 if `$x` is not strictly increasing.
- -4 if C<*nelem*($xe) ¡ 1>.
- -5 if an error has occurred in a lower-level routine, which should never happen.

chfe

Module: PDL::Slatec

```
Signature: (x(n);f(n);d(n);int check();xe(ne);[o]fe(ne);int [o]ierr())
```

Interpolate function values.

Given a piecewise cubic Hermite function — such as from the `chim` entry in the *chim*| manpage — evaluate the function (`$fe`) at a set of points (`$xe`). If derivative values are also required, use the `chfd` entry in the *chfd*| manpage. Set `check` to 0 to skip checks on the input data.

Error status returned by `$ierr`:

- 0 if successful.
- ¿0 if extrapolation was performed at `ierr` points (data still valid).
- -1 if C<*nelem*($x) ¡ 2>
- -3 if `$x` is not strictly increasing.
- -4 if C<*nelem*($xe) ¡ 1>.

chia

Module: PDL::Slatec

```
Signature: (x(n);f(n);d(n);int check();a();b();[o]ans();int [o]ierr())
```

Integrate (x,f(x)) over arbitrary limits.

Evaluate the definite integral of a a piecewise cubic Hermite function over an arbitrary interval, given by `[$a,$b]`. See the `chid` entry in the *chid*| manpage if the integration limits are data points. Set `check` to 0 to skip checks on the input data.

The values of `$a` and `$b` do not have to lie within `$x`, although the resulting integral value will be highly suspect if they are not.

Error status returned by `$ierr`:

- 0 if successful.
- 1 if `$a` lies outside `$x`.
- 2 if `$b` lies outside `$x`.
- 3 if both 1 and 2 are true.
- -1 if C<*nelem*($x) ¡ 2>
- -3 if `$x` is not strictly increasing.
- -4 if an error has occurred in a lower-level routine, which should never happen.

chic

Module: PDL::Slatec

```
Signature: (int ic(two=2);vc(two=2);mflag();x(n);f(n);[o]d(n);wk(nwk);int [o]ierr())
```

Calculate the derivatives of (x,f(x)) using cubic Hermite interpolation.

Calculate the derivatives at the given points (`$x,$f`, where `$x` is strictly increasing). Control over the boundary conditions is given by the `$ic` and `$vc` piddles, and the value of `$mflag` determines the treatment of points where monotoncity switches direction. A simpler, more restricted, interface is available using the `chim` entry in the *chim*| manpage.

The first and second elements of `$ic` determine the boundary conditions at the start and end of the data respectively. If the value is 0, then the default condition, as used by the `chim` entry in the *chim*| manpage, is adopted. If greater than zero, no adjustment for monotonicity is made, otherwise if less than zero the derivative will be adjusted. The allowed magnitudes for `ic(0)` are:

- 1 if first derivative at `x(0)` is given in `vc(0)`.
- 2 if second derivative at `x(0)` is given in `vc(0)`.
- 3 to use the 3-point difference formula for `d(0)`. (Reverts to the default b.c. if `n < 3`)
- 4 to use the 4-point difference formula for `d(0)`. (Reverts to the default b.c. if `n < 4`)

- 5 to set `d(0)` so that the second derivative is continuous at `x(1)`. (Reverts to the default b.c. if `n < 4`)

The values for `ic(1)` are the same as above, except that the first-derivative value is stored in `vc(1)` for cases 1 and 2. The values of `$vc` need only be set if options 1 or 2 are chosen for `$ic`.

Set `$mflag = 0` if interpolant is required to be monotonic in each interval, regardless of the data. This causes `$d` to be set to 0 at all switch points. Set `$mflag` to be non-zero to use a formula based on the 3-point difference formula at switch points. If `$mflag > 0`, then the interpolant at swich points is forced to not deviate from the data by more than `$mflag*dfloc`, where `dfloc` is the maximum of the change of `$f` on this interval and its two immediate neighbours. If `$mflag < 0`, no such control is to be imposed.

The piddle `$wk` is only needed for work space. However, I could not get it to work as a temporary variable, so you must supply it; it is a 1D piddle with `2*n` elements.

Error status returned by `$ierr`:

- 0 if successful.
- 1 if C<*ic*(0) ¡ 0> and `d(0)` had to be adjusted for monotonicity.
- 2 if C<*ic*(1) ¡ 0> and `d(n-1)` had to be adjusted for monotonicity.
- 3 if both 1 and 2 are true.
- -1 if `n < 2`.
- -3 if `$x` is not strictly increasing.
- -4 if C<*abs*(ic(0)) ¿ 5>.
- -5 if C<*abs*(ic(1)) ¿ 5>.
- -6 if both -4 and -5 are true.
- -7 if `nwk < 2*(n-1)`.

chid

Module: PDL::Slatec

```
Signature: (x(n);f(n);d(n);int check();int ia();int ib();[o]ans();int [o]ierr())
```

Integrate (x,f(x)) between data points.

Evaluate the definite integral of a a piecewise cubic Hermite function between `x($ia)` and `x($ib)`.

See the `chia` entry in the *chia|* manpage for integration between arbitrary limits.

Although using a fortran routine, the values of `$ia` and `$ib` are zero offset. Set `check` to 0 to skip checks on the input data.

Error status returned by `$ierr`:

- 0 if successful.
- -1 if C<*nelem*($x) ¡ 2>.
- -3 if `$x` is not strictly increasing.
- -4 if `$ia` or `$ib` is out of range.

chim

Module: PDL::Slatec

```
Signature: (x(n);f(n);[o]d(n);int [o]ierr())
```

Calculate the derivatives of (x,*f*(x)) using cubic Hermite interpolation.

Calculate the derivatives at the given set of points (`$x,$f`, where `$x` is strictly increasing). The resulting set of points — `$x,$f,$d`, referred to as the cubic Hermite representation — can then be used in other functions, such as the `chfe` entry in the *chfe*| manpage, the `chfd` entry in the *chfd*| manpage, and the `chia` entry in the *chia*| manpage.

The boundary conditions are compatible with monotonicity, and if the data are only piecewise monotonic, the interpolant will have an extremum at the switch points; for more control over these issues use the `chic` entry in the *chic*| manpage.

Error status returned by `$ierr`:

- 0 if successful.
- ¿ 0 if there were `ierr` switches in the direction of monotonicity (data still valid).
- -1 if C<*nelem*($x) ¡ 2>.
- -3 if `$x` is not strictly increasing.

chsp

Module: PDL::Slatec

```
Signature: (int ic(two=2);vc(two=2);x(n);f(n);[o]d(n);wk(nwk);int [o]ierr())
```

Calculate the derivatives of (x,*f*(x)) using cubic spline interpolation.

Calculate the derivatives, using cubic spline interpolation, at the given points (`$x,$f`), with the specified boundary conditions. Control over the boundary conditions is given by the `$ic` and `$vc` piddles. The resulting values — `$x,$f,$d` - can be used in all the functions which expect a cubic Hermite function.

The first and second elements of `$ic` determine the boundary conditions at the start and end of the data respectively. The allowed values for `ic(0)` are:

- 0 to set `d(0)` so that the third derivative is continuous at `x(1)`.

- 1 if first derivative at `x(0)` is given in `vc(0)`.
- 2 if second derivative at `x(0)` is given in `vc(0)`.
- 3 to use the 3-point difference formula for `d(0)`. (Reverts to the default b.c. if `n < 3`.)
- 4 to use the 4-point difference formula for `d(0)`. (Reverts to the default b.c. if `n < 4`.)

The values for `ic(1)` are the same as above, except that the first-derivative value is stored in `vc(1)` for cases 1 and 2. The values of `$vc` need only be set if options 1 or 2 are chosen for `$ic`.

The piddle `$wk` is only needed for work space. However, I could not get it to work as a temporary variable, so you must supply it; it is a 1D piddle with `2*n` elements.

Error status returned by `$ierr`:

- 0 if successful.
- -1 if C<*nelem*($x) ¡ 2>.
- -3 if `$x` is not strictly increasing.
- -4 if C<*ic*(0) ¡ 0> or C<*ic*(0) ¿ 4>.
- -5 if C<*ic*(1) ¡ 0> or C<*ic*(1) ¿ 4>.
- -6 if both of the above are true.
- -7 if `nwk < 2*n`.
- -8 in case of trouble solving the linear system for the interior derivative values.

circ_mean

Module: PDL::ImageND

Smooths an image by applying circular mean. Optionally takes the center to be used.

```
circ_mean($im);
circ_mean($im,{Center => [10,10]});
```

circ_mean_p

Module: PDL::ImageND

Calculates the circular mean of an n-dim image and returns the projection. Optionally takes the center to be used.

```
$cmean=circ_mean_p($im);
$cmean=circ_mean_p($im,{Center => [10,10]});
```

clip

Module: PDL::Primitive

Clip a piddle by (optional) upper or lower bounds.

```
$b = $a->clip(0,3);
$c = $a->clip(undef, $x);
```

Clog

Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

log (a) = log (cabs (a)) + i ∗ carg (a)

clump

Module: PDL::Slices

```
Signature: (P(); C(); int n)
```

"clumps" the first `n` dimensions into one large dimension

If, for example, `$a` has dimensions `(5,3,4)` then after

```
$b = $a->clump(2);   # Clump 2 first dimensions
```

the variable `$b` will have dimensions `(15,4)` and the element `$b->at(7,3)` refers to the element `$a->at(1,2,3)`.

Cmod

Module: PDL::FFTW

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

modulus of a complex piddle.

```
$out_pdl_real = Cmod($a_pdl_cplx);
```

Cmod2

Module: PDL::FFTW

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

Returns squared modulus of a complex number.

```
$out_pdl_real = Cmod2($a_pdl_cplx);
```

Cmul

Module: PDL::FFTW

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

Complex multiplication

```
$out_pdl_cplx = Cmul($a_pdl_cplx,$b_pdl_cplx);
```

cmul

Module: PDL::FFT

```
Signature: (ar(); ai(); br(); bi(); [o]cr(); [o]ci())
```

Complex multiplication

combcoords

Module: PDL::Graphics::TriD::Rout

```
Signature: (x(); y(); z();
              float [o]coords(tri=3);)
```

Combine three coordinates into a single piddle.

Combine x, y and z to a single piddle the first dimension of which is 3. This routine does dataflow automatically.

cont

Module: PDL::Graphics::PGPLOT

Display image as contour map

```
Usage: cont ( $image,  [$contours, $transform, $misval], [$opt] )
```

Notes: `$transform` for image/cont etc. is used in the same way as the `TR()` array in the underlying PGPLOT FORTRAN routine but is, fortunately, zero-offset.

Options recognised:

```
   CONTOURS - A piddle with the contour levels
     FOLLOW - Follow the contour lines around (uses pgcont rather than
              pgcons) If this is set >0 the chosen linestyle will be
              ignored and solid line used for the positive contours
              and dashed line for the negative contours.
     LABELS - An array of strings with labels for each contour
LABELCOLOUR - The colour of labels if different from the draw colour
              This will not interfere with the setting of draw colour
              using the colour keyword.
    MISSING - The value to ignore for contouring
  NCONTOURS - The number of contours wanted for automatical creation,
              overridden by CONTOURS
  TRANSFORM - The pixel-to-world coordinate transform vector
```

The following standard options influence this command:

```
AXIS, BORDER, COLOUR, JUSTIFY, LINESTYLE, LINEWIDTH

$x=sequence(10,10);
$ncont = 4;
$labels= ['COLD', 'COLDER', 'FREEZING', 'NORWAY']
# This will give four blue contour lines labelled in red.
cont $x, {NCONT => $ncont, LABELS => $labels, LABELCOLOR => RED,
          COLOR => BLUE}
```

contour_segments

Module: PDL::Graphics::TriD::Rout

This is the interface for the pp routine contour_segments_internal — it takes 3 piddles as input

`$c` is a contour value (or a list of contour values)

`$data` is an [m,n] array of values at each point

`$points` is a list of [3,m,n] points, it should be a grid monotonically increasing with m and n.

contour_segments returns a reference to a Perl array of line segments associated with each value of `$c`. It does not (yet) handle missing data values.

Algorthym

The data array represents samples of some field observed on the surface described by points. For each contour value we look for intersections on the line segments joining points of the data. When an intersection is found we look to the adjoining line segments for the other *end*(s) of the line *segment*(s). So suppose we find an intersection on an x-segment. We first look down to the left y-segment, then to the right y-segment and finally across to the next x-segment. Once we find one in a box (two on a point) we can quit because there can only be one. After we are done with a given x-segment, we look to the leftover possibilities for the adjoining y-segment. Thus the contours are built as a collection of line segments rather than a set of closed polygons.

conv2d

Module: PDL::Image2D

```
Signature: (a(m,n); kern(p,q); [o]b(m,n); int opt)
```

2D convolution of an array with a kernel (smoothing)

For large kernels, using a FFT routine, such as the `fftconvolve` entry in the *PDL::FFT::fftconvolve()|PDL::FFT* manpage, will be quicker.

```
$new = conv2d $old, $kernel, {OPTIONS}

$smoothed = conv2d $image, ones(3,3), {Boundary => Reflect}

Boundary - controls what values are assumed for the image when kernel
           crosses its edge:
           => Default  - periodic boundary conditions
                         (i.e. wrap around axis)
           => Reflect  - reflect at boundary
           => Truncate - truncate at boundary
```

convert

Module: PDL::Core

Generic datatype conversion function

```
$y = convert($x, $newtype);

$y = convert $x, long
$y = convert $x, ushort
```

`$newtype` is a type number, for convenience they are returned by `long()` etc when called without arguments.

convmath

Module: PDL::FFT

```
Signature: ([o,nc]a(m); [o,nc]b(m))
```

Internal routine doing maths for convolution

convolve

Module: PDL::ImageND

```
Signature: (a(m); b(n); int adims(p); int bdims(q); [o]c(m))
```

convolve

Module: PDL::ImageND

N-dimensional convolution algorithm.

$new = convolve $a, $kernel

Convolve an array with a kernel, both of which are N-dimensional.

Note because of the algorithm used (writing N-dim routines is not easy on the brain!) the boundary conditions are a bit strange. They wrap, but up to the NEXT row/column/cube-slice/etc. If this is a problem consider using zero-padding or something.

copy
Module: PDL::Core

Make a physical copy of a piddle

```
$new = $old->copy;
```

Since `$new = $old` just makes a new reference, the `copy` method is provided to allow real independent copies to be made.

cos Module: PDL::Ops

```
Signature: (a(); [o]b())
```

the cos function

```
$b = cos $a;
$a->inplace->cos;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary `cos` operator/function.

cosh
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard hyperbolic function.

Cp2r
Module: PDL::Complex

```
Signature: (r(m=2); [o]p(m=2))
```

convert complex numbers in polar (mod,arg) form to rectangular form

Cpow
Module: PDL::Complex

```
Signature: (a(m=2); b(m=2); [o]c(m=2))
```

complex `pow()` (`**`-operator)

Cproj
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

compute the projection of a complex number to the riemann sphere

cquant

Module: PDL::ImageRGB

quantize and reduce colours in 8-bit images

```
($out, $lut) = cquant($image [,$ncols]);
```

This function does color reduction for ¡=8bit displays and accepts 8bit RGB and 8bit palette images. It does this through an interface to the ppm_quant routine from the pbmplus package that implements the median cut routine which intellegently selects the 'best' colors to represent your image on a ¡= 8bit display (based on the median cut algorithm). Optional args: $ncols sets the maximum nunmber of colours used for the output image (defaults to 256). There are images where a different color reduction scheme gives better results (it seems this is true for images containing large areas with very smoothly changing colours).

Returns a list containing the new palette image (type PDL_Byte) and the RGB colormap.

Cr2p

Module: PDL::Complex

```
Signature: (r(m=2); float+ [o]p(m=2))
```

convert complex numbers in rectangular form to polar (mod,arg) form

Croots

Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2,n); int n => n)
```

Compute the `n` roots of `a`. `n` must be a positive integer. The result will always be a complex type!

crossp

Module: PDL::Primitive

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

Cross product of two 3D vectors

After

```
$c = crossp $a, $b
```

the inner product `$c*$a` and `$c*$b` will be zero, i.e. `$c` is orthogonal to `$a` and `$b`

Cscale
Module: PDL::FFTW

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

Complex by real multiplation.

```
Cscale($a_pdl_cplx,$b_pdl_real);
```

Csin
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))

sin (a) = 1/(2*i) * (exp (a*i) - exp (-a*i))
```

Csinh
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))

sinh (a) = (exp (a) - exp (-a)) / 2
```

Csqrt
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

ctab
Module: PDL::Graphics::PGPLOT

Load an image colour table.

Usage:

```
ctab ( $name, [$contrast, $brightness] ) # Builtin col table
ctab ( $ctab, [$contrast, $brightness] ) # $ctab is Nx4 array
ctab ( $levels, $red, $green, $blue, [$contrast, $brightness] )
ctab ( '', $contrast, $brightness ) # use last color table
```

Note: See the *PDL::Graphics::LUT*|*PDL::Graphics::LUT* manpage for access to a large number of colour tables.

Ctanh
Module: PDL::Complex

```
Signature: (a(m=2); [o]c(m=2))
```

cumuprodover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b(n))
```

Cumulative product

This function calculates the cumulative product along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

The sum is started so that the first element in the cumulative product is the first element of the parameter.

```
$a = cumuprodover($b);

$spectrum = cumuprodover $image->xchg(0,1)
```

cumusumover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b(n))
```

Cumulative sum

This function calculates the cumulative sum along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

The sum is started so that the first element in the cumulative sum is the first element of the parameter.

```
$a = cumusumover($b);

$spectrum = cumusumover $image->xchg(0,1)
```

Datatype_conversions

Module: PDL::Core

byte|short|ushort|long|float|double convert shorthands

```
$y = double $x; $y = ushort [1..10];
# all of byte|short|ushort|long|float|double behave similarly
```

When called with a piddle argument, they convert to the specific datatype.

When called with a numeric or list / listref argument they construct a new piddle. This is a convenience to avoid having to be long-winded and say `$x = long(pdl(42))`

Thus one can say:

```
$a = float(1,2,3,4);           # 1D
$a = float([1,2,3],[4,5,6]);   # 2D
$a = float([[1,2,3],[4,5,6]]); # 2D
```

Note the last two are equivalent — a list is automatically converted to a list reference for syntactic convenience. i.e. you can omit the outer `[]`

When called with no arguments return a special type token. This allows syntactical sugar like:

```
$x = ones byte, 1000,1000;
```

This example creates a large piddle directly as byte datatype in order to save memory.

In order to control how undefs are handled in converting from perl lists to PDLs, one can set the variable `$PDL::undefval`; see the function the `pdl` entry in the *pdl()*| manpage for more details.

```
perldl> p $x=sqrt float [1..10]
[1 1.41421 1.73205 2 2.23607 2.44949 2.64575 2.82843 3 3.16228]
perldl> p byte $x
[1 1 1 2 2 2 2 2 3 3]
```

dev Module: PDL::Graphics::PGPLOT

Open PGPLOT graphics device

```
Usage: dev $device, [$nx,$ny];
```

`$device` is a PGPLOT graphics device such as "/xserve" or "/ps", if omitted defaults to last used device (or value of env var `PGPLOT_DEV` if first time). `$nx`, `$ny` specify sub-panelling.

diagonal

Module: PDL::Core

Returns the multidimensional diagonal over the specified dimensions.

```
$d = $x->diagonal(dim1, dim2,...)
```

```
perldl> $a = zeroes(3,3,3);
perldl> ($b = $a->diagonal(0,1))++;
perldl> p $a
[
 [
  [1 0 0]
  [0 1 0]
  [0 0 1]
```

```
 ]
 [
  [1 0 0]
  [0 1 0]
  [0 0 1]
 ]
 [
  [1 0 0]
  [0 1 0]
  [0 0 1]
 ]
]
```

diagonalI

Module: PDL::Slices

```
Signature: (P(); C(); SV *list)
```

Returns the multidimensional diagonal over the specified dimensions.

The diagonal is placed at the first (by number) dimension that is diagonalized. The other diagonalized dimensions are removed. So if `$a` has dimensions `(5,3,5,4,6,5)` then after

```
$b = $a->diagonal(0,2,5);
```

the piddle `$b` has dimensions `(5,3,4,6)` and `$b->at(2,1,0,1)` refers to `$a->at(2,1,2,0,1,2)`.

NOTE: diagonal doesn't handle threadids correctly. XXX FIX

dice

Module: PDL::Slices

Dice rows/columns/planes out of a PDL using indexes for each dimension.

This function can be used to extract irregular subsets along many dimension of a PDL, e.g. only certain rows in an image, or planes in a cube. This can of course be done with the usual dimension tricks but this saves having to figure it out each time!

This method is similar in functionality to the the `slice` entry in the *slice*| manpage method, but the `slice` entry in the *slice*| manpage requires that contiguous ranges or ranges with constant offset be extracted. (i.e. the `slice` entry in the *slice*| manpage requires ranges of the form `1,2,3,4,5` or `2,4,6,8,10`). Because of this restriction, the `slice` entry in the *slice*| manpage is more memory efficient and slightly faster than dice

```
$slice = $data->dice([0,2,6],[2,1,6]); # Dicing a 2-D array
```

The arguments to dice are arrays (or 1D PDLs) for each dimension in the PDL. These arrays are used as indexes to which rows/columns/cubes,etc to dice-out (or extract) from the `$data` PDL.

```
perldl> $a = sequence(10,4)
perldl> p $a
[
 [ 0  1  2  3  4  5  6  7  8  9]
 [10 11 12 13 14 15 16 17 18 19]
 [20 21 22 23 24 25 26 27 28 29]
 [30 31 32 33 34 35 36 37 38 39]
]
perldl> p $a->dice([1,2],[0,3]) # Select columns 1,2 and rows 0,3
[
 [ 1  2]
 [31 32]
]
```

As this is an index function, any modifications to the slice change the parent.

dice_axis

Module: PDL::Slices

Dice rows/columns/planes from a single PDL axis (dimension) using index along a specified axis

This function can be used to extract irregular subsets along any dimension, e.g. only certain rows in an image, or planes in a cube. This can of course be done with the usual dimension tricks but this saves having to figure it out each time!

```
$slice = $data->dice_axis($axis,$index);

perldl> $a = sequence(10,4)
perldl> $idx = pdl(1,2)
perldl> p $a->dice_axis(0,$idx) # Select columns
[
 [ 1  2]
 [11 12]
 [21 22]
 [31 32]
]
perldl> $t = $a->dice_axis(1,$idx) # Select rows
perldl> $t.=0
perldl> p $a
[
 [ 0  1  2  3  4  5  6  7  8  9]
```

```
 [ 0  0  0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0  0  0]
 [30 31 32 33 34 35 36 37 38 39]
]
```

The trick to using this is that the index selects elements along the dimensions specified, so if you have a 2D image `axis=0` will select certain `X` values — i.e. extract columns

As this is an index function, any modifications to the slice change the parent.

dims

Module: PDL::Core

Return piddle dimensions as a perl list

```
@dims = $piddle->dims;  @dims = dims($piddle);

perldl> p @tmp = dims zeroes 10,3,22
10 3 22
```

divide

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

divide two piddles

```
$c = divide $a, $b, 0;     # explicit call with trailing 0
$c = $a / $b;           # overloaded call
$a->inplace->divide($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `/` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

doflow

Module: PDL::Core

Turn on/off dataflow

```
$x->doflow;  doflow($x);
```

dog Module: PDL::Core

Opposite of 'cat' :). Split N dim piddle to list of N-1 dim piddles

Takes a single N-dimensional piddle and splits it into a list of N-1 dimensional piddles. The breakup is done along the last dimension. Note the dataflown connection is still preserved by default, e.g.:

```
perldl> $p = ones 3,3,3
perldl> ($a,$b,$c) = dog $p
perldl> $b++; p $p
[
 [
  [1 1 1]
  [1 1 1]
  [1 1 1]
 ]
 [
  [2 2 2]
  [2 2 2]
  [2 2 2]
 ]
 [
  [1 1 1]
  [1 1 1]
  [1 1 1]
 ]
]

Break => 1   Break dataflow connection (new copy)
```

double

Module: PDL::Core

Convert to double datatype — see 'Datatype_conversions'

dummy

Module: PDL::Core

Insert a 'dummy dimension' of given length (defaults to 1)

No relation to the 'Dungeon Dimensions' in Discworld! Negative positions specify relative to last dimension, i.e. `dummy(-1)` appends one dimension at end, `dummy(-2)` inserts a dummy dimension in front of the last dim, etc.

```
$y = $x->dummy($position[,$dimsize]);

perldl> p sequence(3)->dummy(0,3)
[
 [0 0 0]
 [1 1 1]
 [2 2 2]
]
```

eigens

Module: PDL::Math

```
Signature: ([phys]a(m); [o,phys]ev(n,n); [o,phys]e(n))
```

Eigenvalues and -vectors of a symmetric square matrix. If passed an asymmetric matrix, the routine will warn and symmetrize it.

```
($e, $ev) = eigens($a);
```

eigsys

Module: PDL::Slatec

Eigenvalues and eigenvectors of a real positive definite symmetric matrix.

```
($eigvals,$eigvecs) = eigsys($mat)
```

Note: this function should be extended to calculate only eigenvalues if called in scalar context!

env Module: PDL::Graphics::PGPLOT

Define a plot window, and put graphics on 'hold'

```
Usage: env $xmin, $xmax, $ymin, $ymax, [$justify, $axis];
       env $xmin, $xmax, $ymin, $ymax, [$options];
```

`$xmin`, `$xmax`, `$ymin`, `$ymax` are the plot boundaries. `$justify` is a boolean value (default is **0**); if true the axes scales will be the same (see the `justify` entry elsewhere in this document). `$axis` describes how the axes should be drawn (see the `axis` entry elsewhere in this document) and defaults to **0**.

If the second form is used, $justify and $axis can be set in the options hash, for example:

```
env 0, 100, 0, 50, {JUSTIFY => 1, AXIS => 'GRID', CHARSIZE => 0.7};
```

eq Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

binary *equal to* operation (`==`)

```
$c = eq $a, $b, 0;     # explicit call with trailing 0
$c = $a == $b;           # overloaded call
$a->inplace->eq($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `==` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

erf Module: PDL::Math

```
Signature: (a(); [o]b())
```

The error function

erfc

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The complement of the error function

erfi

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The inverse of the error function

errb

Module: PDL::Graphics::PGPLOT

Plot error bars (using `pgerrb()`)

Usage:

```
errb ( $y, $yerrors, [$opt] )
errb ( $x, $y, $yerrors, [$opt] )
errb ( $x, $y, $xerrors, $yerrors, [$opt] )
errb ( $x, $y, $xloerr, $xhierr, $yloerr, $yhierr, [$opt])
```

Options recognised:

```
  TERM - Length of terminals in multiples of the default length
SYMBOL - Plot the datapoints using the symbol value given, either
         as name or number - see documentation for 'points'
```

The following standard options influence this command:

```
AXIS, BORDER, CHARSIZE, COLOUR, JUSTIFY, LINESTYLE, LINEWIDTH

$y = sequence(10)**2+random(10);
$sigma=0.5*sqrt($y);
errb $y, $sigma, {COLOUR => RED, SYMBOL => 18};
```

exp Module: PDL::Ops

```
Signature: (a(); [o]b())
```

the exponential function

```
$b = exp $a;
$a->inplace->exp;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary `exp` operator/function.

ezfftb
Module: PDL::Slatec

```
Signature: ([o]r(n);azero();a(n);b(n);wsave(foo))
```

ezfftf
Module: PDL::Slatec

```
Signature: (r(n);[o]azero();[o]a(n);[o]b(n);wsave(foo))
```

ezffti
Module: PDL::Slatec

```
Signature: (int n();[o]wsave(foo))
```

Subroutine ezffti initializes the work array `wsave()` which is used in both the `ezfftf` entry in the *ezfftf|* manpage and the `ezfftb` entry in the *ezfftb|* manpage. The prime factorization of `n` together with a tabulation of the trigonometric functions are computed and stored in `wsave()`.

fft Module: PDL::FFT

```
Signature: ([o,nc]real(n); [o,nc]imag(n))
```

Complex FFT of the "real" and "imag" arrays [inplace]

fftconvolve
Module: PDL::FFT

N-dimensional convolution

```
$kernel = kernctr($image,$smallk);
fftconvolve($image,$kernel);
```

fftconvolve works inplace, and returns an error array in kernel as an accuracy check — all the values in it should be negligible.

The sizes of the image and the kernel must be the same. the `kernctr` entry in the *kernctr|* manpage centres a small kernel to emulate the behaviour of the direct convolution routines.

The speed cross-over between using straight convolution (the `conv2d` entry in the *PDL::Image2D::conv2d()|PDL::Image2D* manpage) and these fft routines is for kernel sizes roughly 7x7.

fftnd

Module: PDL::FFT

N-dimensional FFT (inplace)

```
fftnd($real,$imag);
```

fftw

Module: PDL::FFTW

calculate the complex FFT of a real piddle (complex input, complex output)

```
$pdl_cplx = fftw $pdl_cplx;
```

fftwconv

Module: PDL::FFTW

ND convolution using complex ffts from the FFTW library

Assumes real input!

```
$conv = fftwconv $im, kernctr $k;
```

fibonacci

Module: PDL::Primitive

```
Signature: ([o]x(n))
```

Constructor — a vector with Fibonacci's sequence

fitgauss1d

Module: PDL::Fit::Gaussian

Fit 1D Gassian to data piddle

($cen, $pk, $fwhm2, $back, $err, $fit) = fitgauss1d($x, $data);

($cen, $pk, $fwhm2, $back, $err, $fit) = fitgauss1d($x, $data);

xval(n); *data*(n); [o] *xcentre()*; [o] *peak_ht()*; [o] *fwhm()*; [o] *background()*;int [o] *err()*; [o] *datafit*(n); [t] *sig*(n); [t] *xtmp*(n); [t] *ytmp*(n); [t] *yytmp*(n); [t] *rtmp*(n);

Fit's a 1D Gaussian robustly free parameters are the centre, peak height, FWHM. The background is NOT fit, because I find this is generally unreliable, rather a median is determined in the 'outer' 10% of pixels (i.e. those at the start/end of the data piddle). The initial estimate of the FWHM is the length of the piddle/3, so it might fail if the piddle is too long. (This is non-robust anyway). Most data does just fine and this is a good default gaussian fitter.

SEE ALSO: *fitgauss1dr()* for fitting radial gaussians

fitgauss1dr

Module: PDL::Fit::Gaussian

Fit 1D Gassian to radial data piddle

($pk, $fwhm2, $back, $err, $fit) = fitgauss1dr($r, $data);

($pk, $fwhm2, $back, $err, $fit) = fitgauss1dr($r, $data);

xval(n); *data*(n); [o] *peak_ht()*; [o] *fwhm()*; [o] *background()*;int [o] *err()*; [o] *datafit*(n); [t] *sig*(n); [t] *xtmp*(n); [t] *ytmp*(n); [t] *yytmp*(n); [t] *rtmp*(n);

Fit's a 1D radial Gaussian robustly free parameters are the peak height, FWHM. Centre is assumed to be X=0 (i.e. start of piddle). The background is NOT fit, because I find this is generally unreliable, rather a median is determined in the 'outer' 10% of pixels (i.e. those at the end of the data piddle). The initial estimate of the FWHM is the length of the piddle/3, so it might fail if the piddle is too long. (This is non-robust anyway). Most data does just fine and this is a good default gaussian fitter.

SEE ALSO: *fitgauss1d()* to fit centre as well.

fitpoly1d

Module: PDL::Fit::Polynomial

Fit 1D polynomials to data using min chi^2 (least squares)

```
Usage: ($yfit, [$coeffs]) = fitpoly1d [$xdata], $data, $order, [Options...]
```

Signature: (x(n); y(n); [o] *yfit*(n); [o] *coeffs*(order))

Uses a standard matrix inversion method to do a least squares/min chi^2 polynomial fit to data. Order=2 is a linear fit (two parameters).

Returns the fitted data and optionally the coefficients.

One can thread over extra dimensions to do multiple fits (except the order can not be threaded over — i.e. it must be one fixed scalar number like "4").

The data is normalised internally to avoid overflows (using the mean of the abs value) which are common in large polynomial series but the returned fit, coeffs are in unnormalised units.

```
$yfit = fitpoly1d $data,2; # Least-squares line fit
($yfit, $coeffs) = fitpoly1d $x, $y, 4; # Fit a cubic

$fitimage = fitpoly1d $image,2  # Fit a quadratic to each row of an image

$myfit = fitpoly1d $line, 2, {Weights => $w}; # Weighted fit

Options:
   Weights    Weights to use in fit, e.g. 1/$sigma**2 (default=1)
```

float

Module: PDL::Core

Convert to float datatype — see 'Datatype_conversions'

floor

Module: PDL::Math

```
Signature: (a(); [o]b())
```

Round to integral values in floating-point format

flows

Module: PDL::Core

Whether or not a piddle is indulging in dataflow

```
something if $x->flows; $hmm = flows($x);
```

ge Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

the binary ¿= (greater equal) operation

```
$c = ge $a, $b, 0;     # explicit call with trailing 0
$c = $a >= $b;           # overloaded call
$a->inplace->ge($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `>=` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

geco

Module: PDL::Slatec

```
Signature: (a(n,n);int [o]ipvt(n);[o]rcond();[o]z(n))
```

Factor a matrix using Gaussian elimination and estimate the condition number of the matrix.

gedi

Module: PDL::Slatec

```
Signature: (a(n,n);int [o]ipvt(n);[o]det(two=2);[o]work(n);int job())
```

Compute the determinant and inverse of a matrix using the factors computed by the `geco` entry in the *geco|* manpage or the `gefa` entry in the *gefa|* manpage.

gefa

Module: PDL::Slatec

```
Signature: (a(n,n);int [o]ipvt(n);int [o]info())
```

Factor a matrix using Gaussian elimination.

gesl

Module: PDL::Slatec

```
Signature: (a(lda,n);int ipvt(n);b(n);int job())
```

Solve the real system `A*X=B` or `TRANS(A)*X=B` using the factors computed by the `geco` entry in the *geco|* manpage or the `gefa` entry in the *gefa|* manpage.

get Module: PDL::Func

```
my $x         = $obj->get( x );
my ( $x, $y ) = $obj->get( qw( x y ) );
```

Get attributes from a PDL::Func object.

Given a list of attribute names, return a list of their values; in scalar mode return a scalar value. If the supplied list contains an unknown attribute, `get` returns a value of `undef` for that attribute.

get_datatype

Module: PDL::Core

Internal: Return the numeric value identifying the piddle datatype

```
$x = $piddle->get_datatype;
```

Mainly used for internal routines.

NOTE: get_datatype returns 'just a number' not any special type object.

getdim

Module: PDL::Core

Returns the size of the given dimension.

```
$dim0 = $piddle->getdim(0);

perldl> p zeroes(10,3,22)->getdim(1)
3
```

gethdr

Module: PDL::Core

Retrieve header information from a piddle

```
$pdl=rfits('file.fits');
$h=$pdl->gethdr;
print "Number of pixels in the X-direction=$$h{NAXIS1}\n";
```

The `gethdr` function retrieves whatever header information is contained within a piddle. The header can be set with the `sethdr` entry in the *sethdr|* manpage and is always a hash reference and has to be dereferenced for access to the value.

It is important to realise that you are free to insert whatever hash reference you want in the header, so you can use it to record important information about your piddle, and that it is not automatically copied when you copy the piddle. See the `hdrcpy` entry in the *hdrcpy|* manpage to enable automatic header copying.

For instance a wrapper around rcols that allows your piddle to remember the file it was read from and the columns could be easily written (here assuming that no regexp is needed, extensions are left as an exercise for the reader)

```
sub ext_rcols {
   my ($file, @columns)=@_;
   my $header={};
   $$header{File}=$file;
   $$header{Columns}=\@columns;

   @piddles=rcols $file, @columns;
   foreach (@piddles) { $_->sethdr($header); }
   return @piddles;
}
```

getndims

Module: PDL::Core

Returns the number of dimensions in a piddle

```
$ndims = $piddle->getndims;

perldl> p zeroes(10,3,22)->getndims
3
```

GLinit

Module: PDL::Graphics::TriD::Tk

GLinit is called internally by a Configure callback in Populate. This insures that the required Tk::Frame is initialized before the TriD::GL window that will go inside.

grabpic3d

Module: PDL::Graphics::TriD

Grab a 3D image from the screen.

```
$pic = grabpic3d();
```

The returned piddle has dimensions (3,$x,$y) and is of type float (currently). XXX This should be altered later.

gradient

Module: PDL::Func

```
my $gi          = $obj->gradient( $xi );
my ( $yi, $gi ) = $obj->gradient( $xi );
```

Returns the derivative and, optionally, the interpolated function for the `Hermite` scheme (PDL::Func).

grandom

Module: PDL::Primitive

Constructor which returns piddle of Gaussian random numbers

```
$a = grandom([type], $nx, $ny, $nz,...);
$a = grandom $b;
```

etc (see the `zeroes` entry in the *zeroes|PDL::Core* manpage).

This is generated using the math library routine `ndtri`.

Mean = 0, Stddev = 1

gt Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

the binary ¿ (greater than) operation

```
$c = gt $a, $b, 0;     # explicit call with trailing 0
$c = $a > $b;           # overloaded call
$a->inplace->gt($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `>` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

hclip

Module: PDL::Primitive

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

clip `$a` by `$b` (`$b` is upper bound)

hdrcpy

Module: PDL::Core

switch on/off/examine automatic header copying

```
print "hdrs will be copied" if $a->hdrcpy;
$a->hdrcpy(1);       # switch on hdr copying
$b = $a->sumover;    # and $b will inherit $a's hdr
$a->hdrcpy(0);       # and now make $a non-infectious again
```

Normally, the optional header of a piddle is not copied automatically in pdl operations. Switching on the hdrcpy flag using the `hdrcpy` method will enable automatic hdr copying. Note that copying is **by reference** for efficiency reasons. `hdrcpy` without an argument just returns the current setting of the flag.

help

Module: PDL::Doc::Perldl

print documentation about a PDL function or module or show a PDL manual

```
Usage: help 'func'

perldl> help 'PDL::Slices'   # show the docs in the PDL::Slices module
perldl> help 'PDL::Intro'    # show the PDL::Intro manual
perldl> help 'slice'         # show docs on the 'slice' function
```

hi2d

Module: PDL::Graphics::PGPLOT

Plot image as 2d histogram (not very good IMHO...)

```
Usage: hi2d ( $image, [$x, $ioff, $bias], [$opt] )
```

Options recognised:

```
IOFFSET - The offset for each array slice. >0 slants to the right
                                           <0 to the left.
   BIAS - The bias to shift each array slice up by.
```

The following standard options influence this command:

```
AXIS, BORDER, JUSTIFY
```

Note that meddling with the `ioffset` and `bias` often will require you to change the default plot range somewhat. It is also worth noting that if you have TriD working you will probably be better off using the `mesh3d` entry in the *mesh3d|PDL::Graphics::TriD* manpage or a similar command — see the *PDL::Graphics::TriD* manpage.

```
$r=sequence(100)/50-1.0;
$y=exp(-$r**2)*transpose(exp(-$r**2))
hi2d $y, {IOFF => 1.5, BIAS => 0.07};
```

hist

Module: PDL::Basic

Create histogram of a piddle

```
$hist = hist($data,[$min,$max,$step]);
($xvals,$hist) = hist($data,[$min,$max,$step]);
```

If requested, `$xvals` gives the computed bin centres

A nice idiom (with the *PDL::Graphics::PGPLOT|PDL::Graphics::PGPLOT* manpage) is

```
bin hist $data;  # Plot histogram
```

```
perldl> p $y
[13 10 13 10 9 13 9 12 11 10 10 13 7 6 8 10 11 7 12 9 11 11 12 6 12 7]
perldl> $h = hist $y,0,20,1; # hist with step 1, min 0 and 20 bins
perldl> p $h
[0 0 0 0 0 0 2 3 1 3 5 4 4 4 0 0 0 0 0 0]
```

histogram

Module: PDL::Primitive

```
 Signature: (in(n); int+[o] hist(m); double step; double min; int msize => m)
```

Calculates a histogram for given stepsize and minimum.

```
$h = histogram($data, $step, $min, $numbins);
$hist = zeroes $numbins;  # Put histogram in existing piddle.
histogram($data, $hist, $step, $min, $numbins);
```

The histogram will contain `$numbins` bins starting from `$min`, each `$step` wide. The value in each bin is the number of values in `$data` that lie within the bin limits.

Data below the lower limit is put in the first bin, and data above the upper limit is put in the last bin.

The output is reset in a different threadloop so that you can take a histogram of `$a(10,12)` into `$b(15)` and get the result you want.

Use the `hist` entry in the *hist|PDL::Basic* manpage instead for a high-level interface.

```
perldl> p histogram(pdl(1,1,2),1,0,3)
[0 2 1]
```

histogram2d

Module: PDL::Primitive

```
 Signature: (ina(n); inb(n); int+[o] hist(ma,mb); double stepa; double mina; int masiz
                      double stepb; double minb; int mbsize => mb;)
```

Calculates a 2d histogram.

```
$h = histogram2d($datax, $datay,
      $stepx, $minx, $nbinx, $stepy, $miny, $nbiny);
$hist = zeroes $nbinx, $nbiny;  # Put histogram in existing piddle.
histogram2d($datax, $datay, $hist,
      $stepx, $minx, $nbinx, $stepy, $miny, $nbiny);
```

The histogram will contain `$nbinx` x `$nbiny` bins, with the lower limits of the first one at `($minx, $miny)`, and with bin size `($stepx, $stepy)`. The value in each bin is the number of values in `$datax` and `$datay` that lie within the bin limits.

Data below the lower limit is put in the first bin, and data above the upper limit is put in the last bin.

```
perldl> p histogram2d(pdl(1,1,1,2,2),pdl(2,1,1,1,1),1,0,3,1,0,3)
[
 [0 0 0]
 [0 2 2]
 [0 1 0]
]
```

howbig

Module: PDL::Core

Returns the size of a piddle datatype in bytes.

```
$size = howbig($piddle->get_datatype);
```

Mainly used for internal routines.

NOTE: NOT a method! This is because get_datatype returns 'just a number' not any special object.

```
perldl> p howbig(ushort([1..10])->get_datatype)
2
```

i2C Module: PDL::Complex

```
Signature: (r(); [o]c(m=2))
```

convert imaginary to complex, assuming a real part of zero

identvaff

Module: PDL::Slices

```
Signature: (P(); C())
```

A vaffine identity transformation (includes thread_id copying).

Mainly for internal use.

ifft

Module: PDL::FFT

```
Signature: ([o,nc]real(n); [o,nc]imag(n))
```

Complex Inverse FFT of the "real" and "imag" arrays [inplace]

ifftnd

Module: PDL::FFT

N-dimensional inverse FFT

```
ifftnd($real,$imag);
```

ifftw

Module: PDL::FFTW

Complex inverse FFT (complex input, complex output).

```
$pdl_cplx = ifftw $pdl_cplx;
```

iis Module: PDL::Graphics::IIS

Displays an image on a IIS device (e.g. SAOimage/Ximtool)

```
iis $image, [ { MIN => $min, MAX => $max,
                TITLE => 'pretty picture',
                FRAME => 2 } ]
iis $image, [$min,$max]
```

```
(image(m,n),[\%options]) or (image(m,n),[min(),max()])
```

Displays image on a IIS device. If `min()` or `max()` are omitted they are autoscaled. A good demonstration of PDL threading can be had by giving `iis()` a data *cube* — `iis()` will be repeatedly called for each plane of the cube resulting in a poor man's movie!

If supplied, `TITLE` is used to label the frame, if no title is supplied, either the `OBJECT` value stored in the image header or a default string is used (the title is restricted to a maximum length of 32 characters).

To specify which frame to draw to, either use the package variable `$iisframe`, or the `FRAME` option.

iiscirc

Module: PDL::Graphics::IIS

Draws a circle on a IIS device (e.g. SAOimage/Ximtool)

```
(x(),y(),radius(),colour())

iiscirc $x, $y, [$radius, $colour]
```

Draws circles on the IIS device with specied points and colours. Because this module uses the *PDL::PP*|*PDL::PP* manpage threading you can supply lists of points via 1D arrays, etc.

An amusing PDL idiom is:

```
perldl> iiscirc iiscur
```

Note the colours are the same as IRAF, viz:

```
201 = cursor color (white)
202 = black
203 = white
204 = red
205 = green
206 = blue
207 = yellow
208 = cyan
209 = magenta
210 = coral
211 = maroon
212 = orange
213 = khaki
214 = orchid
215 = turquoise
216 = violet
217 = wheat
```

iiscur

Module: PDL::Graphics::IIS

Return cursor position from an IIS device (e.g. SAOimage/Ximtool)

```
($x,$y) = iiscur($ch)
```

This function puts up an interactive cursor on the IIS device and returns the `($x,$y)` position and the character typed (`$ch`) by the user.

imag

Module: PDL::Graphics::PGPLOT

Display an image (uses `pgimag()`/`pggray()` as appropriate)

```
Usage: imag ( $image,  [$min, $max, $transform], [$opt] )
```

Notes: `$transform` for image/cont etc. is used in the same way as the `TR()` array in the underlying PGPLOT FORTRAN routine but is, fortunately, zero-offset.

There are several options related to scaling. By default, the image is scaled to fit the PGPLOT default viewport on the screen. Scaling, aspect ratio preservation, and 1:1 pixel mapping are available. (1:1 pixel mapping GREATLY increases the speed of pgimag, and is useful for, eg, movie display; but it's not recommended for final output as it's not device-independent.)

Options recognised:

```
       ITF - the image transfer function applied to the pixel values. It
             may be one of 'LINEAR', 'LOG', 'SQRT' (lower case is
             acceptable). It defaults to 'LINEAR'.
     MIN   - Sets the minimum value to be used for calculation of the
             display stretch
     MAX   - Sets the maximum value for the same
TRANSFORM - The transform 'matrix' as a 6x1 vector for display
     PIX   - Sets the image pixel aspect ratio.  By default, imag
             stretches the image pixels so that the final image aspect
             ratio fits the viewport exactly.  Setting PIX=>1 causes
             the image aspect ratio to be preserved.  (the image is
             scaled to avoid cropping, unless you specify scaling
             manually).  Larger numbers yield "landscape mode" pixels.
   PITCH - Sets the number of image pixels per screen unit, in the Y
             direction.  The X direction is determined by PIX, which
             defaults to 1 if PITCH is specified and PIX is not.  PITCH
             causes UNIT to default to "inches" so that it's easy to say
             100dpi by specifying {PITCH=>100}.  Larger numbers yield
             higher resolution (hence smaller appearing) images.
```

```
     UNIT - Sets the screen unit used for scaling.  Must be one of the
            PGPLOT supported units (inch, mm, pixel, normalized).  You
            can refer to them by name or by number.  Defaults to pixels
            if not specified.
    SCALE - Syntactic sugar for the reciprocal of PITCH.  Makes the
            UNIT default to "pixels" so you can say "{SCALE=>1}"
            to see your image in device pixels.   Larger SCALEs lead
            to larger appearing images.
```

The following standard options influence this command:

```
AXIS, BORDER, JUSTIFY

  To see an image with maximum size in the current window, but square
  pixels, say:
        imag $a,{PIX=>1}
  An alternative approach is to try:
        imag $a,{JUSTIFY=>1}
  To see the same image, scaled 1:1 with device pixels, say:
        imag $a,{SCALE=>1}
  To see an image made on a device with 1:2 pixel aspect ratio, with
  X pixels the same as original image pixels, say
        imag $a,{PIX=>0.5,SCALE=>2}
  To display an image at 100 dpi on any device, say:
        imag $a,{PITCH=>100}
  To display an image with 100 micron pixels, say:
        imag $a,{PITCH=>10,UNIT=>'mm'}
```

imag1

Module: PDL::Graphics::PGPLOT

Display an image with correct aspect ratio

```
 Usage:  imag1 ( $image, [$min, $max, $transform], [$opt] )
```

Notes: This is syntactic sugar for *imag*({PIX=¿1}).

imag3d

Module: PDL::Graphics::TriD

3D rendered image plot, defined by a variety of contexts

```
imag3d piddle(3,x,y), {OPTIONS}
imag3d [piddle,...], {OPTIONS}
```

Example:

```
perldl> imag3d [sqrt(rvals(zeroes(50,50))/2)], {{Lines=>0};
```

```
- Rendered image of surface
```

See module documentation for more information on contexts and options

imagrgb

Module: PDL::Graphics::TriD

2D TrueColor Image plot

```
imagrgb piddle(3,x,y), {OPTIONS}
imagrgb [piddle,...], {OPTIONS}
```

This would be used to plot an image, specifying red, green and blue values at each point. Note: contexts are very useful here as there are many ways one might want to do this.

e.g.

```
perldl> $a=sqrt(rvals(zeroes(50,50))/2)
perldl> imagrgb [0.5*sin(8*$a)+0.5,0.5*cos(8*$a)+0.5,0.5*cos(4*$a)+0.5]
```

imagrgb3d

Module: PDL::Graphics::TriD

2D TrueColor Image plot as an object inside a 3D space

```
imagrdb3d piddle(3,x,y), {OPTIONS}
imagrdb3d [piddle,...], {OPTIONS}
```

The piddle gives the colors. The option allowed is Points, which should give 4 3D coordinates for the corners of the polygon, either as a piddle or as array ref. The default is `[[0,0,0],[1,0,0],[1,1,0],[0,1,0]]`.

e.g.

```
perldl> imagrgb3d $colors, {Points => [[0,0,0],[1,0,0],[1,0,1],[0,0,1]]};
- plot on XZ plane instead of XY.
```

indadd

Module: PDL::Primitive

```
 Signature: (a(); int ind(); [o] sum(m))
```

Threaded Index Add: Add `a` to the `ind` element of `sum`, i.e:

```
sum(ind) += a
```

Simple Example:

```
$a = 2;
$ind = 3;
$sum = zeroes(10);
indadd($a,$ind, $sum);
print $sum
#Result: ( 2 added to element 3 of $sum)
# [0 0 0 2 0 0 0 0 0 0]
```

Threaded Example:

```
$a = pdl( 1,2,3);
$ind = pdl( 1,4,6);
$sum = zeroes(10);
indadd($a,$ind, $sum);
print $sum."\n";
#Result: ( 1, 2, and 3 added to elements 1,4,6 $sum)
# [0 1 0 0 2 0 3 0 0 0]
```

index

Module: PDL::Slices

```
Signature: (a(n); int ind(); [oca] c())
```

These functions provide rudimentary index indirection.

```
$c = a(ind());
$c = a(ind1(),ind2());
```

It would be useful to have a more complete function for this at some point, or at least a perl wrapper, that allows

```
$c = $a->islice("1:2",$ind1,"3:4",$ind2");
```

with many dimensions.

This function is two-way, i.e. after

```
$c = $a->index(pdl[0,5,8]);
$c .= pdl [0,2,4];
```

the changes in `$c` will flow back to `$a`.

index2d

Module: PDL::Slices

```
Signature: (a(na,nb); int inda(); int indb(); [oca] c())
```

These functions provide rudimentary index indirection.

```
$c = a(ind());
$c = a(ind1(),ind2());
```

It would be useful to have a more complete function for this at some point, or at least a perl wrapper, that allows

```
$c = $a->islice("1:2",$ind1,"3:4",$ind2");
```

with many dimensions.

This function is two-way, i.e. after

```
$c = $a->index(pdl[0,5,8]);
$c .= pdl [0,2,4];
```

the changes in `$c` will flow back to `$a`.

infftw

Module: PDL::FFTW

Complex inplace inverse FFT (complex input, complex output).

```
$pdl_cplx = infftw $pdl_cplx;
```

info

Module: PDL::Core

Return formatted information about a piddle.

```
$x->info($format_string);
```

```
print $x->info("Type: %T Dim: %-15D State: %S");
```

Returns a string with info about a piddle. Takes an optional argument to specify the format of information a la sprintf. Format specifiers are in the form `%<width><letter>` where the width is optional and the letter is one of

T Type

D Formatted Dimensions

F Dataflow status

S Some internal flags (P=physical,V=Vaffine,C=changed)

C Class of this piddle, i.e. `ref $pdl`

A Address of the piddle struct as a unique identifier

M Calculated memory consumption of this piddle's data area

init

Module: PDL::Func

```
$obj = init PDL::Func( Interpolate => "Hermite", x => $x, y => $y );
$obj = init PDL::Func( { x => $x, y => $y } );
```

Create a PDL::Func object, which can interpolate, and possibly integrate and calculate gradients of a dataset.

If not specified, the value of Interpolate is taken to be `Linear`, which means the interpolation is performed by the `interpolate` entry in the *PDL::Primitive::interpolate|PDL::Primitive* manpage. A value of `Hermite` uses piecewise cubic Hermite functions, which also allows the integral and gradient of the data to be estimated.

inner

Module: PDL::Primitive

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

Inner product over one dimension

```
c = sum_i a_i * b_i
```

inner2

Module: PDL::Primitive

```
Signature: (a(n); b(n,m); c(m); [o]d())
```

Inner product of two vectors and a matrix

```
d = sum_ij a(i) b(i,j) c(j)
```

Note that you should probably not thread over `a` and `c` since that would be very wasteful. Instead, you should use a temporary for `b*c`.

inner2d

Module: PDL::Primitive

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

Inner product over 2 dimensions.

Equivalent to

```
$c = inner($a->clump(2), $b->clump(2))
```

inner2t

Module: PDL::Primitive

```
Signature: (a(j,n); b(n,m); c(m,k); [t]tmp(n,k); [o]d(j,k)))
```

Efficient Triple matrix product `a*b*c`

Efficiency comes from by using the temporary `tmp`. This operation only scales as `N**3` whereas threading using the `inner2` entry in the *inner2|* manpage would scale as `N**4`.

The reason for having this routine is that you do not need to have the same thread-dimensions for `tmp` as for the other arguments, which in case of large numbers of matrices makes this much more memory-efficient.

It is hoped that things like this could be taken care of as a kind of closures at some point.

innerwt

Module: PDL::Primitive

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

Weighted (i.e. triple) inner product

```
d = sum_i a(i) b(i) c(i)
```

inplace

Module: PDL::Core

Flag a piddle so that the next operation is done 'in place'

```
somefunc($x->inplace); somefunc(inplace $x);
```

In most cases one likes to use the syntax `$y = f($x)`, however in many case the operation `f()` can be done correctly 'in place', i.e. without making a new copy of the data for output. To make it easy to use this, we write `f()` in such a way that it operates in-place, and use `inplace` to hint that a new copy should be disabled. This also makes for clear syntax.

Obviously this will not work for all functions, and if in doubt see the function's documentation. However one can assume this is true for all elemental functions (i.e. those which just operate array element by array element like `log10`).

```
perldl> $x = xvals zeroes 10;
perldl> log10(inplace $x)
perldl> p $x
[      -Inf 0    0.30103 0.47712125 0.60205999    0.69897
0.77815125 0.84509804 0.90308999 0.95424251]
```

inrfftw

Module: PDL::FFTW

Real inplace inverse FFT. Have a look at nrfftw to understand the format. USE ONLY an even first dimension size! (complex input, real output)

```
$pdl_real = infftw $pdl_cplx;
```

integrate

Module: PDL::Func

```
my $ans = $obj->integrate( index => pdl( 2, 5 ) );
my $ans = $obj->integrate( x => pdl( 2.3, 4.5 ) );
```

Integrate the function stored in the PDL::Func object, if the scheme is `Hermite`.

The integration can either be between points of the original `x` array (`index`), or arbitrary x values (`x`). For both cases, a two element piddle should be given, to specify the start and end points of the integration.

index
: The values given refer to the indices of the points in the `x` array.

`x` The array contains the actual values to integrate between.

If the `status` method returns a value of -1, then one or both of the integration limits did not lie inside the `x` array. *Caveat emptor* with the result in such a case.

interlrgb

Module: PDL::ImageRGB

Make an RGB image from a palette image and its lookup table.

```
$rgb = $palette_im->interlrgb($lut)
```

Input should be of an integer type and the lookup table (3,x,...). Will perform the lookup for any N-dimensional input pdl (i.e. 0D, 1D, 2D, ...). Uses the index command but will not dataflow by default. If you want it to dataflow the dataflow_forward flag must be set in the $lut piddle (you can do that by saying $lut-¿*set_dataflow_f*(1)).

interpol

Module: PDL::Primitive

```
Signature: (xi(); x(n); y(n); [o] yi())
```

routine for 1D linear interpolation

```
$yi = interpol($xi, $x, $y)
```

`interpol` uses the same search method as the `interpolate` entry in the *interpolate*| manpage, hence `$x` must be *strictly* ordered (either increasing or decreasing). The difference occurs in the handling of out-of-bounds values; here an error message is printed.

interpolate

Module: PDL::Primitive

```
Signature: (xi(); x(n); y(n); [o] yi(); int [o] err())
```

routine for 1D linear interpolation

```
( $yi, $err ) = interpolate($xi, $x, $y)
```

Given a set of points `($x,$y)`, use linear interpolation to find the values `$yi` at a set of points `$xi`.

`interpolate` uses a binary search to find the suspects, er..., interpolation indices and therefore abscissas (ie `$x`) have to be *strictly* ordered (increasing or decreasing). For interpolation at lots of closely spaced abscissas an approach that uses the last index found as a start for the next search can be faster (compare Numerical Recipes `hunt` routine). Feel free to implement that on top of the binary search if you like. For out of bounds values it just does a linear extrapolation and sets the corresponding element of `$err` to 1, which is otherwise 0.

See also the `interpol` entry in the *interpol|* manpage, which uses the same routine, differing only in the handling of extrapolation — an error message is printed rather than returning an error piddle.

interpolate

Module: PDL::Func

```
my $yi = $obj->interpolate( $xi );
```

Returns the interpolated function at a given set of points (PDL::Func).

A status value of -1, as returned by the `status` method, means that some of the `$xi` points lay outside the range of the data. The values for these points were calculated by extrapolation (the details depend on the scheme being used).

intover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via integral to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the integral along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = intover($b);
```

```
$spectrum = intover $image->xchg(0,1)
```

Notes:

For `n > 3`, these are all `O(h^4)` (like Simpson's rule), but are integrals between the end points assuming the pdl gives values just at these centres: for such 'functions', sumover is correct to `O(h)`, but is the natural (and correct) choice for binned data, of course.

irfftw

Module: PDL::FFTW

Real inverse FFT. Have a look at rfftw to understand the format. USE ONLY an even n1! (complex input, real output)

```
$pdl_real = ifftw $pdl_cplx;
```

isbigendian

Module: PDL::IO::Misc

Determine endianness of machine — returns 0 or 1 accordingly

isempty

Module: PDL::Core

Test whether a piddle is empty

```
print "The piddle has zero dimension\n" if $pdl->isempty;
```

This function returns 1 if the piddle has zero elements. This is useful in particular when using the indexing function which. In the case of no match to a specified criterion, the returned piddle has zero dimension.

```
perldl> $a=sequence(10)
perldl> $i=which($a < -1)
perldl> print "I found no matches!\n" if ($a->isempty);
```

Note that having zero elements is rather different from the concept of being a null piddle, see the the *PDL::FAQ|PDL::FAQ* manpage and the *PDL::Indexing|PDL::Indexing* manpage manpages for discussions of this.

kcur

Module: PDL::Graphics::Karma

Return cursor position from a Karma application (e.g. kview/xray)

```
($x,$y) = kcur($ch, {App=>'karma-app',Coords=>"World|Pixel"})
```

This function connects to a Karma application and returns the ($x,$y) position and the character typed ($ch) by the user. By default world coordinates are returned.

```
print kcur {App=>"kview", Coords=>"World"}
```

kernctr

Module: PDL::FFT

'centre' a kernel (auxiliary routine to fftconvolve)

```
$kernel = kernctr($image,$smallk);
fftconvolve($image,$kernel);
```

kernctr centres a small kernel to emulate the behaviour of the direct convolution routines.

kim Module: PDL::Graphics::Karma

Sends piddle data array to an external Karma application for viewing

```
kim($pdl, [$karma-app, $lut])
```

Sends $pdl data to Karma application viewer. Remembers the last one used [default: kview].

koords

Module: PDL::Graphics::Karma

Starts external Karma application koords

koords([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

koverlay

Module: PDL::Graphics::Karma

```
Signature: (x(); y(); r(); ell(); PA(); fill(); int id(); char* karma_app; char* colo
```

koverlay

Module: PDL::Graphics::Karma

Overlay graphics markers on a Karma application (e.g. kview)

koverlay $x, $y, {Options...}

Currently the only markers supported are ellipses. The default is a circle of radius 10 units,

$x = 10∗*xvals*(10); koverlay $x, *sqrt*($x), {Radius=¿$x/3, Colour=¿'green', App=¿'kpolar'}

```
Radius - [piddle] specify radius of ellipses (major axis if ellipse). Default = 10 un
Ellip  - [piddle] specify ellipticity of ellipses. Default = 0 i.e. circle.
PA     - [piddle] specify principle axis (degrees rotation anticlockwise
                  from the Y axis). Default.
ID     - [piddle] Numeric integer id labels to apply.
Colour - [string] Colour name for overlay (e.g.  'red'). Default = 'blue'
App    - [string] name of Karma app to send too
Fill   - [piddle] whether outlines are filled (0 or 1). (Note filled,
                  ellipses are not yet available in Karma).
Coords - [string] "World" or "Pixel" - type of coordinates for x/y/r.
                  Note pixel implementation rounds to nearest pixel due
                  to Karma overlays not supporting proper IMAGE_PIXEL
                  coordinates.
```

kpvslice

Module: PDL::Graphics::Karma

Starts external Karma application kpvslice

kpvslice([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

krenzo

Module: PDL::Graphics::Karma

Starts external Karma application krenzo

krenzo([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

krgb

Module: PDL::Graphics::Karma

krgb($lut, [$karma-app])

Sends RGB image to an external Karma application for viewing

Does not change current default viewer.

kshell

Module: PDL::Graphics::Karma

Starts external Karma application kshell

kshell([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

kslice_3d

Module: PDL::Graphics::Karma

Starts external Karma application kslice_3d

kslice_3d([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

kstarted

Module: PDL::Graphics::Karma

kstarted([$karma-app])

Tests if a Karma application is running.

It tries to connect to the karma application, returns 1 on success, 0 otherwise

Can be used to check if a karma application has already been started, e.g.

```
xray unless kstarted 'xray';
```

kview

Module: PDL::Graphics::Karma

Starts external Karma application kview

kview([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

lags

Module: PDL::Slices

```
Signature: (P(); C(); int nthdim; int step; int n)
```

Returns a piddle of lags to parent.

I.e. if `$a` contains

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

then

```
$b = $a->lags(0,2,2);
```

is a (5,2) matrix

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

This order of returned indices is kept because the function is called "lags" i.e. the nth lag is n steps behind the original.

lattice3d

Module: PDL::Graphics::TriD

alias for mesh3d

lclip

Module: PDL::Primitive

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

clip `$a` by `$b` (`$b` is lower bound)

le Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

the binary ¡= (less equal) operation

```
$c = le $a, $b, 0;     # explicit call with trailing 0
$c = $a <= $b;           # overloaded call
$a->inplace->le($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `<=` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

lgamma

Module: PDL::Math

```
Signature: (a(); [o]b(); int[o]s())
```

log gamma function

This returns 2 piddles — the first set gives the *log*(gamma) values, while the second set, of integer values, gives the sign of the gamma function. This is useful for determining factorials, amongst other things.

line

Module: PDL::Graphics::PGPLOT

Plot vector as connected points

If the 'MISSING' option is specified, those points in the `$y` vector which are equal to the MISSING value are not plotted, but are skipped over. This allows one to quickly draw multiple lines with one call to `line`, for example to draw coastlines for maps.

```
Usage: line ( [$x,] $y, [$opt] )
```

The following standard options influence this command:

```
AXIS, BORDER, COLO(U)R, JUSTIFY, LINESTYLE, LINEWIDTH, MISSING
```

```
$x = sequence(10)/10.;
$y = sin($x)**2;
# Draw a red dot-dashed line
line $x, $y, {COLOR => 'RED', LINESTYLE=>3};
```

line3d

Module: PDL::Graphics::TriD

3D line plot, defined by a variety of contexts.

```
line3d piddle(3,x), {OPTIONS}
line3d [CONTEXT], {OPTIONS}
```

Example:

```
perldl> line3d [sqrt(rvals(zeroes(50,50))/2)]
- Lines on surface
perldl> line3d [$x,$y,$z]
- Lines over X, Y, Z
perldl> line3d $coords
- Lines over the 3D coordinates in $coords.
```

Note: line plots differ from mesh plots in that lines only go in one direction. If this is unclear try both!

See module documentation for more information on contexts and options

list

Module: PDL::Core

Convert piddle to perl list

```
@tmp = list $x;
```

Obviously this is grossly inefficient for the large datasets PDL is designed to handle. This was provided as a get out while PDL matured. It should now be mostly superseded by superior constructs, such as PP/threading. However it is still occasionally useful and is provied for backwards compatibility.

```
for (list $x) {
  # Do something on each value...
}
```

listindices

Module: PDL::Core

Convert piddle indices to perl list

```
@tmp = listindices $x;
```

`@tmp` now contains the values `0..nelem($x)`.

Obviously this is grossly inefficient for the large datasets PDL is designed to handle. This was provided as a get out while PDL matured. It should now be mostly superseded by superior constructs, such as PP/threading. However it is still occasionally useful and is provied for backwards compatibility.

```
for $i (listindices $x) {
  # Do something on each value...
}
```

lmfit

Module: PDL::Fit::LM

Levenberg-Marquardt fitting of a user supplied model function

```
($ym,$a,$covar,$iters) =
     lmfit $x, $y, $sig, \&expfunc, $a, {Maxiter => 300, Eps => 1e-3};
```

Options:

```
Maxiter:  maximum number of iterations before giving up
Eps:      convergence citerium for fit; success when normalized change
          in chisquare smaller than Eps
```

The user supplied sub routine reference should accept 4 arguments

- a vector of independent values $x
- a vector of fitting parameters
- a vector of dependent variables that will be assigned upon return
- a matrix of partial derivatives with respect to the fitting parameters that will be assigned upon return

As an example take this definition of a single exponential with 3 parameters (width, amplitude, offset):

```
sub expdec {
  my ($x,$par,$ym,$dyda) = @_;
  my ($a,$b,$c) = map {$par->slice("($_)")} (0..2);
  my $arg = $x/$a;
```

```
  my $ex = exp($arg);
  $ym .= $b*$ex+$c;
  my (@dy) = map {$dyda->slice(",($_)")} (0..2);
  $dy[0] .= -$b*$ex*$arg/$a;
  $dy[1] .= $ex;
  $dy[2] .= 1;
}
```

Note usage of the `.=` operator for assignment

In scalar context returns a vector of the fitted dependent variable. In list context returns fitted y-values, vector of fitted parameters, an estimate of the covariance matrix (as an indicator of goodness of fit) and number of iterations performed.

load_wisdom

Module: PDL::FFTW

Loads the wisdom from a file for better FFTW performance.

The wisdom is automatically saved when the program ends. It will be automagically called when the variable `$PDL::FFT::wisdom` is set to a file name. For example, the following is a useful idiom to have in your *.perldlrc* file:

```
$PDL::FFT::wisdom = "$ENV{HOME}/.fftwisdom"; # save fftw wisdom in this file
```

Explicit usage:

```
load_wisdom($fname);
```

log

Module: PDL::Ops

```
Signature: (a(); [o]b())
```

the natural logarithm

```
$b = log $a;
$a->inplace->log;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary `log` operator/function.

long

Module: PDL::Core

Convert to long datatype — see 'Datatype_conversions'

lt

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

the binary ¡ (less than) operation

```
$c = lt $a, $b, 0;     # explicit call with trailing 0
$c = $a < $b;           # overloaded call
$a->inplace->lt($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `<` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

lut_data

Module: PDL::Graphics::LUT

Load in the requested colour table and intensity ramp.

```
( $l, $r, $g, $b ) = lut_data( $table, [ $reverse, [ $ramp ] ] );
```

Returns the levels and r, g, b components of the colour table `$table`. If `$reverse` is 1 (defaults to **0** if not supplied), then the r, g, and b components are reversed before being returned. If not supplied, `$ramp` defaults to **"ramp"** (this is a linear intensity ramp).

The returned values are piddles containing values in the range 0 to 1 inclusive, and are floats.

lut_names

Module: PDL::Graphics::LUT

Return, as a list, the names of the available colour tables.

```
@tables = lut_names();
```

lut_ramps

Module: PDL::Graphics::LUT

Return, as a list, the names of the available intensity ramps.

```
@ramps = lut_names();
```

make_physical

Module: PDL::Core

Make sure the data portion of a piddle can be accessed from XS code.

```
$a->make_physical;
$a->call_my_xs_method;
```

Ensures that a piddle gets its own allocated copy of data. This obviously implies that there are certain piddles which do not have their own data. These are so called *virtual* piddles that make use of the *vaffine* optimisation (see the *PDL::Indexing|PDL::Indexing* manpage). They do not have their own copy of data but instead store only access information to some (or all) of another piddle's data.

Note: this function should not be used unless absolutely neccessary since otherwise memory requirements might be severly increased. Instead of writing your own XS code with the need to call `make_physical` you might want to consider using the PDL preprocessor (see the *PDL::PP|PDL::PP* manpage) which can be used to transparently access virtual piddles without the need to physicalise them (though there are exceptions).

mapflex

Module: PDL::IO::FlexRaw

Memory map a binary file with flexible format specification

```
($x,$y,...) = mapflex("filename" [, $hdr] [, $opts])
```

mapfraw

Module: PDL::IO::FastRaw

Memory map a raw format binary file (see the module docs also)

```
$pdl3 = mapfraw("fname2",{ReadOnly => 1});
```

The `mapfraw` command supports the following options (not all combinations make sense):

Dims, Datatype
: If creating a new file or if you want to specify your own header data for the file, you can give an array reference and a scalar, respectively.

Creat
: Create the file. Also writes out a header for the file.

Trunc
: Set the file size. Automatically enabled with `Creat`. NOTE: This also clears the file to all zeroes.

ReadOnly
: Disallow writing to the file.

maptextfraw

Module: PDL::IO::FastRaw

Memory map a text file (see the module docs also).

Note that this function maps the raw format so if you are using an operating system which does strange things to e.g. line delimiters upon reading a text file, you get the raw (binary) representation.

The file doesn't really need to be text but it is just mapped as one large binary chunk.

This function is just a convenience wrapper which firsts `stat`s the file and sets the dimensions and datatype.

```
$pdl4 = maptextfraw("fname", {options}
```

The options other than Dims, Datatype of `mapfraw` are supported.

matinv
Module: PDL::Slatec

Inverse of a square matrix

```
($inv) = matinv($mat)
```

matmult
Module: PDL::Primitive

```
Signature: (a(x,y),b(y,z),[o]c(x,z))
```

Matrix multiplication

We peruse the inner product to define matrix multiplication via a threaded inner product

max Module: PDL::Primitive

Return the maximum of all elements in a piddle

```
$x = max($data);
```

max2d_ind
Module: PDL::Image2D

```
Signature: (a(m,n); [o]val(); int [o]x(); int[o]y())
```

Return value/position of maximum value in 2D image

Contributed by Tim Jeness

maximum
Module: PDL::Primitive

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

Project via maximum to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the maximum along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = maximum($b);

$spectrum = maximum $image->xchg(0,1)
```

maximum_ind

Module: PDL::Primitive

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

Like maximum but returns the index rather than the value

maximum_n_ind

Module: PDL::Primitive

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

Returns the index of `m` maximum elements

Find minimum and maximum and their indices for a given piddle;

```
perldl> $a=pdl [[-2,3,4],[1,0,3]]
perldl> ($min, $max, $min_ind, $max_ind)=minmaximum($a)
perldl> p $min, $max, $min_ind, $max_ind
[-2 0] [4 3] [0 1] [2 2]
```

See also the `minmax` entry in the *minmax|* manpage, which clumps the piddle together.

med2d

Module: PDL::Image2D

```
Signature: (a(m,n); kern(p,q); [o]b(m,n); int opt)
```

2D median-convolution of an array with a kernel (smoothing)

Note: only points in the kernel ¿0 are included in the median, other points are weighted by the kernel value (medianing lots of zeroes is rather pointless)

```
$new = med2d $old, $kernel, {OPTIONS}
```

```
$smoothed = med2d $image, ones(3,3), {Boundary => Reflect}

Boundary - controls what values are assumed for the image when kernel
           crosses its edge:
           => Default  - periodic boundary conditions (i.e. wrap around axis)
           => Reflect  - reflect at boundary
           => Truncate - truncate at boundary
```

median

Module: PDL::Primitive

Return the median of all elements in a piddle

```
$x = median($data);
```

medover

Module: PDL::Primitive

```
 Signature: (a(n); [o]b(); [t]tmp(n))
```

Project via median to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the median along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = medover($b);

$spectrum = medover $image->xchg(0,1)
```

mesh3d

Module: PDL::Graphics::TriD

3D mesh plot, defined by a variety of contexts

```
mesh3d piddle(3,x,y), {OPTIONS}
mesh3d [piddle,...], {OPTIONS}
```

Example:

```
perldl> mesh3d [sqrt(rvals(zeroes(50,50))/2)]

- mesh of surface
```

Note: a mesh is defined by two sets of lines at right-angles (i.e. this is how is differs from line3d).

See module documentation for more information on contexts and options

min Module: PDL::Primitive

Return the minimum of all elements in a piddle

```
$x = min($data);
```

minimum

Module: PDL::Primitive

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

Project via minimum to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the minimum along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = minimum($b);

$spectrum = minimum $image->xchg(0,1)
```

minimum_ind

Module: PDL::Primitive

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

Like minimum but returns the index rather than the value

minimum_n_ind

Module: PDL::Primitive

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

Returns the index of `m` minimum elements

minmax

Module: PDL::Primitive

Returns an array with minimum, maximum of a piddle.

```
($mn, $mx) = minmax($pdl);
```

Return `$mn` as minimum, `$mx` as maximum, `$mn_ind` as the index of minimum and `$mx_ind` as the index of the maximum.

```
perldl> $x = pdl [1,-2,3,5,0]
perldl> ($min, $max) = minmax($x);
perldl> p "$min $max\n";
-2 5
```

minmaximum

Module: PDL::Primitive

```
Signature: (a(n); [o]cmin(); [o] cmax(); int [o]cmin_ind(); int [o]cmax_ind())
```

info not available

minus

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

subtract two piddles

```
$c = minus $a, $b, 0;     # explicit call with trailing 0
$c = $a - $b;           # overloaded call
$a->inplace->minus($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `-` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

modulo

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

elementwise `modulo` operation

```
$c = $a->modulo($b,0); # explicit function call
$c = $a % $b;    # overloaded use
$a->inplace->modulo($b,0);     # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `op%` function. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

mslice

Module: PDL::Core

Convenience interface to the section on *PDL::Slice/slice* in the *slice|* manpage, allowing easier inclusion of dimensions in perl code.

```
$a = $x->mslice(...);
```

```
# below is the same as $x->slice("5:7,:,3:4:2")
$a = $x->mslice([5,7],X,[3,4,2]);
```

mult

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

multiply two piddles

```
$c = mult $a, $b, 0;     # explicit call with trailing 0
$c = $a * $b;           # overloaded call
$a->inplace->mult($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `*` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

mv Module: PDL::Slices

```
Signature: (P(); C(); int n1; int n2)
```

move a dimension to another position

The command

```
$b = $a->mv(4,1);
```

creates `$b` to be like `$a` except that the dimension 4 is moved to the place 1, so:

```
$b->at(1,2,3,4,5,6) == $a->at(1,5,2,3,4,6);
```

The other dimensions are moved accordingly. Negative dimension indices count from the end.

ndtri

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The value for which the area under the Gaussian probability density function (integrated from minus infinity) is equal to the argument (cf the `erfi` entry in the *erfi*| manpage)

ne Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

binary *not equal to* operation (`!=`)

```
$c = ne $a, $b, 0;     # explicit call with trailing 0
$c = $a != $b;           # overloaded call
$a->inplace->ne($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `!=` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

nelem

Module: PDL::Core

Return the number of elements in a piddle

```
$n = nelem($piddle); $n = $piddle->nelem;

$mean = sum($data)/nelem($data);
```

new Module: PDL::Char

Function to create a byte PDL from a string, list of strings, list of list of strings, etc.

```
# create a new PDL::Char from a perl array of strings
$strpdl = PDL::Char->new( ['abc', 'def', 'ghij'] );

# Convert a PDL of type 'byte' to a PDL::Char
$strpdl1 = PDL::Char->new (sequence (byte, 4, 5)+99);

$pdlchar3d = PDL::Char->new([['abc','def','ghi'],['jkl', 'mno', 'pqr']]);
```

new Module: PDL::Core

new piddle constructor method

```
$x = PDL->new(SCALAR|ARRAY|ARRAY REF);

$x = PDL->new(42);
$y = new PDL [1..10];
```

Constructs piddle from perl numbers and lists.

new_from_specification
Module: PDL::Core

Internal method: create piddle by specification

This is the argument processing method called by the `zeroes` entry in the *zeroes*| manpage and some other functions which constructs piddles from argument listss of the form:

```
[type], $nx, $ny, $nz,...
```

nfftw
Module: PDL::FFTW

Complex inplace FFT (complex input, complex output).

```
$pdl_cplx = nfftw $pdl_cplx;
```

ninterpol
Module: PDL::ImageND

N-dimensional interpolation routine

```
Signature: ninterpol(point(),data(n),[o]value())

    $value = ninterpol($point, $data);
```

`ninterpol` uses `interpol` to find a linearly interpolated value in N dimensions, assuming the data is spread on a uniform grid. To use an arbitrary grid distribution, need to find the grid-space point from the indexing scheme, then call `ninterpol` — this is far from trivial (and ill-defined in general).

norm
Module: PDL::Primitive

```
Signature: (vec(n); [o] norm(n))
```

Normalises a vector to unit Euclidean length

not Module: PDL::Ops

```
Signature: (a(); [o]b())
```

the elementwise *not* operation

```
$b = ! $a;
$a->inplace->not;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary `!` operator/function.

nrfftw

Module: PDL::FFTW

Real inplace FFT. If you want a transformation on a piddle with dimensions [n1,n2,....] you MUST pass in a piddle with dimensions [2*(n1/2+1),n2,...] (real input, complex output).

Use with care due to dimension restrictions mentioned below. For details check the html docs that come with the fftw library.

```
$pdl_cplx = nrfftw $pdl_real;
```

null

Module: PDL::Core

Returns a 'null' piddle.

```
$x = null;
```

`null()` has a special meaning to the section on in the *PDL::PP|PDL::PP* manpage. It is used to flag a special kind of empty piddle, which can grow to appropriate dimensions to store a result (as opposed to storing a result in an existing piddle).

```
perldl> sumover sequence(10,10), $ans=null;p $ans
[45 145 245 345 445 545 645 745 845 945]
```

nullcreate

Module: PDL::Core

Returns a 'null' piddle.

```
$x = PDL->nullcreate($arg)
```

This is an routine used by many of the threading primitives (i.e. the `sumover` entry in the *sumover|PDL::Primitive* manpage, the `minimum` entry in the *minimum|PDL::Primitive* manpage, etc.) to generate a null piddle for the function's output that will behave properly for derived (or subclassed) PDL objects.

For the above usage: If `$arg` is a PDL, or a derived PDL, then `$arg->null` is returned. If `$arg` is a scalar (i.e. a zero-dimensional PDL) then `$PDL->null` is returned.

```
PDL::Derived->nullcreate(10)
  returns PDL::Derived->null.
PDL->nullcreate($pldderived)
  returns $pldderived->null.
```

oddmedian

Module: PDL::Primitive

Return the oddmedian of all elements in a piddle

```
$x = oddmedian($data);
```

oddmedover

Module: PDL::Primitive

```
Signature: (a(n); [o]b(); [t]tmp(n))
```

Project via oddmedian to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the oddmedian along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = oddmedover($b);
```

```
$spectrum = oddmedover $image->xchg(0,1)
```

The median is sometimes not a good choice as if the array has an even number of elements it lies half-way between the two middle values — thus it does not always correspond to a data value. The lower-odd median is just the lower of these two values and so it ALWAYS sits on an actual data value which is useful in some circumstances.

one2nd

Module: PDL::Primitive

Converts a one dimensional index piddle to a set of ND coordinates

```
@coords=one2nd($a, $indices)
```

returns an array of piddles containing the ND indexes corresponding to the one dimensional list indices. The indices are assumed to correspond to array `$a` clumped using `clump(-1)`. This routine is used in the `whichND` entry in the *whichND|* manpage, but is useful on its own occasionally.

```
perldl> $a=pdl [[[1,2],[-1,1]], [[0,-3],[3,2]]]; $c=$a->clump(-1)
perldl> $maxind=maximum_ind($c); p $maxind;
6
perldl> print one2nd($a, maximum_ind($c))
0 1 1
perldl> p $a->at(0,1,1)
3
```

ones

Module: PDL::Core

construct a one filled piddle

```
$a = ones([type], $nx, $ny, $nz,...);
etc. (see 'zeroes')

see zeroes() and add one
```

oneslice

Module: PDL::Slices

```
Signature: (P(); C(); int nth; int from; int step; int nsteps)
```

experimental function — not for public use

```
$a = oneslice();
```

This is not for public use currently. See the source if you have to. This function can be used to accomplish run-time changing of transformations i.e. changing the size of some piddle at run-time.

However, the mechanism is not yet finalized and this is just a demonstration.

or Module: PDL::Primitive

Return the logical or of all elements in a piddle

```
$x = or($data);
```

or2 Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

binary *or* of two piddles

```
$c = or2 $a, $b, 0;     # explicit call with trailing 0
$c = $a | $b;           # overloaded call
$a->inplace->or2($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `|` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

orover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via or to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the or along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = orover($b);
```

```
$spectrum = orover $image->xchg(0,1)
```

outer

Module: PDL::Primitive

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

outer product over one dimension

Naturally, it is possible to achieve the effects of outer product simply by threading over the "`*`" operator but this function is provided for convenience.

patch2d

Module: PDL::Image2D

```
Signature: (a(m,n); int bad(m,n); [o]b(m,n))
```

patch bad pixels out of 2D images,

```
$patched = patch2d $data, $bad;
```

`$bad` is a 2D mask array where 1=bad pixel 0=good pixel. Pixels are replaced by the average of their non-bad neighbours.

pcoef

Module: PDL::Slatec

```
Signature: (int l();c();[o]tc(bar);a(foo))
```

Convert the `polfit` coefficients to Taylor series form. `c` and `a()` must be of the same type.

pdl Module: PDL::Core

piddle constructor — creates new piddle from perl scalars/arrays

```
$a = pdl(SCALAR|ARRAY REFERENCE|ARRAY);

$a = pdl [1..10];             # 1D array
$a = pdl ([1..10]);           # 1D array
$a = pdl (1,2,3,4);           # Ditto
$b = pdl [[1,2,3],[4,5,6]];   # 2D 3x2 array
$b = pdl 42                   # 0-dimensional scalar
$c = pdl $a;                  # Make a new copy
$a = pdl([1,2,3],[4,5,6]);    # 2D
$a = pdl([[1,2,3],[4,5,6]]);  # 2D
```

Note the last two are equivalent — a list is automatically converted to a list reference for syntactic convenience. i.e. you can omit the outer `[]`

`pdl()` is a functional synonym for the 'new' constructor, e.g.:

```
$x = new PDL [1..10];
```

In order to control how undefs are handled in converting from perl lists to PDLs, one can set the variable `$PDL::undefval`. For example:

```
$foo = [[1,2,undef],[undef,3,4]];
$PDL::undefval = -999;
$f = pdl $foo;
print $f
[
 [   1    2 -999]
 [-999    3    4]
]
```

`$PDL::undefval` defaults to zero.

plus

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

add two piddles

```
$c = plus $a, $b, 0;     # explicit call with trailing 0
$c = $a + $b;           # overloaded call
$a->inplace->plus($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `+` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

pnminascii

Module: PDL::IO::Pnm

```
Signature: (type(); byte+ [o] im(m,n); int ms => m; int ns => n;
                       int format; char* fd)
```

Read in an ascii pnm file.

pnminraw

Module: PDL::IO::Pnm

```
Signature: (type(); byte+ [o] im(m,n); int ms => m; int ns => n;
                       int isbin; char* fd)
```

Read in a raw pnm file.

read a raw pnm file. The `type` argument is only there to determine the type of the operation when creating `im` or trigger the appropriate type conversion (maybe we want a byte+ here so that `im` follows *strictly* the type of `type`).

pnmout

Module: PDL::IO::Pnm

```
Signature: (a(m); int israw; int isbin; char *fd)
```

Write a line of pnm data.

This function is implemented this way so that threading works naturally.

poco

Module: PDL::Slatec

```
Signature: (a(n,n);rcond();[o]z(n);int [o]info())
```

Factor a real symmetric positive definite matrix and estimate the condition number of the matrix.

podi

Module: PDL::Slatec

```
Signature: (a(n,n);[o]det(two=2);int job())
```

Compute the determinant and inverse of a certain real symmetric positive definite matrix using the factors computed by the `poco` entry in the *poco*| manpage.

points

Module: PDL::Graphics::PGPLOT

Plot vector as points

```
Usage: points ( [$x,] $y, [$symbol(s)], [$opt] )
```

Options recognised:

```
  SYMBOL - Either a piddle with the same dimensions as $x, containing
           the symbol associated to each point or a number specifying
           the symbol to use for every point, or a name specifying the
           symbol to use according to the following (recognised name in
            capital letters):
            0 - SQUARE   1 - DOT     2 - PLUS     3 - ASTERISK
            4 - CIRCLE   5 - CROSS   7 - TRIANGLE 8 - EARTH
            9 - SUN     11 - DIAMOND 12- STAR
PLOTLINE - If this is >0 a line will be drawn through the points.
```

The following standard options influence this command:

```
AXIS, BORDER, CHARSIZE, COLOUR, JUSTIFY, LINESTYLE, LINEWIDTH
```

```
$y = sequence(10)**2+random(10);
# Plot blue stars with a solid line through:
points $y, {PLOTLINE => 1, COLOUR => BLUE, SYMBOL => STAR};
```

points3d

Module: PDL::Graphics::TriD

3D points plot, defined by a variety of contexts

```
points3d piddle(3), {OPTIONS}
points3d [piddle,...], {OPTIONS}
```

Example:

```
perldl> points3d [sqrt(rvals(zeroes(50,50))/2)];
- points on surface
```

See module documentation for more information on contexts and options

polfit

Module: PDL::Slatec

```
Signature: (x(n);y(n);w(n);int maxdeg();int [o]ndeg();[o]eps();[o]r(n);int [o]ierr();
```

Fit discrete data in a least squares sense by polynomials in one variable. `x()`, `y()` and `w()` must be of the same type.

poly

Module: PDL::Graphics::PGPLOT

Draw a polygon

```
Usage: poly ( $x, $y )
```

Options recognised:

The following standard options influence this command:

```
AXIS, BORDER, COLOUR, FILLTYPE, HATCHING, JUSTIFY, LINESTYLE,
LINEWIDTH

# Fill with hatching in two different colours
$x=sequence(10)/10;
# First fill with cyan hatching
poly $x, $x**2, {COLOR=>5, FILL=>3};
hold;
# Then do it over again with the hatching offset in phase:
poly $x, $x**2, {COLOR=>6, FILL=>3, HATCH=>{PHASE=>0.5}};
release;
```

polycoef

Module: PDL::Slatec

Convenience wrapper routine around the `pcoef slatec` function. Separates supplied arguments and return values.

Convert the `polyfit/polfit` coefficients to Taylor series form.

```
$tc = polycoef($l, $c, $a);
```

polyfill

Module: PDL::Image2D

```
Signature: (int [o,nc] im(m,n); float ps(two=2,np); int col())
```

fill the area inside the given polygon with a given colour

This function works inplace, i.e. modifies `im`.

polyfillv

Module: PDL::Image2D

return the (dataflown) area of an image within a polygon

```
# increment intensity in area bounded by $poly
$im->polyfillv($pol)++; # legal in perl >= 5.6
# compute average intensity within area bounded by $poly
$av = $im->polyfillv($poly)->avg;
```

polyfit

Module: PDL::Slatec

Convenience wrapper routine about the `polfit slatec` function. Separates supplied arguments and return values.

Fit discrete data in a least squares sense by polynomials in one variable.

```
($ndeg, $r, $ierr, $a) = polyfit($x, $y, $w, $maxdeg, $eps);
```

`eps` is modified to contain the rms error of the fit.

polyvalue

Module: PDL::Slatec

Convenience wrapper routine around the `pvalue slatec` function. Separates supplied arguments and return values.

For multiple input x positions, a corresponding y position is calculated.

The derivatives PDL is one dimensional (of size `nder`) if a single x position is supplied, two dimensional if more than one x position is supplied.

Use the coefficients generated by `polyfit` (or `polfit`) to evaluate the polynomial fit of degree `l`, along with the first `nder` of its derivatives, at a specified point.

```
($yfit, $yp) = polyvalue($l, $nder, $x, $a);
```

Populate

Module: PDL::Graphics::TriD::Tk

Used for widget initialization by Tk, this function should never be called directly

pow Module: PDL::Math

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

Synonym for '**'

power

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

raise piddle `$a` to the power `b`

```
$c = $a->power($b,0); # explicit function call
$c = $a ** $b;    # overloaded use
$a->inplace->power($b,0);     # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `op**` function. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

prodover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via product to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the product along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = prodover($b);

$spectrum = prodover $image->xchg(0,1)
```

propndfx

Module: PDL::IO::NDF

Routine to write a PDL to an NDF by copying the extension information from an existing NDF and writing DATA,VARIANCE, QUALITY and AXIS info from a PDL (if they exist).

Extensions, labels and history are propogated from the old NDF. No new extension information is written.

This command has been superseded by the `wndf()` entry in the *wndf()|* manpage.

pvalue

Module: PDL::Slatec

```
Signature: (int l();x();[o]yfit();[o]yp(nder);a(foo))
```

Use the coefficients generated by `polfit` to evaluate the polynomial fit of degree `l`, along with the first `nder` of its derivatives, at a specified point. `x` and `a` must be of the same type.

px Module: PDL::Dbg

Print info about a piddle (or all known piddles)

```
perldl> PDL->px
perldl> $b += $a->clump(2)->px('clumptest')->sumover
perldl> $a->px('%C (%A) Type: %T')
```

This function prints some information about piddles. It can be invoked as a class method (e.g. `PDL->px`) or as an instance method (e.g. C<$pdl-¿*px*($arg)>). If

invoked as a class method
: it prints info about all piddles found in the current package (*excluding* `my` variables). This comes in quite handy when you are not quite sure which pdls you have already defined, what data they hold , etc. `px` is supposed to support inheritance and prints info about all symbols for which an `isa($class)` is true. An optional string argument is interpreted as the package name for which to print symbols:

```
perldl> PDL->px('PDL::Mypack')
```

The default package is that of the caller.

invoked as an instance method
: it prints info about that particular piddle if `$PDL::debug` is true and returns the pdl object upon completion. It accepts an optional string argument that is simply prepended to the default info if it doesn't contain a `%` character. If, however, the argument contains a `%` then the string is passed to the `info` method to control the format of the printed information. This can be used to achieve customized output from `px`. See the documentation of `PDL::info` for further details.

The output of px will be determined by the default formatting string that is passed to the `info` method (unless you pass a string containing `%` to px when invoking as an instance method, see above). This default string is stored in `$PDL::Dbg::Infostr` and the default output format can be accordingly changed by setting this variable. If you do this you should also change the default title string that the class method branch prints at the top of the listing to match your new format string. The default title is stored in the variable `$PDL::Dbg::Title`.

For historical reasons `vars` is an alias for `px`.

qsort
: Module: PDL::Primitive

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

Quicksort a vector into ascending order.

```
print qsort random(10);
```

qsorti
Module: PDL::Primitive

```
Signature: (a(n); int [o]indx(n))
```

Quicksort a vector and return index of elements in ascending order.

```
$ix = qsorti $a;
print $a->index($ix); # Sorted list
```

r2C Module: PDL::Complex

```
Signature: (r(); [o]c(m=2))
```

convert real to complex, assuming an imaginary part of zero

random
Module: PDL::Primitive

Constructor which returns piddle of random numbers

```
$a = random([type], $nx, $ny, $nz,...);
$a = random $b;
```

etc (see the `zeroes` entry in the *zeroes|PDL::Core* manpage).

This is the uniform distribution between 0 and 1 (assumedly excluding 1 itself). The arguments are the same as `zeroes` (q.v.) — i.e. one can specify dimensions, types or give a template.

randsym
Module: PDL::Primitive

Constructor which returns piddle of random numbers

```
$a = randsym([type], $nx, $ny, $nz,...);
$a = randsym $b;
```

etc (see the `zeroes` entry in the *zeroes|PDL::Core* manpage).

This is the uniform distribution between 0 and 1 (excluding both 0 and 1, cf the `random` entry in the *random|* manpage). The arguments are the same as `zeroes` (q.v.) — i.e. one can specify dimensions, types or give a template.

rasc
Module: PDL::IO::Misc

Simple function to slurp in ASCII numbers quite quickly, although error handling is marginal (to nonexistent).

```
$pdl->rasc("filename" [,$noElements]);
    Where:
      filename is the name of the ASCII file to read
      $noElements is the optional number of elements in the file to read.
           (If not present, all of the file will be read to fill up $pdl)

#  (test.num is an ascii file with 20 numbers. One number per line.)
$in = PDL->null;
$num = 20;
$in->rasc('test.num',20);
$imm = zeroes(float,20,2);
$imm->rasc('test.num');
```

rcols

Module: PDL::IO::Misc

Read ASCII whitespaced cols from a file into piddles and perl arrays (also see the `rgrep()` entry elsewhere in this document).

There are two calling conventions — the old version, where a pattern can be specified after the filename/handle, and the new version where options are given as as hash reference. This reference can be given as either the second or last argument.

The default behaviour is to ignore lines beginning with a # character and lines that only consist of whitespace. Options exist to only read from lines that match, or do not match, supplied patterns, and to set the types of the created piddles.

Can take file name or ∗HANDLE, and if no columns are specified, all are assumed. For the allowed types, see the section on *Datatype_conversions* in the *PDL::Core* manpage.

Options:

EXCLUDE or IGNORE — ignore lines matching this pattern (default '/^#/').

INCLUDE or KEEP — only use lines which match this pattern (default '').

LINES — which line numbers to use. Line numbers start at 0 and the syntax is 'a:b:c' to use every c'th matching line between a and b (default '').

DEFTYPE — default data type for stored data (if not specified, use the type stored in `$PDL::IO::Misc::deftype`, which starts off as **double**).

TYPES — reference to an array of data types, one element for each column to be read in. Any missing columns use the DEFTYPE value (default `[]`).

PERLCOLS — an array of column numbers which are to be read into perl arrays rather than piddles. References to these arrays are returned after the requested piddles (default **undef**).

```
Usage:
  ($x,$y,...) = rcols( *HANDLE|"filename", { EXCLUDE => '/^!/' },
                       $col1, $col2, ... )
  ($x,$y,...) = rcols( *HANDLE|"filename", $col1, $col2, ...,
                       { EXCLUDE => '/^!/' } )
  ($x,$y,...) = rcols( *HANDLE|"filename", "/foo/", $col1, $col2, ... )
```

e.g.,

```
$x      = PDL->rcols 'file1';
($x,$y) = rcols *STDOUT;

# read in lines containing the string foo, where the first
# example also ignores lines that with a # character.
($x,$y,$z) = rcols 'file2', 0,4,5, { INCLUDE => '/foo/' };
($x,$y,$z) = rcols 'file2', 0,4,5,
               { INCLUDE => '/foo/', EXCLUDE => '' };

# ignore the first 27 lines of the file, reading in as ushort's
($x,$y) = rcols 'file3', { LINES => '27:-1', DEFTYPE => ushort };
($x,$y) = rcols 'file3',
            { LINES => '27:', TYPES => [ ushort, ushort ] };

# read in the first column as a perl array and the next two as piddles
($x,$y,$name) = rcols 'file4', 1, 2, { PERLCOLS => [ 0 ] };
printf "Number of names read in = %d\n", 1 + $#$name;
```

Notes:

1. Quotes are required on patterns.

2. Columns are separated by whitespace by default, use `$PDL::IO::Misc::colsep` to specify an alternate separator.

3. For PDL-2.003, the meaning of the 'c' value in the LINES option has changed: it now only counts matching lines rather than all lines as in previous versions of PDL.

4. LINES =¿ '−1:0:3' may not work as you expect, since lines are skipped when read in, then the whole array reversed.

rCpolynomial

Module: PDL::Complex

```
Signature: (coeffs(n); x(c=2,m); [o]out(c=2,m))
```

evaluate the polynomial with (real) coefficients `coeffs` at the (complex) *position*(s) `x`. `coeffs[0]` is the constant term.

rcube

Module: PDL::IO::Misc

Read list of files directly into a large data cube (for efficiency)

```
$cube = rcube \&reader_function, @files;

$cube = rcube \&rfits, glob("*.fits");
```

This IO function allows direct reading of files into a large data cube, Obviously one could use *cat()* but this is more memory efficient.

The reading function (e.g. rfits, readfraw) (passed as a reference) and files are the arguments.

The cube is created as the same X,Y dims and datatype as the first image specified. The Z dim is simply the number of images.

rdsa

Module: PDL::IO::Misc

Read a FIGARO/NDF format file.

Requires non-PDL DSA module. Contact Frossie (frossie@jach.hawaii.edu) Usage:

```
([$xaxis],$data) = rdsa($file)

$a = rdsa 'file.sdf'
```

Not yet tested with PDL-1.9X versions

readflex

Module: PDL::IO::FlexRaw

Read a binary file with flexible format specification

```
($x,$y,...) = readflex("filename" [, $hdr])
```

readfraw

Module: PDL::IO::FastRaw

Read a raw format binary file

```
$pdl2 = readfraw("fname");
$pdl2 = PDL->readfraw("fname");
```

realfft

Module: PDL::FFT

One-dimensional FFT of real function [inplace].

The real part of the transform ends up in the first half of the array and the imaginary part of the transform ends up in the second half of the array.

```
realfft($real);
```

realifft

Module: PDL::FFT

Inverse of one-dimensional realfft routine [inplace].

```
realifft($real);
```

rebin

Module: PDL::ImageND

```
Signature: (a(m); [o]b(n); int ns => n)
```

rebin

Module: PDL::ImageND

N-dimensional rebinning algorithm

$new = rebin $a, $dim1, $dim2,..;. $new = rebin $a, $template; $new = rebin $a, $template, {Norm =¿ 1};

Rebin an N-dimensional array to newly specified dimensions. Specifying 'Norm' keeps the sum constant, otherwise the intensities are kept constant. If more template dimensions are given than for the input pdl, these dimensions are created; if less, the final dimensions are maintained as they were.

So if `$a` is a 10 x 10 pdl, then `rebin($a,15)` is a 15 x 10 pdl, while `rebin($a,15,16,17)` is a 15 x 16 x 17 pdl (where the values along the final dimension are all identical).

refresh

Module: PDL::Graphics::TriD::Tk

refresh() causes a display event to be put at the top of the TriD work que. This should be called at the end of each user defined TriD::Tk callback.

reorder

Module: PDL::Slices

Re-orders the dimensions of a PDL based on the supplied list.

Similar to the the `xchg` entry in the *xchg|* manpage method, this method re-orders the dimensions of a PDL. While the the `xchg` entry in the *xchg|* manpage method swaps the position of two dimensions, the reorder method can change the positions of many dimensions at once.

```
# Completely reverse the dimension order of a 6-Dim array.
$reOrderedPDL = $pdl->reorder(5,4,3,2,1,0);
```

The argument to reorder is an array representing where the current dimensions should go in the new array. In the above usage, the argument to reorder `(5,4,3,2,1,0)` indicates that the old dimensions (`$pdl`'s dims) should be re-arranged to make the new pdl (`$reOrderPDL`) according to the following:

```
Old Position   New Position
------------   ------------
5              0
4              1
3              2
2              3
1              4
0              5
```

Example:

```
perldl> $a = sequence(5,3,2);    # Create a 3-d Array
perldl> p $a
[
 [
  [ 0  1  2  3  4]
  [ 5  6  7  8  9]
  [10 11 12 13 14]
 ]
 [
  [15 16 17 18 19]
  [20 21 22 23 24]
  [25 26 27 28 29]
 ]
]
perldl> p $a->reorder(2,1,0); # Reverse the order of the 3-D PDL
[
 [
  [ 0 15]
  [ 5 20]
  [10 25]
 ]
 [
  [ 1 16]
  [ 6 21]
  [11 26]
 ]
 [
  [ 2 17]
  [ 7 22]
```

```
 [12 27]
]
[
 [ 3 18]
 [ 8 23]
 [13 28]
]
[
 [ 4 19]
 [ 9 24]
 [14 29]
]
]
```

The above is a simple example that could be duplicated by calling `$a->xchg(0,2)`, but it demonstrates the basic functionality of reorder.

As this is an index function, any modifications to the result PDL will change the parent.

repulse

Module: PDL::Graphics::TriD::Rout

```
Signature: (coords(nc,np);
                [o]vecs(nc,np);
                int [t]links(np);;
               double boxsize;
               int dmult;
               double a;
               double b;
               double c;
               double d;
      )
```

Repulsive potential for molecule-like constructs.

`repulse` uses a hash table of cubes to quickly calculate a repulsive force that vanishes at infinity for many objects. For use by the module the *PDL::Graphics::TriD::MathGraph*|*PDL::Graphics::TriD::MathGraph* manpage. For definition of the potential, see the actual function.

rescale2d

Module: PDL::Image2D

```
Signature: (I(n,m); O(q,p))
```

The first piddle is rescaled to the dimensions of the second (expandind or meaning values as needed) and then added to it.

reshape

Module: PDL::Core

Change the shape (i.e. dimensions) of a piddle, preserving contents.

```
$x->reshape(NEWDIMS); reshape($x, NEWDIMS);
```

The data elements are preserved, obviously they will wrap differently and get truncated if the new array is shorter. If the new array is longer it will be zero-padded.

Note: an explicit copy is forced — this is the only way (for now) of stopping a crash if `$x` is a slice.

```
perldl> $x = sequence(10)
perldl> reshape $x,3,4; p $x
[
 [0 1 2]
 [3 4 5]
 [6 7 8]
 [9 0 0]
]
perldl> reshape $x,5; p $x
[0 1 2 3 4]
```

rfftw

Module: PDL::FFTW

Real FFT. For an input piddle of dimensions `[n1,n2,...]` the output is `[2,(n1/2)+1,n2,...]` (real input, complex output).

```
$pdl_cplx = fftw $pdl_real;
```

rfftwconv

Module: PDL::FFTW

ND convolution using real ffts from the FFTW library

```
$conv = rfftwconv $im, kernctr $k;
```

rfits

Module: PDL::IO::Misc

Simple piddle FITS reader.

```
$pdl = rfits('file.fits');
```

Suffix magic:

```
# Automatically uncompress via gunzip pipe
$pdl = rfits('file.fits.gz');
# Automatically uncompress via uncompress pipe
$pdl = rfits('file.fits.Z');
```

FITS Headers stored in piddle and can be retrived with $a-¿gethdr.

This header is a reference to a hash where the hash keys are the keywords in the FITS header. It is important to note that for strings, the surrounding quotes are kept to ensure that strings that look like numbers are kept as strings. This is also of importance if you create your own header information and you want the value to be printed out as a string.

Comments in headers are stored as $$h{COMMENT}{¡Keyword¿} where $h is the header retrieved with $a-¿gethdr. History entries in the header are stored as $$h{HISTORY}, which is an anonymous array for each HISTORY entry in the header.

rgbtogr

Module: PDL::ImageRGB

Converts an RGB image to a grey scale using standard transform

```
$gr = $rgb->rgbtogr
```

Performs a conversion of an RGB input image (3,x,....) to a greyscale image (x,.....) using standard formula:

```
Grey = 0.301 R + 0.586 G + 0.113 B
```

rgrep

Module: PDL::IO::Misc

Read columns into piddles using full regexp pattern matching.

Usage

```
($x,$y,...) = rgrep(sub, *HANDLE|"filename")
```

e.g.

```
($a,$b) = rgrep {/Foo (.*) Bar (.*) Mumble/} $file;
```

i.e. the vectors `$a` and `$b` get the progressive values of `$1`, `$2` etc.

rint

Module: PDL::Math

```
Signature: (a(); [o]b())
```

Round to integral values in floating-point format

rld Module: PDL::Slices

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

Run-length decode a vector

Given a vector `$a` of the numbers of instances of values `$b`, run-length decode to `$c`.

```
rld($a,$b,$c=null);
```

rle Module: PDL::Slices

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

Run-length encode a vector

Given vector `$c`, generate a vector `$a` with the number of each element, and a vector `$b` of the unique values. Only the elements up to the first instance of '0' in `$a` should be considered.

```
rle($c,$a=null,$b=null);
```

rndf

Module: PDL::IO::NDF

Reads a piddle from a NDF format data file.

```
$pdl = rndf('file.sdf');
$pdl = rndf('file.sdf',1);
```

The '.sdf' suffix is optional. The optional second argument turns off automatic quality masking and returns a quality array as well.

Header information and NDF Extensions are stored in the piddle as a hash which can be retreived with the `$pdl->gethdr` command. Array extensions are stored in the header as follows:

```
$a - the base DATA_ARRAY
```

If `$hdr = $a->gethdr`;

then:

```
%{$hdr}        contains all the FITS headers plus:
$$hdr{Error}   contains the Error/Variance PDL
$$hdr{Quality} The quality byte array (if reqeusted)
@{$$hdr{Axis}} Is an array of piddles containing the information
               for axis 0, 1, etc.
$$hdr{NDF_EXT} Contains all the NDF extensions
$$hdr{Hist}    Contains the history information
$$hdr{NDF_EXT}{_TYPES} - Data types for non-PDL NDF extensions so that
                         wndf can reconstruct a NDF.
```

All extension information is stored in the header hash array. Extension structures are preserved in hashes, so that the PROJ_PARS component of the IRAS.ASTROMETRY extension is stored in $$hdr{NDF_EXT}{IRAS}{ASTROMETRY}{'PROJ_P
All array structures are stored as arrays in the Hdr: numeric arrays are stored as PDLs, logical and character arrays are stored as plain Perl arrays. FITS arrays are a special case and are expanded as scalars into the header.

rot2d

Module: PDL::Image2D

```
Signature: (im(m,n); float angle(); bg(); int aa(); [o] om(p,q))
```

rotate an image by given `angle`

```
# rotate by 10.5 degrees with antialiasing, set missing values to 7
$rot = $im->rot2d(10.5,7,1);
```

This function rotates an image through an `angle` between −90 and + 90 degrees. Uses/doesn't use antialiasing depending on the `aa` flag. Pixels outside the rotated image are set to `bg`.

Code modified from pnmrotate (Copyright Jef Poskanzer) with an algorithm based on "A Fast Algorithm for General Raster Rotation" by Alan Paeth, Graphics Interface '86, pp. 77−81.

Use the `rotnewsz` function to find out about the dimension of the newly created image

```
($newcols,$newrows) = rotnewsz $oldn, $oldm, $angle;
```

rotate

Module: PDL::Slices

```
Signature: (x(n); int shift(); [oca]y(n))
```

Shift vector elements along with wrap. Flows data back&forth.

routine

Module: PDL::Func

```
my $name = $obj->routine;
```

Returns the name of the last routine called by a PDL::Func object.

This is mainly useful for decoding the value stored in the `err` attribute.

rpic

Module: PDL::IO::Pic

Read images in many formats with automatic format detection.

```
$im = rpic $file;
$im = PDL->rpic 'PDL.jpg' if PDL->rpiccan('JPEG');
```

Options

```
FORMAT  =>  'JPEG'   # explicitly read this format
XTRAFLAGS => '-nolut'  # additional flags for converter
```

Reads image files in most of the formats supported by netpbm. You can explicitly specify a supported format by additionally passing a hash containing the FORMAT key as in

```
$im = rpic ($file, {FORMAT => 'GIF'});
```

This is especially useful if the particular format isn't identified by a magic number and doesn't have the 'typical' extension or you want to avoid the check of the magic number if your data comes in from a pipe. The function returns a pdl of the appropriate type upon completion. Option parsing uses the the *PDL::Options* manpage module and therefore supports minimal options matching.

You can also read directly into an existing pdl that has to have the right *size*(!). This can come in handy when you want to read a sequence of images into a datacube, e.g.

```
$stack = zeroes(byte,3,500,300,4);
rpic $stack->slice(':,:,:,(0)'),"PDL.jpg";
```

reads an rgb image (that had better be of size (500,300)) into the first plane of a 3D RGB datacube (=4D pdl datacube). You can also do transpose/inversion upon read that way.

rpnm

Module: PDL::IO::Pnm

Read a pnm (portable bitmap/pixmap, pbm/ppm) file into a piddle.

Reads a file in pnm format (ascii or raw) into a pdl (magic numbers P1-P6). Based on the input format it returns pdls with arrays of size (width,height) if binary or grey value data (pbm and pgm) or (3,width,height) if rgb data (ppm). This also means for a palette image that the distinction between an image and its lookup table is lost which can be a problem in cases (but can hardly be avoided when using netpbm/pbmplus). Datatype is dependent on the maximum grey/color-component value (for raw and binary formats always PDL_B). rpnm tries to read chopped files by zero padding the missing data (well it currently doesn't, it barfs; I'll probably fix it when it becomes a problem for me ;). You can also read directly into an existing pdl that has to have the right *size*(!). This can come in handy when you want to read a sequence of images into a datacube.

For details about the formats see appropriate manpages that come with the netpbm/pbmplus packages.

```
    $im = rpnm $file;

  $stack = zeroes(byte,3,500,300,4);
  rpnm $stack->slice(':,:,:,(0)'),"PDL.ppm";
```

reads an rgb image (that had better be of size (500,300)) into the first plane of a 3D RGB datacube (=4D pdl datacube). You can also do inplace transpose/inversion that way.

rs Module: PDL::Slatec

```
  Signature: (a(n,n);[o]w(n);int matz();[o]z(n,n);[t]fvone(n);[t]fvtwo(n);int [o]ierr()
```

This subroutine calls the recommended sequence of subroutines from the eigensystem subroutine package (EISPACK) to find the eigenvalues and eigenvectors (if desired) of a REAL SYMMETRIC matrix.

rvals

Module: PDL::Basic

Fills a piddle with radial distance values from some centre.

```
 $r = rvals $piddle,{OPTIONS};
 $r = rvals [OPTIONAL TYPE],$nx,$ny,...{OPTIONS};

 Options:

 Centre => [$x,$y,$z...] # Specify centre
 Center => [$x,$y.$z...] # synonym.

 perldl> print rvals long,7,7,{Centre=>[2,2]}
 [
  [2 2 2 2 2 3 4]
  [2 1 1 1 2 3 4]
  [2 1 0 1 2 3 4]
  [2 1 1 1 2 3 4]
  [2 2 2 2 2 3 4]
  [3 3 3 3 3 4 5]
  [4 4 4 4 4 5 5]
 ]
```

For a more general metric, one can define, e.g.,

```
sub distance {
  my ($a,$centre,$f) = @_;
  my ($r) = $a->allaxisvals-$centre;
  $f->($r);
}
sub l1 { sumover(abs($_[0])); }
sub euclid { use PDL::Math 'pow'; pow(sumover(pow($_[0],2)),0.5); }
sub linfty { maximum(abs($_[0])); }
```

so now

```
distance($a, $centre, \&euclid);
```

will emulate rvals, while `\&l1` and `\&linfty` will generate other well-known norms.

saoimage

Module: PDL::Graphics::IIS

Starts the SAOimage external program

```
saoimage[(command line options)]
```

Starts up the SAOimage external program. Default FIFO devices are chosen so as to be compatible with other IIS module functions. If no suitable FIFOs are found it will offer to create them.

e.g.:

```
perldl> saoimage
perldl> saoimage( -geometry => '800x800' )
```

scheme

Module: PDL::Func

```
my $scheme = $obj->scheme;
```

Return the type of interpolation of a PDL::Func object.

Returns either `Linear` or `Hermite`.

sequence

Module: PDL::Basic

Create array filled with a sequence of values

```
$a = sequence($b); $a = sequence [OPTIONAL TYPE], @dims;
```

etc. see the `zeroes` entry in the *zeroes|PDL::Core* manpage.

```
perldl> p sequence(10)
[0 1 2 3 4 5 6 7 8 9]
perldl> p sequence(3,4)
[
 [ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
]
```

set Module: PDL::Func

```
my $nset = $obj->set( x = $newx, $y => $newy );
my $nset = $obj->set( { x = $newx, $y => $newy } );
```

Set attributes for a PDL::Func object.

The return value gives the number of the supplied attributes which were actually set.

set Module: PDL::Core

Set a single value inside a piddle

```
set $piddle, @position, $value
```

`@position` is a coordinate list, of size equal to the number of dimensions in the piddle. Occasionally useful, mainly provided for backwards compatibility as superseded by use of the `slice` entry in the *slice|PDL::Slices* manpage and assigment operator `.=`.

```
perldl> $x = sequence 3,4
perldl> set $x, 2,1,99
perldl> p $x
[
 [ 0  1  2]
 [ 3  4 99]
 [ 6  7  8]
 [ 9 10 11]
]
```

sethdr

Module: PDL::Core

Set header information of a piddle

```
$pdl=rfits('file.fits');
$h=$pdl->gethdr;
# add a FILENAME field to the header
$$h{FILENAME} = 'file.fits';
$pdl->sethdr( $h );
```

The `sethdr` function sets the header information for a piddle. Normally you would get the current header information with the `gethdr` entry in the *gethdr|* manpage, add/change/remove fields, then apply those changes with `sethdr`.

The `sethdr` function must be given a hash reference. For further information on the header, see the `gethdr` entry in the *gethdr|* manpage and the `hdrcpy` entry in the *hdrcpy|* manpage.

setstr

Module: PDL::Char

Function to set one string value in a character PDL. The input position is the position of the string, not a character in the string. The first dimension is assumed to be the length of the string.

The input string will be null-padded if the string is shorter than the first dimension of the PDL. It will be truncated if it is longer.

```
$char = PDL::Char->new( [['abc', 'def', 'ghi'], ['jkl', 'mno', 'pqr']] );
$char->setstr(0,1, 'foobar');
print $char; # 'string' bound to "", perl stringify function
# Prints:
# [
#  ['abc' 'def' 'ghi']
#  ['foo' 'mno' 'pqr']
# ]
$char->setstr(2,1, 'f');
print $char; # 'string' bound to "", perl stringify function
# Prints:
# [
#  ['abc' 'def' 'ghi']
#  ['foo' 'mno' 'f']      -> note that this 'f' is stored "f\0\0"
# ]
```

shiftleft

Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

leftshift `a$` by `$b`

```
$c = shiftleft $a, $b, 0;     # explicit call with trailing 0
$c = $a << $b;           # overloaded call
$a->inplace->shiftleft($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `<<` operator. Note that when calling

this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

shiftright
Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

leftshift `a$` by `$b`

```
$c = shiftright $a, $b, 0;     # explicit call with trailing 0
$c = $a >> $b;           # overloaded call
$a->inplace->shiftright($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `>>` operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

short
Module: PDL::Core

Convert to short datatype — see 'Datatype_conversions'

sig Module: PDL::Doc::Perldl

prints signature of PDL function

```
sig 'func'
```

The signature is the normal dimensionality of the functions arguments. Calling with different dimensions causes 'threading' — see `PDL::PP` for more details.

```
perldl> sig 'outer'
  Signature: outer(a(n); b(m); [o]c(n,m); )
```

simplex
Module: PDL::Opt::Simplex

Simplex optimization routine

```
($optimum,$ssize) = simplex($init,$initsize,$minsize,
                $maxiter,
                sub {evaluate_func_at($_[0])},
                sub {display_simplex($_[0])}
                );
```

See module PDL::Opt::Simplex for more information.

simq

Module: PDL::Math

```
Signature: ([phys]a(n,n); [phys]b(n); [o,phys]x(n); int [o,phys]ips(n); int flag)
```

Solution of simultaneous linear equations, a x = b.

$a is an n x n matrix (i.e., a vector of length n*n), stored row-wise: that is, a(i,j) = a[ij], where ij = i*n + j. While this is the transpose of the normal column-wise storage, this corresponds to normal PDL usage. The contents of matrix a may be altered (but may be required for subsequent calls with flag = −1).

$b, $x, $ips are vectors of length n.

Set flag=0 to solve. Set flag=-1 to do a new back substitution for different $b vector using the same a matrix previously reduced when flag=0 (the $ips vector generated in the previous solution is also required).

sin Module: PDL::Ops

```
Signature: (a(); [o]b())
```

the sin function

```
$b = sin $a;
$a->inplace->sin;  # modify $a inplace
```

It can be made to work inplace with the $a->inplace syntax. This function is used to overload the unary sin operator/function.

sinh

Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard hyperbolic function.

slice

Module: PDL::Slices

```
Signature: (P(); C(); char* str)
```

Returns a rectangular slice of the original piddle

```
$a->slice('1:3');  #  return the second to fourth elements of $a
$a->slice('3:1');  #  reverse the above
$a->slice('-2:1'); #  return last-but-one to second elements of $a
```

The argument string is a comma-separated list of what to do for each dimension. The current formats include the following, where a, b and c are integers and can take legal array index values (including -1 etc):

: takes the whole dimension intact.

" (nothing) is a synonym for ":" (This means that `$a->slice(':,3')` is equal to `$a->slice(',3')`).

a slices only this value out of the corresponding dimension.

(a) means the same as "a" by itself except that the resulting dimension of length one is deleted (so if `$a` has dims `(3,4,5)` then `$a->slice(':,(2),:')` has dimensions `(3,5)` whereas `$a->slice(':,2,:')` has dimensions `(3,1,5))`.

a:b slices the range a to b inclusive out of the dimension.

a:b:c
slices the range a to b, with step c (i.e. `3:7:2` gives the indices `(3,5,7)`). This may be confusing to Matlab users but several other packages already use this syntax.

'*'
inserts an extra dimension of width 1 and

'*a'
inserts an extra (dummy) dimension of width a.

An extension is planned for a later stage allowing C<$a-¿*slice*('(=1),(=1|5:8),3:*6*(=1),4:6')> to express a multidimensional diagonal of `$a`.

spaceship
Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

binary `<=>` operation

```
$c = $a->spaceship($b,0); # explicit function call
$c = $a <=> $b;    # overloaded use
$a->inplace->spaceship($b,0);     # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary `op<=>` function. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

splitdim
Module: PDL::Slices

```
Signature: (P(); C(); int nthdim; int nsp)
```

Splits a dimension in the parent piddle (opposite of the `clump` entry in the *clump*| manpage)

After

```
$b = $a->splitdim(2,3);
```

the expression

```
$b->at(6,4,x,y,3,6) == $a->at(6,4,x+3*y)
```

is always true (`x` has to be less than 3).

sqrt

Module: PDL::Ops

```
Signature: (a(); [o]b())
```

elementwise square root

```
$b = sqrt $a;
$a->inplace->sqrt;  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the unary `sqrt` operator/function.

squaretotri

Module: PDL::Math

```
Signature: (a(n,n); b(m))
```

Convert a symmetric square matrix to triangular vector storage

stats

Module: PDL::Primitive

Calculates useful statistics on a piddle

```
($mean,$rms,$median,$min,$max) = stats($piddle,[$weights]);
```

This utility calculates all the most useful quantities in one call.

Note: The RMS value that this function returns in the RMS deviation from the mean, also known as the population standard- deviation.

status

Module: PDL::Func

```
my $status = $obj->status;
```

Returns the status of a PDL::Func object.

This method provides a high-level indication of the success of the last method called (except for `get` which is ignored). Returns **1** if everything is okay, **0** if there has been a serious error, and **-1** if there was a problem which was not serious. In the latter case, C<$obj-¿*get*("err")> may provide more information, depending on the particular scheme in use.

string

Module: PDL::Char

Function to print a character PDL (created by 'char') in a pretty format.

```
$char = PDL::Char->new( [['abc', 'def', 'ghi'], ['jkl', 'mno', 'pqr']] );
print $char; # 'string' bound to "", perl stringify function
# Prints:
# [
#  ['abc' 'def' 'ghi']
#  ['jkl' 'mno' 'pqr']
# ]

# 'string' is overloaded to the "" operator, so:
# print $char;
# should have the same effect.
```

sum Module: PDL::Primitive

Return the sum of all elements in a piddle

```
$x = sum($data);
```

sumover

Module: PDL::Primitive

```
 Signature: (a(n); int+ [o]b())
```

Project via sum to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the sum along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = sumover($b);

$spectrum = sumover $image->xchg(0,1)
```

svd Module: PDL::Math

```
Signature: (a(n,m); [o]u(n,m); [o,phys]z(n); [o]v(n,n))
```

Singular value decomposition of array.

```
($u, $s, $v) = svd($a);
```

svdc
Module: PDL::Slatec

```
Signature: (x(n,p);[o]s(p);[o]e(p);[o]u(n,p);[o]v(p,p);[o]work(n);int job();int [o]in
```

singular value decomposition of a matrix

tan Module: PDL::Math

```
Signature: (a(); [o]b())
```

The usual trigonometric function.

tanh
Module: PDL::Math

```
Signature: (a(); [o]b())
```

The standard hyperbolic function.

thread
Module: PDL::Core

Use explicit threading over specified dimensions (see also the *PDL::Indexing* manpage)

```
$b = $a->thread($dim,[$dim1,...])
```

```
$a = zeroes 3,4,5;
$b = $a->thread(2,0);
```

Same as the `PDL::thread1` entry in the *PDL::thread1*| manpage, i.e. uses thread id 1.

thread1
Module: PDL::Core

Explicit threading over specified dims using thread id 1.

```
$xx = $x->thread1(3,1)
```

```
Wibble
```

Convenience function interfacing to the `threadI` entry in the *PDL::Slices::threadI|PDL::Slices* manpage.

thread2

Module: PDL::Core

Explicit threading over specified dims using thread id 2.

```
$xx = $x->thread2(3,1)

Wibble
```

Convenience function interfacing to the `threadI` entry in the *PDL::Slices::threadI|PDL::Slices* manpage.

thread3

Module: PDL::Core

Explicit threading over specified dims using thread id 3.

```
$xx = $x->thread3(3,1)

Wibble
```

Convenience function interfacing to the `threadI` entry in the *PDL::Slices::threadI|PDL::Slices* manpage.

thread_define

Module: PDL::Core

define functions that support threading at the perl level

```
thread_define 'tline(a(n);b(n))', over {
 line $_[0], $_[1]; # make line compliant with threading
};
```

`thread_define` provides some support for threading (see the *PDL::Indexing* manpage) at the perl level. It allows you to do things for which you normally would have resorted to PDL::PP (see the *PDL::PP* manpage); however, it is most useful to wrap existing perl functions so that the new routine supports PDL threading.

`thread_define` is used to define new *threading aware* functions. Its first argument is a symbolic repesentation of the new function to be defined. The string is composed of the name of the new function followed by its signature (see the *PDL::Indexing* manpage and the *PDL::PP* manpage) in parentheses. The second argument is a subroutine that will be called

with the slices of the actual runtime arguments as specified by its signature. Correct dimension sizes and minimal number of dimensions for all arguments will be checked (assuming the rules of PDL threading, see the *PDL::Indexing* manpage).

The actual work is done by the `signature` class which parses the signature string, does runtime dimension checks and the routine `threadover` that generates the loop over all appropriate slices of pdl arguments and creates pdls as needed.

Similar to `pp_def` and its `OtherPars` option it is possible to define the new function so that it accepts normal perl args as well as piddles. You do this by using the `NOtherPars` parameter in the signature. The number of `NOtherPars` specified will be passed unaltered into the subroutine given as the second argument of `thread_define`. Let's illustrate this with an example:

```
PDL::thread_define 'triangles(inda();indb();indc()), NOtherPars => 2',
 PDL::over {
   ${$_[3]} .= $_[4].join(',',map {$_->at} @_[0..2]).",-1,\n";
 };
```

This defines a function `triangles` that takes 3 piddles as input plus 2 arguments which are passed into the routine unaltered. This routine is used to collect lists of indices into a perl scalar that is passed by reference. Each line is preceded by a prefix passed as `$_[4]`. Here is typical usage:

```
$txt = '';
triangles(pdl(1,2,3),pdl(1),pdl(0),\$txt," "x10);
print $txt;
```

resulting in the following output

```
1,1,0,-1,
2,1,0,-1,
3,1,0,-1,
```

which is used in the *PDL::Graphics::TriD::VRML|PDL::Graphics::TriD::VRML* manpage to generate VRML output.

Currently, this is probably not much more than a POP (proof of principle) but is hoped to be useful enough for some real life work.

Check the *PDL::PP|PDL::PP* manpage for the format of the signature. Currently, the `[t]` qualifier and all type qualifiers are ignored.

threadids

Module: PDL::Core

Returns the piddle thread IDs as a perl list

```
@ids = threadids $piddle;
```

tlmfit

Module: PDL::Fit::LM

threaded version of Levenberg-Marquardt fitting routine mfit

```
tlmfit $x, $y, float(1)->dummy(0), $na, float(200), float(1e-4),
      $ym=null, $afit=null, \&expdec;
```

Signature:

```
tlmfit(x(n);y(n);sig(n);a(m);iter();eps();[o] ym(n);[o] ao(m);
          OtherPar => subref)
```

a threaded version of `lmfit` by using perl threading. Direct threading in `lmfit` seemed difficult since we have an if condition in the iteration. In principle that can be worked around by using `where` but .... Send a threaded `lmfit` version if you work it out!

Since we are using perl threading here speed is not really great but it is just convenient to have a threaded version for many applications (no explicit for-loops required, etc). Suffers from some of the current limitations of perl level threading.

topdl

Module: PDL::Core

alternate piddle constructor — ensures arg is a piddle

```
$a = topdl(SCALAR|ARRAY REFERENCE|ARRAY);
```

The difference between the `pdl` entry in the *pdl()*| manpage and `topdl()` is that the latter will just 'fall through' if the argument is already a piddle. It will return a reference and NOT a new copy.

This is particulary useful if you are writing a function which is doing some fiddling with internals and assumes a piddle argument (e.g. for method calls). Using `topdl()` will ensure nothing breaks if passed with '2'.

```
$a = topdl 43;         # $a is piddle with value '43'
$b = topdl $piddle;    # fall through
$a = topdl (1,2,3,4);  # Convert 1D array
```

transpose

Module: PDL::Basic

transpose rows and columns.

```
$b = transpose($a); $b = ~$a;
```

Also bound to the ~ unary operator.

```
perldl> $a = sequence(3,2)
perldl> p $a
[
 [0 1 2]
 [3 4 5]
]
perldl> p transpose( $a )
[
 [0 3]
 [1 4]
 [2 5]
]
```

twiddle3d

Module: PDL::Graphics::TriD

Wait for the user to rotate the image in 3D space.

Let the user rotate the image in 3D space, either for one step or until (s)he presses 'q', depending on the 'keeptwiddling3d' setting. If 'keeptwiddling3d' is not set the routine returns immediately and indicates that a 'q' event was received by returning 1. If the only events received were mouse events, returns 0.

type

Module: PDL::Core

return the type of a piddle as a blessed type object

A convenience function for use with the piddle constructors, e.g.

```
$b = PDL->zeroes($a->type,$a->dims,3);
```

unthread

Module: PDL::Slices

```
Signature: (P(); C(); int atind)
```

All threaded dimensions are made real again.

See [TBD Doc] for details and examples.

unwind

Module: PDL::Core

Return a piddle which is the same as the argument except that all threadids have been removed.

```
$y = $x->unwind;
```

usage

Module: PDL::Doc::Perldl

Prints usage information for a PDL function

```
 Usage: usage 'func'

   perldl> usage 'inner'

   inner           inner prodcuct over one dimension
                   (Module PDL::Primitive)

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

ushort

Module: PDL::Core

Convert to ushort datatype — see 'Datatype_conversions'

using

Module: PDL::Slices

Returns array of column numbers requested

```
 line $pdl->using(1,2);
```

Plot, as a line, column 1 of `$pdl` vs. column 2

```
 perldl> $pdl = rcols("file");
 perldl> line $pdl->using(1,2);
```

vars

Module: PDL::Dbg

Alias for `px`

vect

Module: PDL::Graphics::PGPLOT

Display 2 images as a vector field

```
 Usage: vect ( $a, $b, [$scale, $pos, $transform, $misval] )
```

Notes: `$transform` for image/cont etc. is used in the same way as the `TR()` array in the underlying PGPLOT FORTRAN routine but is, fortunately, zero-offset.

This routine will plot a vector field. `$a` is the horizontal component and `$b` the vertical component.

Options recognised:

```
    SCALE - Set the scale factor for vector lengths.
      POS - Set the position of vectors.
            <0 - vector head at coordinate
            >0 - vector base at coordinate
            =0 - vector centered on the coordinate
TRANSFORM - The pixel-to-world coordinate transform vector
  MISSING - Elements with this value are ignored.
```

The following standard options influence this command:

```
ARROW, ARROWSIZE, AXIS, BORDER, CHARSIZE, COLOUR, JUSTIFY,
LINESTYLE, LINEWIDTH
```

```
$a=rvals(11,11,{Centre=>[5,5]});
$b=rvals(11,11,{Centre=>[0,0]});
vect $a, $b, {COLOR=>YELLOW, ARROWSIZE=>0.5, LINESTYLE=>dashed};
```

vrmlcoordsvert

Module: PDL::Graphics::TriD::Rout

```
Signature: (vertices(n=3); char* space; char* fd)
```

info not available

vsearch

Module: PDL::Primitive

```
Signature: (i(); x(n); int [o]ip())
```

routine for searching 1D values i.e. step-function interpolation.

```
$inds = vsearch($vals, $xs);
```

Returns for each value of `$val` the index of the least larger member of `$xs` (which need to be in increasing order). If the value is larger than any member of `$xs`, the index to the last element of `$xs` is returned.

This function is useful e.g. when you have a list of probabilities for events and want to generate indices to events:

```
$a = pdl(.01,.86,.93,1); # Barnsley IFS probabilities cumulatively
$b = random 20;
$c = vsearch($b, $a); # Now, $c will have the appropriate distr.
```

It is possible to use the the `cumusumover` entry in the *cumusumover|* manpage function to obtain cumulative probabilities from absolute probabilities.

wcols

Module: PDL::IO::Misc

Write ASCII whitespaced cols into file from piddles efficiently.

If no columns are specified all are assumed. Will optionally only process lines matching a pattern. Can take file name or ∗HANDLE, and if no file/filehandle is given defaults to STDOUT.

Options:

HEADER — prints this string before the data. If the string is not terminated by a newline, one is added (default ”).

```
Usage: wcols $piddle1, $piddle2,..., *HANDLE|"outfile", [\%options];
```

e.g.,

```
wcols $x, $y+2, 'foo.dat';
wcols $x, $y+2, *STDERR;
wcols $x, $y+2, '|wc';
wcols $a,$b,$c; # Orthogonal version of 'print $a,$b,$c' :-)

wcols "%10.3f", $a,$b; # Formatted
wcols "%10.3f %10.5g", $a,$b; # Individual column formatting

wcols $a,$b, { HEADER => "#   a   b" };
```

Note: columns are separated by whitespace by default, use $PDL::IO::Misc::colsep to specify an alternate separator.

wfits

Module: PDL::IO::Misc

Simple piddle FITS writer

```
wfits $pdl, 'filename.fits', [$BITPIX];
$pdl->wfits('foo.fits',-32);
```

Suffix magic:

```
# Automatically compress through pipe to gzip
wfits $pdl, 'filename.fits.gz';
# Automatically compress through pipe to compress
wfits $pdl, 'filename.fits.Z';
```

$BITPIX is optional and coerces the output format.

where

Module: PDL::Primitive

Returns indices to non-zero values or those values from another piddle.

```
$i = $x->where($x+5 > 0); # $i contains elements of $x
                          # where mask ($x+5 > 0) is 1
```

Note: `$i` is always 1-D, even if `$x` is ¿1-D. The first argument (the values) and the second argument (the mask) currently have to have the same initial dimensions (or horrible things happen).

It is also possible to use the same mask for several piddles with the same call:

```
($i,$j,$k) = where($x,$y,$z, $x+5>0);
```

There is also the following syntax, retained only for compatibility with PDL versions ¡ 1.99. This use is deprecated, and will be removed in the future. Use the `which` entry in the *which|* manpage instead.

```
$i = where($x > 0);       # indices to $x, equivalent to 'which()'
```

Note: the mask has to be 1-D. See the documentation for the `which` entry in the *which|* manpage

which

Module: PDL::Primitive

```
Signature: (mask(n); int [o] inds(m))
```

Returns piddle of indices of non-zero values.

```
$i = which($mask);
```

returns a pdl with indices for all those elements that are nonzero in the mask. Note that mask really has to be 1-D (use `clump(-1)` if you need to work with ND-images)

If you want to return both the indices of non-zero values and the complement, use the function the section on *which_both* in the *which_both|* manpage.

```
perldl> $x = sequence(10); p $x
[0 1 2 3 4 5 6 7 8 9]
perldl> $indx = which($x>6); p $indx
[7 8 9]
```

which_both

Module: PDL::Primitive

```
Signature: (mask(n); int [o] inds(m); int [o]notinds(q))
```

Returns piddle of indices of non-zero values and their complement

```
($i, $c_i) = which_both($mask);
```

This works just as which, but the complement of `$i` will be in `$c_i`.

```
perldl> $x = sequence(10); p $x
[0 1 2 3 4 5 6 7 8 9]
perldl> ($small, $big) = which_both ($x >= 5); p "$small\n $big"
[5 6 7 8 9]
[0 1 2 3 4]
```

whichND

Module: PDL::Primitive

Returns the coordinates for non-zero values

```
@coords=whichND($mask);
```

returns an array of piddles containing the coordinates of the elements that are non-zero in `$mask`.

```
perldl> $a=sequence(10,10,3,4)
perldl> ($x, $y, $z, $w)=whichND($a == 203); p $x, $y, $z, $w
[3] [0] [2] [0]
perldl> print $a->at(list(cat($x,$y,$z,$w)))
203
```

whistogram

Module: PDL::Primitive

```
 Signature: (in(n); float+ wt(n);float+[o] hist(m); double step; double min; int msize
```

Calculates a histogram from weighted data for given stepsize and minimum.

```
$h = whistogram($data, $weights, $step, $min, $numbins);
$hist = zeroes $numbins;  # Put histogram in existing piddle.
whistogram($data, $weights, $hist, $step, $min, $numbins);
```

The histogram will contain `$numbins` bins starting from `$min`, each `$step` wide. The value in each bin is the sum of the values in `$weights` that correspond to values in `$data` that lie within the bin limits.

Data below the lower limit is put in the first bin, and data above the upper limit is put in the last bin.

The output is reset in a different threadloop so that you can take a histogram of `$a(10,12)` into `$b(15)` and get the result you want.

```
perldl> p whistogram(pdl(1,1,2), pdl(0.1,0.1,0.5), 1, 0, 4)
[0 0.2 0.5 0]
```

whistogram2d

Module: PDL::Primitive

```
 Signature: (ina(n); inb(n); float+ wt(n);float+[o] hist(ma,mb); double stepa; double
                     double stepb; double minb; int mbsize => mb;)
```

Calculates a 2d histogram from weighted data.

```
$h = whistogram2d($datax, $datay, $weights,
      $stepx, $minx, $nbinx, $stepy, $miny, $nbiny);
$hist = zeroes $nbinx, $nbiny;  # Put histogram in existing piddle.
whistogram2d($datax, $datay, $weights, $hist,
      $stepx, $minx, $nbinx, $stepy, $miny, $nbiny);
```

The histogram will contain `$nbinx` x `$nbiny` bins, with the lower limits of the first one at `($minx, $miny)`, and with bin size `($stepx, $stepy)`. The value in each bin is the sum of the values in `$weights` that correspond to values in `$datax` and `$datay` that lie within the bin limits.

Data below the lower limit is put in the first bin, and data above the upper limit is put in the last bin.

```
perldl> p whistogram2d(pdl(1,1,1,2,2),pdl(2,1,1,1,1),pdl(0.1,0.2,0.3,0.4,0.5),1,0,3,1,
[
 [  0   0   0]
 [  0 0.5 0.9]
 [  0 0.1   0]
]
```

wmpeg

Module: PDL::IO::Pic

Write an image sequence ((x,y,n) piddle) as an MPEG animation.

```
   $anim->wmpeg("GreatAnimation.mpg");
```

Writes a stack of rgb images as an mpeg movie. Expects a 4-D pdl of type byte as input. First dim has to be 3 since it is interpreted as interlaced RGB. Some of the input data restrictions will have to be relaxed in the future but routine serves as a proof of principle at the moment. It uses the program mpeg_encode from the Berkeley multimedia package (see also text at the top of this package). Mpeg parameters written by this routines haven't been tweaked in any way yet (in other words, lots of room for improvement). For an example how to use the routine see appropriate test that comes with this package. Currently, wmpeg doesn't

allow modification of the parameters written through its calling interface. This will change in the future as needed.

In the future it might be much nicer to implement a movie perl object that supplies methods for manipulating the image stack (insert, cut, append commands) and a final movie-¿*make()* call would invoke mpeg_encode on the picture stack (which will only be held on disk). This should get around the problem of having to hold a huge amount of data in memory to be passed into wmpeg (when you are, e.g. writing a large animation from PDL3D rendered fly-throughs). Having said that, the actual storage requirements might not be so big in the future any more if you could pass 'virtual' transform pdls into wmpeg that will only be actually calculated when accessed by the wpic routines, you know what I mean...

wndf

Module: PDL::IO::NDF

Writes a piddle to a NDF format file:

```
$pdl->wndf($file);
wndf($pdl,$file);
```

wndf can be used for writing PDLs to NDF files. The '.sdf' suffix is optional. All the extensions created by rndf are supported by wndf. This means that error, axis and quality arrays will be written if they exist. Extensions are also reconstructed by using their name (ie FIGARO.TEST would be expanded as a FIGARO extension and a TEST component). Hdr keywords Label, Title and Units are treated as special cases and are written to the label, title and units fields of the NDF.

Header information is written to corresponding NDF extensions. NDF extensions can also be created in the {NDF} hash by using a key containing '.', ie {NDF}{'IRAS.DATA'} would write the information to an IRAS.DATA extension in the NDF. rndf stores this as $$hdr{NDF}{IRAS}{DATA} and the two systems are interchangeable.

rndf stores type information in {NDF}{'_TYPES'} and below so that wndf can reconstruct the data type of non-PDL extensions. If no entry exists in _TYPES, wndf chooses between characters, integer and double on a best guess basis. Any perl arrays are written as CHAR array extensions (on the assumption that numeric arrays will exist as PDLs).

wpic

Module: PDL::IO::Pic

Write images in many formats with automatic format selection.

```
Usage: wpic($pdl,$filename[,{ options... }])
```

```
wpic $pdl, $file;
$im->wpic('web.gif',{LUT => $lut});
for (@images) {
  $_->wpic($name[0],{CONVERTER => 'ppmtogif'})
}
```

Write out an image file. Function will try to guess correct image format from the filename extension, e.g.

```
$pdl->wpic("image.gif")
```

will write a gif file. The data written out will be scaled to byte if input is of type float/double. Input data that is of a signed integer type and contains negative numbers will be rejected (assuming the user should have the desired conversion to an unsigned type already). A number of options can be specified (as a hash reference) to get more direct control of the image format that is being written. Valid options are (key =¿ example_value):

```
CONVERTER  => 'ppmtogif',   # explicitly specify pbm converter
FLAGS      => '-interlaced -transparent 0',  # flags for converter
IFORM      => 'PGM',        # explicitly specify intermediate format
XTRAFLAGS  => '-imagename iris', # additional flags to defaultflags
FORMAT     => 'PCX',        # explicitly specify output image format
COLOR      => 'bw',         # specify color conversion
LUT        => $lut,         # use color table information
```

Option parsing uses the the *PDL::Options* manpage module and therefore supports minimal options matching. A detailed explanation of supported options follows.

CONVERTER

directly specify the converter, you had better know what you are doing, e.g.

```
CONVERTER  => 'ppmtogif',
```

FLAGS

flags to use with the converter; ignored if !*defined*($$hints{CONVERTER}), e.g. with the gif format

```
FLAGS      => '-interlaced -transparent 0',
```

IFORM

intermediate PNM/PPM/PGM/PBM format to use; you can append the strings 'RAW' or 'ASCII' to enforce those modes, eg IFORMAT=¿'PGMRAW' or

```
IFORM    => 'PGM',
```

XTRAFLAGS
additional flags to use with an automatically chosen converter, this example works when you write SGI files (but will give an error otherwise)

```
XTRAFLAGS => '-imagename iris',
```

FORMAT
explicitly select the format you want to use. Required if wpic cannot figure out the desired format from the file name extension. Supported types are currently TIFF,GIF,SGI,PNM,JPEG,PS,RAST(Sun Raster),IFF,PCX, e.g.

```
FORMAT     => 'PCX',
```

COLOR
you want black and white (value **bw**), other possible value is **bwdither** which will write a dithered black&white image from the input data, data conversion will be done appropriately, e.g.

```
COLOR      => 'bw',
```

LUT This is a palette image and the value of this key should be a pdl containg an RGB lookup table (3,x), e.g.

```
LUT        => $lut,
```

Using the CONVERTER hint you can also build a pipe and perform several netpbm operations to get the special result you like. Using it this way the first converter/filecommand in the pipe should be specified with the CONVERTER hint and subsequent converters + flags in the FLAGS hint. This is because wpic tries to figure out the required format to be written by wpnm based on the first converter. Be careful when using the PBMBIN var as it will only be prepended to the converter. If more converters are in the FLAGS part specify the full path unless they are in your PATH anyway.

Example:

```
$im->wpic('test.ps',{CONVERTER  => 'pgmtopbm',
                     FLAGS => "-dither8 | pnmtops" })
```

Some of the options may appear silly at the moment and probably are. The situation will hopefully improve as people use the code and the need for different/modified options becomes clear. The general idea is to make the function perl compliant: easy things should be easy, complicated tasks possible.

wpnm
Module: PDL::IO::Pnm

Write a pnm (portable bitmap/pixmap, pbm/ppm) file into a file.

Writes data in a pdl into pnm format (ascii or raw) (magic numbers P1-P6). The $format is required (normally produced by **wpic**) and routine just checks if data is compatible with that format. All conversions should already have been done. If possible, usage of **wpic** is preferred. Currently RAW format is chosen if compliant with range of input data. Explicit control of ASCII/RAW is possible through the optional $raw argument. If RAW is set to zero it will enforce ASCII mode. Enforcing RAW is somewhat meaningless as the routine will always try to write RAW format if the data range allows (but maybe it should reduce to a RAW supported type when RAW == 'RAW'?). For details about the formats consult appropriate manpages that come with the netpbm/pbmplus packages.

```
$im = wpnm $pdl, $file, $format[, $raw];
```

writeflex

Module: PDL::IO::FlexRaw

Write a binary file with flexible format specification

```
$hdr = writeflex($file, $pdl1, $pdl2,...)
```

writefraw

Module: PDL::IO::FastRaw

Write a raw format binary file

```
writefraw($pdl,"fname");
```

wtstat

Module: PDL::Primitive

```
Signature: (a(n); wt(n); avg(); [o]b(); int deg)
```

Weighted statistical moment of given degree

This calculates a weighted statistic over the vector `a`. The formula is

```
b() = (sum_i wt_i * (a_i ** degree - avg)) / (sum_i wt_i)
```

xchg

Module: PDL::Slices

```
Signature: (P(); C(); int n1; int n2)
```

exchange two dimensions

Negative dimension indices count from the end.

The command

```
$b = $a->xchg(2,3);
```

creates `$b` to be like `$a` except that the dimensions 2 and 3 are exchanged with each other i.e.

```
$b->at(5,3,2,8) == $a->at(5,3,8,2)
```

ximtool

Module: PDL::Graphics::IIS

Starts the Ximtool external program

```
ximtool[(command line options)]
```

Starts up the Ximtool external program. Default FIFO devices are chosen so as to be compatible with other IIS module functions. If no suitable FIFOs are found it will offer to create them.

e.g.

```
perldl> ximtool
perldl> ximtool (-maxColors => 64)
```

xlinvals

Module: PDL::Basic

X axis values between endpoints (see the `xvals` entry in the *xvals*| manpage).

```
$a = zeroes(100,100);
$x = $a->xlinvals(0.5,1.5);
$y = $a->ylinvals(-2,-1);
# calculate Z for X between 0.5 and 1.5 and
# Y between -2 and -1.
$z = f($x,$y);
```

`xlinvals`, `ylinvals` and `zlinvals` return a piddle with the same shape as their first argument and linearly scaled values between the two other arguments along the given axis.

xor Module: PDL::Ops

```
Signature: (a(); b(); [o]c(); int swap)
```

binary *exclusive or* of two piddles

```
$c = xor $a, $b, 0;     # explicit call with trailing 0
$c = $a ^ $b;           # overloaded call
$a->inplace->xor($b,0);  # modify $a inplace
```

It can be made to work inplace with the `$a->inplace` syntax. This function is used to overload the binary ^ operator. Note that when calling this function explicitly you need to supply a third argument that should generally be zero (see first example). This restriction is expected to go away in future releases.

xray

Module: PDL::Graphics::Karma

Starts external Karma application xray

xray([OPTIONS])

```
perldl> kview (-num_col => 42)
perldl> xray
```

xvals

Module: PDL::Basic

Fills a piddle with X index values

```
$x = xvals($somearray);
$x = xvals([OPTIONAL TYPE],$nx,$ny,$nz...);
```

etc. see the `zeroes` entry in the *zeroes|PDL::Core* manpage.

```
perldl> print xvals zeroes(5,10)
[
 [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]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
 [0 1 2 3 4]
]
```

ylinvals

Module: PDL::Basic

Y axis values between endpoints (see the `yvals` entry in the *yvals|* manpage).

See the `xlinvals` entry in the *xlinvals|* manpage for more information.

yvals

Module: PDL::Basic

Fills a piddle with Y index values

```
$x = yvals($somearray); yvals(inplace($somearray));
$x = yvals([OPTIONAL TYPE],$nx,$ny,$nz...);
```

etc. see the `zeroes` entry in the *zeroes|PDL::Core* manpage.

```
perldl> print yvals zeroes(5,10)
[
 [0 0 0 0 0]
 [1 1 1 1 1]
 [2 2 2 2 2]
 [3 3 3 3 3]
 [4 4 4 4 4]
 [5 5 5 5 5]
 [6 6 6 6 6]
 [7 7 7 7 7]
 [8 8 8 8 8]
 [9 9 9 9 9]
]
```

zcheck

Module: PDL::Primitive

Return the check for zero of all elements in a piddle

```
$x = zcheck($data);
```

zcover

Module: PDL::Primitive

```
Signature: (a(n); int+ [o]b())
```

Project via != 0 to N-1 dimensions

This function reduces the dimensionality of a piddle by one by taking the != 0 along the 1st dimension.

By using the `xchg` entry in the *xchg|PDL::Slices* manpage etc. it is possible to use *any* dimension.

```
$a = zcover($b);
```

```
$spectrum = zcover $image->xchg(0,1)
```

zeroes

Module: PDL::Core

construct a zero filled piddle from dimension list or template piddle.

Various forms of usage,

(i) by specification or (ii) by template piddle:

```
# usage type (i):
$a = zeroes([type], $nx, $ny, $nz,...);
$a = PDL->zeroes([type], $nx, $ny, $nz,...);
$a = $pdl->zeroes([type], $nx, $ny, $nz,...);
# usage type (ii):
$a = zeroes $b;
$a = $b->zeroes
zeroes inplace $a;     # Equivalent to   $a .= 0;
$a->inplace->zeroes;   #  ""

perldl> $z = zeroes 4,3
perldl> p $z
[
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
]
perldl> $z = zeroes ushort, 3,2 # Create ushort array
[ushort() etc. with no arg returns a PDL::Types token]
```

zlinvals

Module: PDL::Basic

Z axis values between endpoints (see the `zvals` entry in the *zvals*| manpage).

See the `xlinvals` entry in the *xlinvals*| manpage for more information.

zvals

Module: PDL::Basic

Fills a piddle with Z index values

```
$x = zvals($somearray); zvals(inplace($somearray));
$x = zvals([OPTIONAL TYPE],$nx,$ny,$nz...);
```

etc. see the `zeroes` entry in the *zeroes*|*PDL::Core* manpage.

```
perldl> print zvals zeroes(3,4,2)
[
 [
  [0 0 0]
  [0 0 0]
  [0 0 0]
  [0 0 0]
 ]
 [
  [1 1 1]
```

```
    [1 1 1]
    [1 1 1]
    [1 1 1]
  ]
]
```