The PDL Book

March 2013

for PDL 2.006

PDL Book - Table of Contents

Suggested Reading Orders

We assume you know Perl, but that you are new to PDL.

First, try out the PDL command line by going through FirstSteps. PDL has several ways of displaying two-dimensional images and producing publication quality plots, and so we have PGPLOT and PLplot for producing two dimensional plots either in a computer window or as written file formats (PostScript, PNG, JPEG and more), and we also have the capability to produce three dimensional plots in TriD.

The power of PDL is in the ability to carry out threading (known as broadcasting in Python) over N-dimensional PDLs. When you code with threading you eliminate the multiple FOR loops that are the source of many slow-downs in code. Reading Threading and Functions will get you up to speed and in the right mind-set.

If you require the speed of C routines in your PDL code, there is also the powerful PDL:PP capability of PDL - you can write C code INLINE in your PDL code, and it will be compiled and run when you call your Perl/PDL scripts!

PDL is primarily used by scientists who want access to Scientific libraries and data types, so we have Complex numbers handled by PDL and the capabilities of PDL::Transform, the Slatec libraries accessible in PDL::Slatec, and any other libraries that you can access through Perl.

First Steps with PDL

"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 basics 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 section 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 sections will focus on looking at the features introduced here, in more depth.

Alright, let's do something

We'll assume PDL is correctly installed and set up on your computer system (see *http://pdl.perl.org/* 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 editing (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 v1.357
 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, NiceSlice, MultiLines  enabled
Reading PDL/default.perldlrc...
Found docs database /usr/lib/perl5/.../PDL/pdldoc.db
Type 'help' for online help
Type 'demo' for online demos
Loaded PDL v2.006 (supports bad values)
pdl>
```

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 `pdl>` prompt at which we can type commands. This kind of interactive program is called a 'shell'. There is also `pdl2` which is a newer version of the PDL shell with additional features. It is still under development but completely usable.

Let's create something, and display it:

```
pdl> use PDL::Graphics::Simple
pdl> imag (sin(rvals(200,200)+1))
```

The result should look like the image below - 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. (Try '`imag(rvals(200,200))`' and you will see better what we mean!) `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 generate the same display, even given the existence of some convenient graphics library.

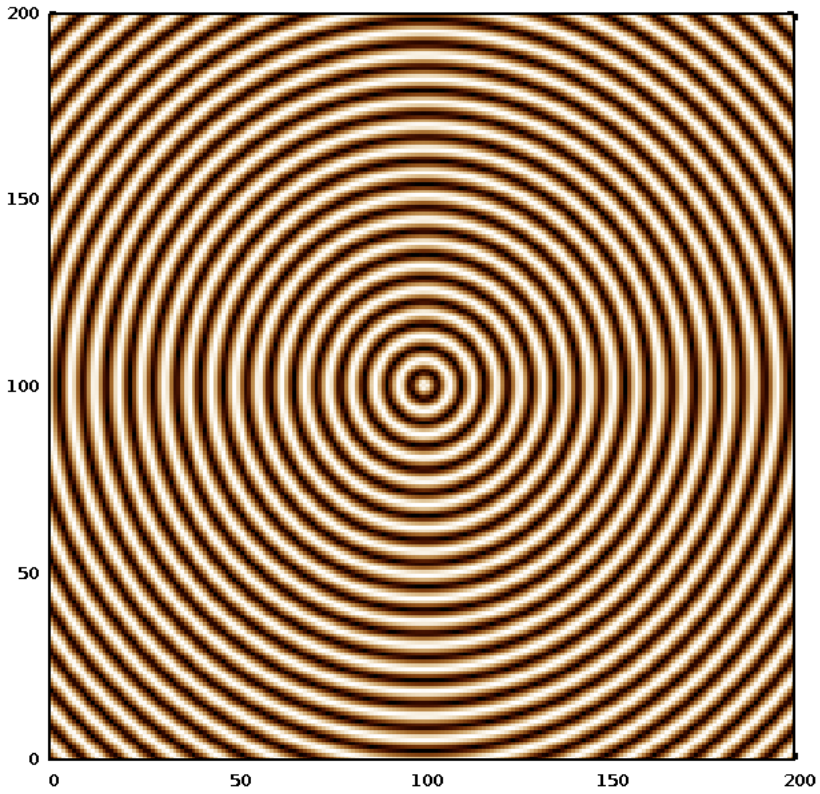


Figure of a two dimensional C<sin> function.

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

That's it. This is a complete perl/PDL program. One could run it by typing `perl filename`. (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 - then the `perlrun` manual page.)

Two comments:

1. The statements are all terminated by the '`;`' character. Perl is like C in this regard. When entering code at the `pdl` 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 `pdl` 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 interested in knowing there are other ways of starting PDL which allows one to select which bits of it you want).

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 can download any needed input files from *http://sourceforge.net/projects/pdl/files/PDL/PDL%20Book%20Example%20Data%20Set/* and we'll assume you are running any of these examples in the same directory as you have downloaded the input data files.

We'll be playing with an image of the famous spiral galaxy discovered by Charles Messier, known to astronomers 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 gray than GIF or JPEG, but PDL can read these formats too).

```
pdl> $a = rfits("m51_raw.fits");   # m51_raw.fits is in current directory
Reading IMAGE data...
BITPIX =  -32  size = 262144 pixels
Reading  1048576  bytes
BSCALE =  &&  BZERO =
```

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 its 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.

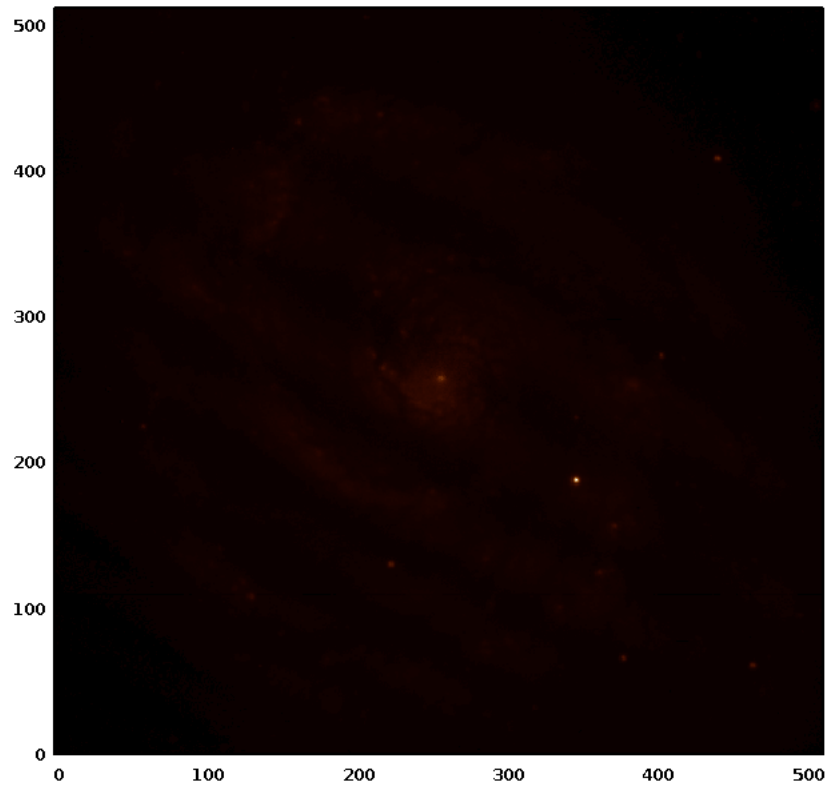
Before we start seriously playing around with M51 it is worth noting that we can also say:

```
pdl> $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

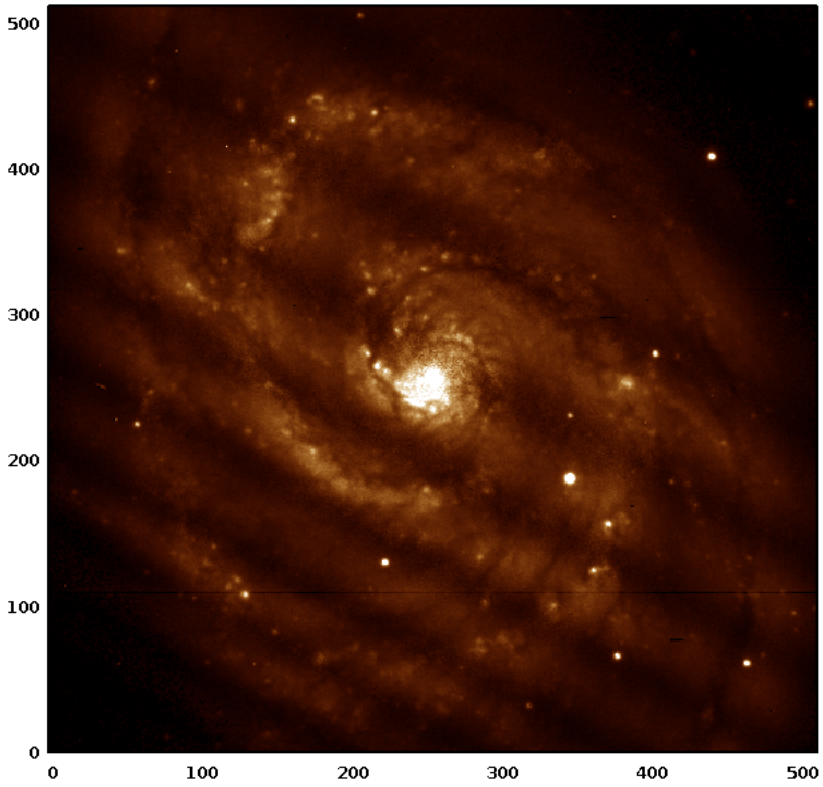
Let's look at M51:

```
pdl> imag $a;
```

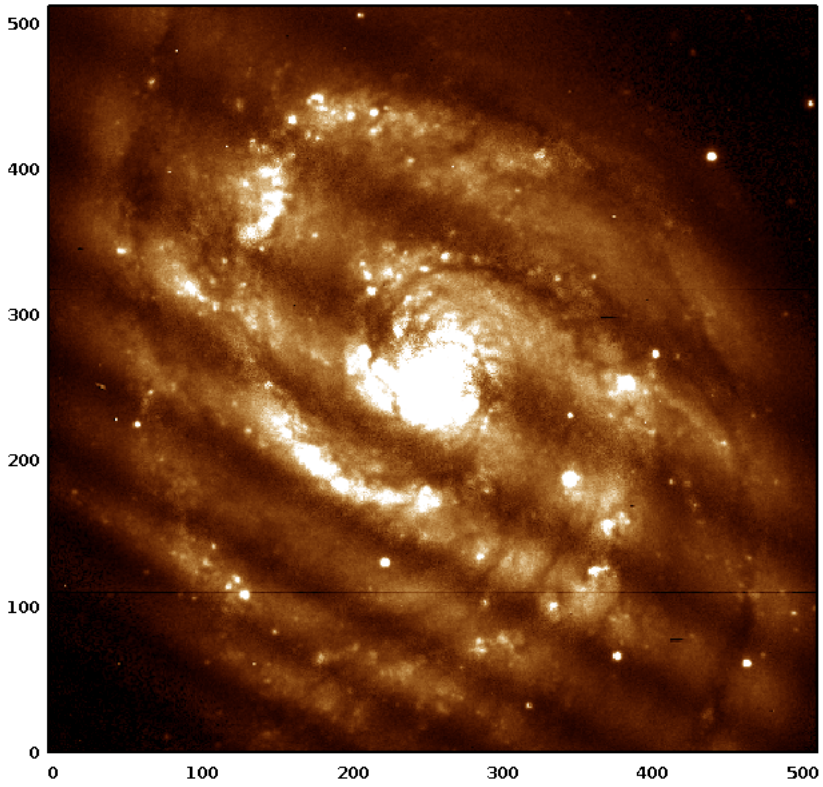



```
Figure of the raw image C<m51_raw.fits> shown with
progressively greater contrast using the C<imag> command.
```

A couple of bright spots can be seen, 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!):




```
pdl> imag $a,0,1000; # More contrast
```



```
pdl> 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 `pdl` prompt we can find out all about the function.

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 sensitivity of the detector used to record the image, a common artifact in digital imaging. We can correct for this using an image of a uniformly illuminated screen, what is commonly known as a 'flatfield'.

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

This is shown in the next figure. 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:

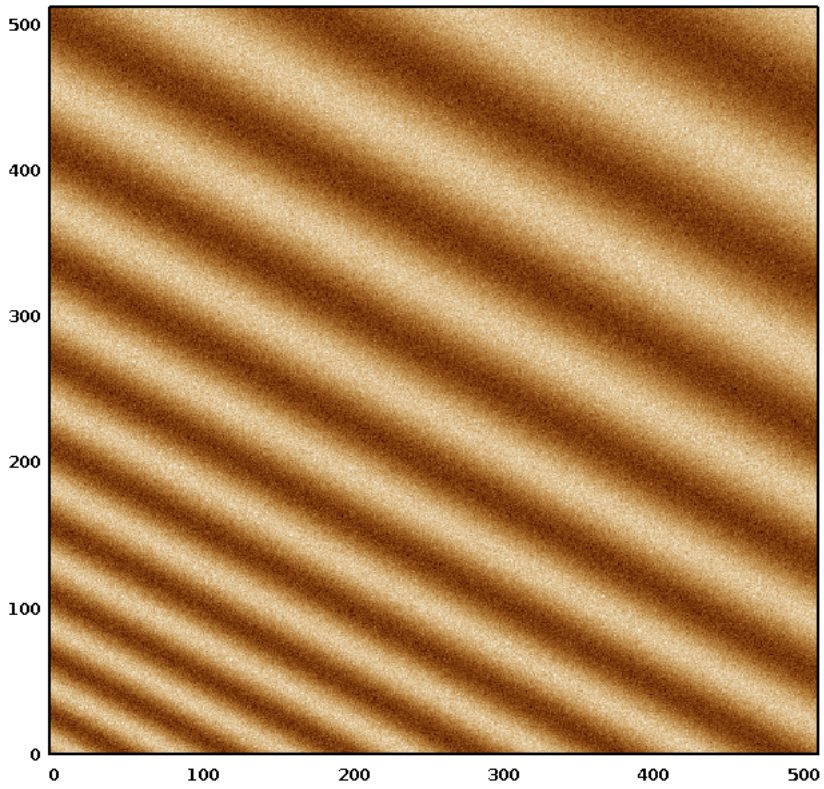


Figure: The 'flatfield' image showing the detector sensitivity of the raw data.

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

Well that's a lot better. 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 *vectorized* operations. In PDL each of these is implemented by heavily optimized 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:

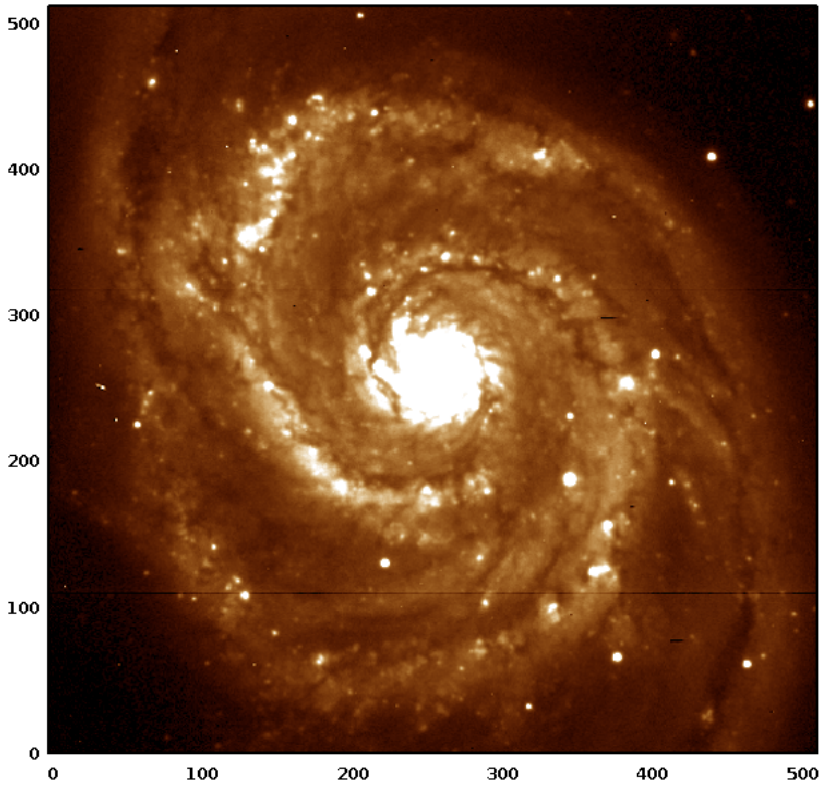


Figure: The M51 image corrected for the flatfield.

```
pdl> $funny = log(($gal/300)**2 - $gal/100  + 4);
pdl> 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.

Measuring the brightness of M51

How might we extract some useful scientific information out of this image? A simple quantity 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:

```
pdl> 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 1E36 Watts, so we can work out what one data value corresponds to in physical units:

```
pdl> p 10**36/sum($gal)
```

```
5.58159992096455e+28
```

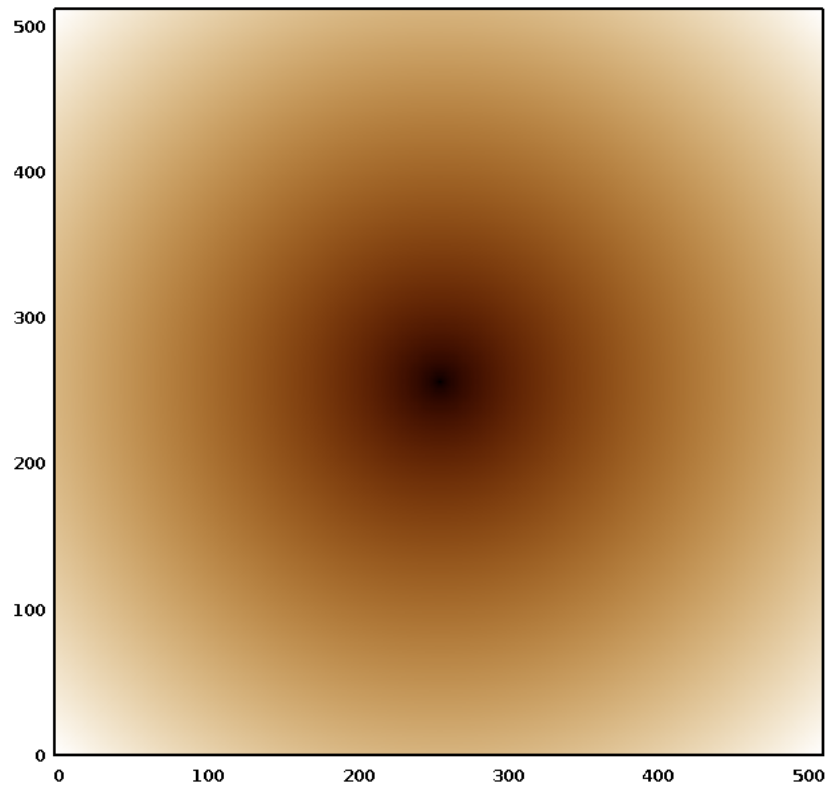
This is also about 200 solar luminosities, (Note we have switched to using p as a shorthand for `print` - which only works in the `pdl` and `pdl2` shells) which gives 4 billion solar luminosities for the whole galaxy.

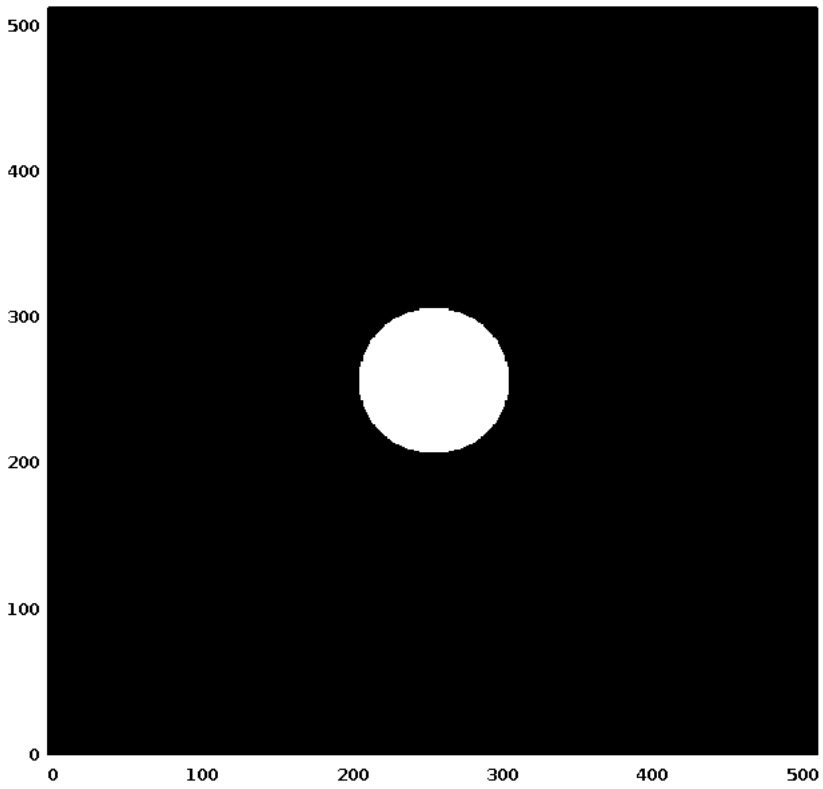
OK we do not need PDL for this simple arithmetic, let's get back to computations that involve the whole image. How can we get the sum of a piece of an image, e.g. near the centre? Well in PDL there is more than one way to do it (Perl aficionados call this phenomenon TIMTOWTDI). In this case, because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

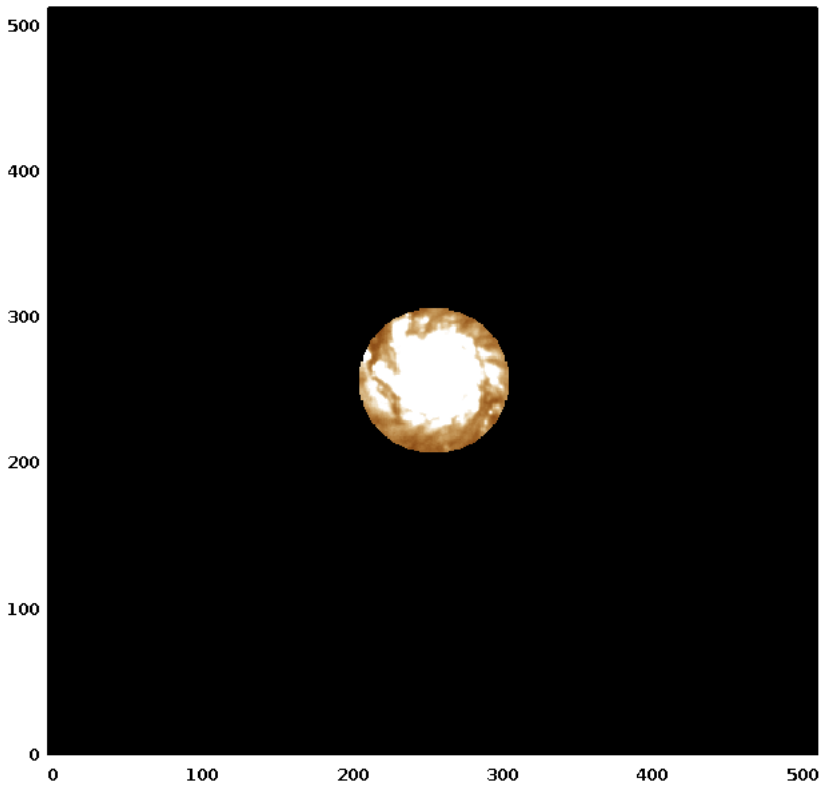
```
pdl> $r = rvals $gal;
pdl> imag $r;
...
```

Remember `rvals`? It replaces all the pixels in an image with its distance from the centre. We can turn this into a *mask* with a simple operation like:

```
pdl> $mask = $r<50;
pdl> imag $mask;
...
```




500
400
300
200
100
0
0
100
200
300
400
500


500
400
300
200
100
0
0
100
200
300
400
500

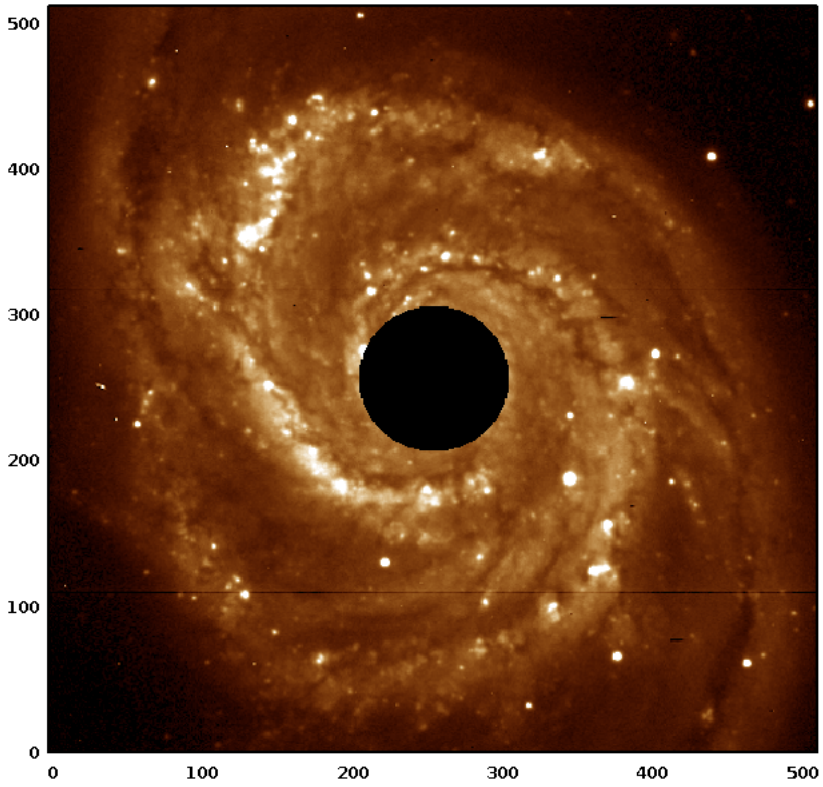


Figure: Using `rvals` to generate a mask image to isolate the galaxy bulge and disk. Top row: radial gradient image `$r`, and radial gradient masked with less than operator `$r < 50`. Bottom row: Bulge and disk of the galaxy.

The Perl *less than operator* is applied to all pixels in the image. You can see the result 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 (lower row) and it is then very easy to find the brightness of both pieces of the M51 galaxy:

```
pdl> $bulge = $mask * $gal
pdl> imag $bulge,0,300
...
pdl> print sum $bulge;
3011125

pdl> $disk = $gal * (1-$mask)
pdl> imag $disk,0,300
...
pdl> print sum $disk
14904884
```

You can see that the disk is about 5 times brighter than the bulge in total, despite its 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:

```
pdl> print sum($bulge)/sum($mask);
pdl> 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 the above figure, which tells astronomers its stellar density is much higher.

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

```
  pdl> print ( avg($gal->where(rvals($gal)<50)) /
avg($gal->where(rvals($gal)>=50)) )
  6.56590509414673
```

Twinkle, twinkle, little star

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

```
  pdl> $section = $gal(337:357,178:198);
  pdl> imag $section; # 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 337 to 357 along the first dimension, and 178 through 198 along the second. Remember pdl data dimension indexes start from zero. We'll talk some more about *slicing and dicing* later on. 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. (Answer: about 2%) 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 centered on the star, you could think if you want about how you might determine the centroid automatically using the `xvals` and `yvals` functions). Well it is simple enough to get the distance from the centre:

```
  pdl> $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:

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


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

You should see a nice graph with points like those in the figure below 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 the plot it looks like this is about 2-3 pixels - pretty compact!

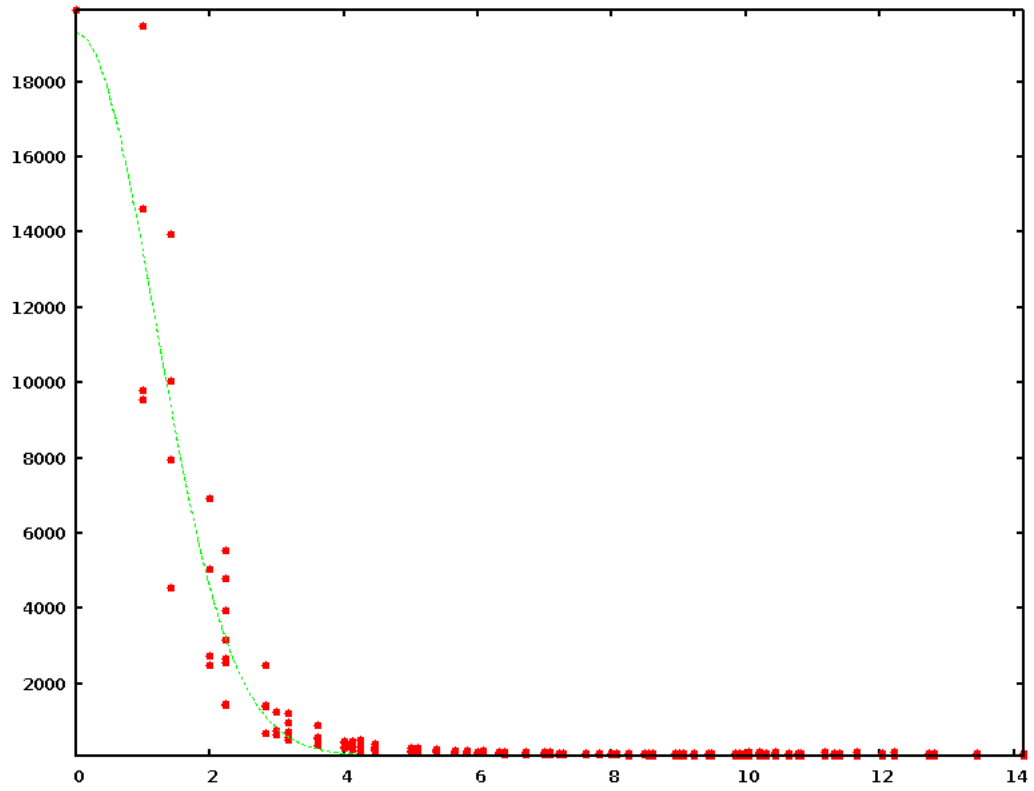


Figure: Radial light profile of the bright star with fitted curve.

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 surprisingly enough a special function for this very purpose. (One could use more general fitting packages like `PDL::Fit::LM` or `PDL::Opt::Simplex` but that would require more care).

```
pdl> 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`:

```
pdl> 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 function (integrated from minus
     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 `pdl` or `pdl2` shells.

Let's fit a Gaussian:

```
pdl> use PDL::Fit::Gaussian;
pdl> ($peak, $fwhm, $background) = fitgauss1dr($rr, $sec);
pdl> p $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.

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


# Generate Gaussian with given FWHM


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

Note the use of a new function, `sequence(N)`, which generates a new piddle with N values ranging 0..(N-1). We are simply using this to generate the horizontal axis values for the plot. Now let's overlay it on the previous plot.

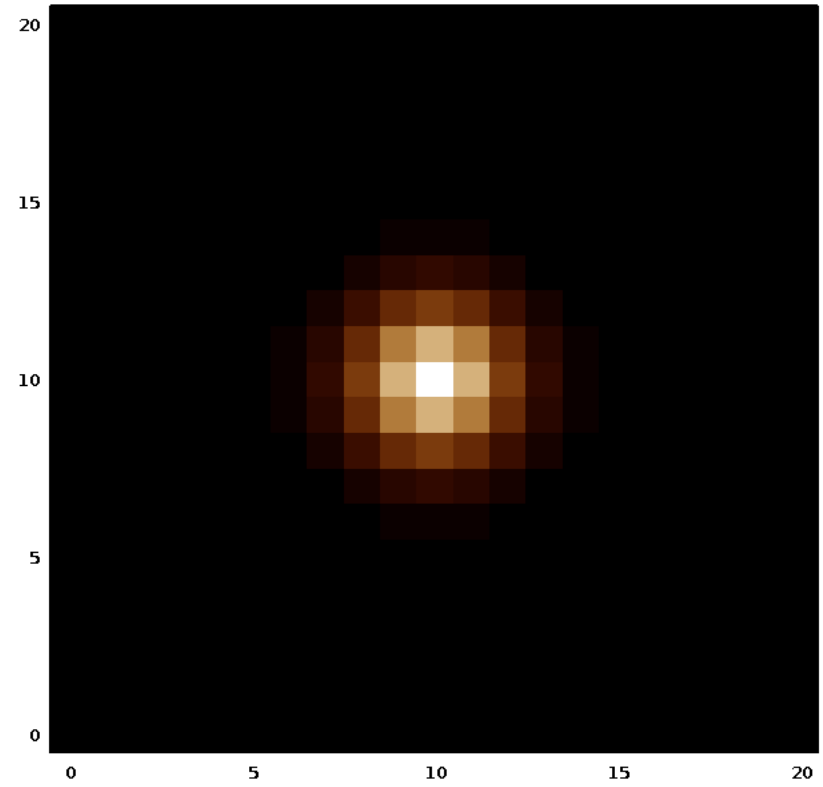
```
pdl> hold; # This command stops new plots starting new pages
pdl> line $rrr, $fit, {Colour=>2} ; # Line plot
```

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 *PDL::Book::PGPLOT*. The result should look a lot like the figure above. 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:

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

But the figure below is a boring. So far we have been using simple 2D graphics from the `PDL::Graphics::Simple` 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 (see the right hand panel);



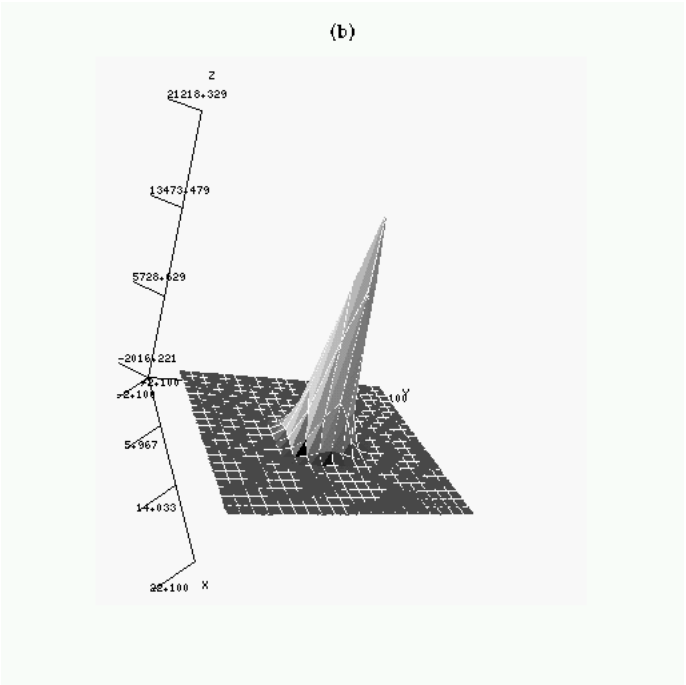


Figure: Two different views of the 2D simulated Point Spread Function.

```
pdl> use PDL::Graphics::TriD; # Load the 3D graphics module
pdl> 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 *PDL::Book::TriD*.

```
To continue: Select the TriD window and type q
```

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

```
pdl> $newsection = $gal(50:70,70:90);
pdl> $newsection +=  $fit2d;
pdl> 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:

```
pdl> $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::Simple;
use PDL::NiceSlice;  # must use in each program file

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);
   $y1 = int(rand(470)+10);
   addstar($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 the figure below and takes virtually no time to compute.

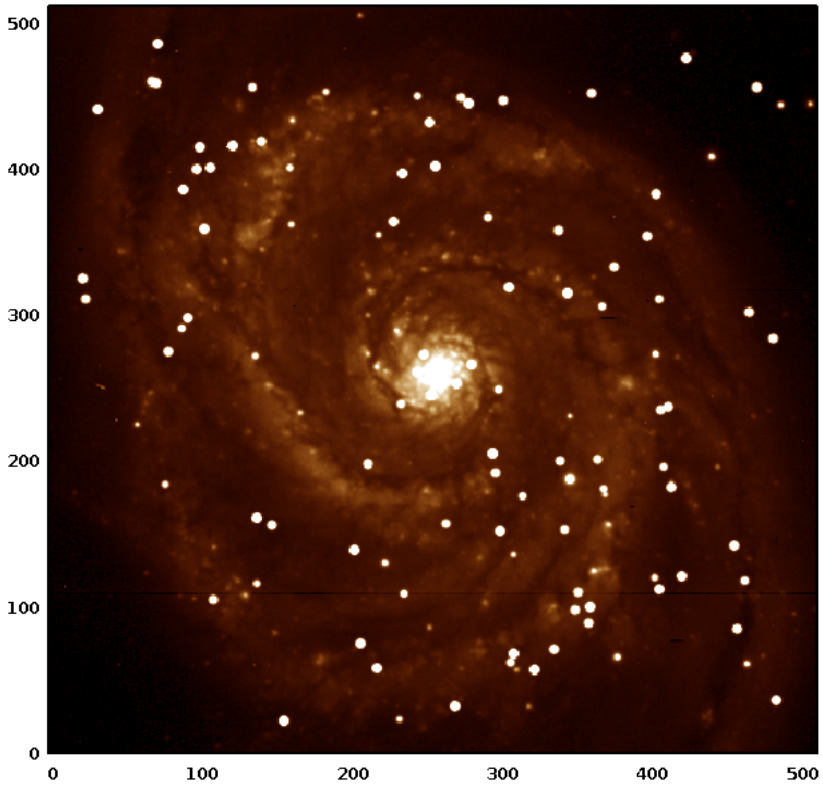


Figure: M51 covered in fake stars.

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:

```
pdl> use PDL::FFT; # Load Fast Fourier Transform package
pdl> $gal = rfits "fixed_gal.fits";
```

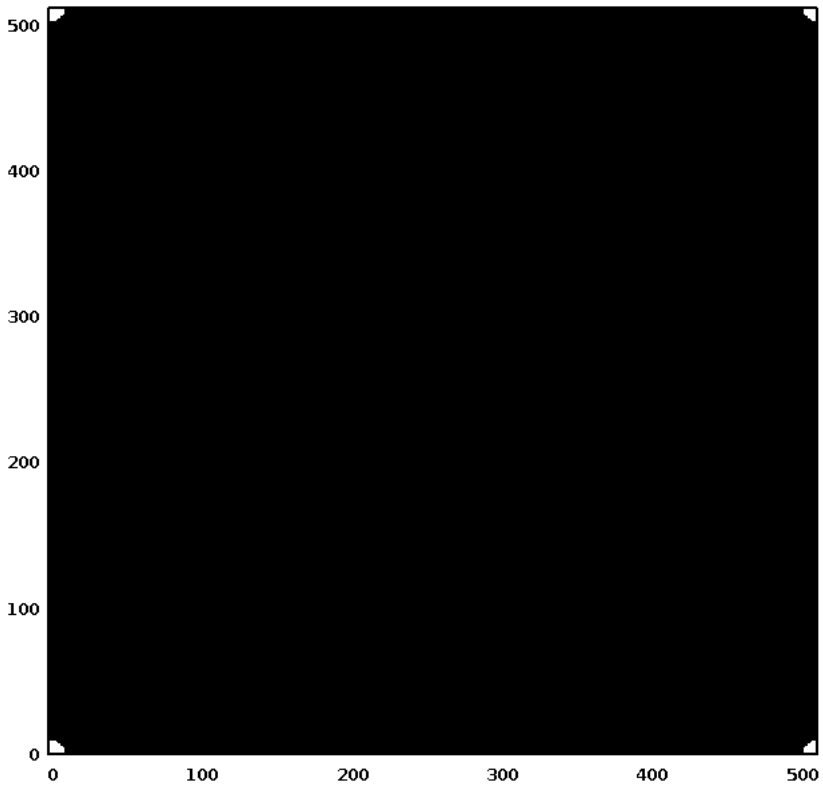
Now `$gal` contains real values, to do the Fourier transform it has to have complex values. We create a variable `$imag` to hold the imaginary component and set to zero.(For reasons of efficiency complex numbers are represented in PDL by separate real and imaginary arrays - more about this in Chapter 2.)

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

```
pdl> $tmp = rvals($gal)<10;        # Radially-symmetric filter function
pdl> use PDL::ImageND;             # provides kernctr()
pdl> $filter = kernctr $tmp, $tmp; # Shift origin to 0,0
pdl> 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 lets through low-frequency wavelengths. So we multiply by the filter and FFT back to see the result (`cmul` is complex multiplication):

```
pdl> ($gal2, $imag2) = cmul $gal, $imag, $filter, 0;
pdl> ifftnd $gal2, $imag2;
pdl> imag $gal2,0,300;
```

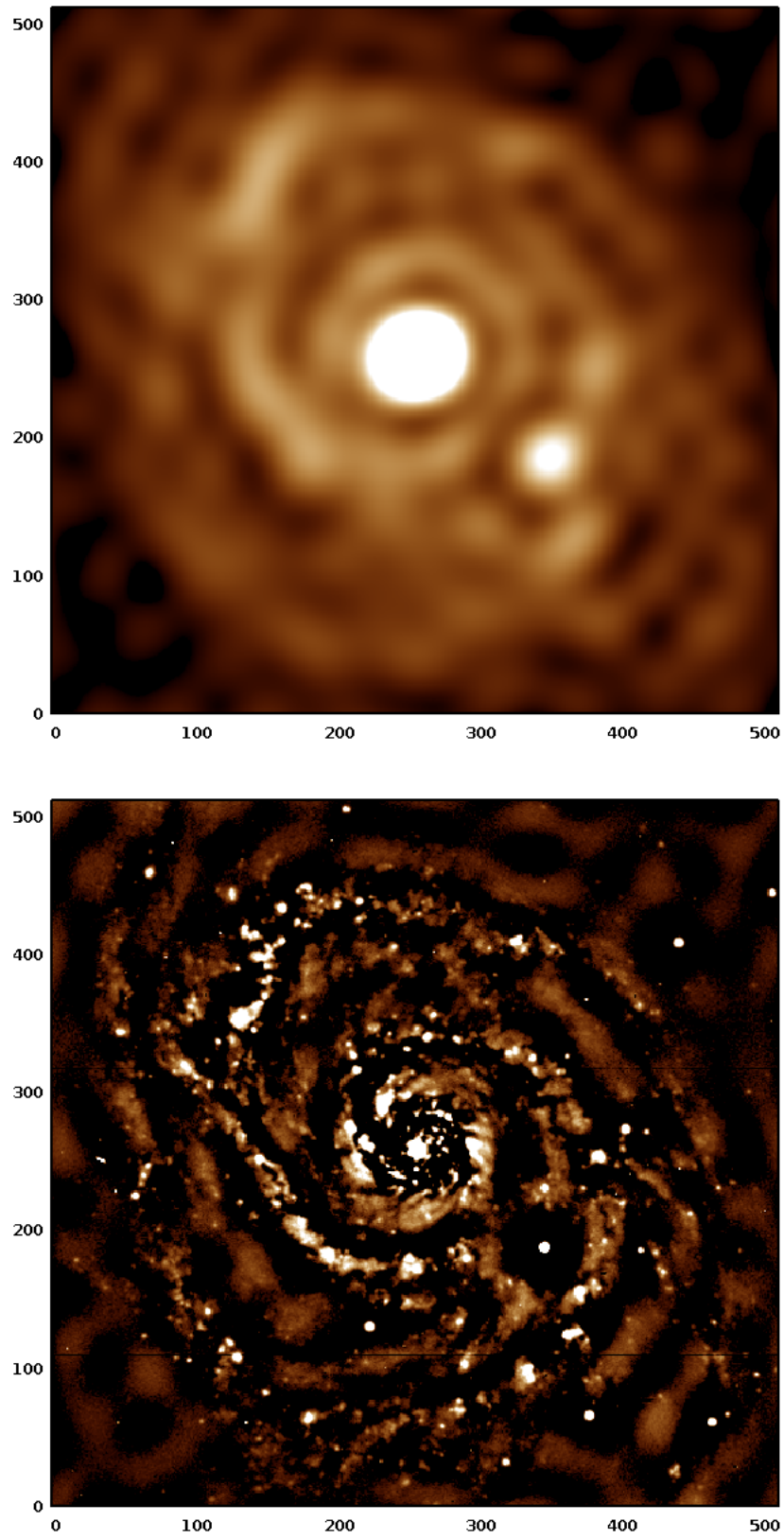


Figure: Fourier filtered smoothed image and contrast enhanced image with the smoothed image subtracted.

Well that looks quite a bit different! 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 to form the right hand image.

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

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 fits neatly 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.

What is a Piddle?

PDL uses Perl `objects' to hold PDL data, affectionately called a piddle. 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.

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.

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

By Perl `arrays' we of course 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:

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.

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'.

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.

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.

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.

Perl lists do not support the range of data types 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'.

Constructing PDLs

PDL variables are a new class of object within Perl. There are three main ways to construct them: via the pdl constructor; via one of the special index PDL constructors; or by reading in some external data. In addition, there are hooks for stuffing your own raw data into a PDL variable. The more basic constructors are here.

The basic constructor, pdl()

The most basic way to make a PDL is with the function `pdl()`. You can feed pdl just about anything that makes sense: a perl scalar, a perl list, a nested perl list, another PDL, or even a perl list of PDLs. It will return an appropriately-dimensioned PDL containing those values. Here are some examples:

```
$a = pdl( 5 );                 # double-precision scalar
$a = pdl( short,5 );           # short-integer scalar
$a = pdl( 1,2,3 );             # 3-PDL (one dim)
$a = pdl( [1,2,3] );           # 3-PDL, another way (just one dim)
$a = pdl( [[1,2,3]] );         # 3x1-PDL (two dims)
$a = pdl( [[1,2,3],[4,5,6]] ); # 3x2-PDL (two dims)
$a = pdl "[[1,2,3],[4,5,6]]";  # Even strings from print output!
```

In the last couple of examples, notice that the innermost nested lists form the 0th dimension of the PDL.

If you aren't sure whether a particular variable contains a PDL or not (and sometimes you care: there's a slight difference between a scalar PDL and a perl scalar!) you can always safely wrap a `pdl` call around it to be sure.

Array allocation: zeroes() and ones()

The two operations zeroes and ones generate PDLs full of the value 0 and of the value 1, respectively. (well, what did you expect?) They're useful for allocating data in a hurry. If you feed in a list of perl scalars, they are used as a list of dimensions for the new PDL that gets returned. If you feed in a PDL, either one will simply match the size of the PDL. Examples:

```
$a = zeroes(3,3);          # $a becomes a 3x3 array filled with 0
$a = zeroes(byte,3,3);     # ditto, only bytes instead of doubles
$b = ones($a);             # $b becomes a 3x3 array filled with 1
$p = pdl(1,2,3);           # A PDL containing [1 2 3]
$c = zeroes($p);           # A 3-PDL containing [0 0 0]
```

```
$d = zeroes($p->list);      # A 1x2x3-PDL ($p->list is a Perl list)
```

Index PDLs: xvals, yvals, rvals, sequence, ndcoords

It is surprisingly useful to be able to generate "index PDLs": arrays whose elements merely enumerate their coordinates. PDL supplies a passel of index PDL constructors.

The basics are `xvals`, `yvals` and `zvals`, which work like `zeroes` and `ones`, but construct an index PDL that works along the 0, 1, or 2 axis, respectively. For example:

```
pdl> print xvals(3,3)
[
  [0 1 2]
  [0 1 2]
  [0 1 2]
]
pdl> print yvals(3,3)
[
  [0 0 0]
  [1 1 1]
  [2 2 2]
]
```

If you want more generality or higher dimensionality, `axisvals` works the same way but lets you specify the index dimension by number.

Sometimes you want a PDL that contains radii from a given point. You could always apply the Pythagorean theorem explicitly:

```
$x=xvals(10,10)-5;
$y=yvals(10,10)-5;
$a=sqrt( $x*$x + $y*$y );
```

but it's much easier to use `rvals`, which does that stuff for you:

```
pdl> $a = rvals(3,3); print $a;
[
  [ 1.4142136        1  1.4142136]
  [         1        0          1]
  [ 1.4142136        1  1.4142136]
]
```

As with the others, `rvals` works in any number of dimensions, and can either take a dimension list or another PDL to match. There are a number of adjustments that you can make to `rvals`; see the online documentation for details.

Finally, sometimes you want to create a full vector index PDL; for example, to enumerate all the coordinates in a 100x100 image you would want a 2x100x100-PDL. You can assemble one from `xvals`, `yvals`, or just use `ndcoords`. Here's how to do it either way:

```
$a = pdl( xvals(100,100), yvals(100,100) )->mv(0,1); # slow way
$a = ndcoords(100,100);                              # fast way
```

`ndcoords` works like all the other index constructors, except that it adds an additional dim to the beginning of its return value, to handle the fact that each index is a vector that points into an N-dimensional array. `ndcoords` and `range` together can be used to chop up an image into

manageable tiles; see Section [sub:Range] , below.

Specialty constructors

PDL contains two important internal constructors, `PDL::new_from_specification` and `null`, that are useful for importing data en masse or for other special applications. If you're just starting out, you probably don't really need to know this stuff just yet - you'll probably find the various data import techniques in [sec:Getting-values-into] more useful. So skip ahead if you like.

`null` takes no arguments and returns a null PDL. A null PDL has no values, but (unlike the empty PDL) can be assigned to. Null PDLs are placeholders that automatically resize themselves to fit any dimensional context. They're mainly intended for internal use, but you might find them helpful in odd contexts (for example, you can pass a null PDL into a function as a write-back return value).

`PDL::new_from_specification` is the engine that `zeroes`, `rvals`, and such use for initial construction. It takes the same sort of arguments as `zeroes` (an optional type and a PDL template or a size list), but doesn't bother with any initialization of the newly allocated RAM. This is especially useful if you're just going to stuff your own values into the new PDL anyway.

Getting values into and out of PDLs

Unless you can get data in and out of your PDLs they won't do you much good. Most large blocks of data are handled by direct file I/O (Chapter [cha:File-I/O]), but you will also want to get normal Perl values into and out of your PDLs. Here are the basic ways to get data into your PDLs (from perl, other PDLs, or random chunks of memory), and back out again (into perl, into random chunks of memory, or into ASCII). For displaying your data you will want to look at Chapter [cha:Graphics].

Construction: slurping Perl arrays

The simplest way to turn a bunch of Perl data into a PDL is by calling `pdl()`, the PDL constructor. The constructor pokes and prods the array structure of its argument(s), and creates a PDL that contains all the values in whatever nested array you've come up with. For example,

```
$pdl_all = pdl(@pdl_source);
$pdl_3x3 = pdl([00,01,02],[10,11,12],[20,21,22]);
```

That is certainly the most convenient (and probably the fastest) way to stuff a bunch of values from Perl variables into a PDL.

Assignment with .=

PDL distinguishes between two kinds of assignment: *global assignment* (the usual = operator) and *threaded (computed) assignment* (the `.=` operator).

PDLs are best thought of as something like perl refs or C pointers: the variable points to the location in memory where the data reside. That makes array indexing and slicing straightforward, since you can hold a slice of a larger array in a related variable, without expensive memory copies. The global = operator is used to set the value of the pointer. The threaded `.=` operator is used to set the value of the data that are contained in the PDL. The two operators work quite differently. For example:

```
$a = xvals(3);      # 1D-PDL: values are (0,1,2)
$b = zeroes(3,4);   # 3x4 array of zeroes
$c = zeroes(3,4);   # 3x4 array of zeroes
$b = $a;            # $b becomes a clone of $a
$c .= $a;           # $c becomes 4 copies of $a
```

puts two quite different values into `$b` and `$c`. At the end of the code, `$b` and `$a` are linked (they point to the same area of memory), so assigning to the elements of `$b` changes `$a` too. But `$c` remains a separate variable, whose elements happen to have received values from the corresponding elements of `$a`.

But that's not all! $b and $c end up with completely different shapes. Because $c started out as a 3x4-PDL, the threading engine duplicates $a (which is a 3-PDL) for each row of $c. The .= operator is called threaded assignment, because it causes its right-hand argument to be expanded and vectorized exactly as any other operand would. Threading is explained in detail in Section [sec:Dimensionality-and-Threading] .

Importing data directly from memory: get_dataref

PDL lets you access the memory of a PDL variable directly, using a perl string variable. You normally won't have to use this mechanism, but I include it here for completeness - if you are just learning PDL, you can probably skip this subsection.

The string variable mechanism gives you access to the low-level representation of the data (which is the same as your C compiler would use). The string access routines are `get_dataref` and `upd_data`. `get_dataref` takes a PDL argument and returns a perl scalar ref that points to the PDL's data as a perl string. If you change the string, Perl might move it in memory, so you must then update the pointers in the PDL variable to match. That is what `upd_data` is for.

Here's a brief example of how to import a large hunk of memory into a PDL. In this case, the hunk is three 1000x1000 image planes that you have somehow imported into a perl string, e.g. by reading from a file or executing a PerlXS script. The three image planes are to represent R, G, and B in a PDL with dimensions (3x1000x1000).

```
$pdl = PDL->new_from_specification(byte,1000,1000,3);
$dref = $pdl->get_dataref;  # $$dref is the PDL data as a string.
$$dref = $data;   # Overwrite the string.
$pdl->upd_data(); # Make sure the PDL knows it changed.
$rgb = $pdl->mv(2,0); # 3x1000x1000.
```

Here, `$$dref` is a Perl string that occupies the same location in RAM as the data in `$pdl`. Unless you're using 2-byte Unicode strings, the string has as many characters as there are bytes in the machine representation of the PDL. This example has a 3MB string - but a double-precision PDL with the same dimensions would have a 24MB string. Remember, `new_from_specification` allocates the PDL but doesn't initialize its contents - so initially the string is full of whatever garbage happened to be in that chunk of memory. Overwriting the string with a simple copy (or perl sysread operation) rapidly loads the binary data directly into `$pdl`, with no type conversion at all. (Warning - you can hose yourself if you shorten the string, which will de-allocate the end of the PDL!) Afterward, you have to update the internal data pointers in `$pdl`, in case Perl moved the string around. The final mv call makes sure that the dimensionality is right, without shuffling the actual bytes around.

If you use this low-level mechanism, you are responsible for making sure that the data you put into the new PDL has the same form as the PDL's formal data type! You are also responsible for figuring out byte swapping for your machine - the bytes in the string are in machine order, not network order.

Conversion to Perl types: at and list

You can get a PDL scalar out into the Perl world with `at`, which requires the index of the scalar to pull out:

```
pdl> $a = xvals(5)*2;  # $a is a PDL
pdl> $a4 = $a->at(4);  # $a4 is a perl scalar
```

You can also export a whole PDL with `list`:

```
pdl> @a = $a->list;
pdl> for($a->list) { print $_, - ; }
0 - 2 - 4 - 6 - 8 -
```

Be careful with `at`, as you almost never want to use it - it is tedious for anything nontrivial, and extremely slow! Particularly if you find yourself placing an `at` call inside a `for` loop, you should probably stop and think about how to use threading for your problem - see below.

Data Types and Contexts

Because PDL is a hybrid language, it's important to understand Perl's data structures as well as PDL's. Normal Perl variables are represented in a way that makes sense for Perl's original application - small to medium sized "glue" tasks - while PDL variables and arrays ("PDLs") have a more traditional typing scheme.

Unlike most other languages, ordinary Perl uses "polymorphous" (or "behind-your-back") typing. While the traditional simple types (boolean, string, short, long, float, double) are all represented, the language doesn't distinguish between the different types. The Perl engine keeps track of each variable's representation, and delivers to you the most appropriate representation depending on context. For example, + is an arithmetic operator, so the expression `"5" + 2` yields the number `7` even though one of its terms began life as a string.

PDL variables are implemented on top of Perl's normal variable system. A PDL is effectively a new type of perl scalar, that can contain a whole array of numeric values. PDLs are strongly typed, but are still slightly influenced by Perl's notion of context. In particular, PDLs behave slightly differently in numeric, boolean, and string context. In Perl numeric contexts, PDLs act normally. In boolean/logical contexts, they act like boolean values in the C language - the only false value is `0`, and any nonzero value is treated as true (note: Not all languages treat nonzero values as logical-true, which may come as a surprise to C or Perl programmers. For example, some FORTRANs and RSI's IDL language use the least-significant bit of integer variables as the boolean truth value of the integer).

In Perl string contexts, PDLs act like descriptive multiline strings (or the string `"TOO LONG TO PRINT"`). See the following subsections for details.

Refresher on Perl Data Types & Contexts

While the underlying representations of objects change, Perl itself recognizes only a few distinct variable types. These are " scalar", "ref", "array" (also called "list"), and "hash". (PDLs are implemented as special refs that are opaque to perl itself; perl treats them as scalars). The ones relevant to PDLs are scalars, lists, and hashes.

- **Scalar variables** or expressions hold a single value - a string, a number, the undefined value, or a reference ("ref") to one of the other basic types (see below). A scalar - even one that carries a numeric value - is slightly different than a PDL with one element.
- **List values** (often called "arrays" by the general Perl community) are collections of scalars that are indexed by number. Unlike normal arrays or PDLs, perl lists are expanded automatically as needed, so you can address any element whether it exists or no. List elements can contain any perl scalar value.
- **Hash values** are collections of scalars that are indexed by string. Hashes act like lookup tables or dynamic structures. Instead of being numbered, each element is addressed by name.
- **Refs** are special scalar values that hold pointers to other data types. They have a different name than pointers, to remind you that they are podiatrically friendly - it's much harder to shoot yourself in the foot with refs than it is with pointers. Perl variables maintain a reference count (like UNIX files) and are automatically deallocated when the last reference disappears - so you don't have to keep track of whether a ref is valid or no. Refs come in four basic flavors: scalar refs, list refs, hash refs, and code refs.

  Refs can be used to "roll up" large data structures (like lists) into a single scalar value; this is how Perl implements multi-dimensional lists and complex data structures. In addition, refs may be "blessed" into a particular object class; this is the mechanism that Perl uses for object

oriented programming. Blessing merely associates the target of the ref with a particular kind of object. PDLs are implemented as blessed Perl refs, so that a PDL (which may hold a million values) may appear wherever you can put a Perl scalar.

PDL Data Types

PDLs are strongly typed: when you create a PDL, it gets a particular representation and stays that way. The basic types are familiar to C programmers: byte, short, unsigned short, long, long long, float, and double. You can compile 64-bit support into your copy of PDL, and have access to wide doubles and other such exotica. Complex numbers are supported as a subclass of PDL; see Chapter [cha:Subclass-Smorgasbord].

PDL types are automatically converted as necessary within arithmetic expressions, at some cost in speed. Numeric expressions run faster between PDLs of the same type than between PDLs of different types, but all numeric expressions work more or less the way a C programmer would expect, with data types being automagically promoted to the highest complexity type that is used in each expression.

PDLs and Perl Contexts

While the representation of each PDL is fixed, the interpretation is different in each of the three main Perl scalar contexts:

Numeric context is what you get if you use PDLs in the usual way - adding, subtracting, and such. Normal numeric operations act elementwise, and each array preserves its storage class (char/byte, short-int, long-int, float, double, etc.). If you mix a PDL with a Perl variable in numeric context (for example, `pdl(2,3,4)+5`), then the Perl variable is "promoted" to a PDL.

Boolean context is what you get if you use a PDL in a branch statement like `if` or `while` or even the && and || operators. Multi-element PDLs are not allowed in this context, to avoid the confusion inherent in non-deterministic branching (&& and || are short-circuit operators that don't evaluate the second term if doing so would be redundant). Single-element PDLs are treated as TRUE if they are nonzero and FALSE if they are zero. (Note that the bitwise logical operators, such as & apply numeric, rather than boolean, context - so you can do elementwise Boolean arithmetic with &, |, and ^ - but not with &&, ||, and ^^.

String context is what you get if you use a PDL as a string. The PDL gets converted to a human-readable string suitable for printing. The new `pdl()` string input capability allows one to convert printed piddles back into the original object. The feature includes support for MATLAB-style `[ ; ]` syntax as well.

Because string conversion is intended for use with `print`, PDLs that are moderately large (more than about 1,000 elements) don't get converted - the string that you get back is `TOO LONG TO PRINT`. String context is easy to remember as "just" a way to give you direct access to the output of `print`: use a PDL as if it were a string, and you get the string that would be printed.

BAD Values

PDL lets you propagate bad/missing values in your data. You can set a particular numeric value that will be treated as BAD and ignored by the underlying code.

You can mark values BAD with the `setbadif` and `setbadat` methods. Bad values are treated as truly missing by statistical routines and collapse operators (that summarize each row of a PDL) and as poisonous by arithmetic routines. For example, `average` and `sumover` ignore bad values completely, multiplication will mark appropriate output values as bad, and `convolve` and `convolveND` will cause bad patches to spread throughout a block of data.

Dataflow

"Dataflow" is the concept that multiple variables can remain connected to one another (so that data flows between them). PDL allows you to keep multiple variables that refer to the same underlying

data. For example, if you extract a subfield of a large data array you can pass it to subroutines and other expressions just like any other PDL, but changes will still propagate back to the large array unless you indicate otherwise.

In general, PDL's element-selection operators (such as slicing and indexing) maintain dataflow connections unless they are explicitly severed. To support dataflow, PDL has two different kinds of assignment: the global assignment operator = and the computed assignment operator `.=`.

Global assignment is used to create new PDLs, and computed assignment is used to insert values into existing PDLs. Many other languages, such as FORTRAN and IDL, don't maintain dataflow for slices of arrays except in the special case where the slice operation is on the left-hand side of an assignment; in that case, those languages assume computed assignment rather than global assignment. That nuance sweeps under the rug the differences between the two types of assignment. It also yields many special cases that do not work correctly in those languages - for example, array subroutine parameters in IDL are passed by reference and can hence be used to change the original array - but array slices are copied before being passed, so the original array does not change. C sidesteps the issue by not (directly) supporting array slices. One result is that you can keep multiple representations of your data, and work on the representation that is most convenient.

For example:

```
pdl> $a = xvals(5);
pdl> $b = $a(2); # global
pdl> $b .= 100;  # computed - flows back to $a
pdl> print $a;
[0 1 100 3 4]
```

Here, $a and $b remain connected by the slicing/indexing operation, so the change in $b flows back to $a. Most indexing operations maintain dataflow.

At times, you want to ensure that your variables remain separate or to make a physical copy of your data.

The copy operator makes a physical copy of its argument and returns it. In general, if you want a real copy of something, just ask for it:

```
pdl> $a = xvals(5);
pdl> $b = $a(2)->copy;
pdl> $b .= 100;
pdl> print $a;
[0 1 2 3 4]
```

or, even more straightforwardly,

```
pdl> $a = xvals(5);
pdl> $c = $a;       # $c and $a remain connected
pdl> $b = $a->copy; # $b is a (separate) copy of $a
```

The `sever` operator is slightly more subtle. It acts in place on its argument, cutting most kinds of dataflow connection. It cannot disconnect two variables that were cloned with Perl's =; it can only sever the dataflow connection between related PDLs. The wart is present in current versions that rely on the Perl 5 engine, because it is not possible to overload the built-in = operator in Perl 5.

```
pdl> $a = xvals(5);
pdl> $b = $a(2:3)->sever; # $b is a slice of $a: gets separated
pdl> $b += 100; print $a; # changing $b doesn't affect $a.
[0 1 2 3 4]
```

```
pdl> $c = $a->sever;      # $c is a clone of $a: still connected
pdl> $c += 100; print $a; # changing $b affects $a.
[100 101 102 103 104]
```

Threading

Array languages like PDL perform basic operations by looping over an entire array, applying a simple operation to each element, row, or column of the array. This process is called **threading**. Threading is accomplished by the threading engine, which matches up the sizes of different variables and ensures that they "fit". The threading engine is based on constructs from linear algebra (but is slightly more forgiving than most math professors).

Most operations act on the first few dimensions of a PDL. These first dimensions are active dimensions and any dimensions after that are called thread dimensions. The active dimensions must match any requirements of the operator, and the thread dimensions are automatically looped over by the threading engine. The operator sets the number of active and thread dimensions. A given operator may have 0 active dimensions (e.g. addition, +), 1 active dimension (e.g. reduce operators like `sumover` and vector operators like `cross`), 2 active dimensions (e.g. matrix multiplication), or even more.

You can rearrange the way that an operator acts on a PDL by rearranging the dim list of that PDL, to bring dims down into the active position(s) for an operation or to bring them up to be threaded over. These rearrangements are a generalization of *matrix transposition*, though in general they are quite fast as they don't actually transpose the data in memory - only rearrange PDL's internal metadata that explain how the block of memory is to be used.

Threading rules

PDL operators that act on two or more operands require the thread dimensions of each operand to match up. The threading engine follows these rules for each dim (starting with the 0 dim and iterating through to the highest dim in either operand):

- If both operands have the dim and it has a size greater than 1 in each operand, then the size must be the same for both!
  - `print pdl(1,2,3) * pdl(3,4)` doesn't work, because dim 0 of the left operand has size 3 and dim 0 of the right operand has size 2.
  - `print pdl(1,2,3)*pdl(4,5,6)` prints the string `[4 10 18]`.
- If both operands have the dim and it has size 1 in at least one operand (it is a trivial dim), then the dim is "extended" as a dummy dimension. This is a generalization of scalar multiplication in linear algebra.
  - print pdl(1,2,3) * pdl(2) prints the string `[2 4 6]`.
- If a dimension exists in one operand and not in the other, it is treated as a virtual trivial dim
  - `print pdl([1,2],[3,4]) * pdl(3)` prints the string `[ [3 6] [9 12] ]`.
- If one operand is a PDL and the other is a Perl scalar, the scalar is PDL-ified before the operation
  - `print pdl([1,2],[3,4]) * 3` prints the string `[ [3 6] [9 12] ]`.

Controlling threading and dimension order: xchg, mv, reorder, flat, clump, and reshape

Because rearranging the dim list of a PDL (i.e. transposing it) is the way to control the threading engine, PDL has many operators that are devoted to rearranging dim lists. Here are six of them:

transpose - matrix transposition

`$at=$a->transpose` will yield the transpose of a matrix `$a` (that is, with the 0 and 1 dims exchanged); you can use `$a->inplace->transpose` to change the variable itself. Of course, if `$a` has more than two dims, it is treated as a collection of matrices (the other dims are threaded over).

xchg - generalized transposition

You can generalize transpose to any two dims with xchg - just give the index numbers and those two indices get exchanged (transposed): `$at = $a->xchg(0,1)` is the same as using transpose, but you can also say (for example) `$ax = $a->xchg(0,3)`.

mv - dim reshuffling

Using mv shifts a dim from its original location to a new location; all the other dims stay in the same relative order but get shifted to make room and/or fill up the old slot. You can say, for example,`$b = $a->mv(3,0)` to move dimension 3 to the 0 slot. Afterward, `$b` will have the dimensions of `$a` in the order (3,0,1,2).

reorder - arbitrary redimensioning

This is useful for carrying out many transpositions at once. You specify the order in which the old dimensions should appear in the new PDL: `$b=$a->reorder(3,0,1,2)` is the same as `$b=$a->mv(3,0)`, and `$at=$a->reorder(1,0)` does the same thing as `$at=$a->transpose`. You can reorder all the dimensions of your PDL or just the first few - if you ignore later dimensions they carried along "for the ride", keeping the same order in which they came.

flat - flatten a PDL

Flat reduces a PDL of arbitrary dimension to one with a single long dimension. The 0 dimension runs fastest in the resulting 1-D PDL, and the last dimension runs slowest. For example, if `$a` is a 120x120 image then `$a->flat` is a 1-D array of 14400 values. That is useful, for example, for making a reduce operator (see Section [sub:Collapse-Operators]) work on a whole PDL at once. In the above example, `$a->average` would return a 120-array of row average brightnesses, but `$a->flat->average` would return the average brightness of the whole image (or, if `$a` had more dimensions) the average brightness of the whole collection.

clump - flatten specific dims

Clump is useful for making an operation that normally works on one dimension work on more at once. For example, `$im->average` reduces an NxMx3 RGB image into a Mx3 array of row-average brightnesses. If you want the average brightness of each color throughout the whole image, you can say either `$im->average->average` or `$im->clump(2)->average`, to get a 3-array of average brightnesses for R, G, and B.

reshape - allocate dims yourself

With reshape you can reassign the block of memory that makes up a PDL, cutting it up however you please. For example, if `$a` is a 60x60 image, you can say `$b=$a->reshape(100,36)` to create instead a 100x36 image. The product of the new dimensions should be less than or equal to the product of the old dimensions, or strange things may happen!

Dummy Dimensions

Dummy dimensions are bookkeeping dimensions that act to the threading engine like complete dimensions but in fact repeat the same data in each position in the new dimension. A dummy dimension is simply a convenient bookkeeping convention; no extra memory is allocated for it. You create dummy dimensions with the dummy operator or via the slicing syntax explained elsewhere.

The dummy operator takes two parameters: a position at which the dummy dimension is to be inserted into the dim list, and a size. For example, if `$a` is a 100-array, then `$b=$a->dummy(0,50)` makes `$b` a 50x100 image - except that each column of `$b` points to the same piece of memory, so

that assigning to any element of $b changes a whole row.

You can "physicalize" a dummy dimension by making an explicit copy. For example, `$b=$a->dummy(0,50)->copy` makes $b a 50x100 image, each column of which happens to contain the same data, but in this case every pixel of $b is allocated separately from memory, so that assigning to $b works in the normal way.

Collapse/Reduce Operators and Reduction

PDL contains many "collapse operators": enough of them that they deserve special attention as a group. A collapse operator has a single active dim. It summarizes elements along each row (the 0 dim) of a PDL, returning the summary of that row as a single number. Thus, a collapse operator will reduce a D-dimensional PDL to D-1 dimensions. The average, sumover, and andover operators are examples of collapse operators: each one has a single active dim and produces the average, sum, or logical AND (respectively) of everything along that dim of the argument PDL. To average over a dim other than the 0 dim, you must move that dim to the 0 position. For example, to convert a color image that is (NxMx3) to a black-and-white image that is (NxM) you can say $bw=$rgb->mv(2,0)->average. For historical reasons, some documentation refers to them as "reduce operators", because they reduce the dimensionality of their operands.

PDL Headers

Every PDL can contain a "header" - a perl hash ref (that is, a collection of keyword/value pairs) that stores metadata about the PDL itself. Some of the built-in routines are aware of the FITS WCS format for metadata about scientific images, and use the header slot to store a WCS coordinate system about the PDL; but most operations do not use or affect the header at all. You are free to store whatever data you like in it.

An internal flag associated with each PDL controls whether the header is propagated to derived PDLs. Copying the header can be a time-consuming operation, many times slower than arithmetic on small PDLs - but it can be quite convenient as well. PDL keeps the copying flag false by default on most new PDLs, but if you set it to true (using the hdrcpy method, see below), then the both the header and the copy flag will be copied to derived PDLs.

Convenient interfaces exist to use an *Astro::FITS::Header* tied hash instead of a normal Perl hash ref. *Astro::FITS::Header* tied hashes act like normal Perl hashes but force case-insensitivity and provide some control over the card structure of the underlying FITS header.

hdr & fhdr - access PDL header elements

You can access elements the header of a PDL by inlining the hdr or fhdr method into a hash dereference: `$a->hdr->{keyword}=$value;`, or `$val=$a->hdr->{keyword}`. If the header doesn't exist, then it is autogenerated. The only difference between hdr and fhdr is that, if no header exists, fhdr autogenerates tied FITS header objects while hdr autogenerates normal Perl hashes.

gethdr & sethdr - manipulate a complete PDL header

You can get or store the current header of a PDL with the `gethdr` and `sethdr` methods. `$a->gethdr` returns either a hash ref (which could be a tied object such as a FITS header object) or the undefined value. `$a->sethdr($hdr)` accepts either a hash ref or the undefined value, and assigns it to the pdl's header.

hdr_copy - return a deep copy of a PDL's header

The gethdr method makes a shallow copy of the PDL's header - it returns a ref that points to the original header data. If instead you want a complete, deep copy (that you can modify without affecting the original PDL) you want `hdr_copy` instead.

hdrcpy - control header copying

If you apply an operator to a PDL with a header set, you can arrange to have the header copied to the result PDL. The underlying hash or object is deep-copied, which is somewhat expensive; so you must set a flag on the source PDL to make it happen. `$a->hdrcpy()` returns the state of the copying flag; `$a->hdrcpy($flag)` sets it. False values (the default) turn the feature off, true values turn it on.

Slicing, Dicing and Threading dims with PDL

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

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

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

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

Finding piddle dimensions.

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

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

See also the `shape` function which returns the pdl shape as a pdl:

```
$datashape = shape($data);
```

The number of elements in a piddle is equally easy:

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

The slice function - regular subsets along axes

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

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

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

```
$section = $gal->slice('337:357,178:198');
```

Use the on-line documentation:

```
pdl> help slice
```

to explore the full set of options. `slice` is probably the most frequently used PDL function so we will explore it in some detail. But first we notice that `slice` is implemented via a named function.

Through the magic of `PDL::NiceSlice` and source filtering you can access `slice` functionality in a form very similar to the vector array syntax found in many array computation languages such as FORTRAN-90 and MATLAB. The chief difference being that the argument to the `slice` method call is a *string* describing the elements to be selected. For the new `PDL::NiceSlice` syntax, you don't use the method or function call and the argument does not need to be wrapped up in a string.

In this chapter, we will usually show the `PDL::NiceSlice` syntax but refer to the operation as a slice even though with the new syntax there is no longer an explicit `slice` method being called.

The basic slicing specification.

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

```
$data = zeroes(1000);
$sec = $data(0:20);
$data = zeroes(100,100,20);
$sec = $data(0:20,40:60,1:3);
```

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

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

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

```
pdl> $x = sequence(24); # Create a piddle of increasing value
pdl> 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]
pdl> print $x(16:22:2)
[16 18 20 22]
```

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

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

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

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

You also can use perl scalars to construct the slicing specifications:

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

Modifying slices.

Here's the biggy:

```
pdl> $x = sequence(24);
pdl> 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]
pdl> $slice = $x(4:20:2);
pdl> print $slice;
[4 6 8 10 12 14 16 18 20]
```

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

```
pdl> $slice .= 0;
pdl> print $slice;
[0 0 0 0 0 0 0 0 0]
pdl> 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 modifies the original data! However it is done (`$slice++` etc. work just as well).

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

Does a slice consume memory?

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

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

If you monitor the memory consumed by the PDL process on your computer (UNIX/Linux users can try the `top` command) you will see that the amount of memory consumed does not go up - **even when the slice is modified** . This is because the way PDL is written allows many of the simple operations on slices to be optimized - 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 optimization is not possible. But the book-keeping of propagating the changes back to the original piddle is handled automatically.

Advanced slice syntax

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

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

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

And then one can put them back again using "*":

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

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

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

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

PDL's Method notation

At this point we would like to introduce the varied notations for calling `slice` and it's friends. This is because it will be commonly seen in PDL code and is very handy. While at first unfamiliar 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 supply) can be re-used for different objects.

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

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

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

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

The dice and dice_axis functions - irregular subsets along axes

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

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

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

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

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

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

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

Using mv, xchg and reorder - transposing dimensions

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

`xchg` simply swaps two dimensions:

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

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

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

Finally one can completely re-order dimensions:

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

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

Combining dimensions with clump

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

```
pdl> $a = xvals(5,3,2);
pdl> 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:

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

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

```
pdl> 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:

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

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

Adding dimensions with dummy

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

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

```
pdl> 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 :

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

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

```
pdl> 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:

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

Completely general subsets of data with index , which and where

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
# Increment first and last values


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

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

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

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

```
# e.g.  make a cube with a sphere of 1's in the middle:
$cube = rvals(100,100,100);
```

```
$tmp = $cube->where($cube<20);
$cube .= 0;
$tmp .= 1;
```

PDL threading and signatures

Slicing and indexing arbitrary subsets of data is certainly a fundamental aspect of any array processing language and PDL is no exception (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) vectorized iteration of "elementary operations" over arbitrary slices of multidimensional data. First we have to explain what we mean by **threading** in the context of PDL, especially since the term **threading** already has a distinct meaning in computer science that only partly agrees with its usage within PDL. In the following we will explain the general use of PDL threading and highlight the close interplay with those slicing and dicing routines that you have just become familiar with (`slice`, `xchg`, `mv`, etc). But first things first: what is PDL threading?

Threading

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

A simple example

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

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

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

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

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

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

that generates the following output:

```
Input Byte D [256,256]
Result Byte D [256]
```

```
Since is important to keep track of the dimensions
of piddles when using (and especially when
introducing) B<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 and we made the function return the piddle argument so that it can be seamlessly integrated into method invocation chains:

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

A small utility routine

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

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

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

Now it should be no surprise that

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

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

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

Why bother?

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

More examples

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

```
pdl> $im3d = sequence short, 5,10,3; # a 3D image (volume)
pdl> $max = $im3d->xchg(0,1)->pdim('Input')->maximum;
pdl> print $max->pdim('Result') .  " \n";
```

generates

```
Input   Short D [5,10,3]
Result  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 result has the intended dimensionality. Formulating your problem at hand in a way that makes use of threading rather than resorting to nested `for`-loops at the perl level can make the difference between a script that is executed faster than you can type and one that is crawling along and giving you plenty of time to have your long overdue lunch break.

Why threading and why call it threading ?

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

Before we get into the details of the why and how let's admit: PDL is by no means the first data

language that supports this type of automatic implicit looping facility: the authors have in fact been inspired by several previous data language implementations, most notably `Yorick`

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

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

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

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

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

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

Our benchmarking script looks like this

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

If we run this script it generates

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

(n=39)That proves our point: while the example using threading is executed at a rate of nearly 13,000 per second using explicit loops has brought down the speed to less than 13/second, a very significant difference.

Obviously, the difference between threading and explicit looping depends somewhat on the nature of the elementary operation and the piddles in question. The difference becomes most striking the more elementary operations are involved and the faster an individual elementary operation can be performed. The advantage of threading will level off as the time for performing the elementary operation becomes comparable or even greater than that required to execute the explicit looping code.

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

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

Even though PDL threading makes your life much easier in one respect by taking care of some of the "messy" details it leaves you with another task: you have to find the places in your algorithm/problem where threading can effectively be used and help to make for speedy execution even when using an (almost inevitably slower) scripting language. But finding such places and making use of these vectorized 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. In fact, as of PDL-2.4.10, there is a new capability where PDL now support automatic parallelization of the PDL threadloops via POSIX threads:

```
use Benchmark qw(:hireswallclock);
use PDL;
$a = zeros(2_000_000);
$b = zeros(2_000_000);
set_autopthread_size(0);

set_autopthread_targ(10);  # split across 10 threads
timethese(20,{threaded => '$a **= 1.3'});

set_autopthread_targ(0); # Set target to 0 for unthreaded
timethese(20,{unthreaded => '$b **= 1.3'});
```

For a Vista/Cygwin system with a quad-core i5 processor we see an greater than 2.5X reduction in wall clock time by using multiple processor cores. See documentation for *PDL::ParallelCPU* using `help PDL::ParallelCPU` in one of the PDL shells, or with `pdldoc PDL::ParallelCPU` from the command line.

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.:

```
pdl> 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 unnecessary 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:

```
$pdl(($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

```
$pdl(($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 subslices. 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:

```
$pdl(:,($i),($j)) $i=0..3; $j=0..4
```

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

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

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

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

Alright, the principles are simple. But nothing is better than a few examples. Again a typical imaging processing task is our starting point. We want to convert a colour image to grayscale. 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 gray 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:

$$i = \frac{77}{256} r + \frac{150}{256} g + \frac{29}{256} b$$

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:

$$c = \sum_{i=0}^{n-1} a_i \, b_i$$

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

```
pdl> sig inner

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

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

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

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

```
use PDL::IO::Pic; # IO for popular image formats
$im = rpic 'pdllogo.jpg'; # a colour image from the book dataset
$gray = inner($im->pdim('COLOR'),$conv);
   # threaded inner product over all pixels
$gb = $gray->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 grayscale 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 $gray
Signature a(n); b(n); [o]c()
Dims (3)[500,300] (3)[] ()[500,300]
Slices ":,($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 grayscale 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 $gray = inner $im, $conv;
   return $gray->convert($im->type); # convert to original type
}
```

You can write your own threading routines

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

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

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

There is some new syntax in this signature that we haven't seen before: writing `tri=3` signifies that in a elementary operation `rgbtogr` will work on 1D subslices (we have encountered this before); additionally, the size of the first dimension (named suggestively `tri`) **must** be three. You get the idea. What we have just seen is worth keeping in mind! By using PDL functions in our own subroutines we can make new functions with the ability to thread over subslices. Obviously, this is useful. We will come back to this feature when we talk about other ways of defining threading functions using PDL::PP below.

Matching threading dimensions

After this small digression, back to the subject at hand: what happens when both piddle arguments have extra dimensions? Well, the extra dimensions have to match. Otherwise we wouldn't know how to sensibly pair the subslices, right? So when do extra dimensions match? It is quite simple: corresponding extra dimensions have to have the same size in both piddle arguments. Corresponding extra dimensions are those that occur in both piddles. However, one piddle can have more extra dimensions than the other without causing a mismatch. That sounds strange? Ok, here is an example. We use one of the fundamental arithmetic operations in PDL, addition 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 (with NiceSlice syntax):

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

How do we achieve that by threading?

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

is not the right way.

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

```
PDL: PDL::Ops::mult(a,b,c): Parameter 'b'
```

```
PDL: Mismatched implicit thread dimension 0: should be 3, is 2
Caught at file (eval 344), line 4, pkg main
```

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

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

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

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

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

Using our symbolic way of writing down the slices that are paired in a 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

```
$AB(($i),($j)) = inner $A(:,($j)), $B(($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.

Writing your own functions into PDL

Using PDL Functions

PDL shares the Perl method for building functions for code that perform a commonly repeated function - you can define a function with `sub`, a function name, and a pair of curly braces.

Here's a simple function in a PDL script:

```
#!/usr/bin/perl
use PDL;
$a = sequence(10);
$b = $a * 4;

$result = my_sums($a, $b);
print $result;

print my_sums(pdl(10,20,30), pdl(3,4,5) );

sub my_sums {
    my ($a1, $a2) = @_; # pdls passed in the perl array @_
    my $c = $a1 + $a2;
    my $difference = $a1 - $a2;
    return($c, $difference);
}
```

As you can see, the function is called `my_sums` and it is defined at the end of this script, but you can be define functions anywhere in the script. In the example above, we call `my_sums` twice, printing out the answers as we go.

You can return as many PDLs from the function as you want, by passing them out in a comma separated list.

You can define the functions in any part of your Perl script, even after the point in the program that you started using the function. The input variables to the function are passed through the `@_` array, and we can put any set of piddles into the function. The function also has local scope, so variables

inside the routine are not seen by anything calling that function. Remember, though, that the variables *outside* the function **can** be seen inside the function! It's good practice to have a `use strict;` inside your functions whilst writing them, though, as this will help catch bugs.

Moving Functions into Separate Files

It gets tiring to copy and paste your useful functions from script to script, and so PDL provides a way to have your functions stored in a file that can be read by many scripts.

Two important notes:

- The filename has `.pdl` at its end, not `.pl`
- The file has `1;` as the last line outside the curly braces of the statement.

Use a file editor and cut and paste the text below into a file with the name `my_sums.pdl`.

```
sub my_sums {
    ($a1, $a2) = @_;
    my $c = $a1 + $a2;
    my $difference = $a1 - $a2;
    return($c, $difference);
}
1;
```

Now create a separate script in the same directory:

```
#!/usr/bin/perl
use PDL;
use PDL::AutoLoader;
$a = sequence(10);
$b = $a * 4;
$result = my_sums($a, $b);
print $result;
print my_sums(pdl(10,20,30), pdl(3,4,5) );
```

Running this Perl script will include PDL, and PDL will automatically look for a file called `my_sums.pdl` (remember the extension has to have `.pdl`) and use it.

Getting PDL to look for your functions in other places

After a while, you will have many PDL functions scattered over many directories, and so it makes sense to collect all your functions into a separate directory and have PDL look for them there.

You can set an environment variable in your shell called `PDLLIB` to look within a given directory. One convention is to use `PDLLIB=${HOME}/lib/pdl+` to store all your functions. When defined in your system shell, this will inform PDL where you've put your commonly used functions. The directory path is specified to your PDL library directory. The + sign at the end of the path tells PDL to also look in all the subdirectories below the PDLLIB directory.

Documenting your Functions

Just like any other Perl script, you can add Plain Old Documentation (POD) inside your PDL functions.

For a detailed look at what you can have in a POD, look at *perlpod* with `perldoc perlpod` or look online for a tutorial.

At the bare minimum, the POD should say what the PDL function does, what are its inputs and outputs. Further detail may include one or two examples so that a new user can test it and check they

understand what the function behaves, look at *perlpod* with `perldoc perlpod` or look online for a tutorial. =for comment :!podchecker PGPLOT.pod && pod2html --infile=PGPLOT.pod --outfile=PGPLOT.html open PGPLOT.html and click 'reload' also conversion for PDF reading is: pod2pdf PGPLOT.pod --icon-scale 0.25 --title "PDL Book" --icon logo2.png --output-file PGPLOT.pdf

Plotting and Labeling Data and Images using PGPLOT

A central requirement of any data analysis package is the possibility of visualization 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.

Here we will cover the use of the `PDL::Graphics::PGPLOT` package which uses the freely available PGPLOT subroutine package written by Tim Pearson. 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 *Using PGPLOT commands directly* 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 this section is to familiarize 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.

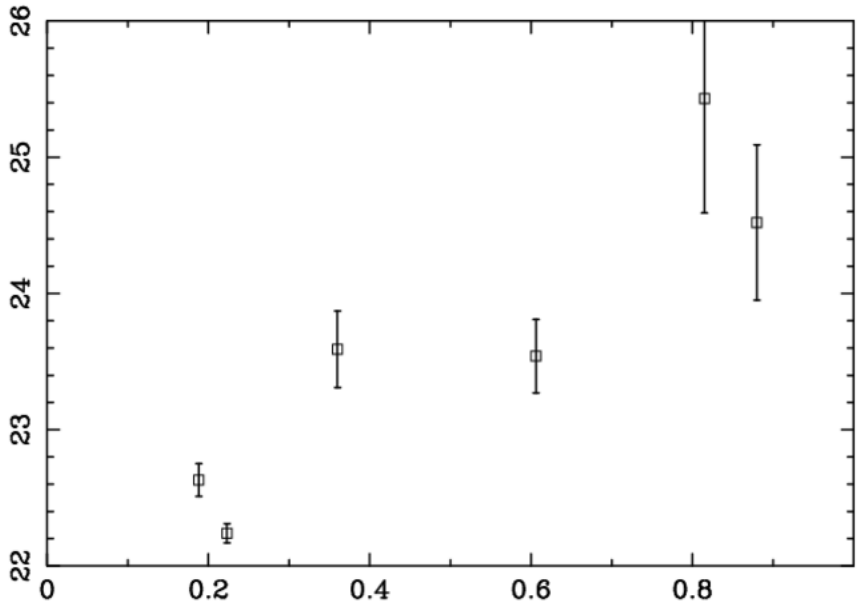
To use `PDL::Graphics::PGPLOT` it is necessary to have the PGPLOT package installed, and in addition have the Perl PGPLOT module (written by Karl Glazebrook and available through CPAN) installed and working. In the following we will assume that you have this all set up.

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`

```
pdl> 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 1.



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

```
> perldl
Type 'help' for online help
Type 'demo' for online demos
Loaded PDL v2.4.3
pdl> 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:

```
pdl> 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 1, so we need an x-variable that is both positive and negative.

```
pdl> $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:

```
pdl> 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:

```
pdl> hold
```

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

```
pdl> 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:

```
pdl> 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* and *y* variables and the error on *y*.

```
pdl> $x = pdl(0.88, 0.223, 0.815, 0.606, 0.188, 0.360)
pdl> $y = pdl(24.52, 22.24, 25.43, 23.54, 22.63, 23.59)
pdl> $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:

```
pdl> 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:

```
pdl> errb $x, $v, $dv,  {Symbol => 'Square'};
```

And here we go! 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.

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 behavior of the plotting routine and are very useful so we will first discuss the standard options and how options are specified.

Options in plot commands

As mentioned above and seen in the brief introduction to the PGPLOT interface earlier, we use options to modify the behavior of plot commands. Below we will often see examples of **specific** options, those that are only recognized by a particular plot command. However in addition there are **general** options that are recognized 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 realized 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 parallel 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

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

or using the `pdldoc` command

```
bash$ pdldoc 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 option 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`

sets 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 code:

```
pdl> $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 extension to the syntax for the PGPLOT command `pgsah` . The equivalent to the above is

```
pdl> $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.

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

`Axis`

Set the axis type (see the `env` command below in *Setting up the plot area*). It can either be specified as a number, or by a name as in the following table

```
Name        Number  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 (i.e. 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       4   Blue        8   Orange
1   Black       5   Cyan        14  Dark  gray
2   Red         6   Magenta     16  Light Gray
3   Green       7   Yellow
```

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-recognized 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 recognized 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 *Text and legends* 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 filltype 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(height,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 *Drawing lines and plotting points* 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 consult the PGPLOT documentation 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 normalized 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 (recognized name in capital letters):

```
0   Square      4   Circle      9   Sun
1   Dot         5   Cross       11  Diamond
2   Plus        7   Triangle    12  Star
3   Asterisk    8   Earth       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 can draw regular polygons with *n*-sides by setting the symbol to *-n*, so that `$opt = {Symbol => -n };` but be aware that *-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.

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 *Setting up the plot area*) or redefined using the method outlined in the next section.

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 '`help\InsetSpace ~perldl`' in the perldl shell or '`pdldoc pdl`'). 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 ('setpgplotoptions');
setpgplotoptions('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. However, character size, device default plot symbol, border and other options can be conveniently be specified in this way.

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 .

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')`

Colour Postscript files - `dev('filename.ps/cps')`

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 applied when the window is created, but it should be updated as soon as you plot to it. The default value is 0.618, i.e. 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:

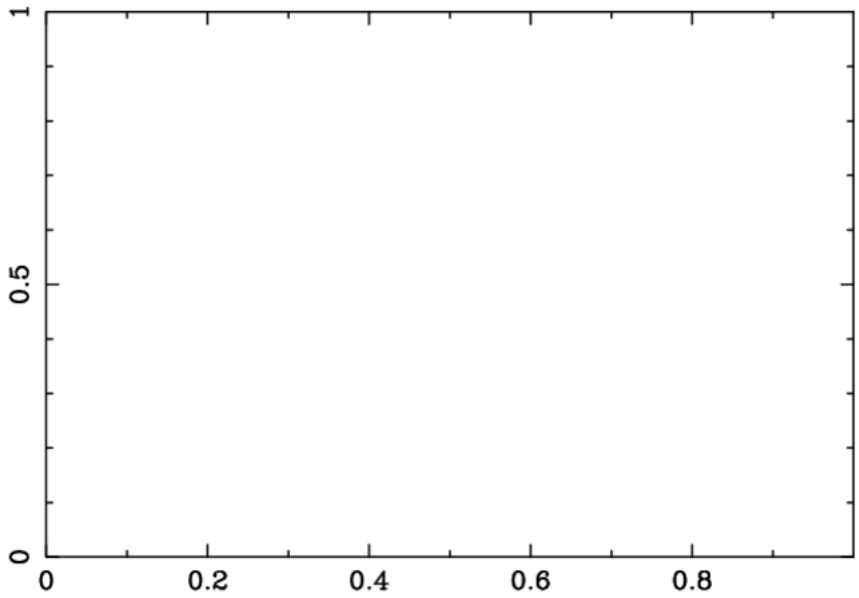
```
pdl> dev('/xs', {NX => 4, NY => 2})
```

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

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

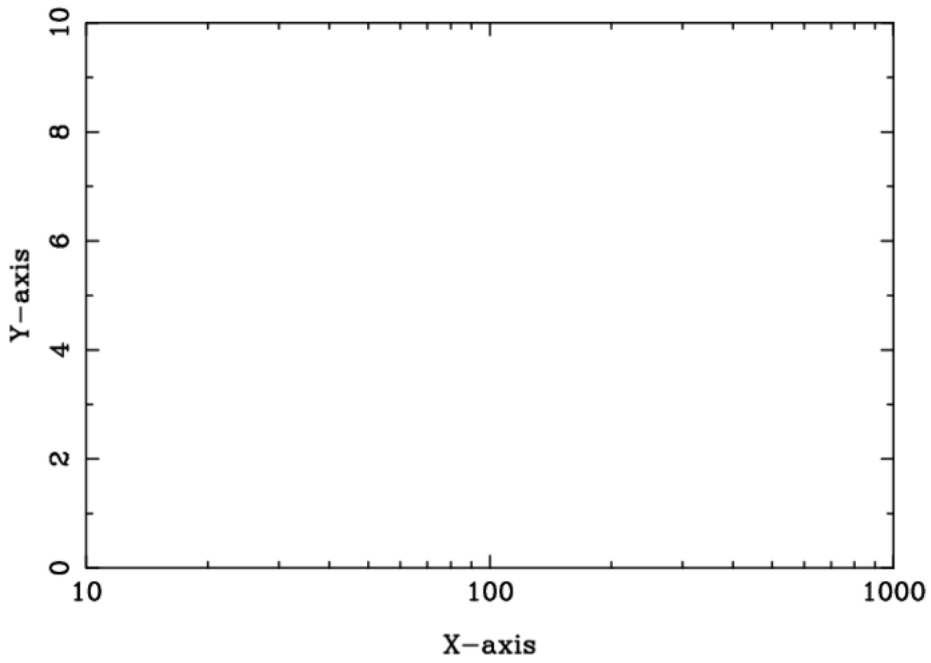
However dev does not actually draw anything for you, it merely selects the output device. To set up a plot you either call a plot command directly, or if you want more control over the axis ranges you use the command `env`. This useful command takes the upper and lower limits in X and Y as input:

```
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'});
```

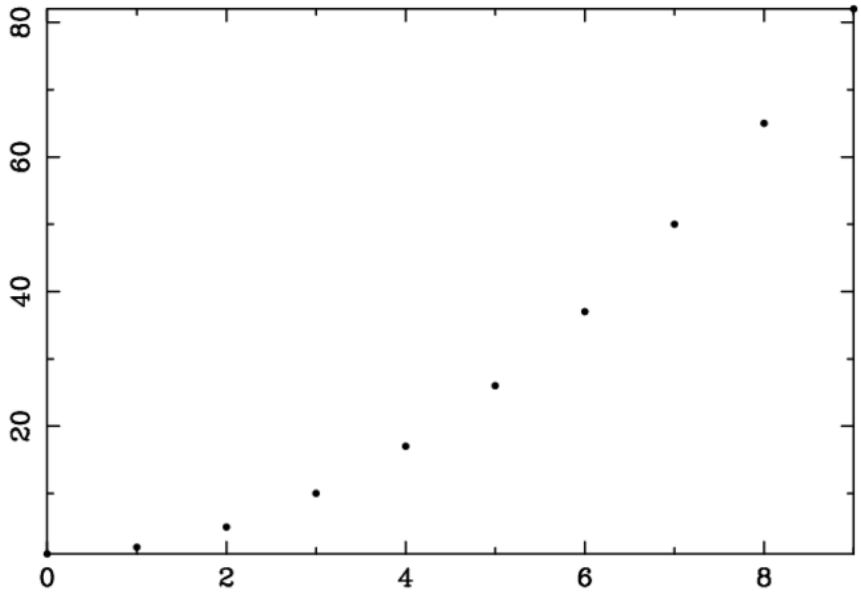


Further information on the `Axis` option can be found in *Options in plot commands*.

It is important to realize 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.

Drawing lines and plotting points

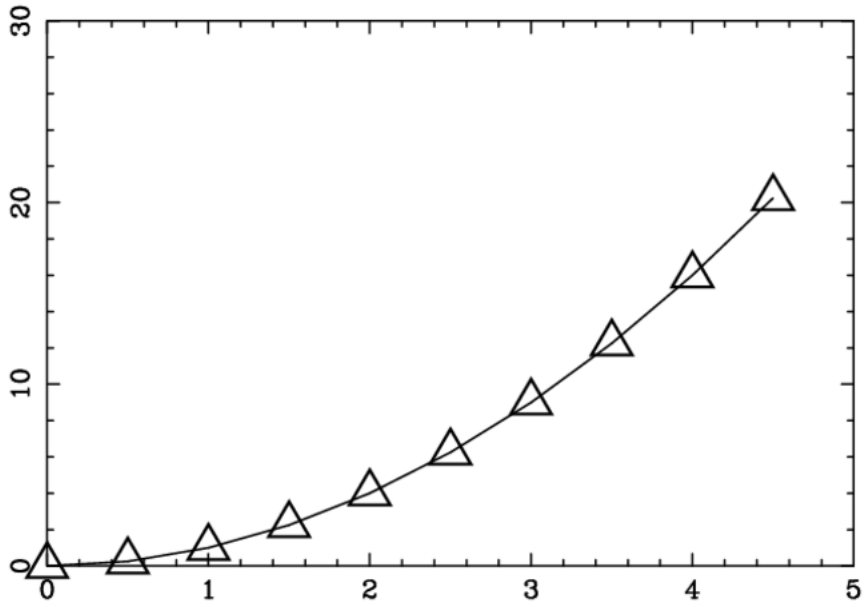
The most important commands in the graphics package are probably the line drawing and point plotting commands `line` and `points` . The most basic command is `points` which plots particular symbols at given x and y values:




```
pdl> $x = sequence(10)
pdl> $y = $x*$x + 1
pdl> 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:

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

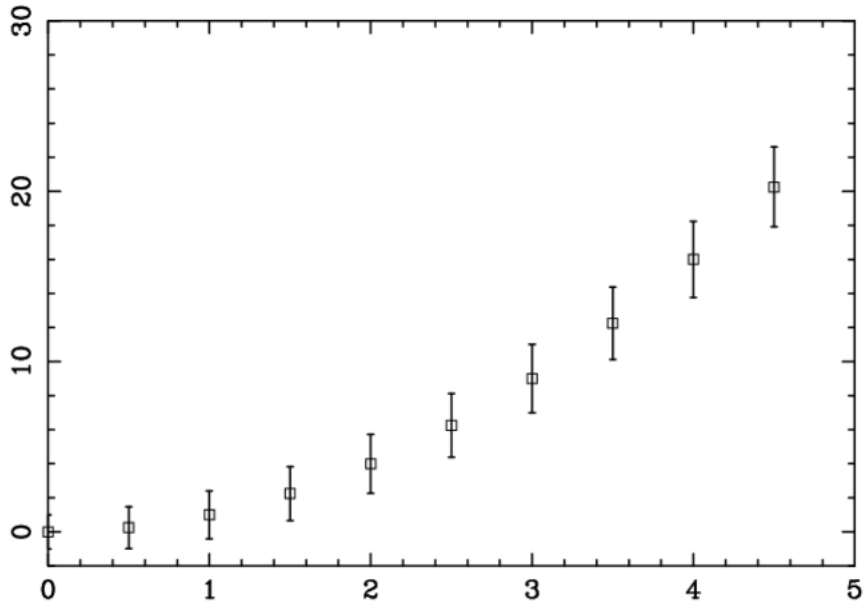


The string `Triangle` is equivalent to symbol number 7 and in general symbols will have to be accessed using the numerical system, but there are textual equivalents for many commonly used symbols (see *Options in plot commands*). The `points` command does also accept a piddle as the symbol value, in 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

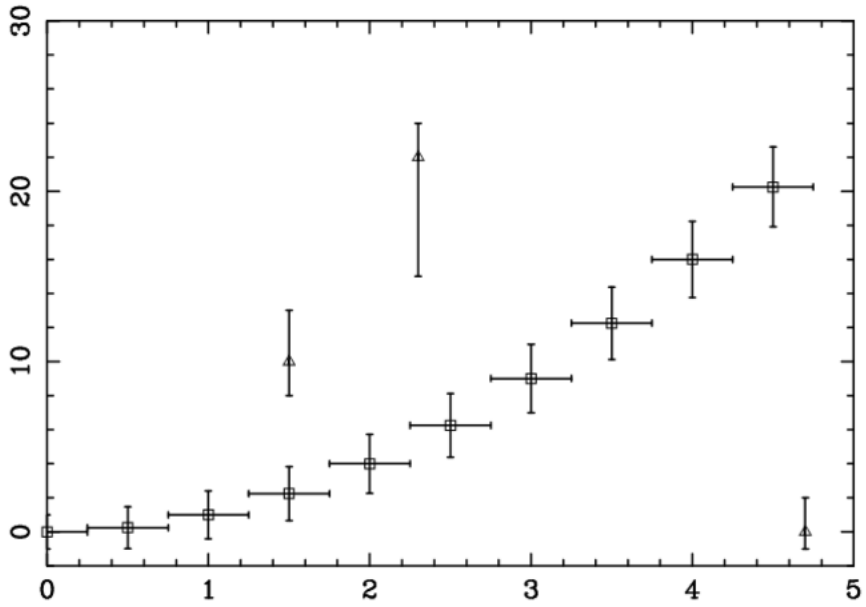
Closely related to `points` is the routine for plotting symbols with error-bars, `errb` . This can be called in a variety of ways to allow for various ways of giving errorbars and whether horizontal or vertical errorbars are required. A typical call is:

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



which plots squares with symmetrical vertical error-bars. To get error bars in the horizontal direction one gives 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:

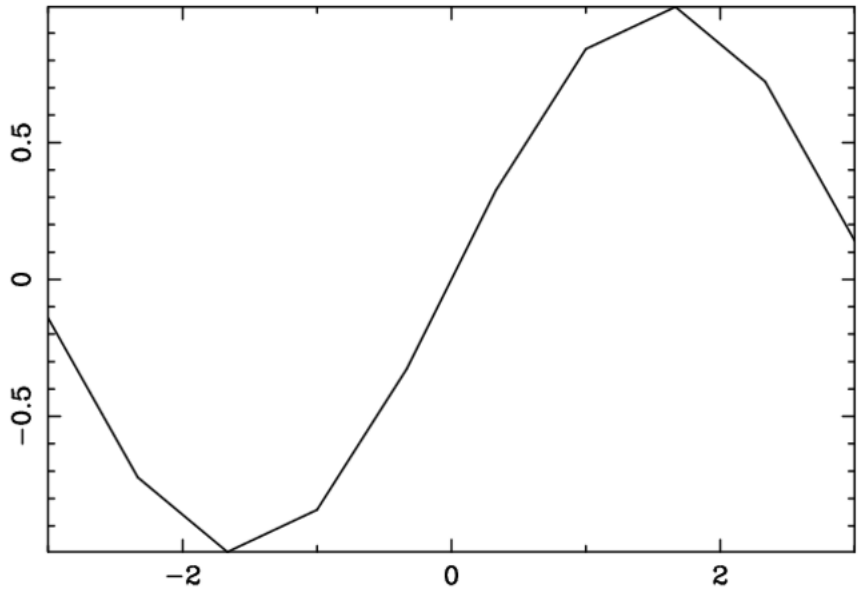
```
pdl> $x2 = pdl(1.5, 2.3, 4.7)
pdl> $y2 = pdl(10, 22, 0)
pdl> $dx = $x2->zeroes(); # No X-errors
pdl> $yu= pdl(12,29,1)-$y2
pdl> $yl= $y2 - pdl(7, 20, -2)
pdl> errb $x2, $y2, $dx, $dx, $yl, $yu, {Symbol => 'Triangle'}
```



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.

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

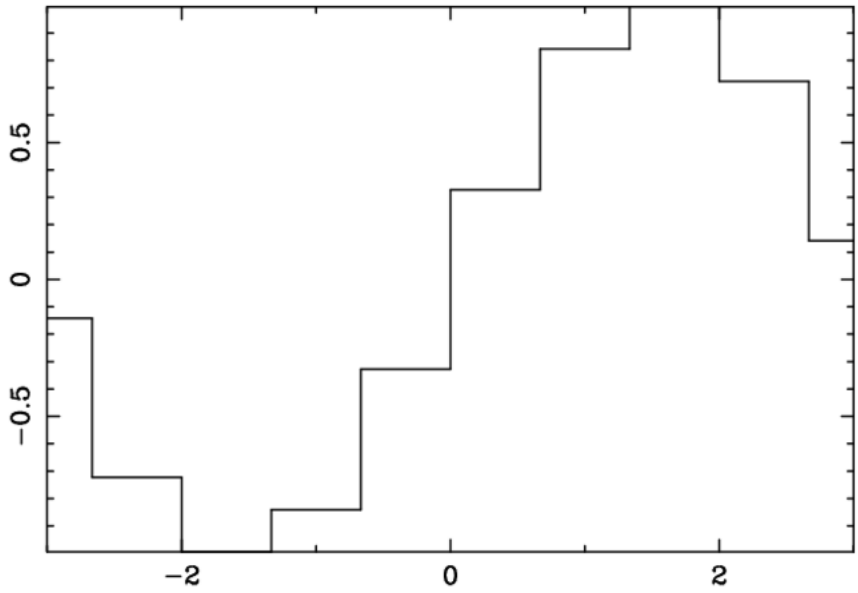


The style, width and colour of the line can be changed with the options `Style`, `LineWidth` and `Colour` / `Color` respectively as outlined in *Options in plot commands*.

Plotting histograms

A very similar command is `bin` which is useful for plotting histograms. This command draws horizontal lines between *x(i)* and *x(i+1)* with the value *y(i)*.

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

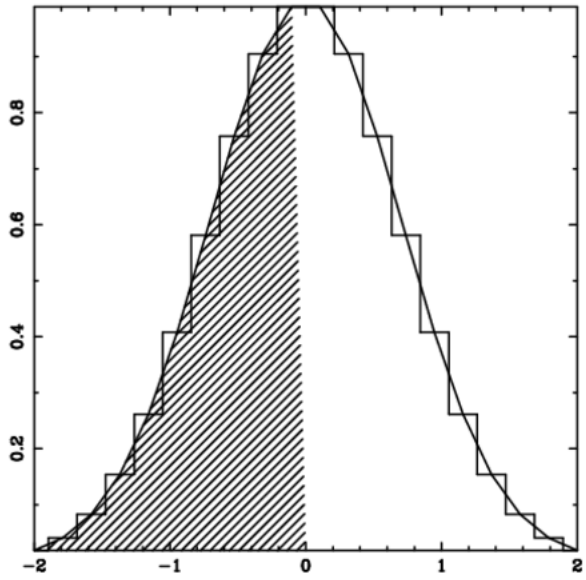


By default the routine assumes that the X-values are the start points of the bin, if instead your values are for the centers of the bins, you need to set the option `Centre`/`Center` to a true value. In addition the appearance of the lines can be modified using the same options as for the `line` command.

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 *Setting default values for options*, or you can set it explicitly as in the following example:

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

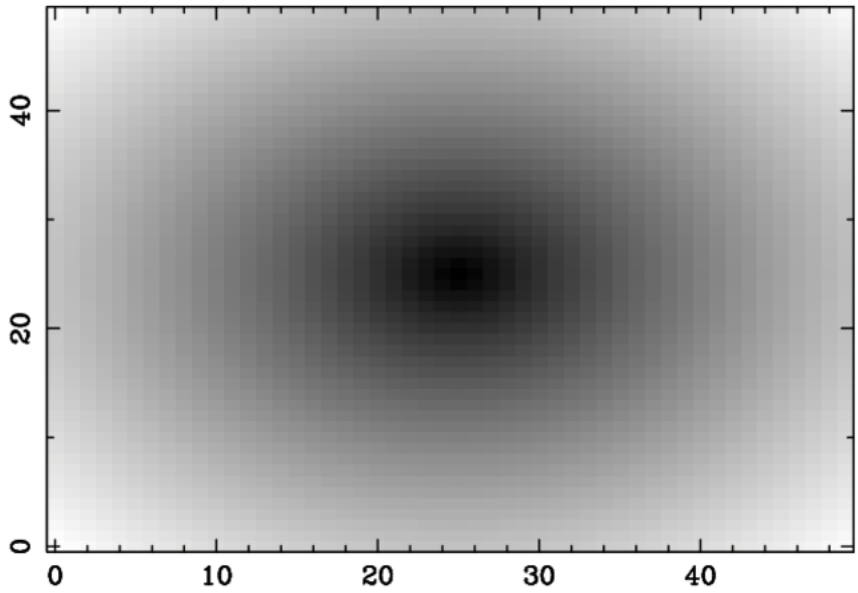


In this example it is worth noting the added complications to ensure that the polygon is closed. In addition we have used the option `FillType` to change the style of fill used. This can be finely adjusted if necessary, for further examples see *PDL::Graphics::PGPLOT* and the discussion of `FillType`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

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:

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



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 *Setting up the plot area*).

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:

```
pdl> 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 only adjusts 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 parlance) 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.

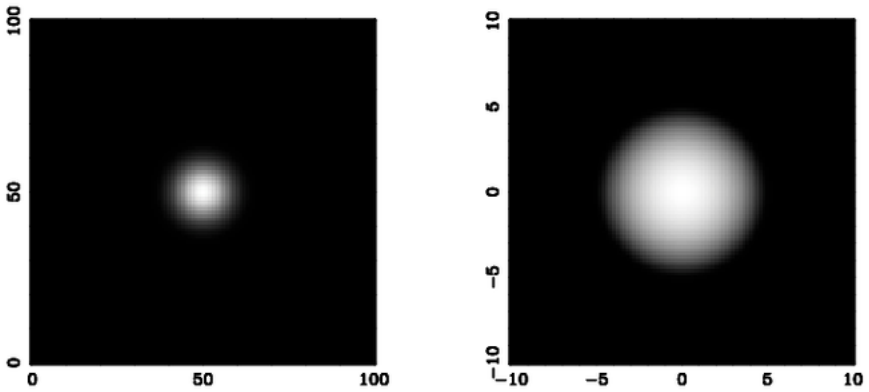
Transforms

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) = T0 + T1(i) + T2(j)
Y(ij) = T3 + T4(i) + T5(j)
```

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.

```
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
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'};
```



Here we are 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. 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 desirable 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 behavior 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`

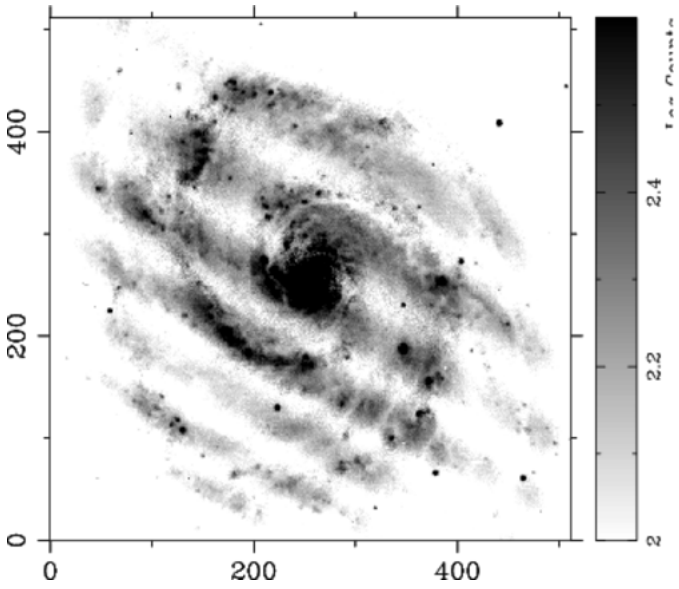
The value to set the foreground colour to. This can be referred to as `Fg` as well. The default is the max value used by `imag` when drawing the image.

`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.




```
    dev '/xs', {WindowWidth => 6, Aspect => 1};
    $im = rfits('Frei/n4013lJ.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
```

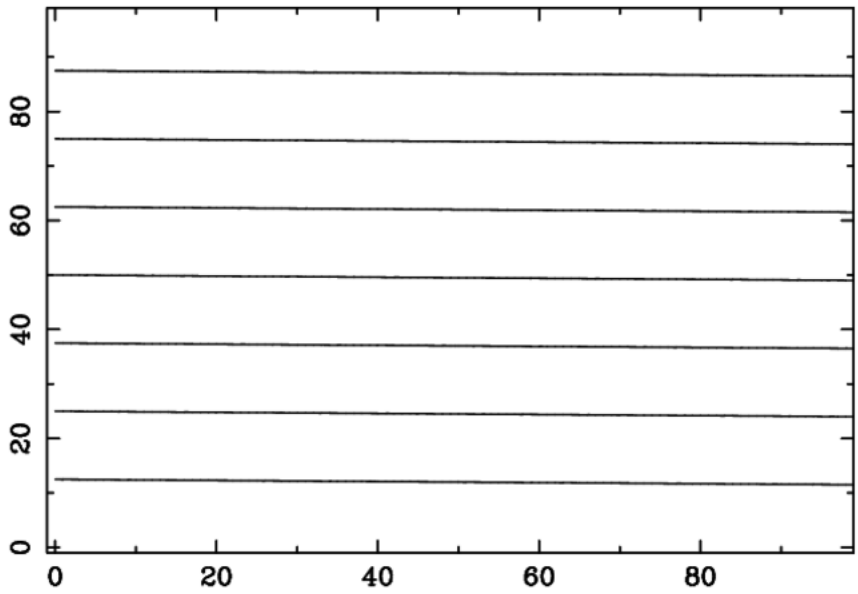
```
=> 1}});
```

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 the Figure above where we display a galaxy and display a look-up table next to it.

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

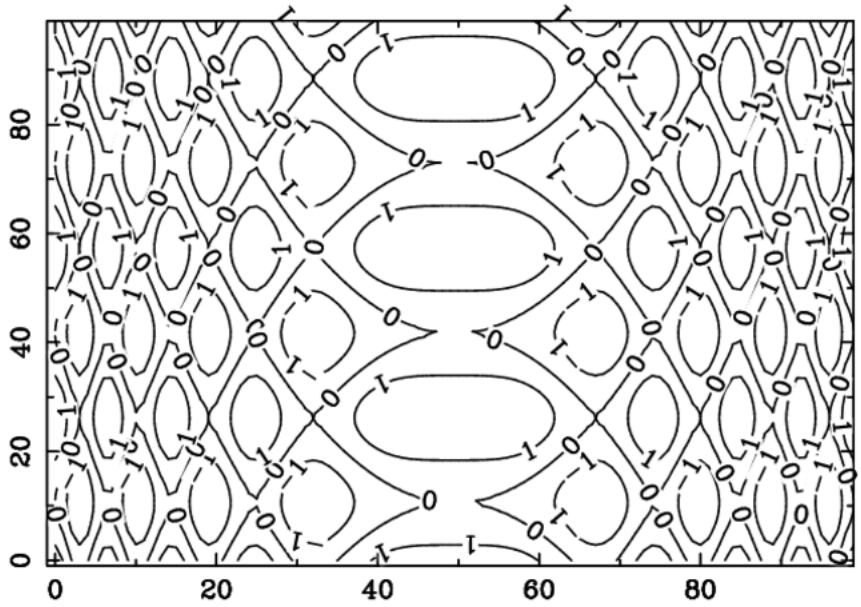
```
$a = sequence(100,100); cont $a;
```



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']};
```



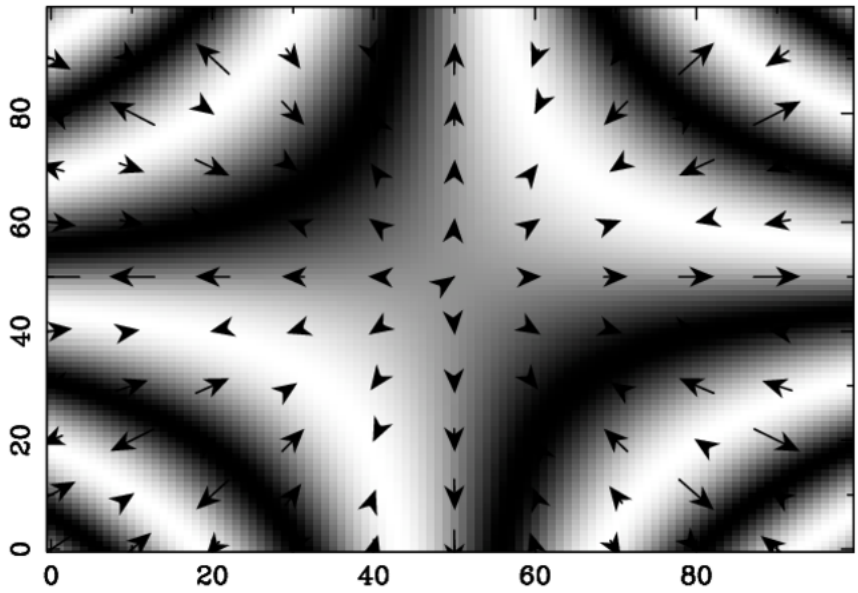
In addition it is possible to colour the labels differently from the contour lines (`LabelColour`), to specify the number of contours instead of their values (`NContours`) and to draw negative contours as dashed lines and positive as solid lines by setting the option `Follow` to a value *>0*.

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):

```
pdl> ctab('Fire');
pdl> imag $z; hold;
pdl> 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 horizontal 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 x 256* arrows on top of a similarly sized image will quickly be unreadable. The result of using this technique is shown below together with the code that produced the plot.




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

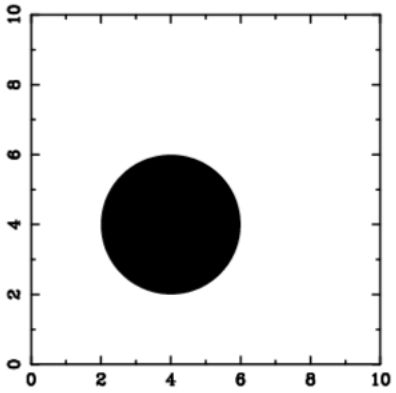
Drawing simple shapes

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 we 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 *Setting up the plot area*. The user specifies the radius and the x and y position of the center:

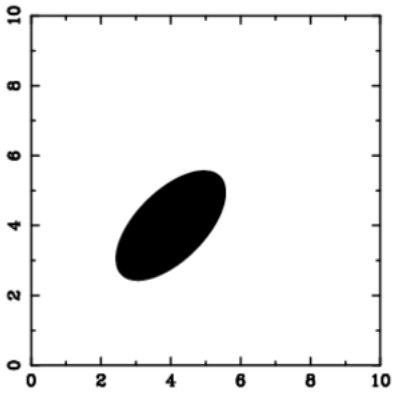
```
pdl> dev '/xs', {Aspect => 1, WindowWidth => 5}
pdl> env 0, 10, 0, 10
pdl> $radius=2; ($x, $y) = (4, 4)
```

```
pdl> circle $x, $y, $radius, {LineWidth => 3}
```



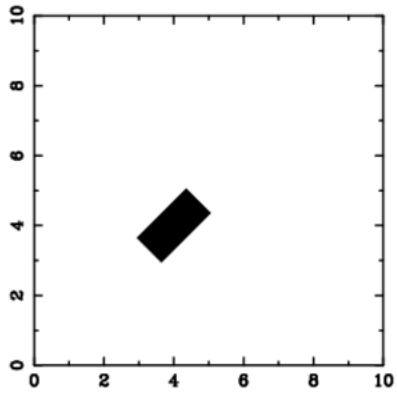
The `ellipse` function is like the `circle` function but it requires the user to specify the minor and major axis and the angle between the major axis and the horizontal. For ease of use it is probably better to specify these as options, but if you remember the order you can also give them directly as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

```
    pdl> dev '/xs', {Aspect => 1, WindowWidth => 5}
    pdl> env 0, 10, 0, 10
    pdl> 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 horizontal. 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.

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



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

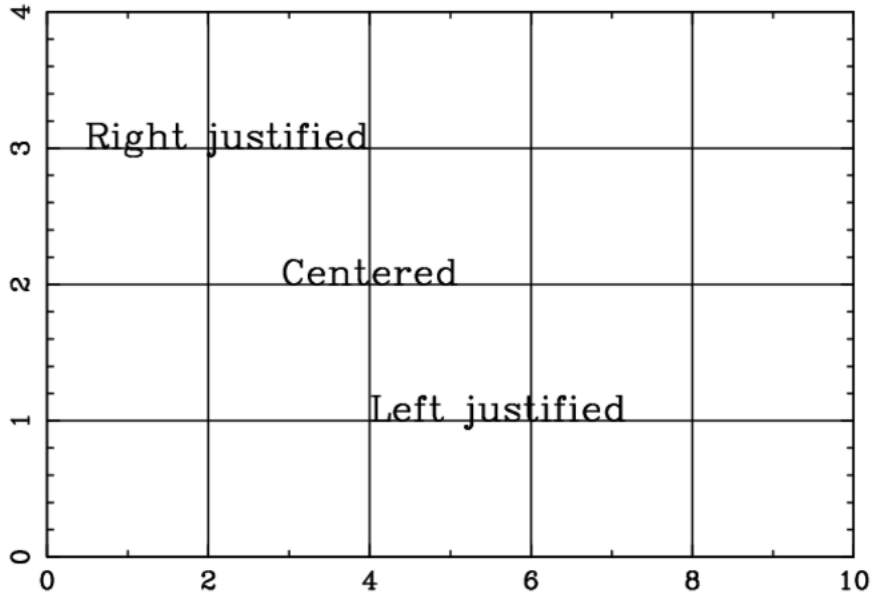
In addition you can set the sides to be similar by setting the `Side` option to the length you require. The lengths are all specified in data-coordinates (which is why you should do a plot or call `env` before using any of these commands).

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

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* and *y* position:

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



Here we 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* and *y* as options (the best advice is to either do all or none).

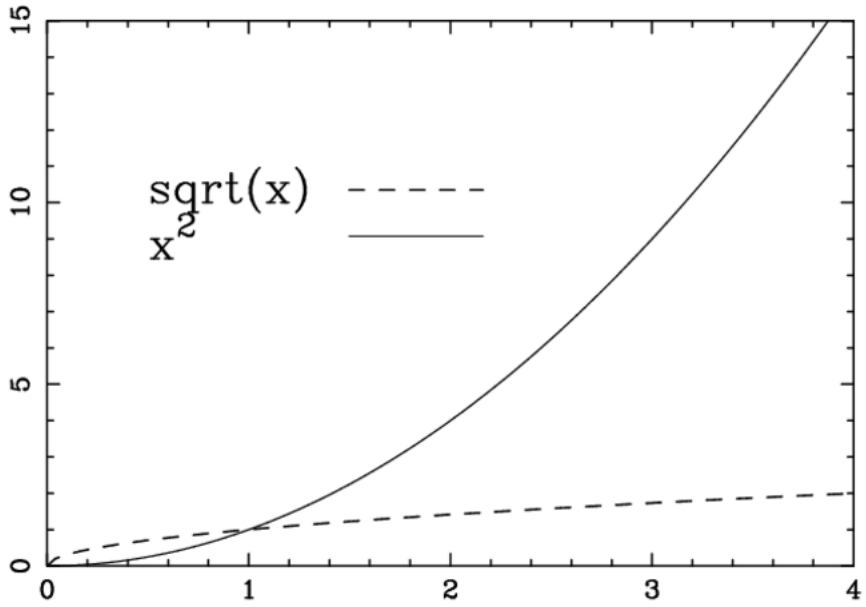
```
pdl> text {XPos => 1, YPos=> 4, Angle => 25, Text => 'Tilted'}
```

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. Consult the PGPLOT documentation for a full list.

Labeling your figures in PGPLOT

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).




```
pdl> $x = sequence(100) / 5; $y1 = sqrt($x); $y2 = $x**2;
pdl> env(0, 4, 0, 15);
pdl> line $x, $y1, {LineStyle => 'Dashed', Colour => 'Red'}
pdl> line $x, $y2, {LineWidth => 3, Colour => 'Blue'}
pdl> legend ['sqrt(x)', 'x \backslash u2'], 0.5, 10,
               {LineStyle => ['Dashed', undef],
               LineWidth => [undef, 3], Colour => ['Red', 'Blue'] }
             # ,Width => 1.0 } makes x**2 legend disappear, why?
```

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.

The figure above is 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 and 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 and y width to the appropriate settings and redoing the plot, possibly using the replay mechanism, see *Recording and playing back plot commands*.

The legend command has several options, the main of which are illustrated above. 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 the legend box. The default value is 0.1.

`VertSpace`

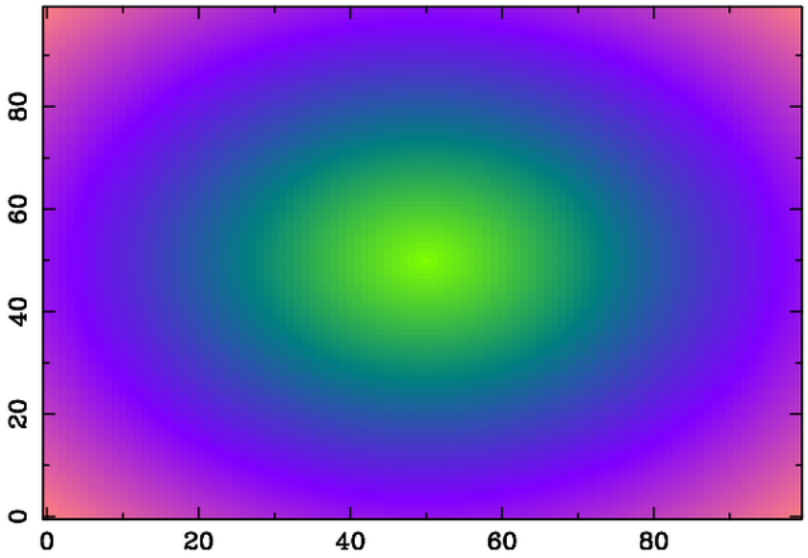
By default the text lines are separated by one character height (in the sense that if the separation were 0 then they would lie on top of each other). The `VertSpace` option allows you to increase (or decrease) this gap in units of the character height; a value of 0.5 would add half a character height 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).

Using colour

PGPLOT has a two disjoint sets of colours. One set determines the colour table used when displaying images and is initialized 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 *Options in plot commands* 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:

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

```
    pdl> use PDL::Graphics::LUT;
    pdl> print "Available tables: ".join(', ', lut_names());
    Available tables: aips0, backgr, bgyrw, blue, blulut, color, green,
    heat, idl11, idl12, idl14, idl15, idl2, idl4, idl5, idl6, isophot,
light,
    manycol, pastel, rainbow, rainbow1, rainbow2, rainbow3,
    rainbow4, ramp, random, random1, random2, random3,
```

```
random4, random5, random6, real, red, smooth, smooth1,
smooth2, smooth3, staircase, stairs8, stairs9, standard
pdl> ctab( lut_data \series default ('rainbow1'));
pdl> imag rvals(100,100);
```

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

All the colour tables with their names overlaid can be generated with this script:

```
use PDL::Graphics::PGPLOT;
use PDL::Graphics::LUT;
dev("/xs",3,15);
foreach(lut_names()){
    print"$_\n";
    ctab(lut_data($_));
    imag sequence(250,1);
    text $_,20,-0.2,{CHARSIZE=>20,LINEWIDTH=>20,COLOUR=>0};
    text $_,20,-0.2,{CHARSIZE=>20,LINEWIDTH=>1,COLOUR=>1};
}
```

And the resultant figure is shown below:

Threading in PDL::Graphics::PGPLOT

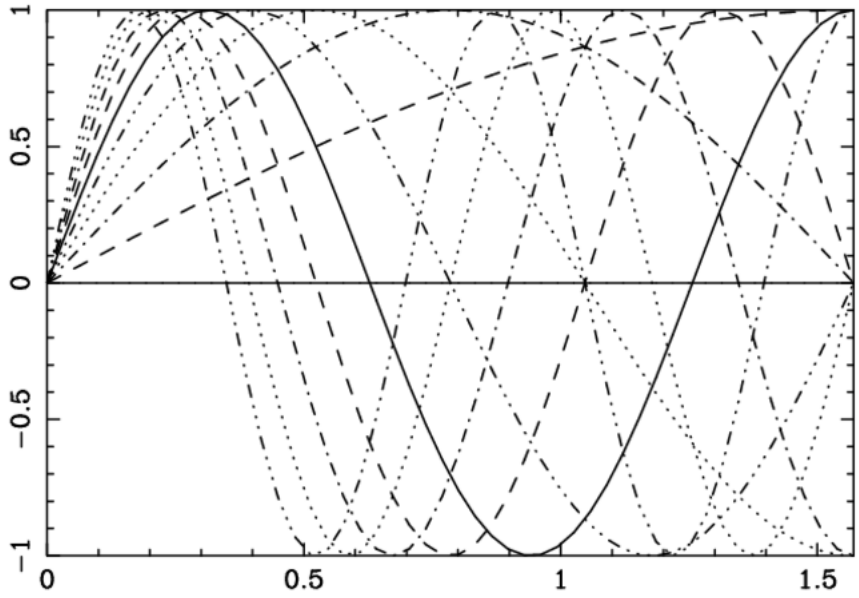
The plot commands do not always lend themselves to easy threading because it can sometimes be difficult to know what the user intends to do when (say) an array of images is passed to the `imag` command. Are they to be displayed in several plot panels, are they to be plotted on top of each other, seamlessly 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 occurrence 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:

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



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:

```
pdl> $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:

```
pdl> 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 initializes 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:

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

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

```
pdl> 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 trying this out.

We are going to use our 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:

```
pdl> $x = sequence(10); $y = $x*$x
pdl> line $x, $y;
pdl> $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:

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

This should give you a part of a parabola from `x=3` to `x=12`, but now defined by the equation `y=pow((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.

In the previous example the limits in the plot window adjusted to the new values for *x* and *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?

```
pdl> $x = sequence(10); $y = $x*$x
pdl> env (0, 9, 0, 81)
pdl> line $x, $y;
```

```
pdl> $s = retrievestate()
pdl> $x += 3; replay $s
```

The result now should be as shown in Figure XXXXXXXX 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

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

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

```
pdl> $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

```
pdl> $x = sequence(10); $y = $x*$x
pdl> line $x, $y; $s=retrievestate()
# Now let us transfer this to a sine plot
pdl> $y -= $y; $y += sin($x)
pdl> replay $s
```

And voila! 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.

The object oriented approach

Assume 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. 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 $integrandwindow = PDL::Graphics::PGPLOT::Window->new($opt);
my $integralwindow = PDL::Graphics::PGPLOT::Window->new($opt);
for (my $i=0; $i<$n; $i++) {
   $integrand = func($x, $i);
   $integrandwindow->points($x, $integrand);
   $sum += $integrand;
}
$integralwindow->points($x, $sum/$n);
```

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 favor 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.

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 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 not 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 PDL::Graphics::PGPLOT 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 *Drawing lines and plotting points* 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 (i.e. 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:

```
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. You are limited by your imagination!

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.

Using PGPLOT commands directly

The Perl module PGPLOT 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 and y coordinates, thus:

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

will plot three points with the x and 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->getdataref, $y->getdataref, 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 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 then let us know! =for comment :!podchecker PLplot.pod && pod2html --infile=PLplot.pod --outfile=PLplot.html open PLplot.html and click 'reload' also conversion for PDF reading is: pod2pdf PLplot.pod --icon-scale 0.25 --title "PDL Book" --icon logo2.png --output-file PLplot.pdf

Graphics with PLplot

The **PDL::Graphics::PLplot** perl module, is an interface to the *http://plplot.sourceforge.net* PLplot C library. It is a 2d plotting library, but also does 1-D bargraphs and 3-D projection graphs.

Many of the examples, discussed below, are from the PDL::Graphics-PLplot/t subdirectory, of the source module. These are written in the functional style, and are direct translations of the examples which come with the **PLplot C library**

The rest of the examples, are object-oreiented, derived from David Merten's slideshow on **PDL::Graphics::PLplot**. His very informative slideshow can be downloaded or viewed at *http://www.slideshare.net/dcmertens/p-lplot-talk*

Introducing PDL::Graphics::PLplot

The basic methods available:

```
new, close ->  create and finalize plot objects
xyplot, stripplots ->   2D plotting
shadeplot ->   `topographical' 3D data representation
histogram ->   plot distribution of 1D data
bargraph ->    plot distribution of categorical data
text ->   annotate plots
setparm ->  set various plotting parameters
```

Once you specify a plotting option, the option will carry over to future calls on the same PLplot object.

The first thing you will notice about invoking PLplot, is that it will prompt you for an output device, or maybe a file to save to, if you do not specify one either in **perldl** or a script

```
Plotting Options:
 < 1> xwin        X-Window (Xlib)
 < 2> tk          Tcl/TK Window
 < 3> ps          PostScript File (monochrome)
 < 4> psc         PostScript File (color)
 < 5> xfig        Fig file
 < 6> null        Null device
 < 7> tkwin       New tk driver
 < 8> mem         User-supplied memory device
 < 9> wxwidgets   wxWidgets Driver
 <10> svg         Scalable Vector Graphics (SVG 1.1)
 <11> bmpqt       Qt Windows bitmap driver
 <12> jpgqt       Qt jpg driver
 <13> pngqt       Qt png driver
 <14> ppmqt       Qt ppm driver
 <15> tiffqt      Qt tiff driver
 <16> svgqt       Qt SVG driver
 <17> qtwidget    Qt Widget
 <18> epsqt       Qt EPS driver
 <19> pdfqt       Qt PDF driver
 <20> extqt       External Qt driver
 <21> memqt       Memory Qt driver
 <22> xcairo      Cairo X Windows Driver
 <23> pdfcairo    Cairo PDF Driver
 <24> pscairo     Cairo PS Driver
 <25> svgcairo    Cairo SVG Driver
 <26> pngcairo    Cairo PNG Driver
 <27> memcairo    Cairo Memory Driver
 <28> extcairo    Cairo External Context Driver

Enter device number or keyword:
```

You can specify a device

```
   To specify the output device:

      pdl> dev('/xwin')

   or in a script

      use PDL::Graphics::PLplot;
      # display the image in the xwindow
      my $pl = PDL::Graphics::PLplot->new(
                   DEV => 'xwin'
                   );
```

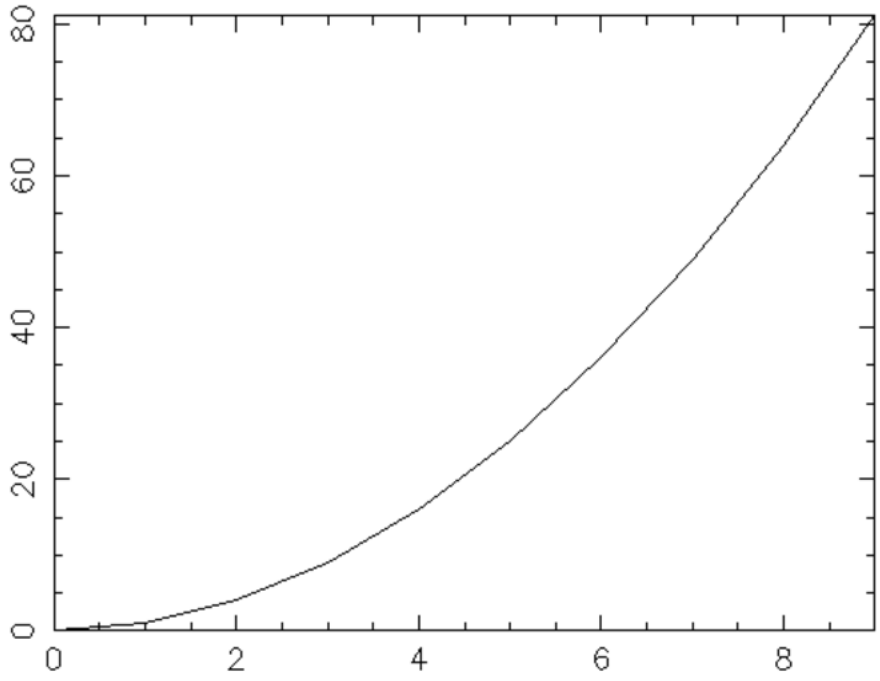
Plotting a simple parabola

This code:

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

```
use strict;
use PDL;
use PDL::Graphics::PLplot;
       my $pl = PDL::Graphics::PLplot->new( DEV => "png", FILE => "$0.png"
);
my $x  = sequence( 10 );
my $y  = $x**2;
$pl->xyplot( $x, $y );
$pl->close;
```

produces a nice parabola, in a PNG file.



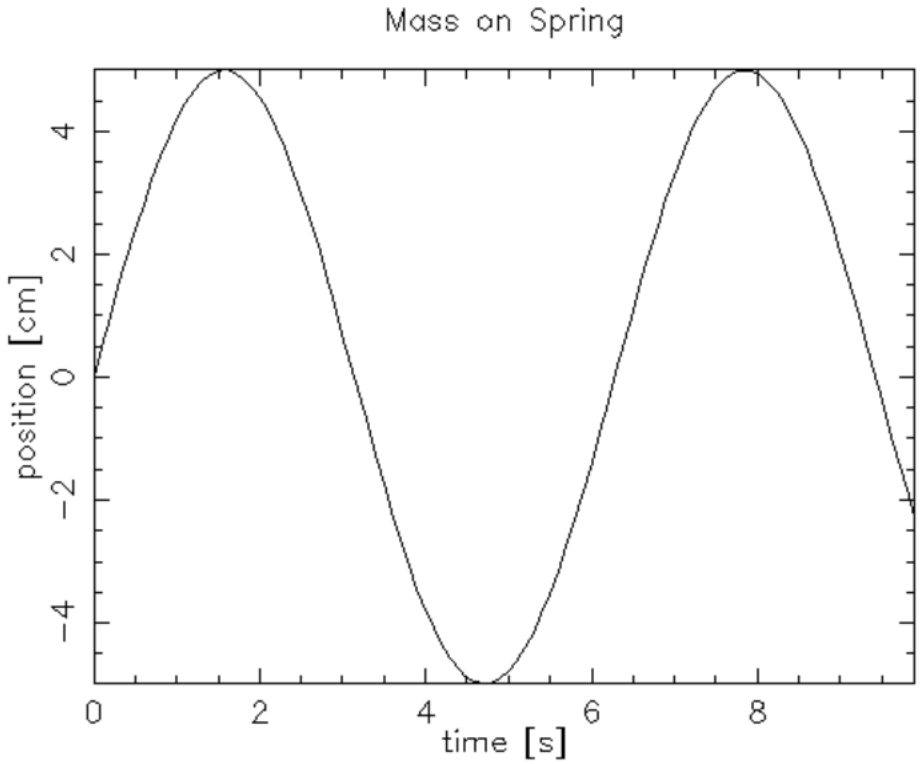
Object Oriented Examples

This section shows how to use the object oriented methods of the **Perl** interface to **PDL::Graphics::PLplot**

Axis labelling and titles

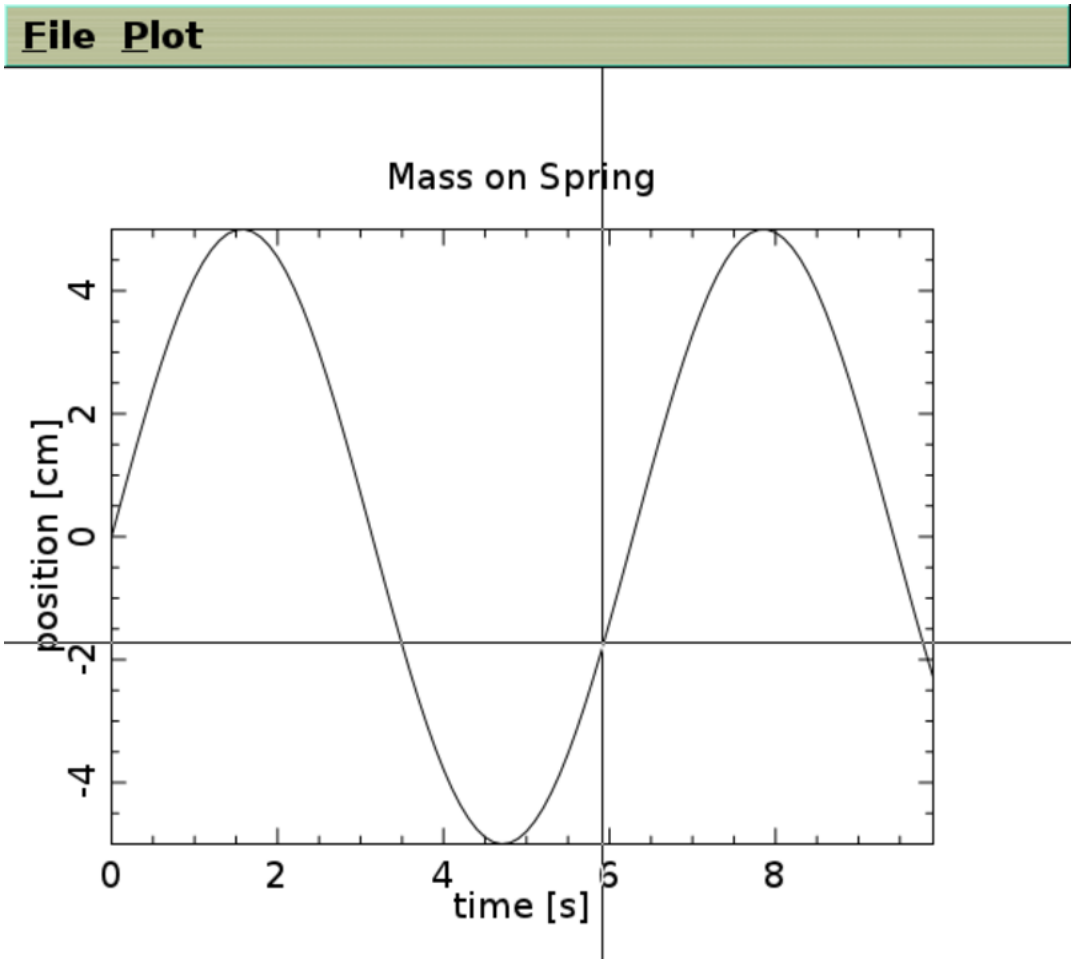
```
use PDL;
use PDL::Graphics::PLplot;

# Generate a time series
my $time = sequence(100)/10;
my $sinewave = 5 * sin($time);
# Create the PLplot object: use xwin for display
my $pl = PDL::Graphics::PLplot->new( DEV => "xwin");
```

```
# Plot the time series
$pl->xyplot($time, $sinewave
     , XLAB => 'time [s]'
     , YLAB => 'position [cm]'
     , TITLE => 'Mass on Spring'
     );
# Close the PLplot object to finalize
$pl->close;
```



Interactive crosshairs with the wxwidgets output device

```
# Create the PLplot object: use wxwidgets for display
# wxwidgets allows saving to many file types, and
# has a Locate function under the Plot menu entry
# providing interactive crosshairs to read individual plot values

my $pl = PDL::Graphics::PLplot->new( DEV => "wxwidgets");
```



setting the DEV and FILE options, and using the aliased option for new()

```
  There are 2 ways to call new(), and the aliased module makes the syntax a
 bit easier.

  The conventional way:
      use PDL::Graphics::PLplot;
      my $pl = PDL::Graphics::PLplot->new( DEV => "xwin");

  The aliased way:
      use aliased 'PDL::Graphics::PLplot';
      my $pl = PLplot->new( DEV => "xwin" );

  Specify the DEV in your call to new.
      For output to a window:
         -- option DEV must be set to xwin, wxwidgets, or similar
  For output to a file:
         -- option DEV must be set to xfig, svg, pscairo, or similar
         -- option FILE must give the output file's name
  For output to a memory buffer:
         -- option DEV must be set to mem or memcairo
         --option MEM must be passed a piddle where the results will be
plot
```

Outputting postscript

```
# Save the image to a postscript file
my $pl = PDL::Graphics::PLplot->new(
             DEV => 'ps'
           , FILE => 'myfile.eps'
       );
```

Tools for plotting points

You can plot lines, symbols, or both by using the PLOTTYPE option. You specify error bars in x and y by passing a scalar or a piddle with those errors to XERRORBAR and YERRORBAR.

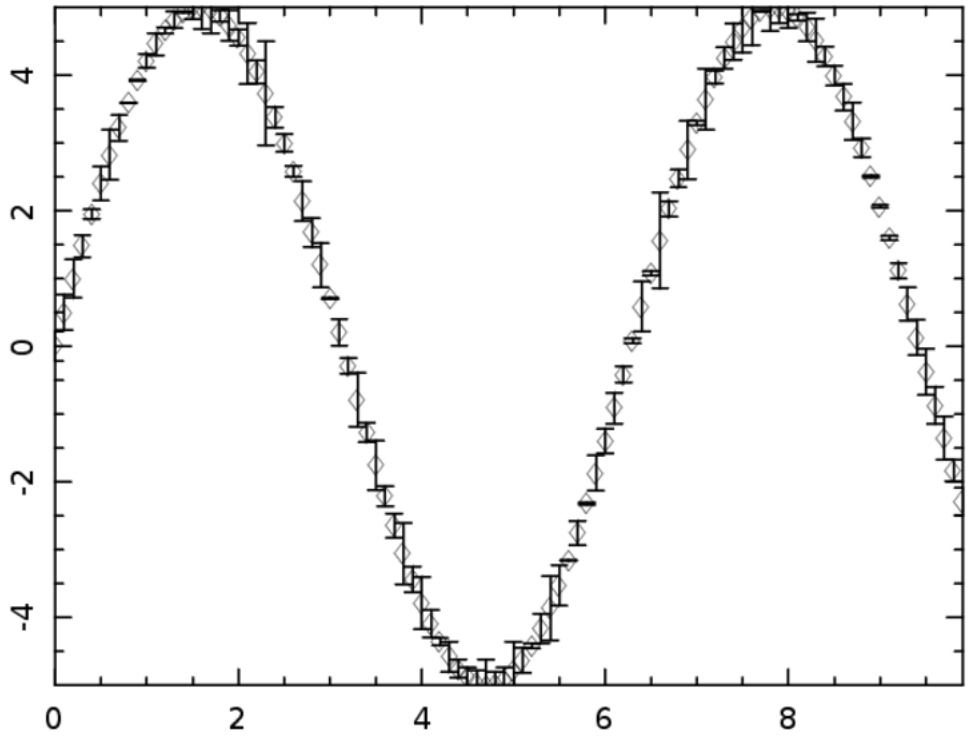
```
-- PLOTTYPE => LINE plots data as lines (default)
-- PLOTTYPE => POINTS plots data as points
-- PLOTTYPE => LINEPOINTS plots data as lines and points
-- PLplot's built in error-bars can plot asymmetric error bars,
   but the high-level PDL bindings do not support this.
```

To set the symbol type and size, use the SYMBOL and SYMBOLSIZE options.

```
--  Symbol sizes are measured as multiples of the default size
-- Symbol sizes can be fractional, such as 0.7 or 4.5
-- Symbols are identified by their number
```

A Symbols example

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use PDL::Graphics::PLplot;

# Generate a time series
my $time = sequence(100)/10;
my $sinewave = 5 * sin($time);

# Save the image to a postscript file

my $pl = PDL::Graphics::PLplot->new(
     DEV => 'pscairo'
   , FILE => 'Symbols.eps'
);

# Plot the time series as points
$pl->xyplot($time, $sinewave
        , PLOTTYPE => 'POINTS'
        , SYMBOL       => 843
        , YERRORBAR => grandom($time)/2
   );

$pl->close;
```



Plotting multiple curves

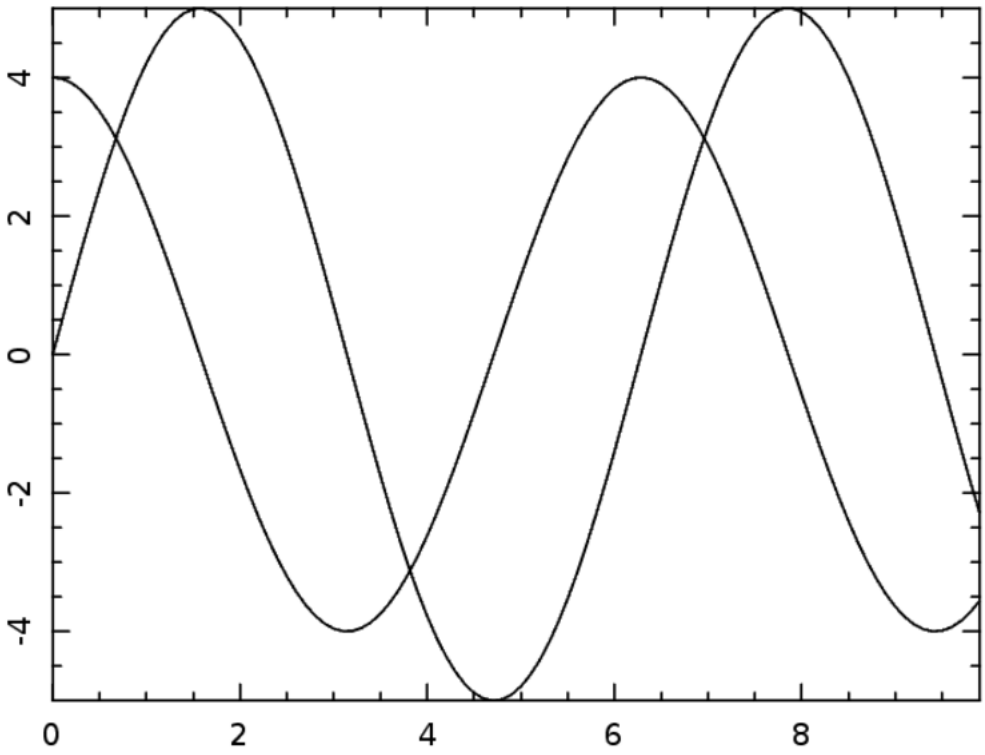
Depending on what you want, there are at least five ways to plot multiple curves.

```
-- plot a multidimensional piddle
-- call xyplot multiple times
-- use stripplots
-- specify SUBPAGES in the constructor
-- create insets using the VIEWPORT option
```

Plotting multiple curves with a multi-dimensional piddle

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

# Generate a time series
my $time = sequence(100)/10;
my $sinewave = 5 * sin($time);
my $cosinewave = 4 * cos($time);
my $toplot = cat($sinewave, $cosinewave);

# Save the image to a postscript file

my $pl = PLplot->new(
```

```
        DEV => 'pscairo',
        FILE => 'Multidimensional.eps'
    );

# Plot the time series
$pl->xyplot($time, $toplot);

$pl->close;
```



Colorizing multiple data sets

Use color to differentiate different data sets:

```
-- For multidimensional piddles, plot as POINTS and use the
   COLORMAP and PALETTE options.
-- For multiple calls to xyplot, use POINTS, COLORMAP, and
   PALETTE, or use COLOR option.
```

The COLORMAP option lets you specify a third value for each (x, y) pair, making it (x, y, colorval).

Which color is associated with the minimum colorval? Which color is associated with the maximum value? All of these are set with the PALETTE.

Valid PALETTEs include:

```
RAINBOW - from red to violet through the spectrum
REVERSERAINBOW - violet through red
```

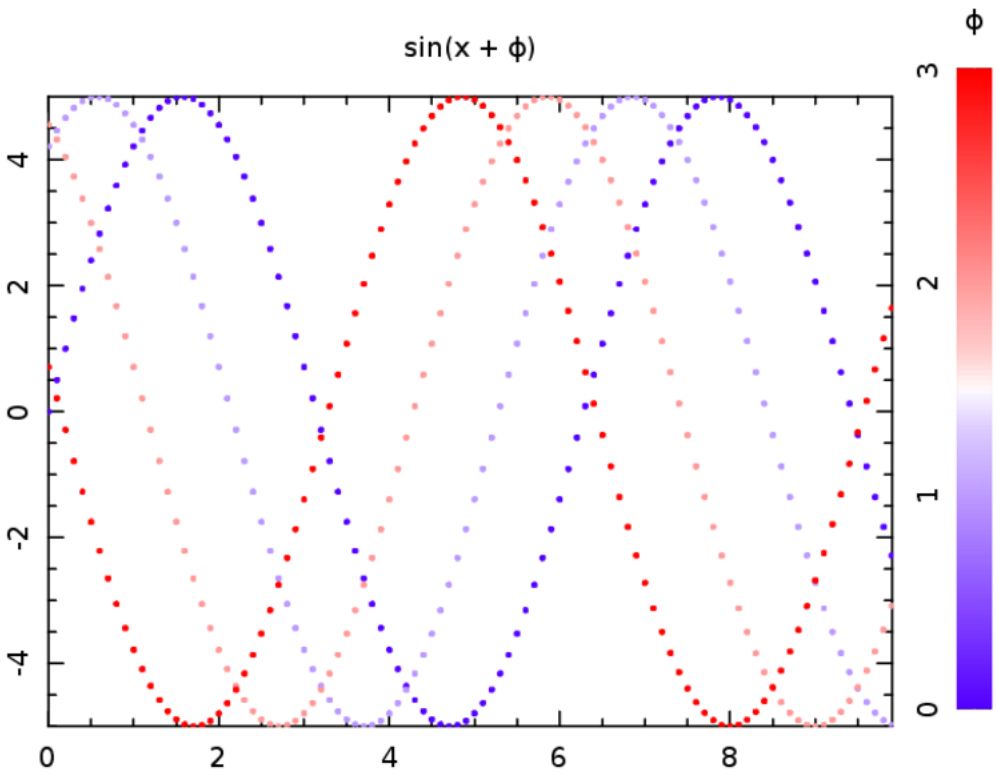
```
GREYSCALE - from black to white via grey
REVERSEGREYSCALE - from white to black via grey
GREENRED - from green to red
REDGREEN - from red to green
```

Note:

```
 --  the default palette is not named
 --  this only works when plotting points, not lines or error bars
```

A multi-colored multi-curve plot

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
  DEV => 'pscairo',
  FILE => 'Multidimensional2.eps');

# Generate a time series and phase offset
my $time = sequence(100)/10;
my $phi = zeroes(4)->xlinvals(0, 3)->transpose;

my $sinewaves = 5*sin($time + $phi);
# Plot the time series and phi color key
$pl->xyplot($time, $sinewaves,
  PLOTTYPE => 'POINTS',
  COLORMAP => $phi,
  TITLE => 'sin(x + #gf)');

$pl->colorkey($phi, 'v',
   TITLE => '#gf',
   VIEWPORT => [0.93, 0.96, 0.15, 0.85]);
$pl->close;
```



Plotting multiple curves with differently colored calls to xyplot

An alternative to plotting a multi-dimensional piddle, you can plot multiple curves by multiple calls to xyplot, specifying a different color for each plot.

Legal colors are:

```
BLACK        GREEN          WHEAT
BLUE         RED            AQUAMARINE
GREY         BLUEVIOLET     YELLOW
PINK         BROWN          CYAN
TURQUOISE    MAGENTA        SALMON
WHITE        ROYALBLUE      DEEPSKYBLUE
VIOLET       STEELBLUE1     DEEPPINK
MAGENTA      DARKORCHID1    PALEVIOLETRED2
TURQUOISE1   LIGHTSEAGREEN  SKYBLUE
FORESTGREEN  CHARTREUSE3    GOLD2
SIENNA1      CORAL          HOTPINK
LIGHTCORAL   LIGHTPINK1     LIGHTGOLDENROD
```

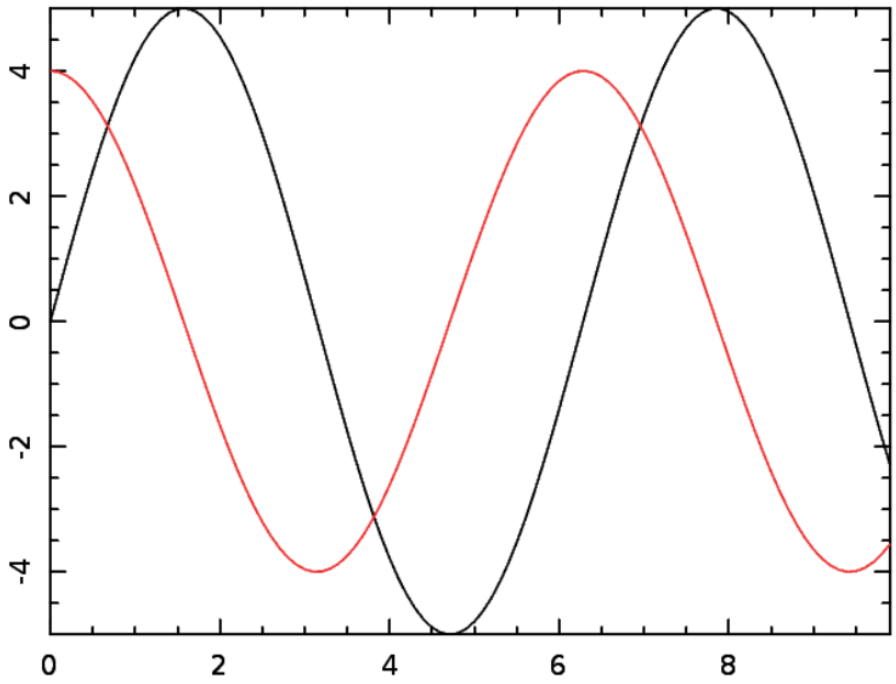
Notes:

```
   --  Curve clipping - the first plot sets the plotting boundaries
        and later plots fall outside of those boundaries
   --  Changing `current' color - the first plot sets the `current'
        color and the second does not specify a color
   --  PLplots has a discrete color limit of 16, including foreground and
background color.
```

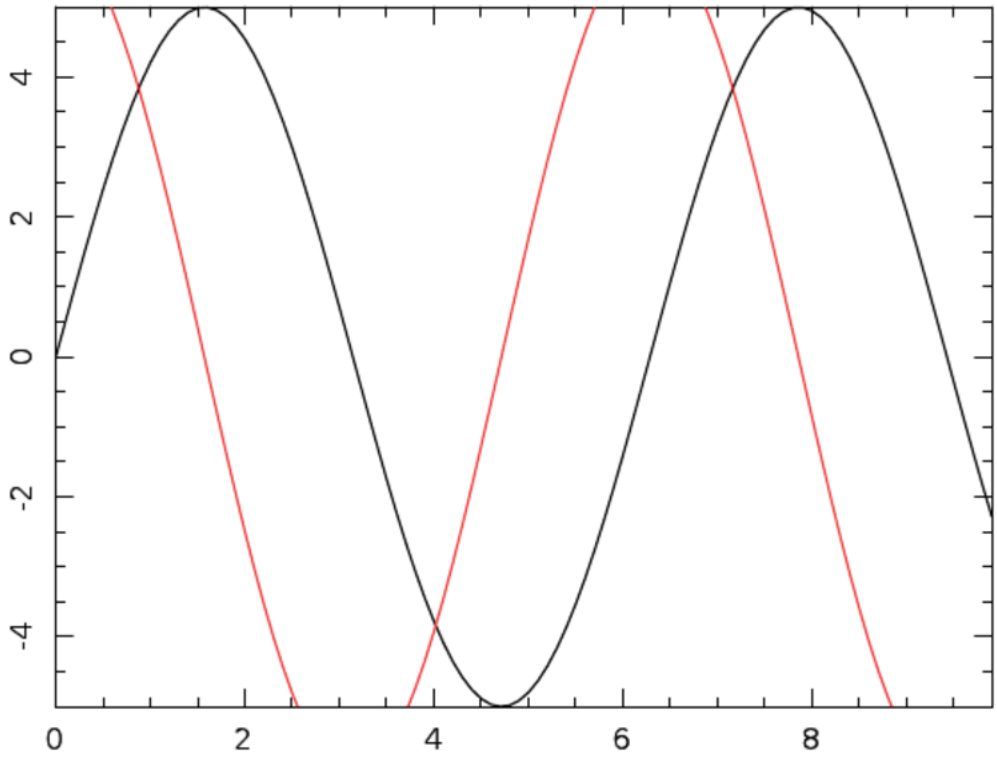
When plotting multiple curves, the first plot sets the boundaries, and can result in subsequent plots being clipped. The obvious solution, is to plot your curve with largest values first. To force a separate color from the first set default color, always specify the colors in xyplot.

A multiple curve with xyplot

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

# Generate a time series
my $time = sequence(100)/10;
my $sinewave = 5 * sin($time);
my $cosinewave = 4 * cos($time);

# Save the image to a postscript file
my $pl = PLplot->new(
      DEV => 'pscairo',
      FILE => 'Multiple curves.eps'
 );

# Plot the sine in black, cosine in red
$pl->xyplot($time, $sinewave);
$pl->xyplot($time, $cosinewave , COLOR => 'RED');

$pl->close;
```



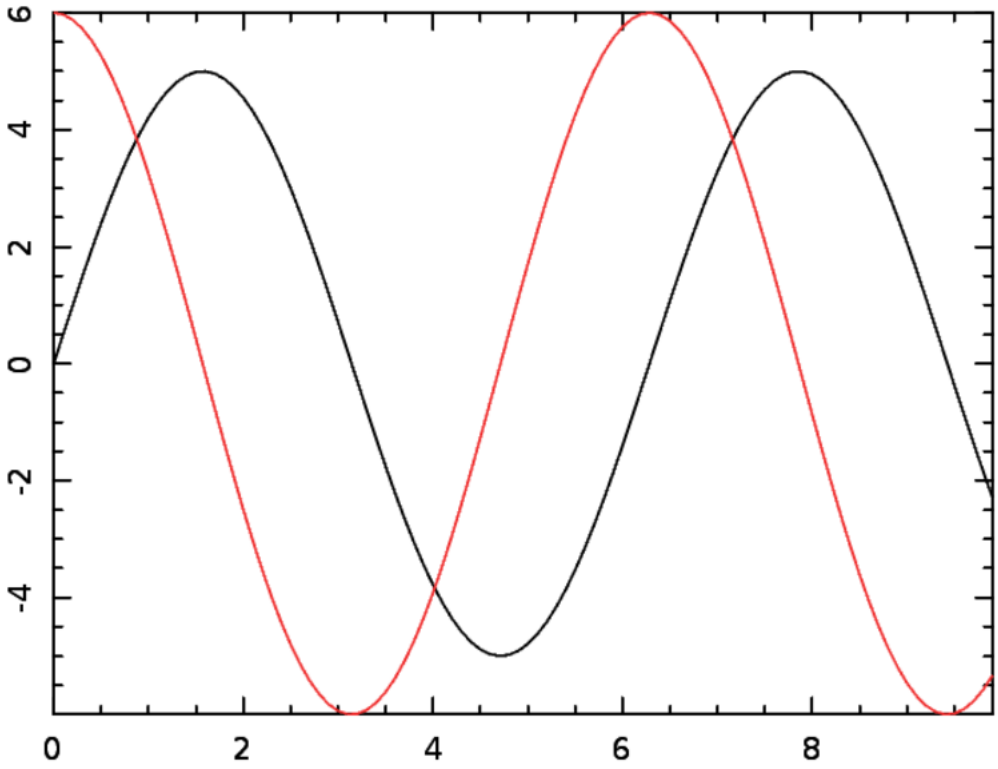
Solving curve clipping on multiple xyplots with the BOX option



When you have multiple xyplots, with widely separated values, you can use the `xyplot BOX` option to prevent clipping.

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

# Generate a time series
my $time = sequence(100)/10;
my $sinewave = 5 * sin($time);
my $cosinewave = 6 * cos($time);

# Save the image to a postscript file
my $pl = PLplot->new(
    DEV => 'pscairo',
   FILE => 'Multiple curves3.eps'
);

# Plot the sine with full bounds
$pl->xyplot($time, $sinewave,
    BOX => [$time->minmax, $cosinewave->minmax]);
```

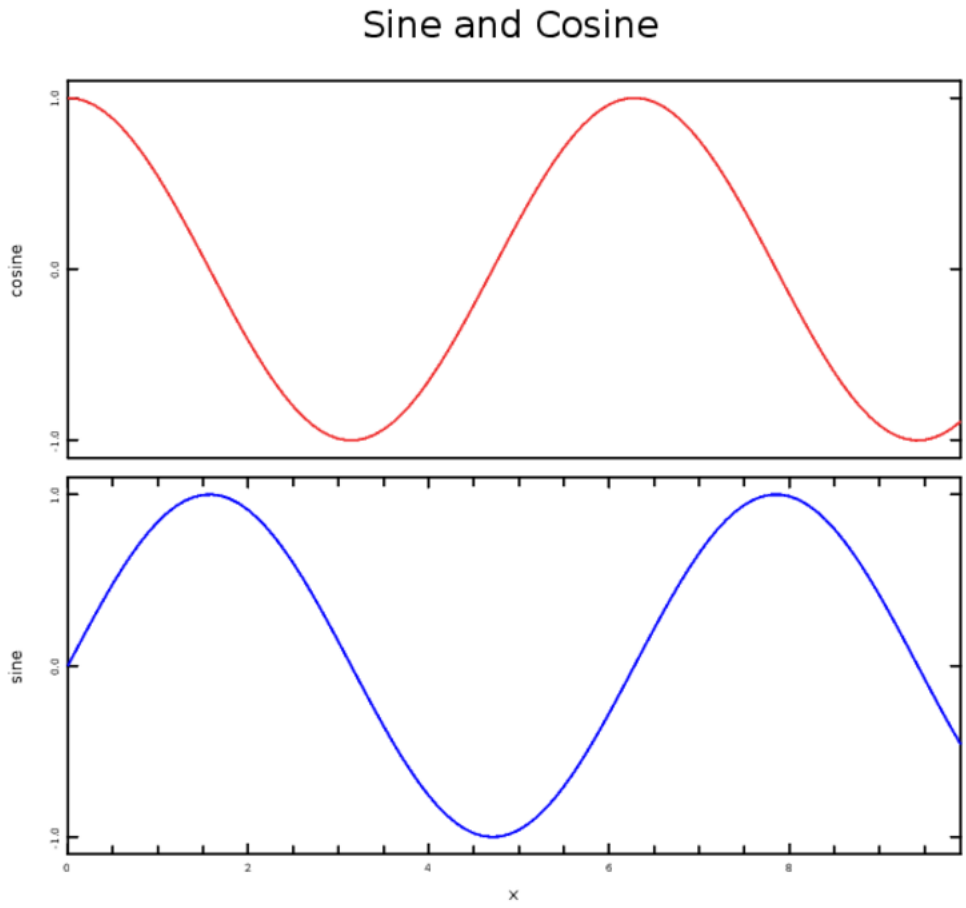
```
# Plot the cosine in red
$pl->xyplot($time, $cosinewave , COLOR => 'RED');

$pl->close;
```



Plotting multiple curves with stripplot

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

# Save the image to a postscript file
my $pl = PLplot->new(
    DEV => 'pscairo',
    FILE => 'stripplots.eps'
 );

# Generate a time series
my $time = sequence(100)/10;

# Make stripplots with the
#      different time series
$pl->stripplots($time,
     [sin($time), cos($time)],
     XLAB => 'x',
```

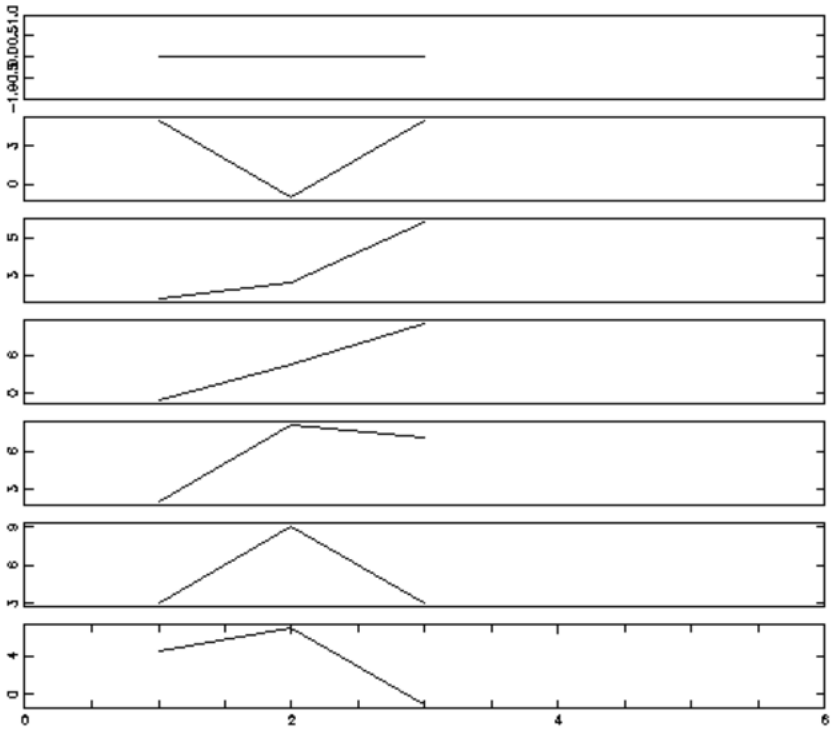
```
        YLAB => ['sine', 'cosine'],
        COLOR => ['BLUE', 'RED'],
        TITLE => 'Sine and Cosine'
    );

$pl->close;
```



Stripplots and reading DATA with rcols

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use PDL::Graphics::PLplot;

my ($t, $data) = rcols(*DATA, 0, []);

my $pl = PDL::Graphics::PLplot->new( DEV => "xwin" );

# Make stripplots with the different time series
# notice data must be transposed
$pl->stripplots($t, $data->transpose);
$pl->close;
```

```
__DATA__
#    t       x1      x2      x3
     1       4       6      -1
     2       3       9       3
     3       2       8       7
     3      -1       4      10
     5       1       2       6
     6       5      -1       5
```



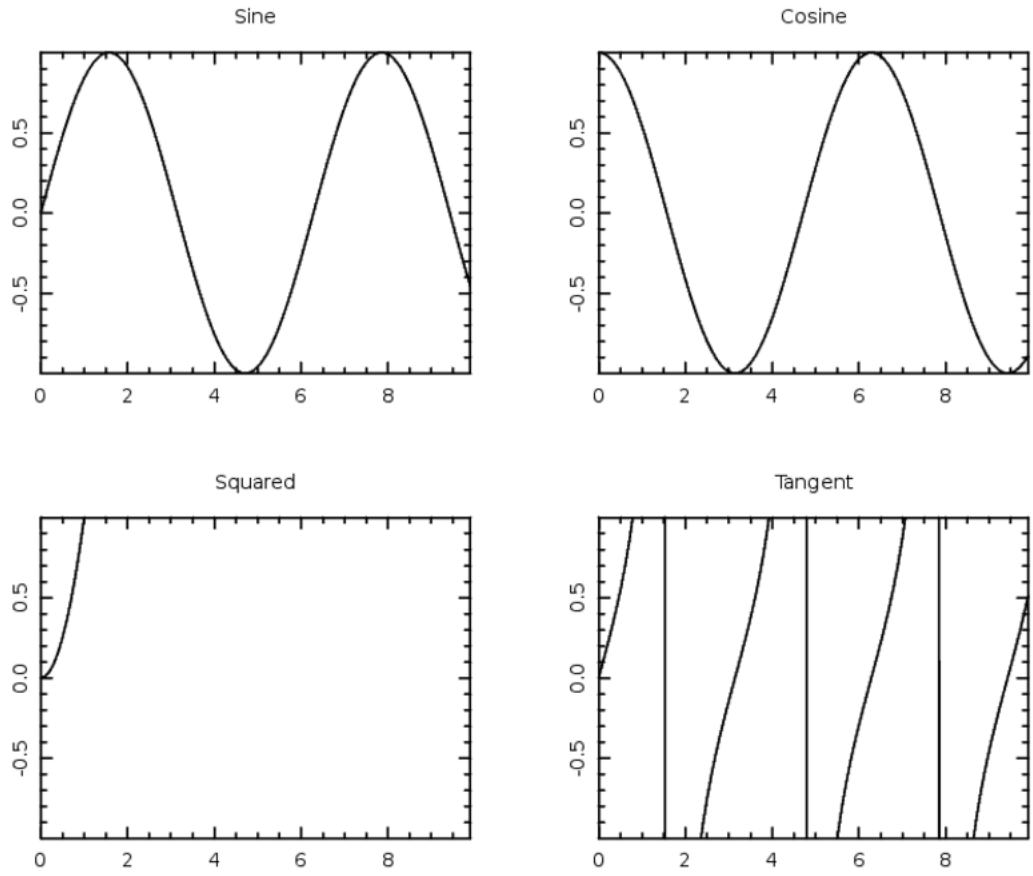
Multiple plots with SUBPAGE

When you create your PLplot object, you can carve the canvas into immutable subpages. my $pl = PDL::Graphics::PLplot->new(# ... , SUBPAGES => [$nx, $ny]);

For example:

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

# Generate a time series
my $time = sequence(100)/10;

# Save the image to a postscript file
```

```
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'subpages.eps',
     SUBPAGES => [2,2]);

# Plot the time series
$pl->xyplot($time, sin($time), TITLE => 'Sine');

$pl->xyplot($time, cos($time), TITLE => 'Cosine',
          SUBPAGE => 0);

$pl->xyplot($time, tan($time), TITLE => 'Tangent',
          SUBPAGE => 4);

$pl->xyplot($time, $time**2, TITLE => 'Squared',
          SUBPAGE => 3);

$pl->close;
```

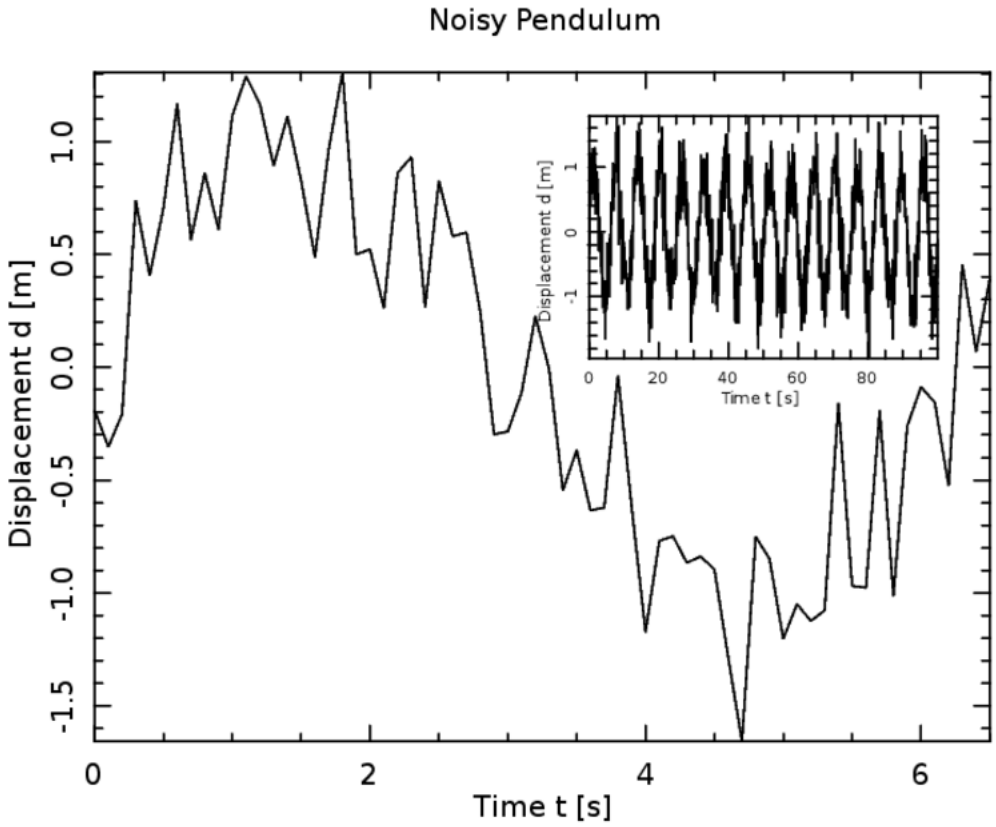


Boxes and Viewports

Using Insets

ometimes you want a small inset in one of the corners of your plot. If you ant to do this you should:

```
  --  Specify the VIEWPORT
  --  Specify the BOX
  --  Use a smaller CHARSIZE
  --  If the underlying plot has a title, you should probably undefine it
  --  Undefine or change the XLAB and YLAB unless you want to use the
        values from the underlying plot

  #!/usr/bin/perl
  use strict;
  use warnings;
  use PDL::Graphics::PLplot;
  use PDL;
  use PDL::NiceSlice;

  # Generate a noisy time series
  my $time = sequence(1000) /10;
  my $sinewave = 1 * sin($time) + grandom($time) / 3;

  # Save the image to a postscript file
  my $pl = PDL::Graphics::PLplot->new( DEV => 'pscairo', FILE =>
'inset.eps');

  # Plot subset as the main plot
  $pl->xyplot($time(0:65), $sinewave(0:65), TITLE => 'Noisy Pendulum',
          YLAB => 'Displacement d [m]', XLAB => 'Time t [s]');

  # Plot full data set as inset
  $pl->xyplot($time, $sinewave,
     TITLE       => undef,
     VIEWPORT => [0.525, 0.825, 0.525, 0.775],
     BOX         => [$time->minmax, $sinewave->minmax],
     CHARSIZE => 0.6
    );
  $pl->close;
```



Basics of viewports

PLplot has three distinct measurements for your plot at any point:

```
-- the plotting surface's dimensions
-- the viewport's relative extent
-- the `natural' coordinates within the viewport
```

Surface dimensions

The dimensions of the canvas or surface that you are using can be specified in the constructor (and cannot be changed later):

```
my $pl = PDL::Graphics::PLplot->new(
        # other options...
        PAGESIZE => [$width, $height]
       # other options...
   );
```

These are measured either in pixels or milimeters depending on whether the underlying format is a raster or vector format.

Viewport positioning

The viewport carves out a chunk of space on the canvas for plotting and can be changed with each plotting function.

```
$pl->xyplot($x, $y
```

```
          # other options
          , VIEWPORT => [$xmin, $xmax, $ymin, $ymax]
          # other options
     );
# Plot on right half of the page
VIEWPORT => [0.5, 1, 0, 1]
# Plot in upper half of the page
VIEWPORT => [0, 1, 0, 0.5]
# Vertically centered, horizontally offset
VIEWPORT => [0.5, 0.7, 0.4, 0.6]
```

Viewport values are fractions of the full page (or sub-page) width all four values should be a number between 0 and 1.

The clipping box

If the viewport indicates the chunk of space you will be graphing on, the clipping box indicates the coordinates within that chunk of space.

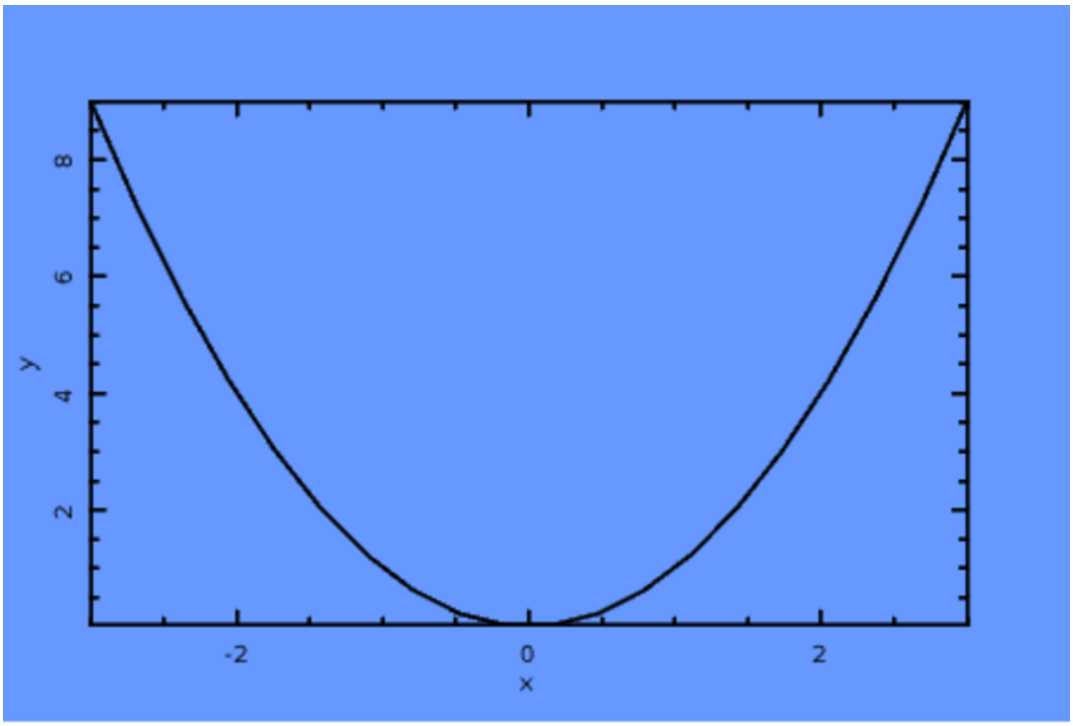
```
$pl->xyplot($x, $y
          # other options...
          , BOX => [$xmin, $xmax, $ymin, $ymax]
          # other options...
     );

 # x runs from 0 to 10, y from -8 to 8:
 BOX => [0, 10, -8, 8]
 # piddles have the minmax method:
 BOX => [$x pdl->minmax, $y pdl->minmax]
```

When plotting using the specified box, a data point near (0, -8) will be plotted near the lower left corner and a data point near (5, 0) will be plotted at the center. Viewports define where plots are drawn. Tick labels, axis labels, and plot titles are drawn outside the viewport.

Page size

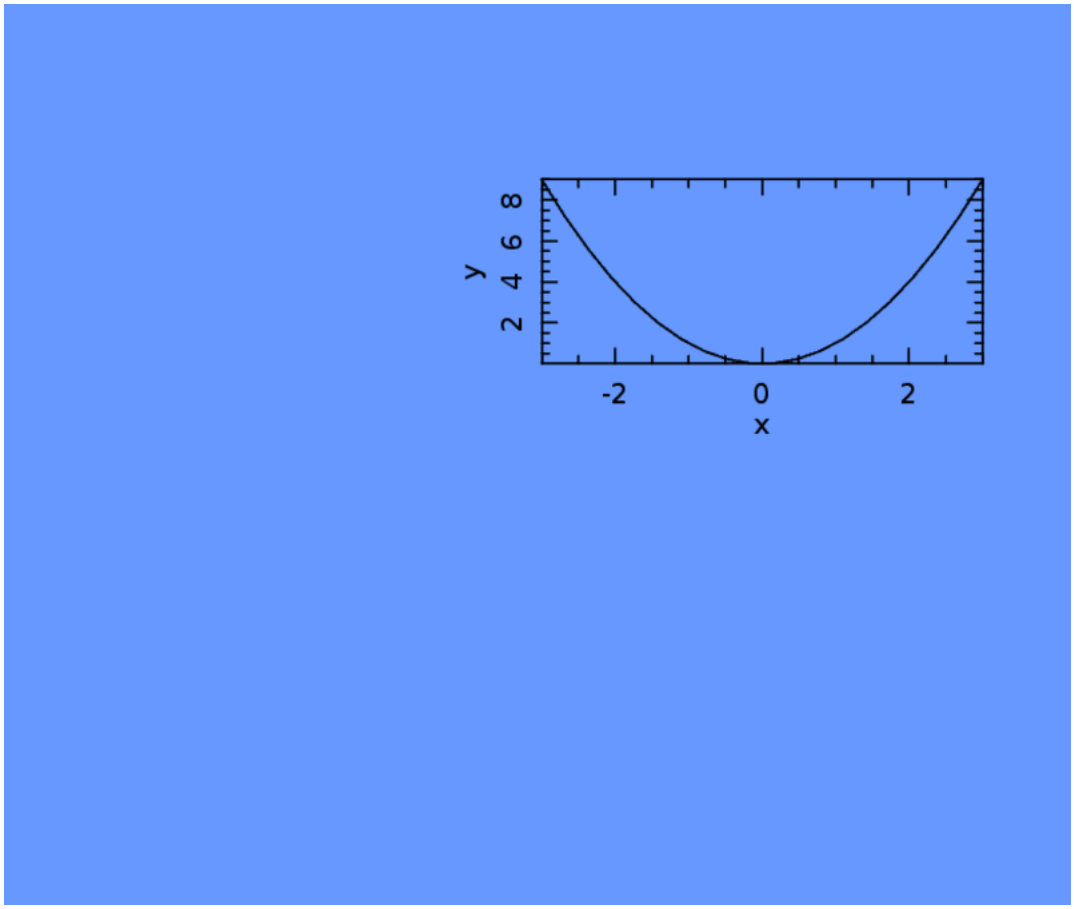
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $y = $x**2;

# Set a custom page size
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 2.eps',
     BACKGROUND => 'SKYBLUE',
     PAGESIZE => [360, 240]
  );

# Plot a quadratic function:
$pl->xyplot($x, $y, YLAB => 'y', XLAB => 'x');
```

```
$pl->close
```



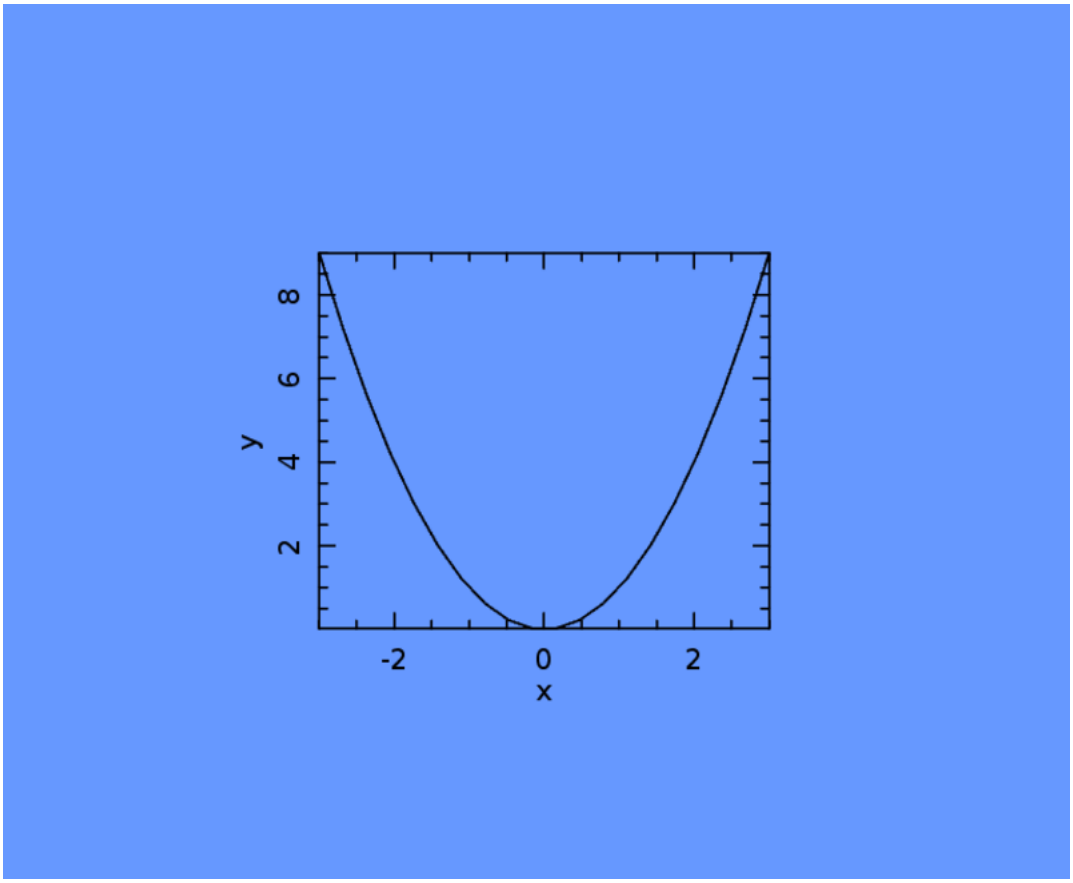
Viewport upper right

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $y = $x**2;

my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 3.eps',
     BACKGROUND => 'SKYBLUE'
);

# Put the plot in the upper right:
$pl->xyplot($x, $y,
     YLAB => 'y',
     XLAB => 'x',
     VIEWPORT => [0.5, 0.9, 0.6, 0.8]
     );

$pl->close;
```



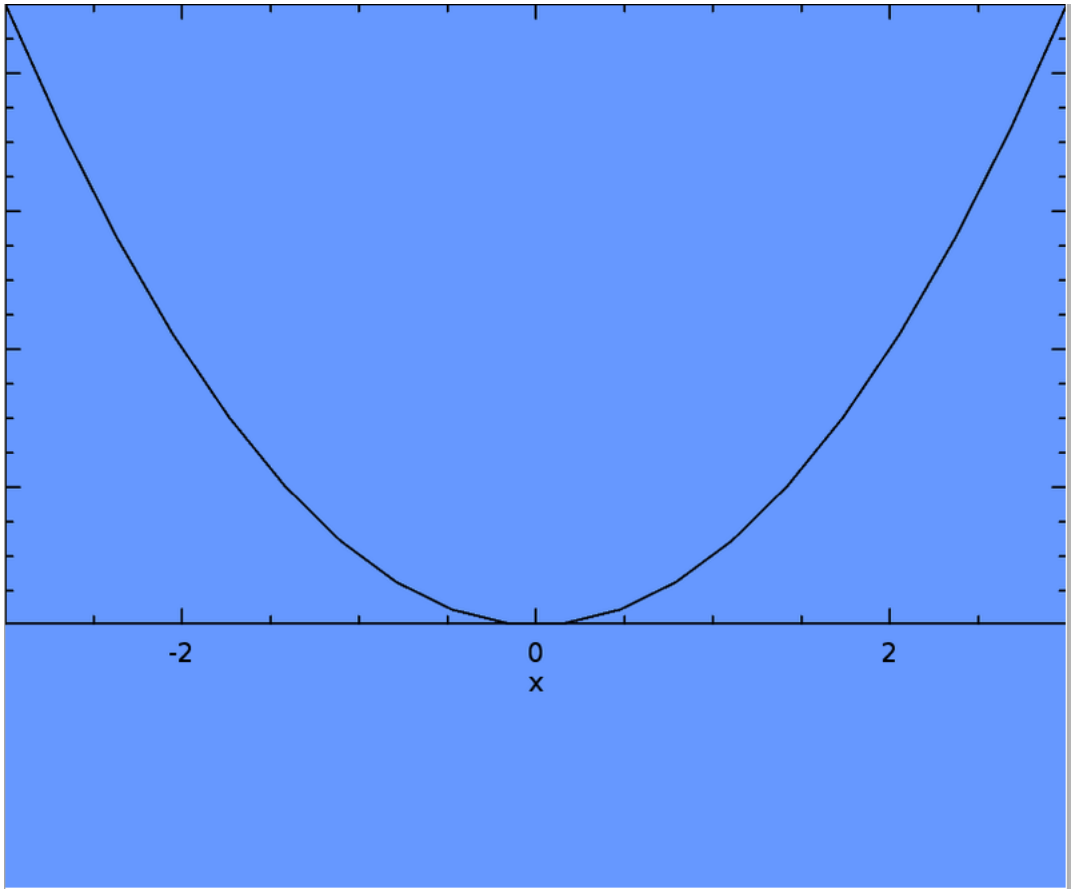
Viewport centered

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $y = $x**2;
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 4.eps',
     BACKGROUND => 'SKYBLUE'
    );

# Center the plot
$pl->xyplot($x, $y,
    YLAB => 'y', XLAB => 'x',
    VIEWPORT => [0.3, 0.7, 0.3, 0.7]
    );

$pl->close;
```



Viewport extreme bounds

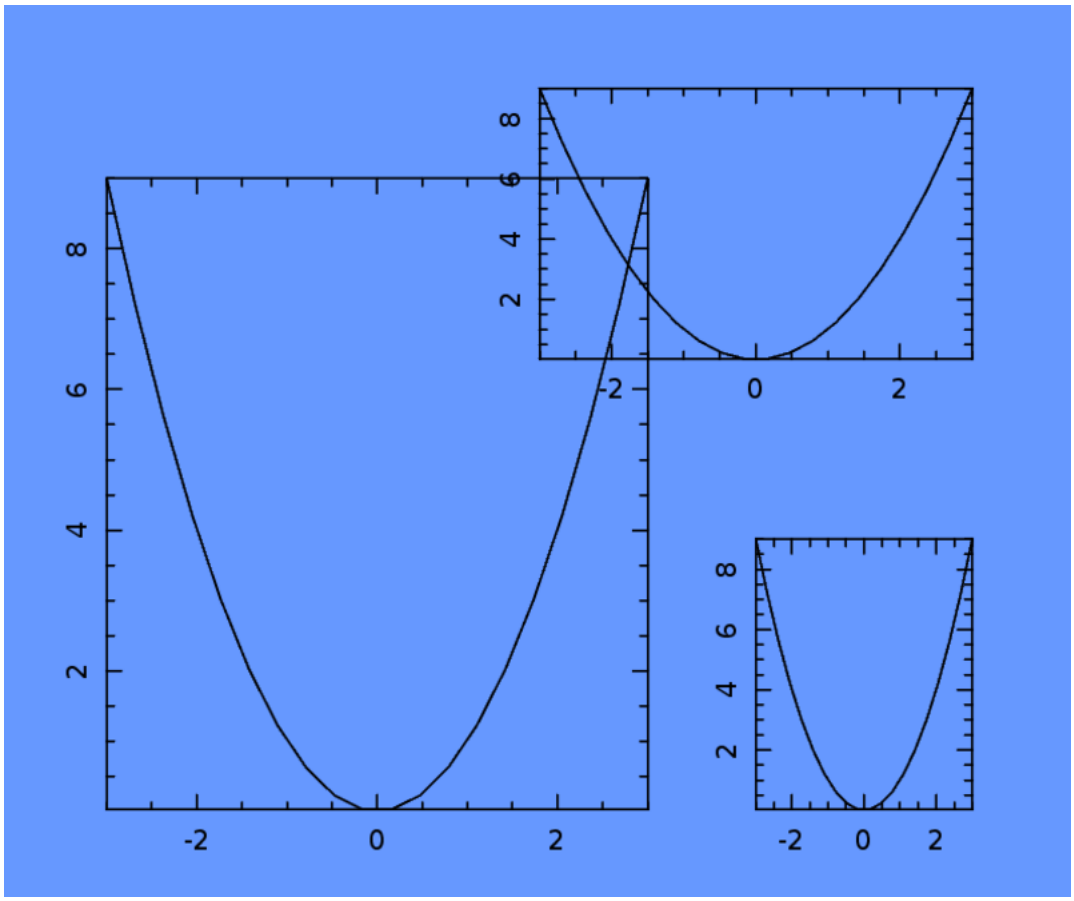
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $y = $x**2;
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 5.eps',
     BACKGROUND => 'SKYBLUE'
    );

# Try extreme bounds for the viewport
$pl->xyplot($x, $y ,
     YLAB => 'y',
     XLAB => 'x',
     VIEWPORT  => [0, 1, 0.3, 1]
     );

$pl->close;
```



Viewport multiple plots

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $y = $x**2;
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 6.eps',
     BACKGROUND => 'SKYBLUE');

# Big plot on left
$pl->xyplot($x, $y, VIEWPORT
     => [0.1, 0.6, 0.1, 0.8]);

# Medium plot on upper right
$pl->xyplot($x, $y, VIEWPORT
     => [0.5, 0.9, 0.6, 0.9]);

# Small plot on lower right
$pl->xyplot($x, $y, VIEWPORT
```

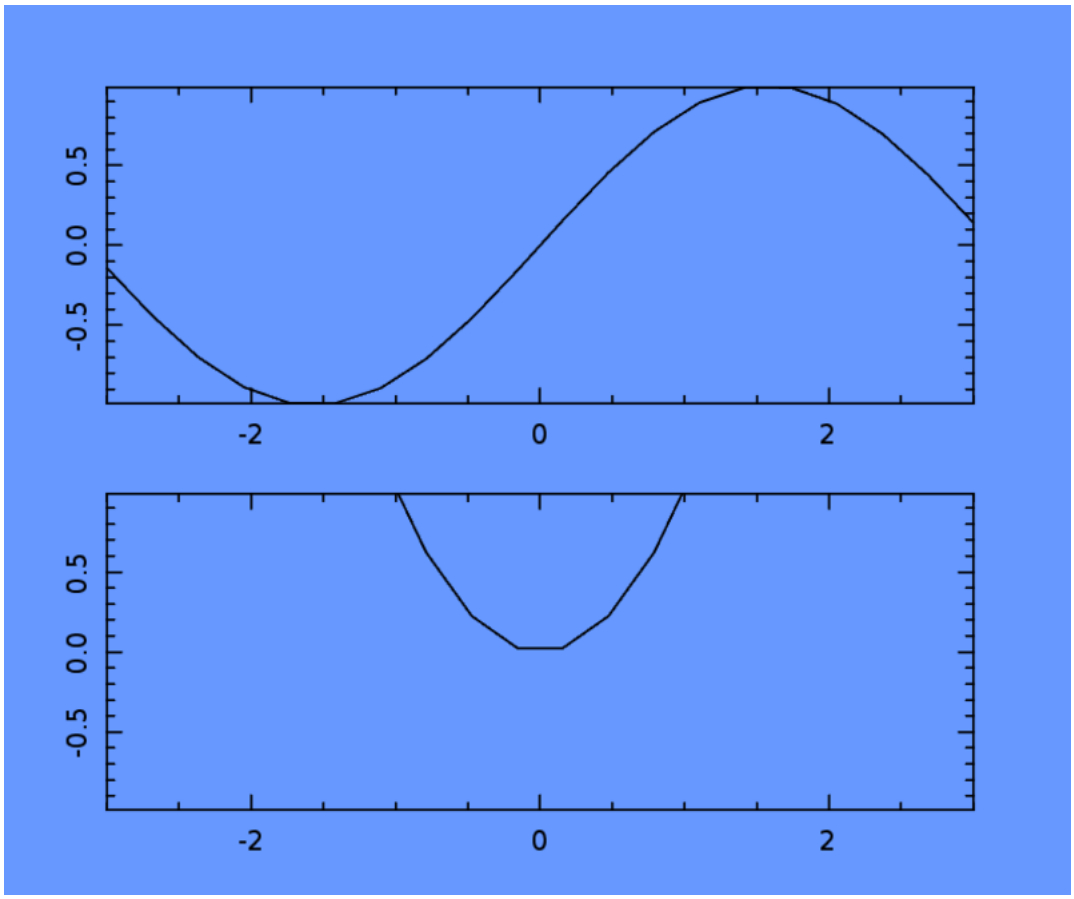
```
     => [0.7, 0.9, 0.1, 0.4]);

$pl->close;
```



The basic box

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 7.eps',
     BACKGROUND => 'SKYBLUE');

# Sine wave on top
$pl->xyplot($x, sin($x),
   VIEWPORT => [0.1, 0.9, 0.55, 0.9]);

# Quadratic on bottom
# BOX is inherited from first plot
$pl->xyplot($x, $x**2,
```

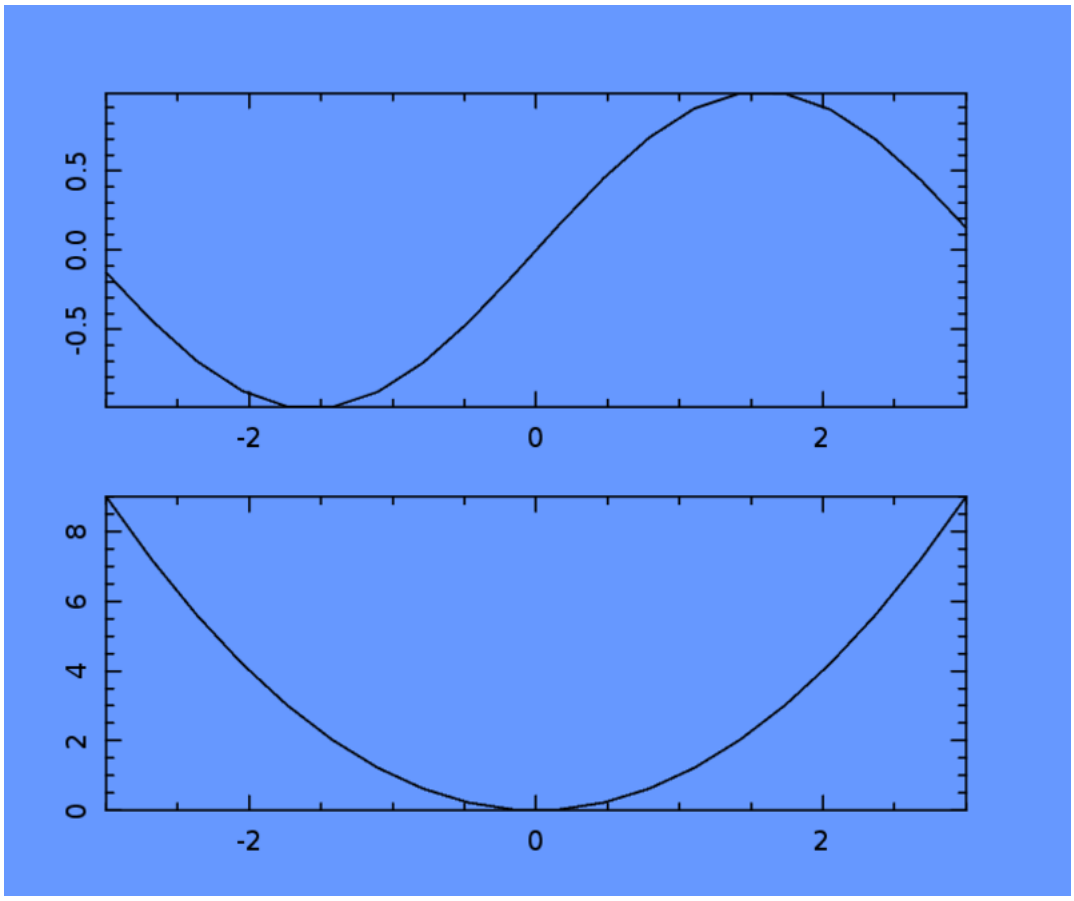
```
    VIEWPORT => [0.1, 0.9, 0.1, 0.45]);

$pl->close;
```



The tweaked box

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 8.eps',
     BACKGROUND => 'SKYBLUE');

# Sine wave on top
 $pl->xyplot($x, sin($x),
     VIEWPORT => [0.1, 0.9, 0.55, 0.9]);

# Quadratic on bottom
 $pl->xyplot($x, $x**2,
     VIEWPORT => [0.1, 0.9, 0.1, 0.45],
```

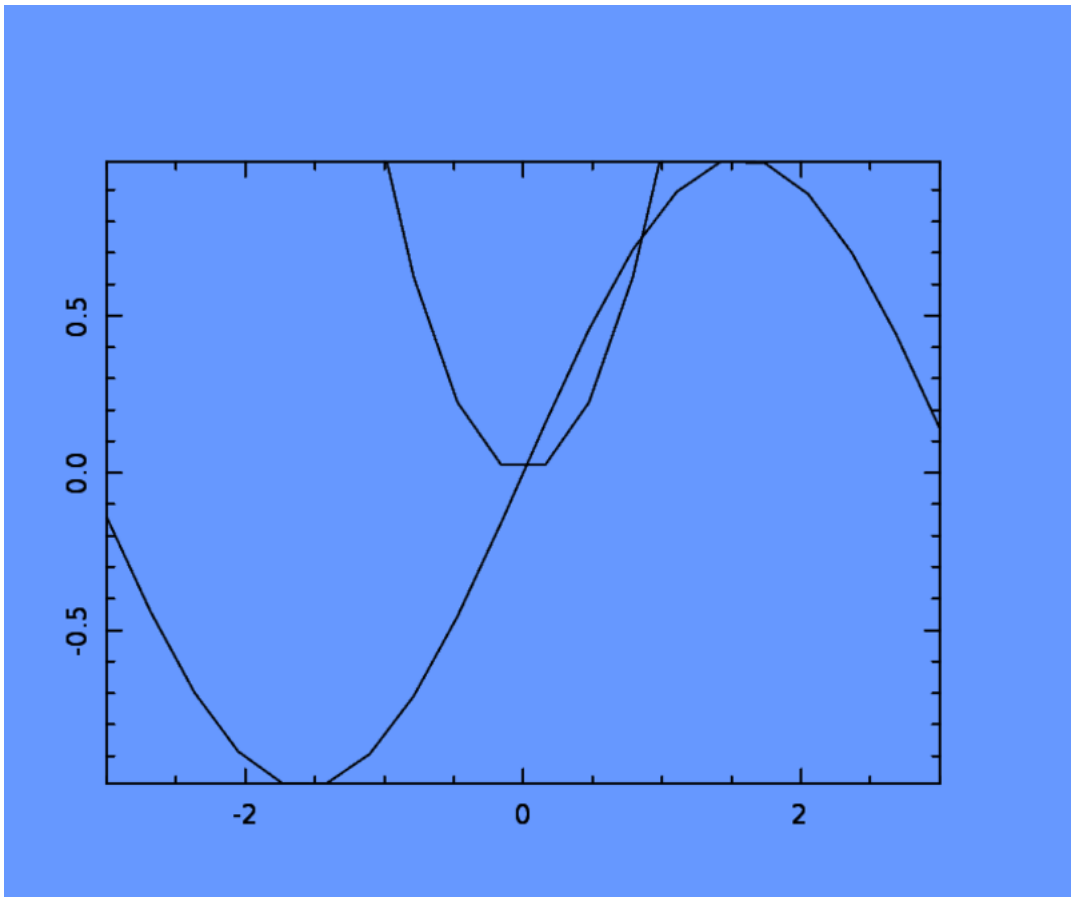
```
     BOX => [-3, 3, 0, 9]);

$pl->close;
```



Box with 2 plots

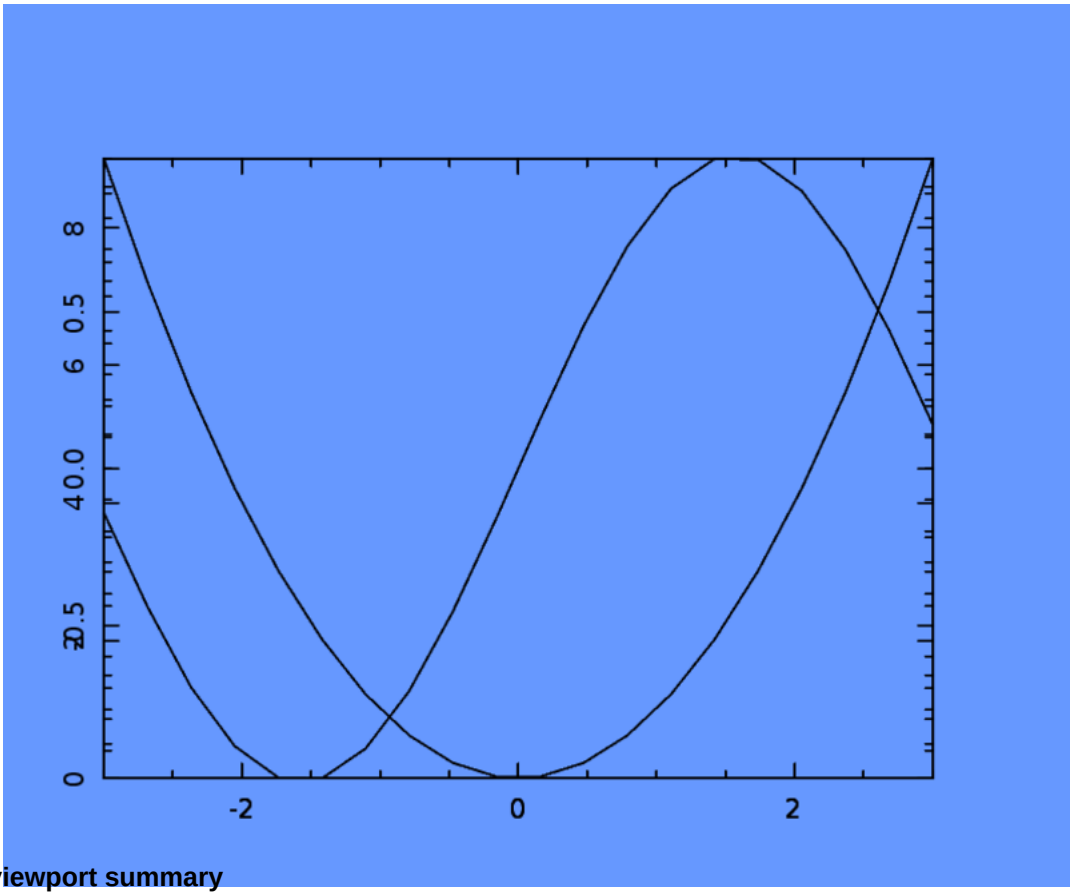
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 9.eps',
     BACKGROUND => 'SKYBLUE');

# Sine wave
$pl->xyplot($x, sin($x));

# Plotting a quadratic on top works
# but the bounds are not good
$pl->xyplot($x, $x**2);
```

```
$pl->close;
```



Multiple plots, changing the box within a single viewport

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $x = zeroes(20)->xlinvals(-3, 3);
my $pl = PLplot->new(
     DEV => 'pscairo',
     FILE => 'box example 10.eps',
     BACKGROUND => 'SKYBLUE');

# Sine wave
$pl->xyplot($x, sin($x));

# Changing the box for the quadratic
# does not work - bad y ticks
$pl->xyplot($x, $x**2,
     BOX => [-3, 3, 0, 9]);

$pl->close;
```



Box and viewport summary

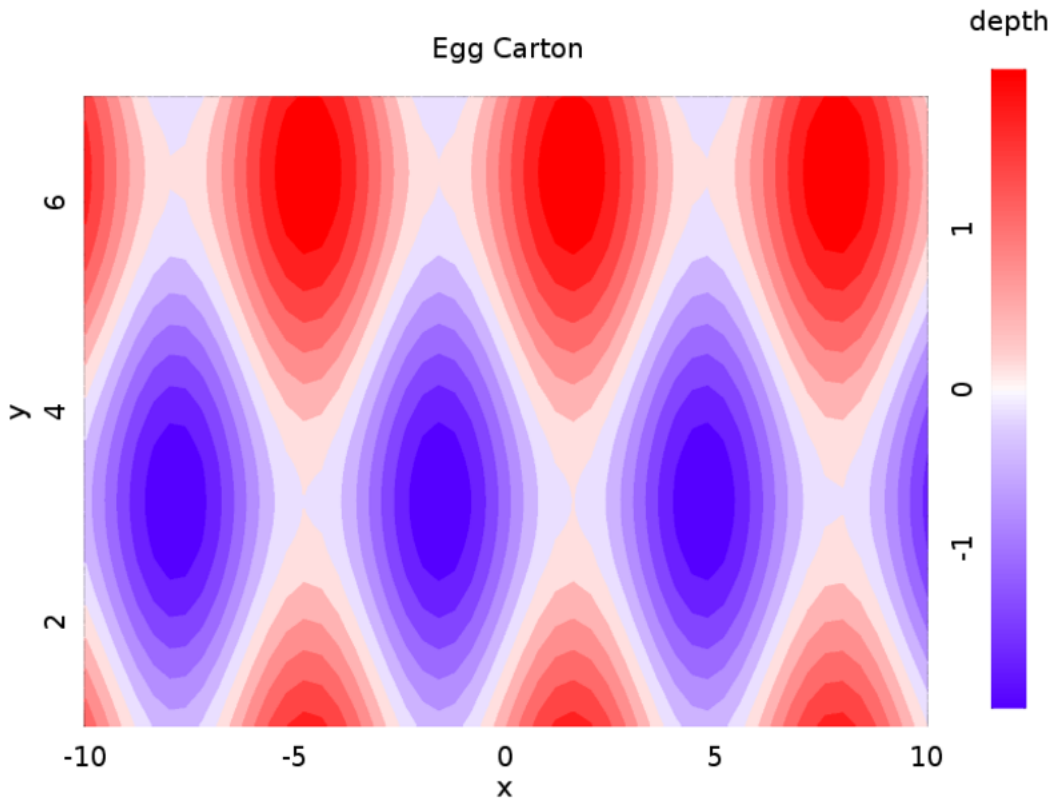
For multiple plots on the same viewport, set the box with the first call to xyplot For non-overlapping plots (on different viewports), specify the box as necessary The viewport specifies the extent of the plotting region; tick labels, axis labels, and titles are drawn outside the viewport

Other types of plot

Shadeplot

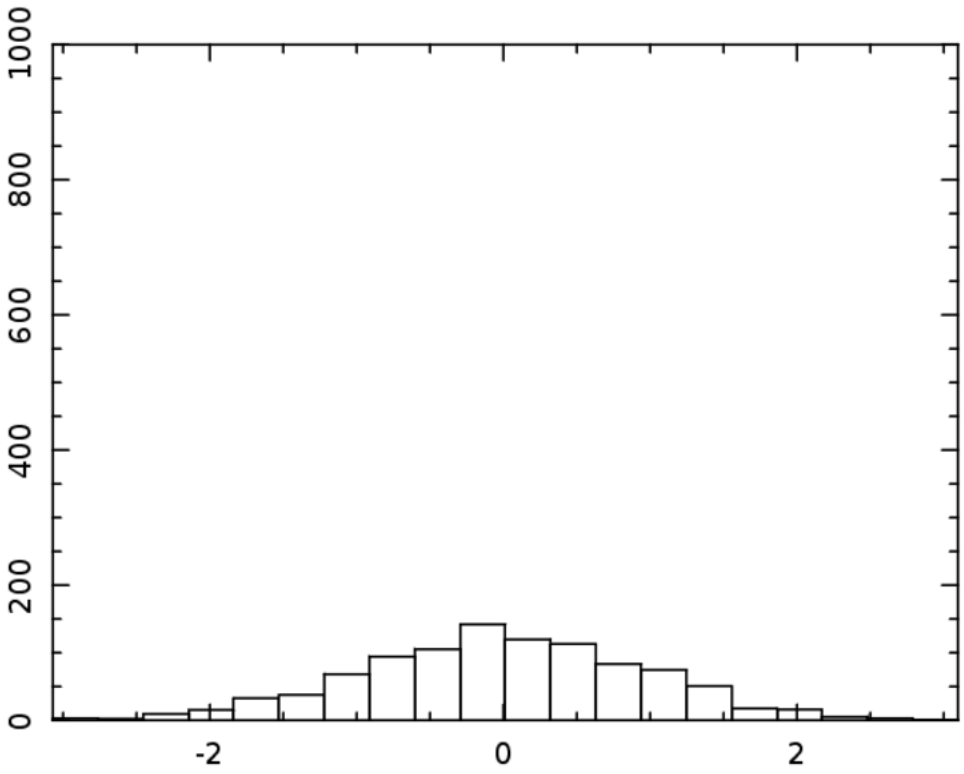
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';
my $pl = PLplot->new(
    DEV => 'pscairo',
    FILE => 'shadeplot3.eps');
# Define z = sin(x) + cos(y), a 2D piddle:
my $x=zeroes(51)->xlinvals(-10, 10);
my $y=zeroes(51)->xlinvals(1, 7);
my $z=sin($x) + cos($y->transpose);
# Make a shade plot with 15 color steps:
$pl->shadeplot($z, 15,
    BOX => [$x->minmax, $y->minmax],
    XLAB => 'x', YLAB => 'y',
    TITLE => 'Egg Carton');
# Add a 'vertical' color key:
$pl->colorkey($z, 'v', VIEWPORT => [0.93, 0.96, 0.15, 0.85],
        XLAB => '', YLAB => '', TITLE => 'depth');
```

```
$pl->close;
```



Histogram

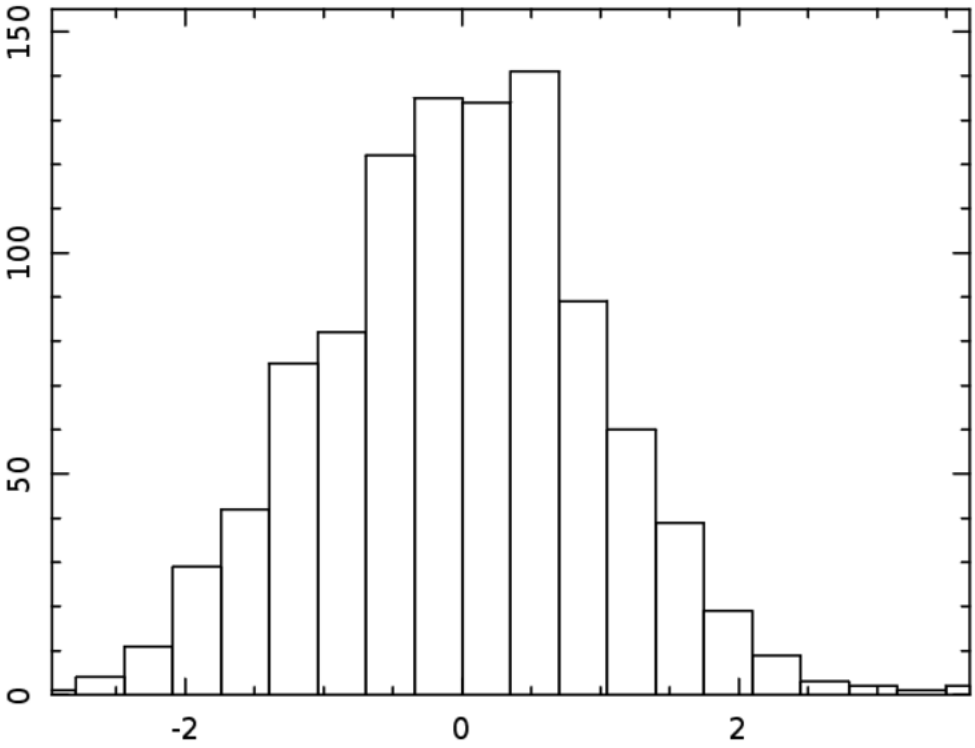
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
    DEV => 'pscairo',
    FILE => 'histogram.eps');

# Generate some data:
my $data = grandom(1000);

# Make a histogram of that data in 20 bins:
$pl->histogram($data, 20);

$pl->close;
```



Histogram height

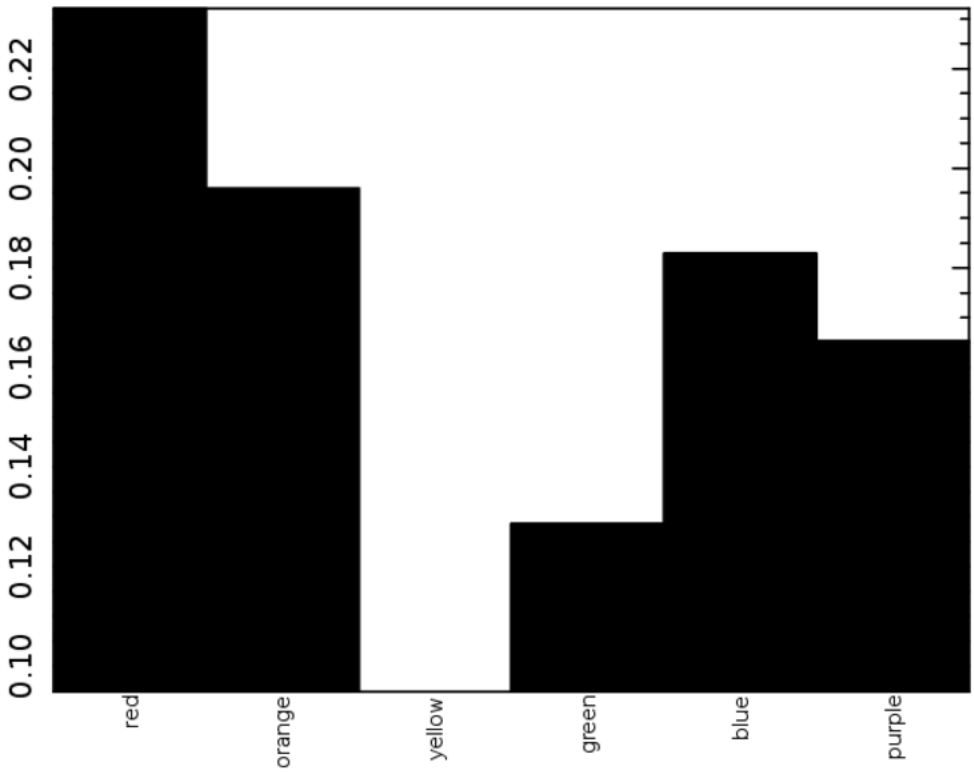
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
    DEV => 'pscairo',
    FILE => 'histogram2.eps');

# Generate some data:
my $data = grandom(1000);

# Get approximate binning:
my $nbins = 20;
my $binwidth =  ($data->max-$data->min) / $nbins;

my ($x, $y) = hist($data , $data->minmax, $binwidth);

# Make a histogram of that data in 20 bins:
my $fudgefactor = 1.1;

$pl->histogram($data, $nbins,
        BOX => [$x->minmax, 0, $y->max * $fudgefactor]);
```

```
$pl->close;
```



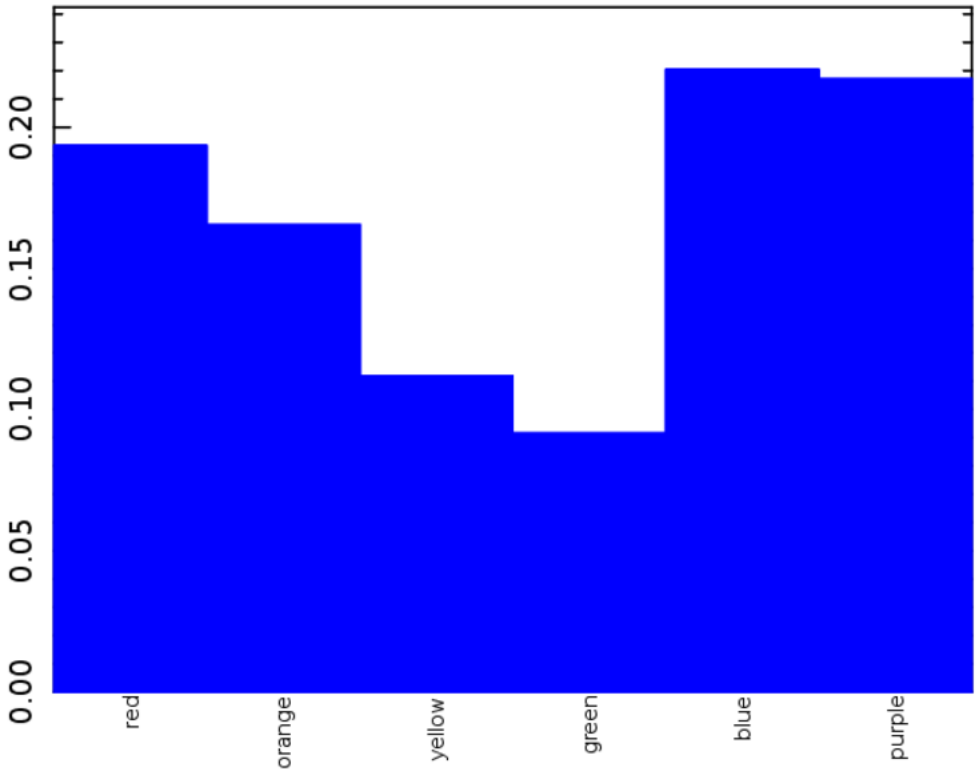
Bargraph

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
    DEV => 'pscairo',
    FILE => 'bargraph.eps');

# Generate some data:
my @colors = qw(red orange yellow green blue purple);
my $votes = random(scalar(@colors));

# Normalize the votes
$votes /= $votes->sum;

# Make a barchart of the votes.
$pl->bargraph(\@colors, $votes);

$pl->close;
```



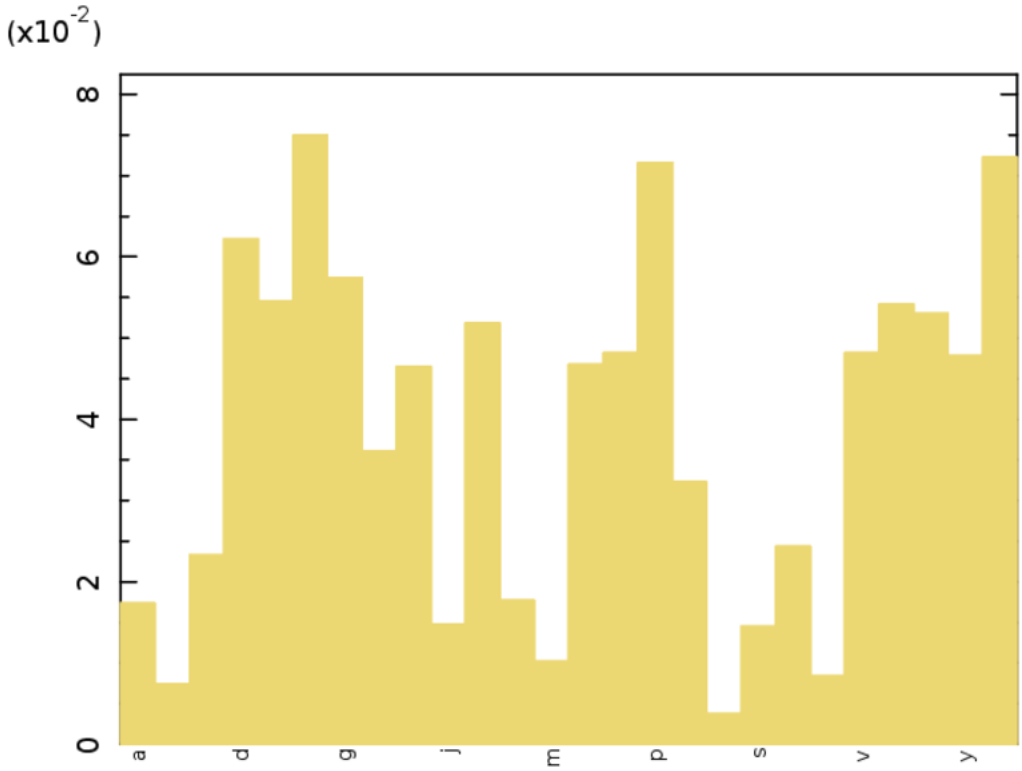
Bargraph color and bar height

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
    DEV => 'pscairo',
    FILE => 'bargraph2.eps');

# Generate some data:
my @colors = qw(red orange yellow green blue purple);
my $votes = random(scalar(@colors));

# Normalize the votes
$votes /= $votes->sum;

# Make a barchart of the votes.
$pl->bargraph(\@colors, $votes,
     COLOR => 'BLUE',
     BOX => [0, scalar(@colors), 0, 1.1 * $votes->max]
  );

$pl->close;
```



Bargraph with labelling

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
      DEV => 'pscairo',
      FILE => 'bargraph3.eps');

# voting on letters:
my @letters = ('a' .. 'z');
my $votes = random(0 + @letters);

# Normalize the votes
$votes /= $votes->sum;

# Make a barchart of the votes.
$pl->bargraph(\@letters, $votes,
      COLOR => 'LIGHTGOLDENROD',
      BOX => [0, scalar(@letters) , 0, 1.1 * $votes->max],
      MAXBARLABELS => 10
  );

$pl->close;
```



Using the MEM device

Use the MEM device to: -- load an image and plot over that image -- plot to a custom windowing device -- animated plots

The way the MEM device works, is that it needs an RGB or RGBA (RGB with alpha transparency) buffer to write on top of.

Creating a MEM memory buffer

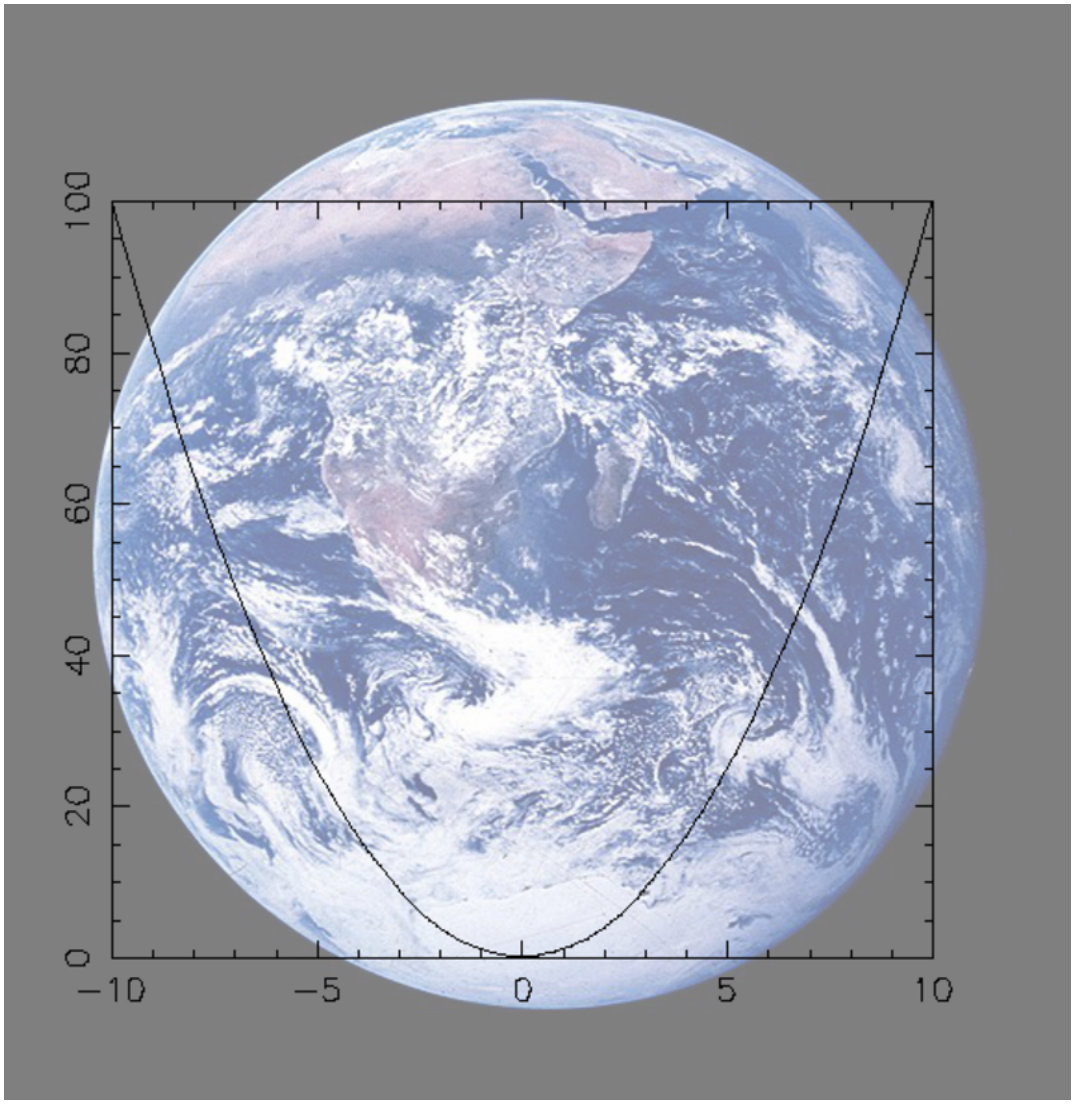
There are 2 drivers which handle the MEM device, mem and memcairo. mem is for plain RGB. memcairo can handle transparency values.

```
use PDL;
## creating the mem device buffer ##

# the mem device
# Allocate the buffer for plain rgb
my $buffer = zeroes(byte, 3, $width, $height);

# Create the PLplot object
my $pl = PDL::Graphics::PLplot->new(
                DEV => 'mem',
                MEM => $buffer
        );

## For the memcairo device which handles tranparencies ##

# Allocate the buffer
my $buffer = zeroes(byte, 4, $width, $height);
```

```
# Create the PLplot object
my $pl = PDL::Graphics::PLplot->new(
               DEV => 'memcairo',
               MEM => $buffer
        );
```

Plotting over an image with the MEM device

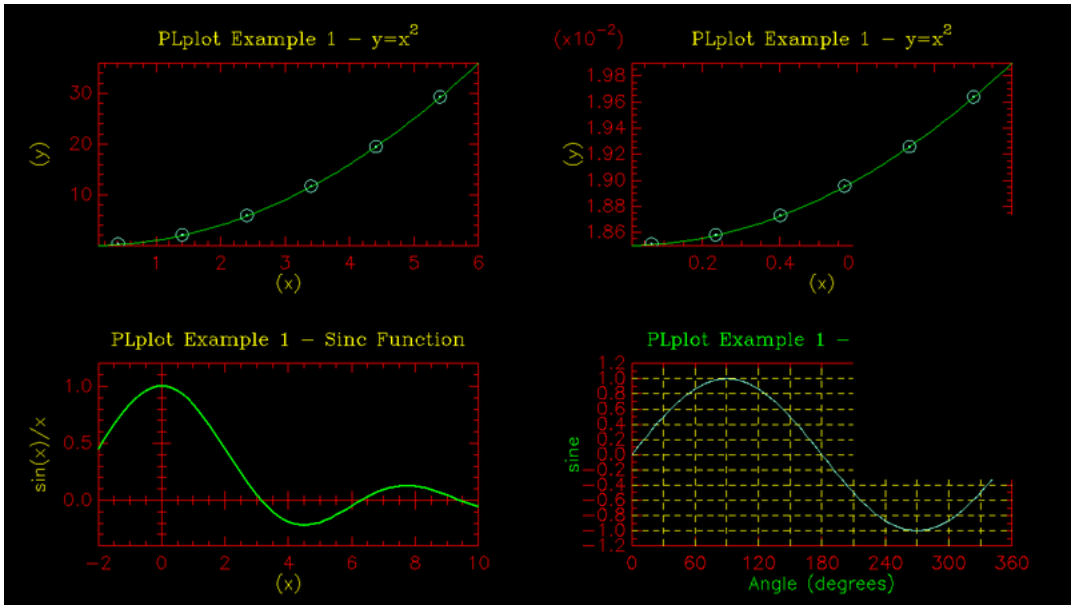
```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';
use PDL::IO::Pic;

# Load an image
# (has dims 3 x width x height)
my $pic = rpic('earth.jpg');

# Flip the y axis
$pic = $pic->slice(':,:,-1:0:-1');
# Whiten the image a bit
$pic = 127 + $pic / 2;

my $pl = PLplot->new( DEV => 'mem',
                      MEM => $pic);

# Plot a quadratic curve over the image
my $x=zeroes(51)->xlinvals(-10, 10);
$pl->xyplot($x, $x**2);
$pl->close;

# flip the y axis back and save the image
$pic = $pic->slice(':,:,-1:0:-1');
wpic($pic, 'earth_plot.png');
```



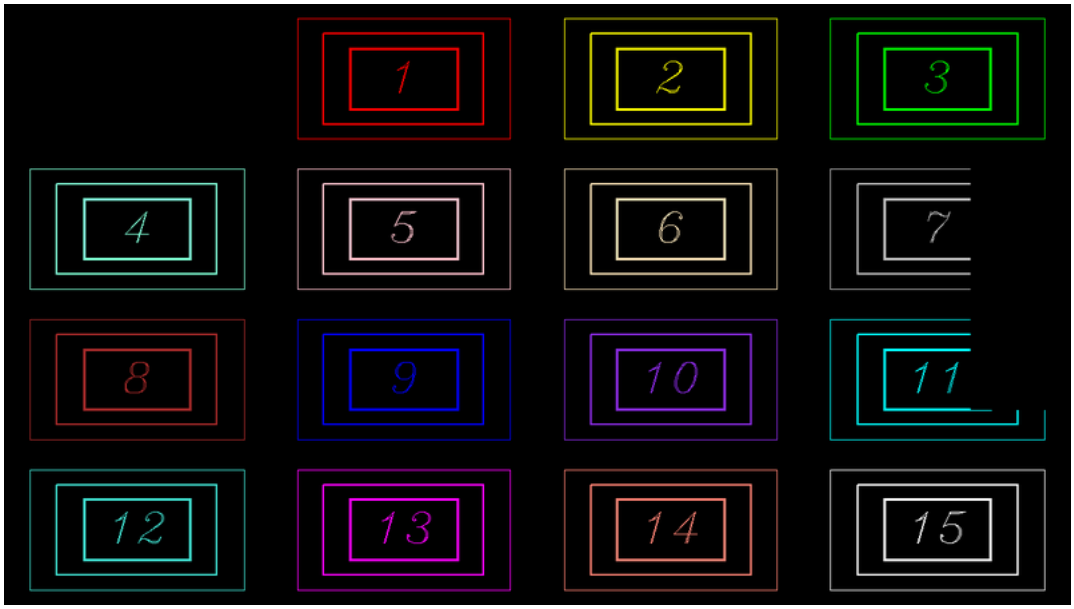
Functional programming style examples

This section uses the functional programming style of the original C library examples.

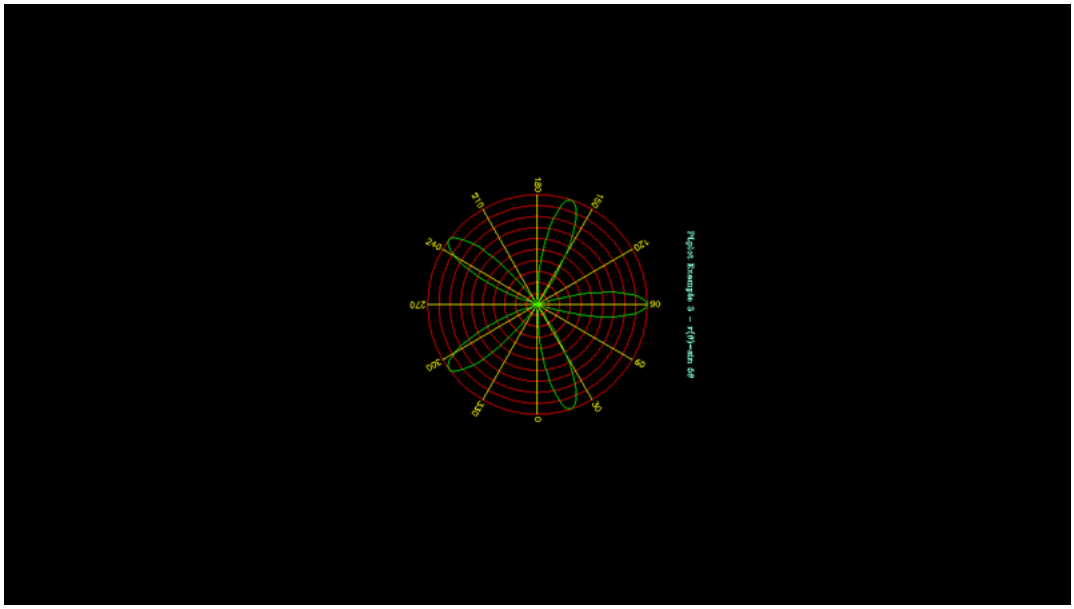
Simple line plot and multiple windows demo x01



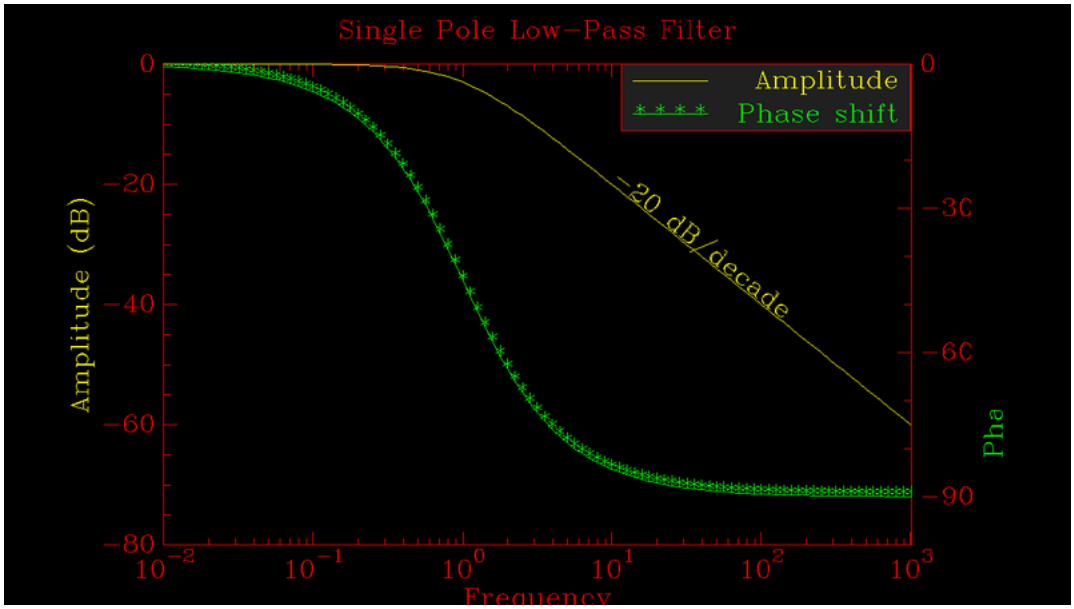
Multiple window and color map 0 demo x02



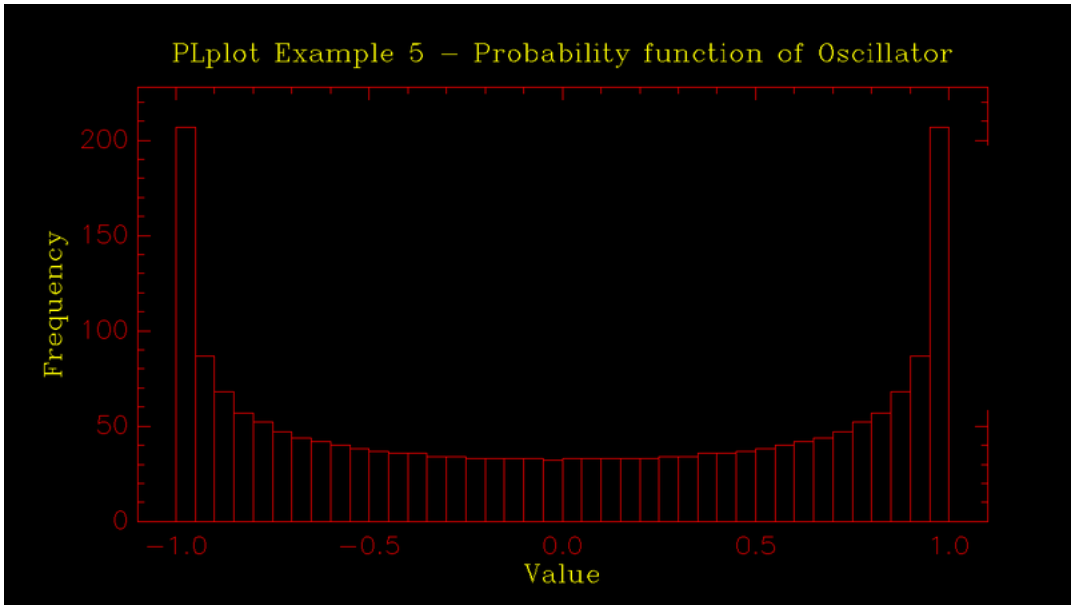
Polar plot demo x03



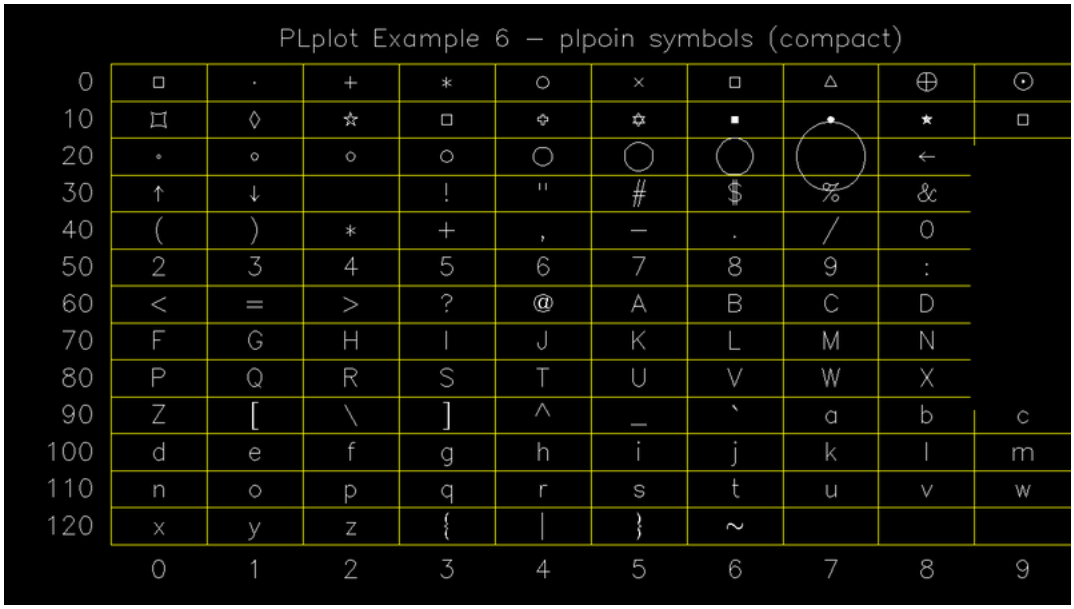
Log plot demo x04



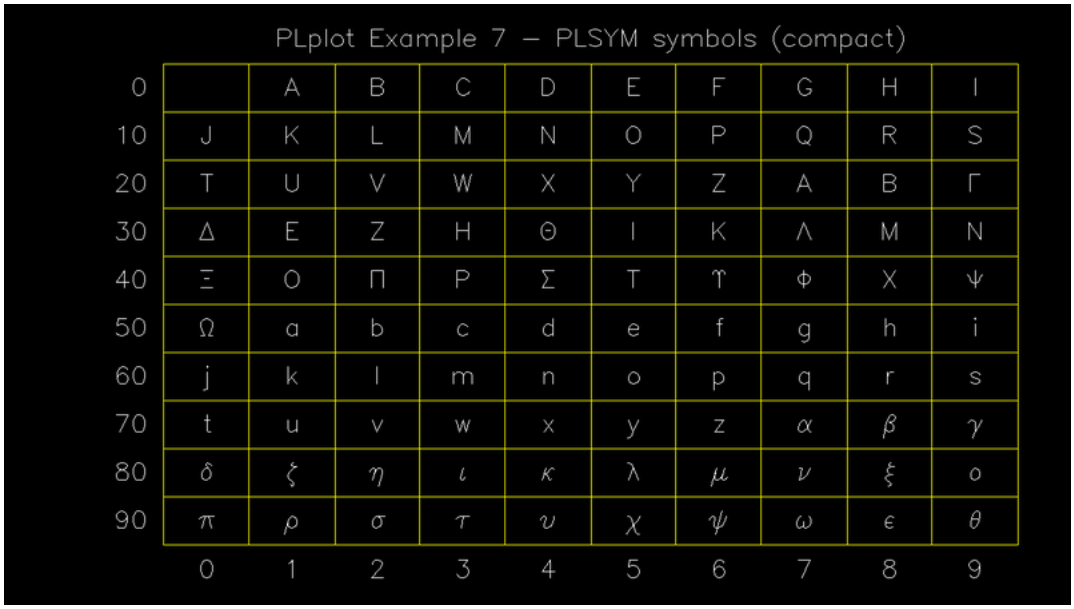
Histogram demo x05



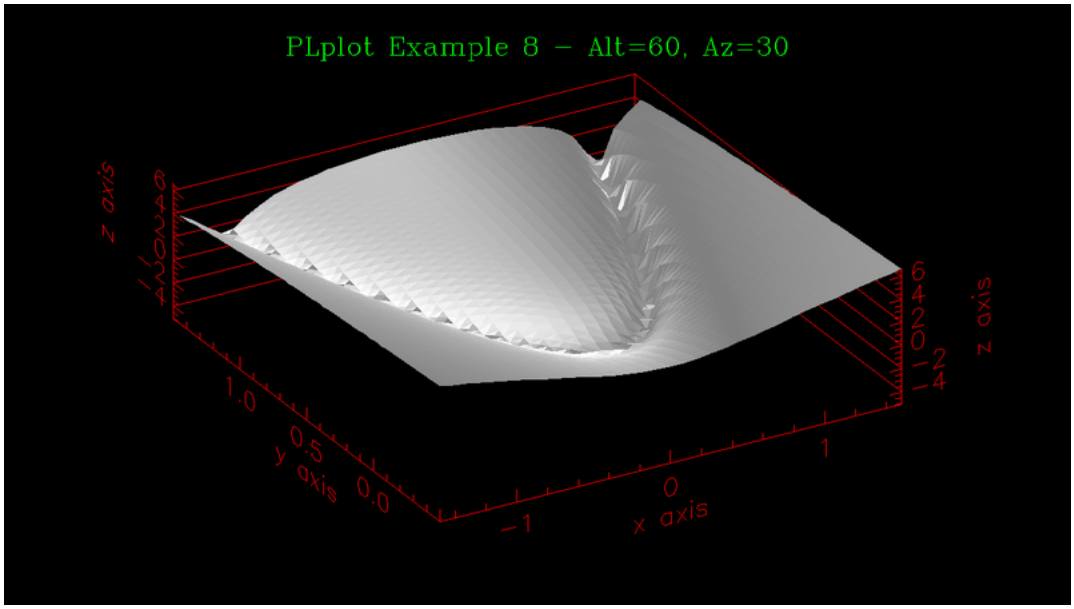
Font demo x06



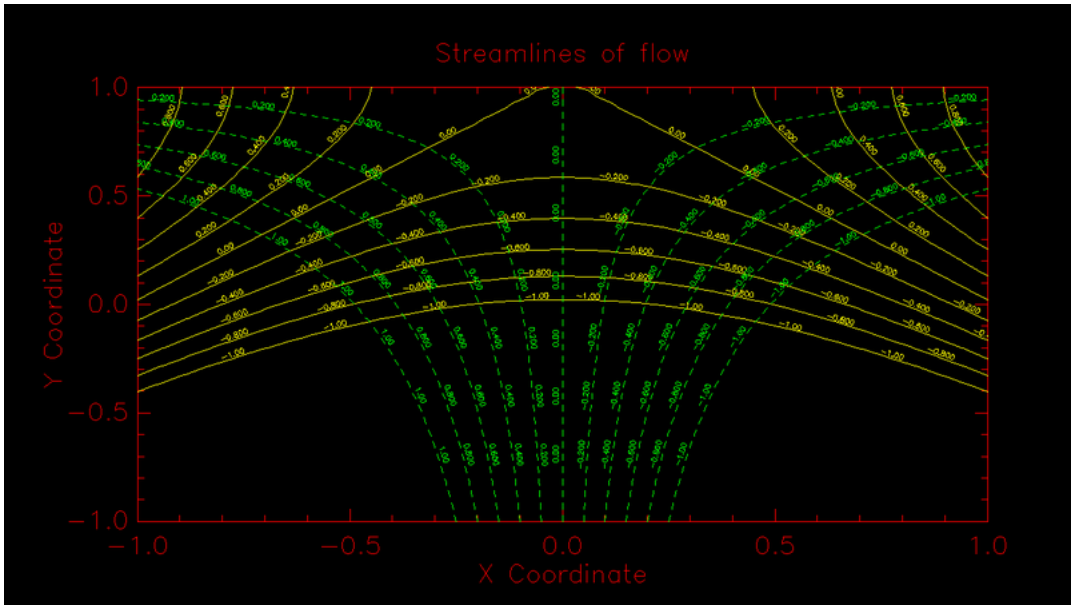
Font demo x07



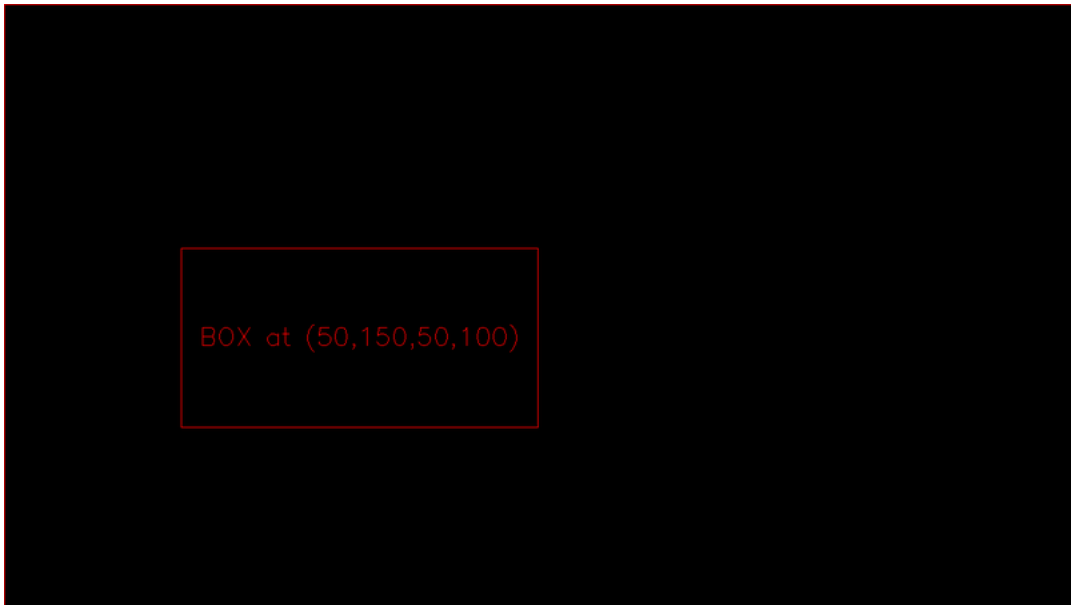
3-d plot demo x08



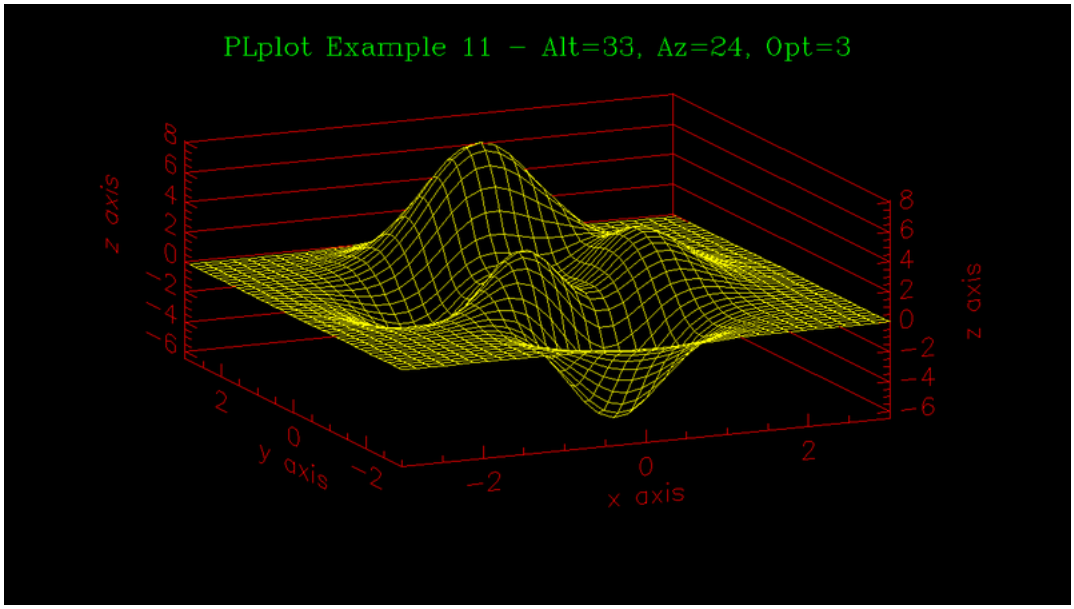
Contour plot demo x09



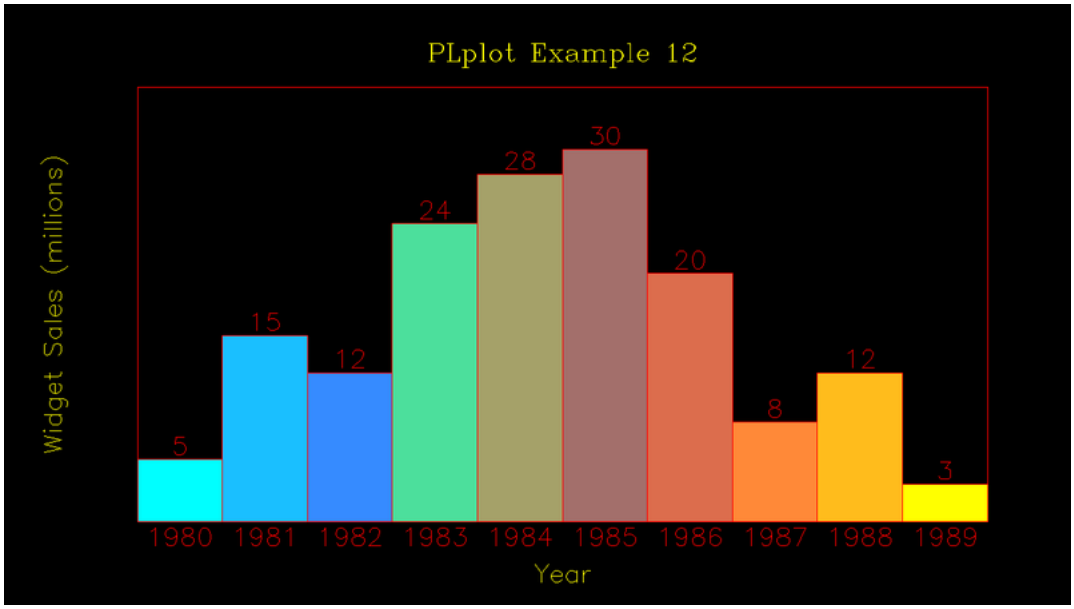
Window positioning demo x10



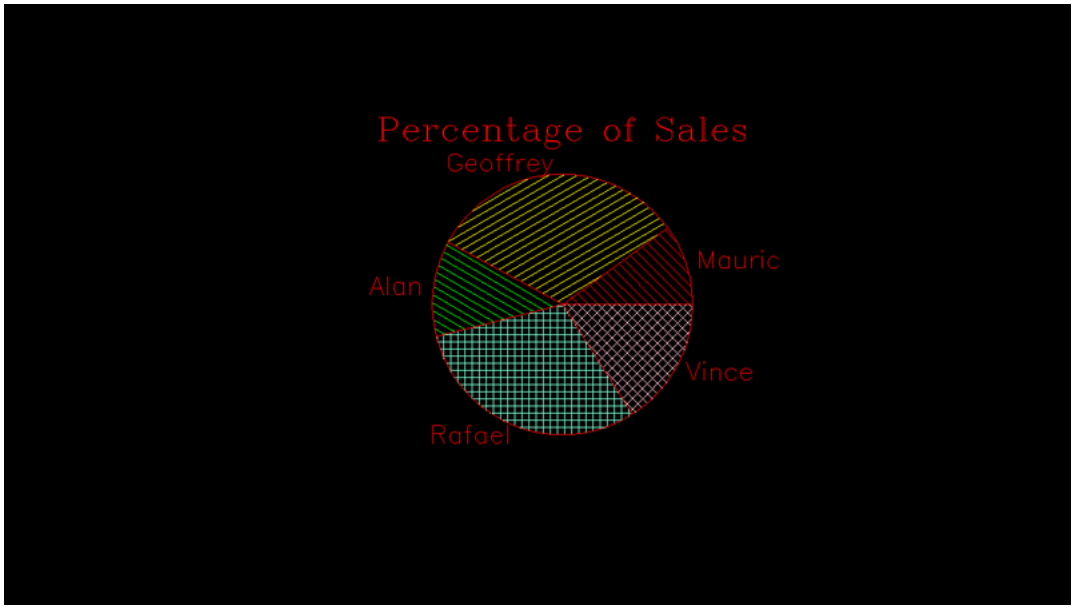
Mesh plot demo x11



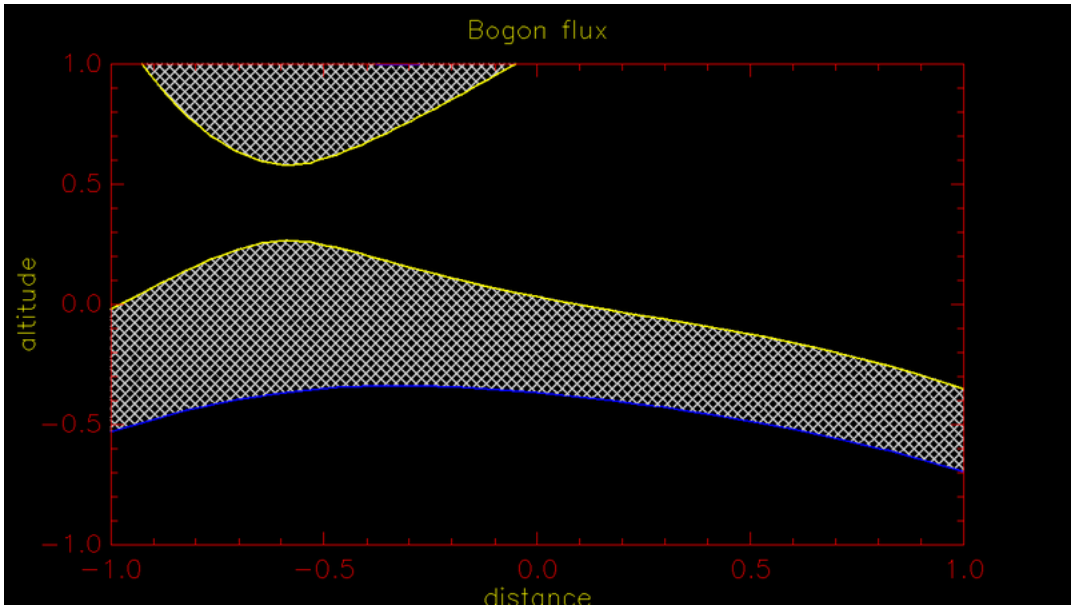
Bar chart demo x12



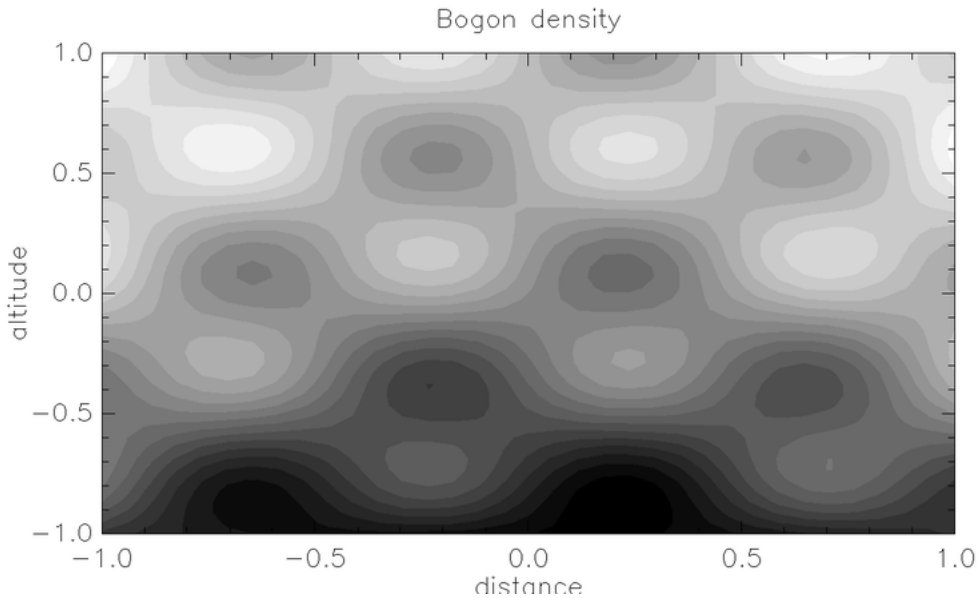
simple pie chart x13



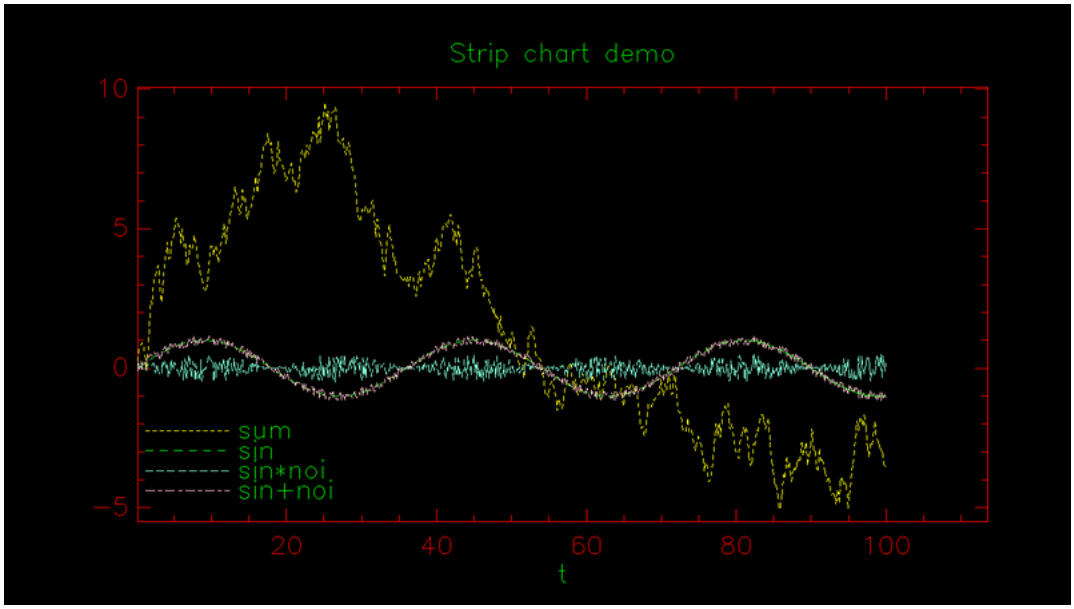
Shade plot demo x15



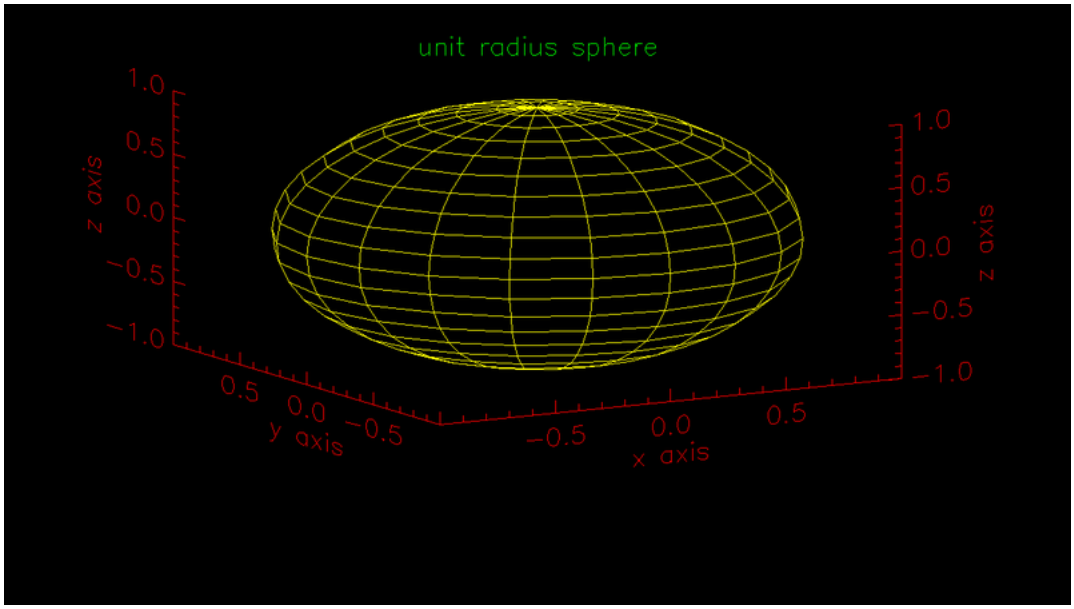
plshade demo, using color fill x16



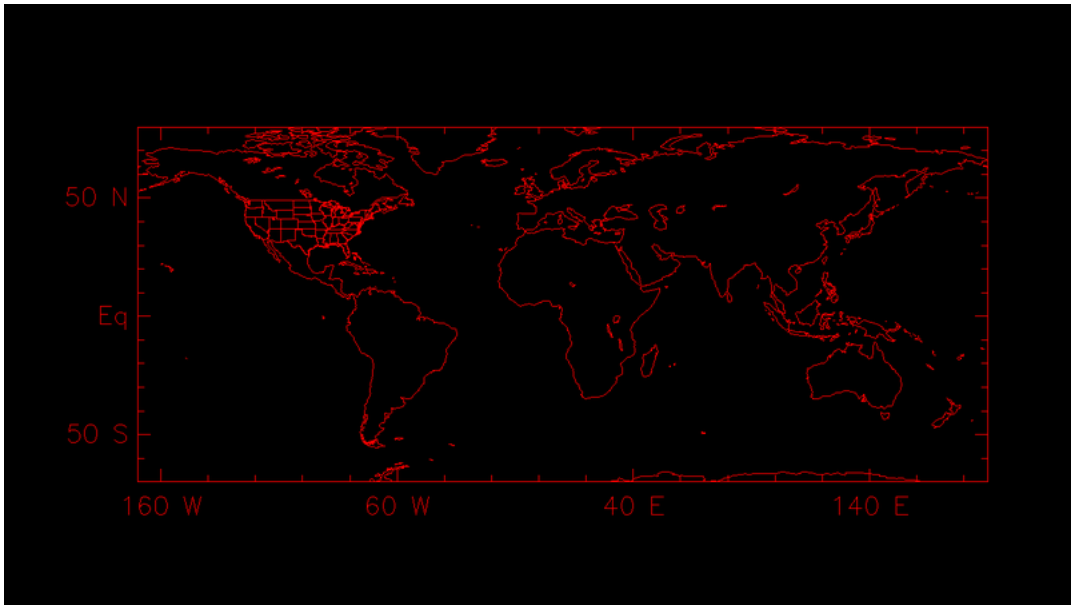
A simple stripchart with four pens x17



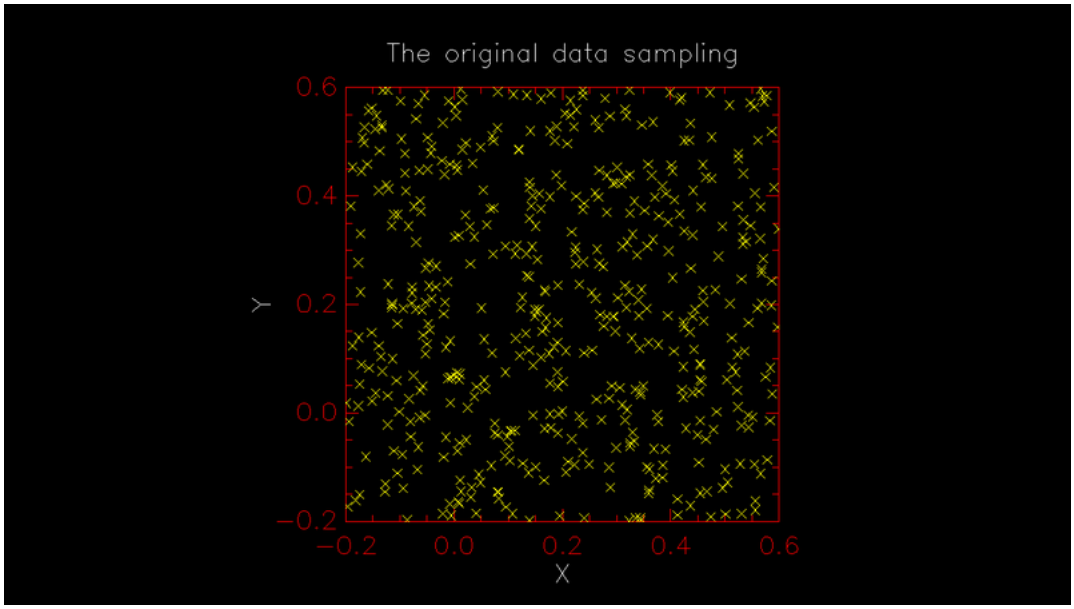
3-d line and point plot demo x18



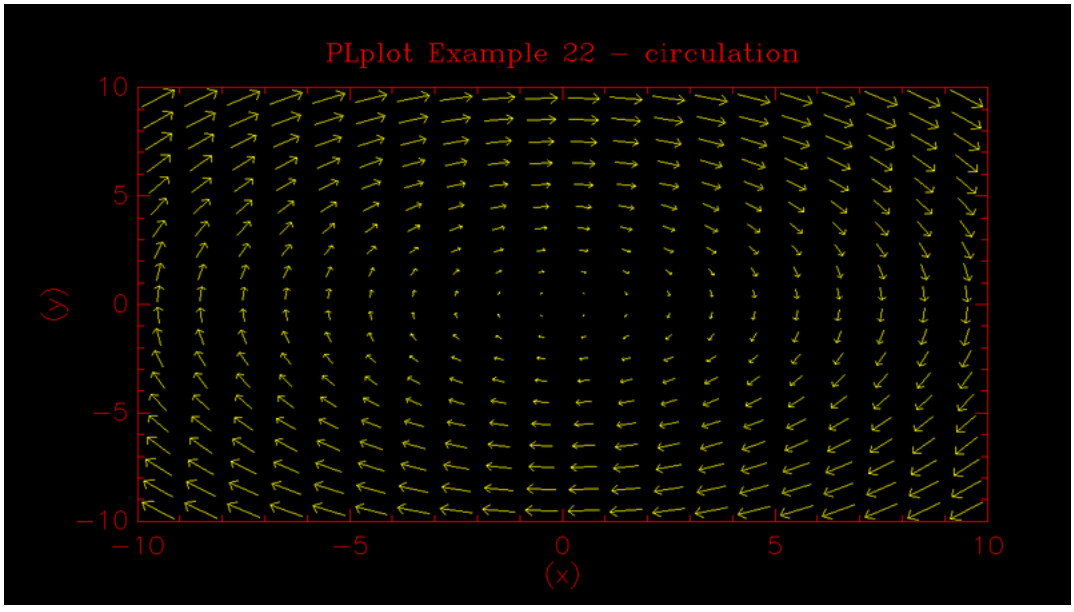
Backdrop plotting of world, US maps. x19



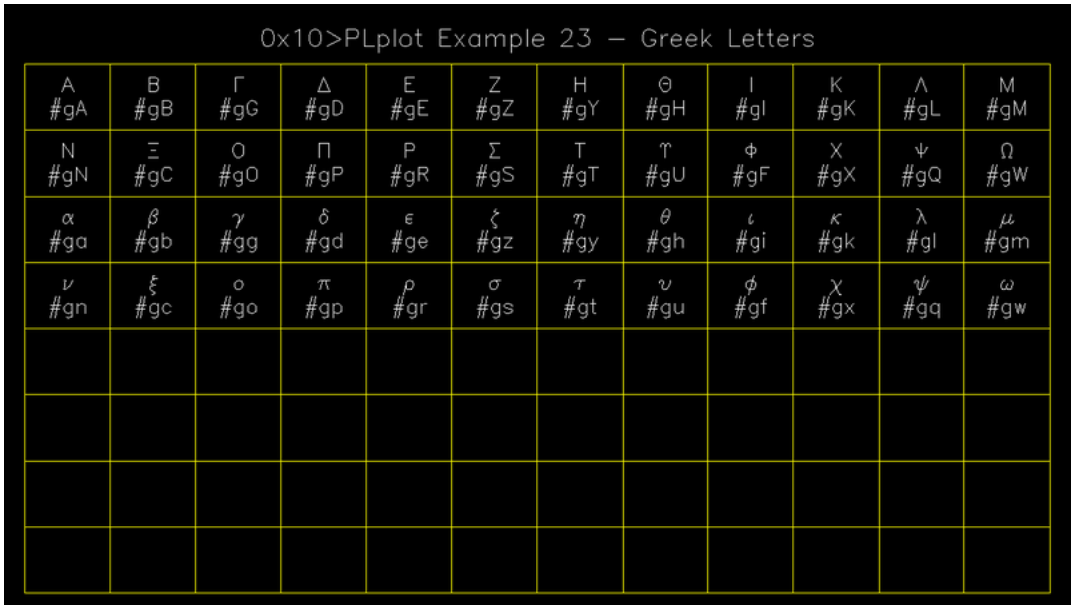
Grid data demo x21



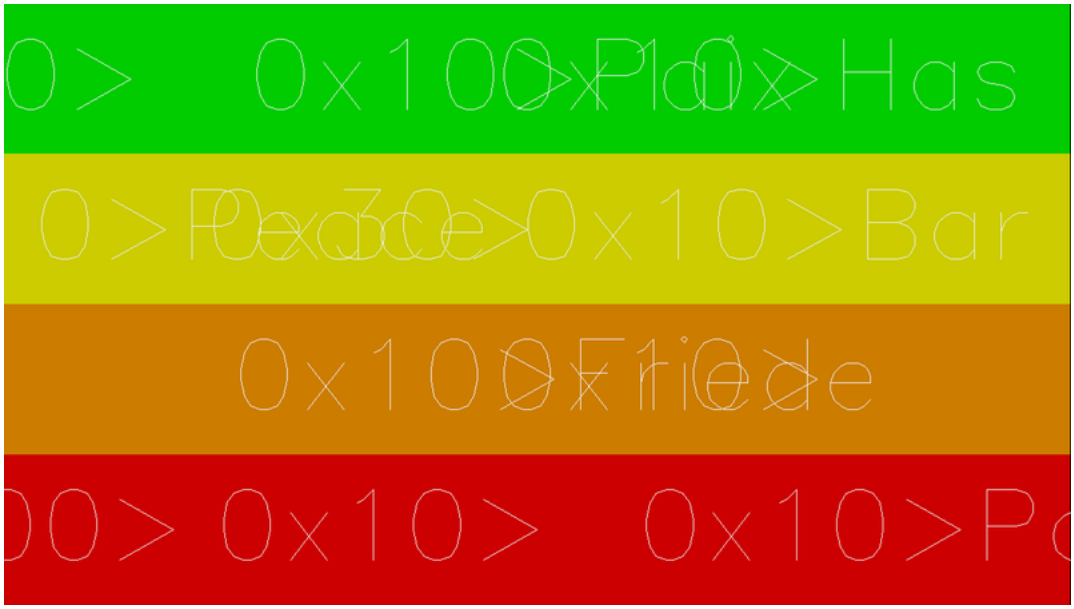
Simple vector plot x22



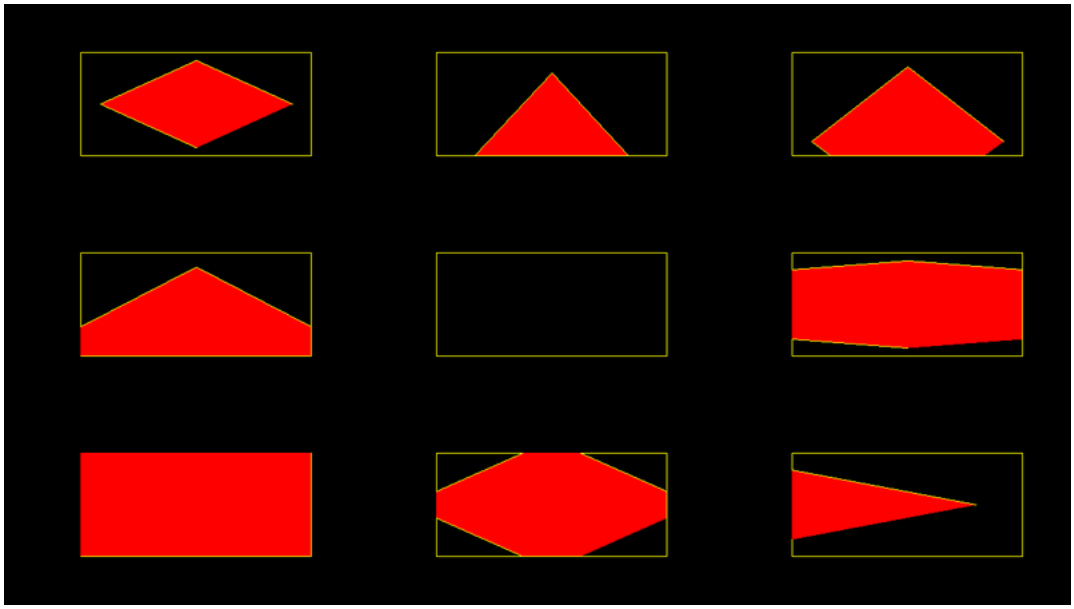
Displays Greek letters and mathematically interesting Unicode ranges x23



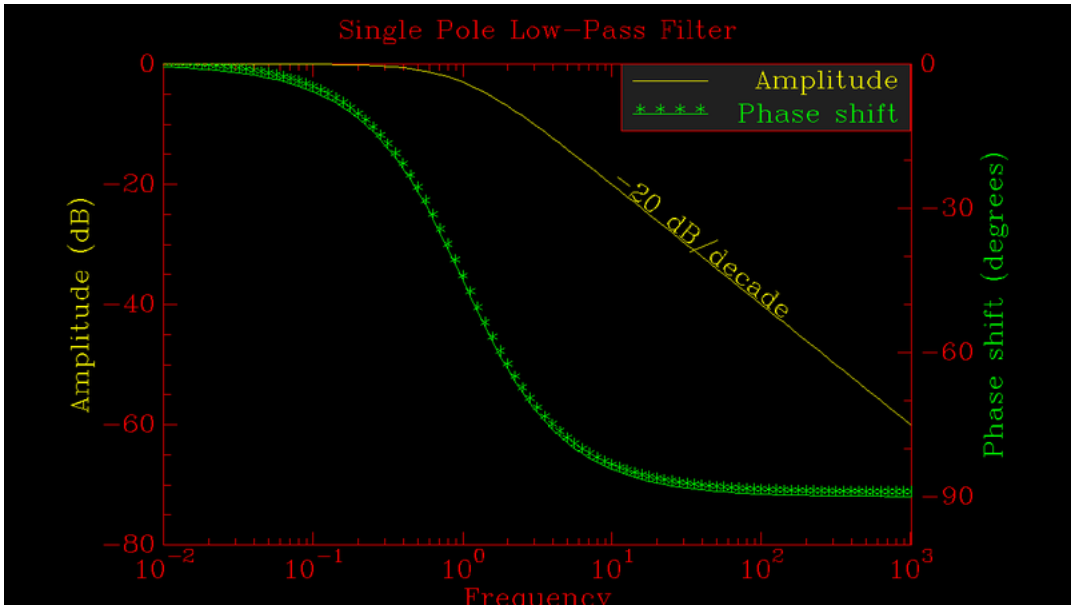
Unicode Pace Flag x24



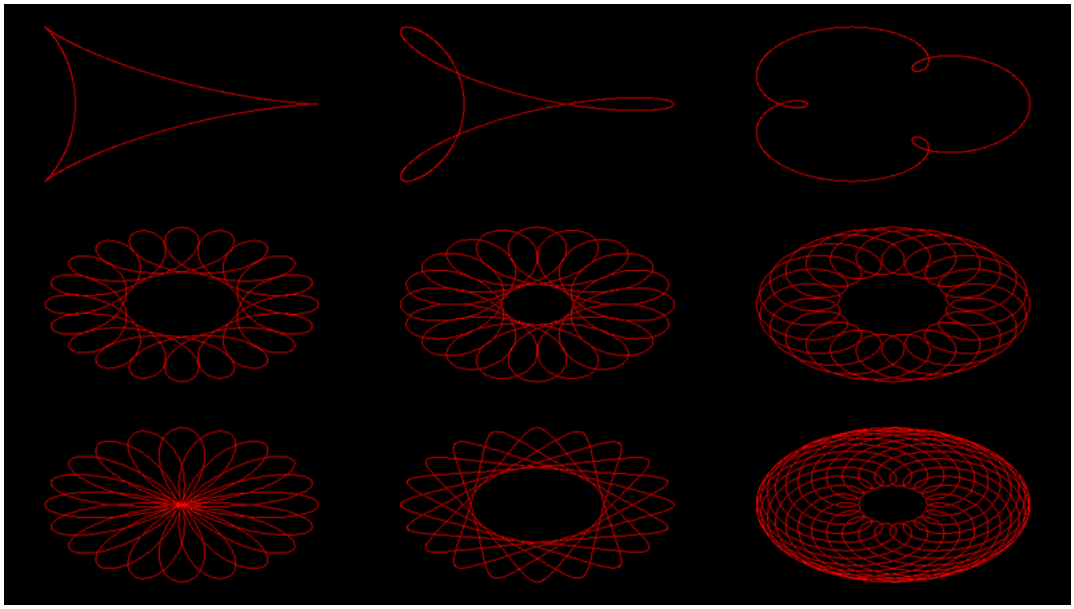
Drawing polygons x25



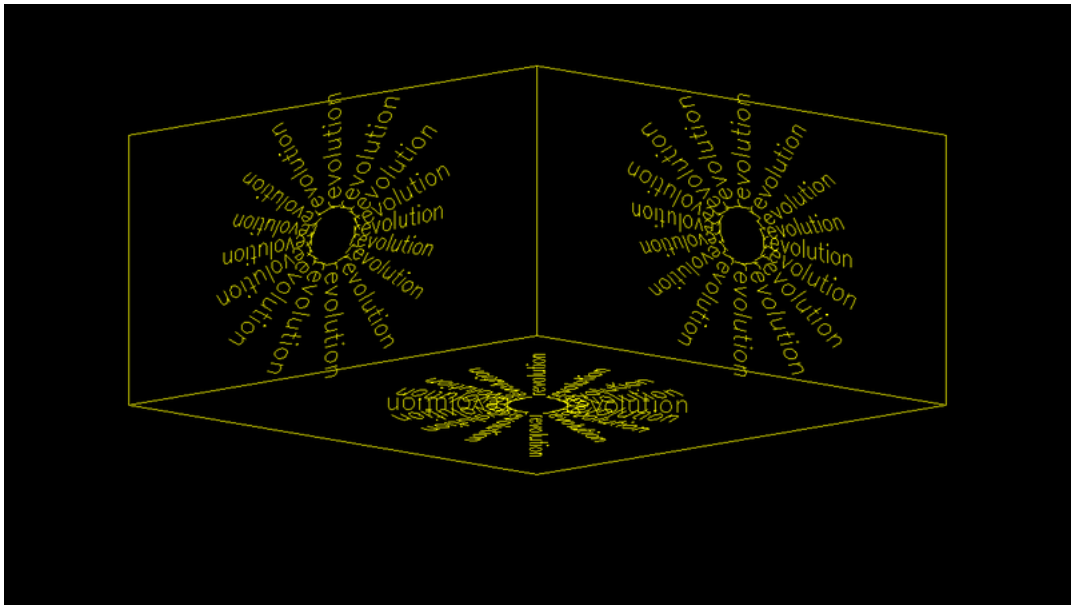
Frequency Amplitude and Phase x26



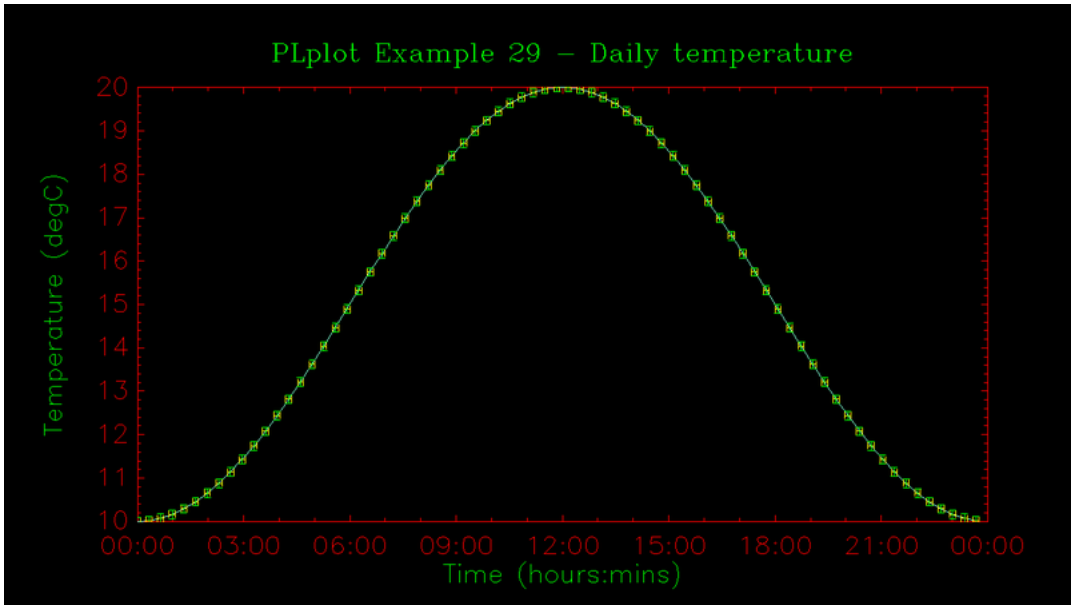
Spirograph curves - epitrochoids, cycolids, roulettes x27



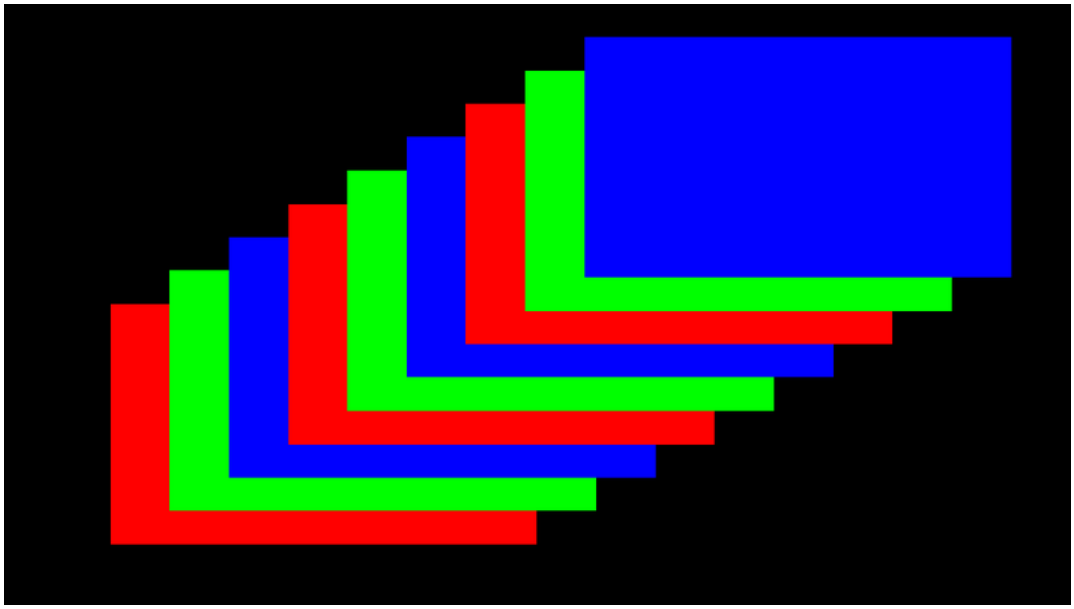
plmtex3, plptex3 demo x28



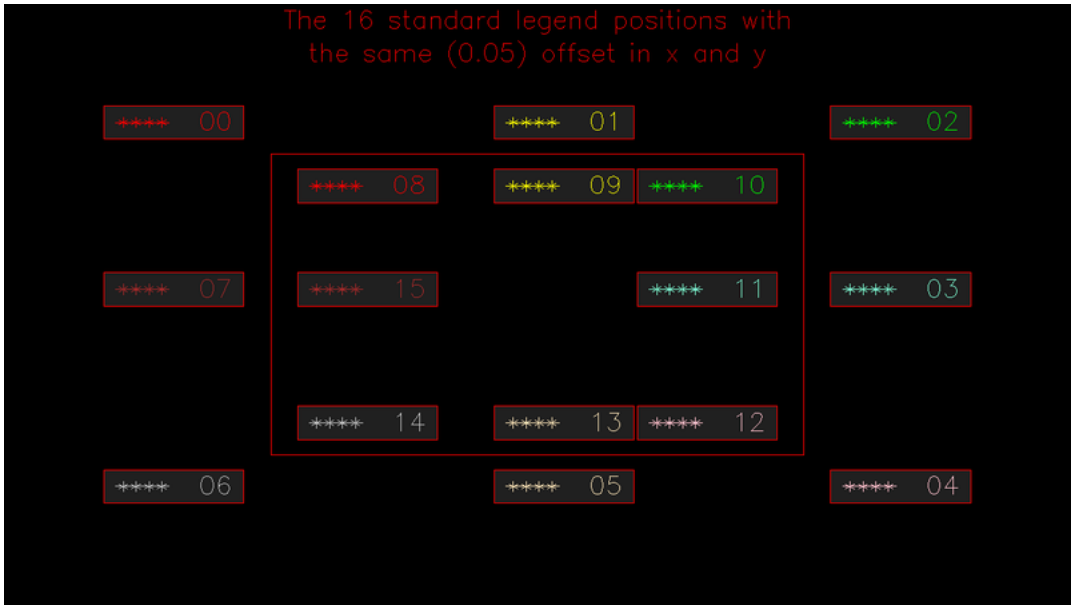
Plots using date / time formatting for axes x29



Alpha color values demonstration x30



Using pllegend including unicode symbols x33



Typesetting, greek letters, symbols

Use escape sequences to insert superscripts, subscripts, Greek letters, etc.

```
#u - superscript until the next #d
#d - subscript until the next #u
#- - toggle underline mode
#+ - toggle overline mode
#fn - switch to normal (sans-serif) font
#fr - switch to Roman (serif) font
#fi - switch to italic font
#fs - switch to script font
```

```
# Use greek symbol rho for density:
    $pl->xyplot($radius, $density,
    YLAB => 'density #gr'
     # ...
    );
```

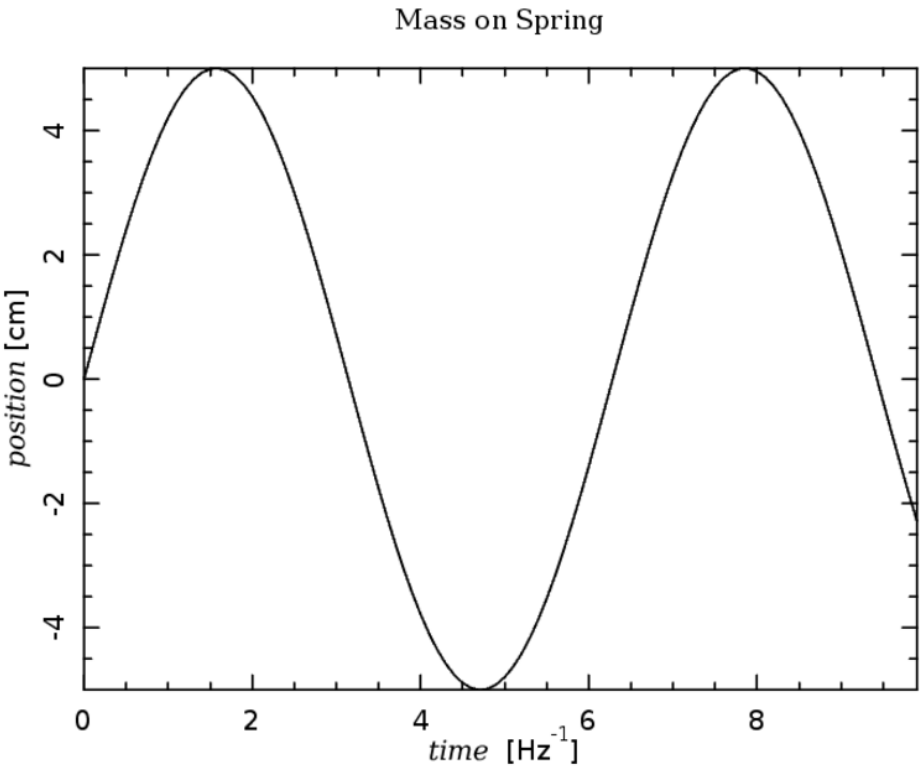
Unicode is supported.

Use the string #gx to print the Greek letter equivalent of x:

A	B	G	D	E	Z	Y	H	I	K	L	M
A	B	Γ	Δ	E	Z	H	Θ	I	K	Λ	M
N	C	O	P	R	S	T	U	F	X	Q	W
N	Ξ	O	Π	P	Σ	T	Y	Φ	X	Ψ	Ω
a	b	g	d	e	z	y	h	i	k	l	m
α	β	γ	δ	ϵ	ζ	η	θ	ι	κ	λ	μ
n	c	o	p	r	s	t	u	f	x	q	w
ν	ξ	o	π	ρ	σ	τ	υ	ϕ	χ	ψ	ω

A basic typsetting example

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

# Generate a time series
my $time = sequence(100)/10;
my $sinewave = 5 * sin($time);

# Create the PLplot object:
my $pl = PLplot->new(
      DEV => 'pscairo',
      FILE => 'Typesetting.eps');

# Plot the time series
$pl->xyplot($time, $sinewave,
      XLAB => '#fi time #fn [Hz#u-1#d]',
      YLAB => '#fiposition#fn [cm]',
      TITLE => '#frMass on Spring'
    );

# Close the PLplot object to finalize
$pl->close;
```



psfrag

For LATEX typsetting, post-process eps images with psfrag.

```
--  replaces simple strings with any valid LATEX text.
--  ensures consistent fonts for both images and documents
--  Do not use the pscairo device. Use ps or psc.
```

annotations and TEXTPOSITION

To add text to a plot, use the text method, specifying the TEXTPOSITION option. The TEXTPOSITION takes either four or five arguments. The four-argument form places text outside the viewport along one of its edges:

```
$pl->text($string, TEXTPOSITION => [$side, $disp, $pos, $just]);
```

$side is one of 't', 'b', 'l', or 'r' indicating the top, bottom, left, or right edge

$disp is the number of character heights out from the edge

$pos is the position of the string's reference point along the edge of the viewport, from 0 to 1

$just indicates the location of the reference point of the string.

0 means the reference point is the string's left edge; 1 indicates the right edge

The five-argument form places the text within the viewport at an arbitrary position and slope:

```
$pl->text($string, TEXTPOSITION => [$x, $y, $dx, $dy, $just]);
```

$x, $y are the location of the string's reference point within the clipping box

$dx, $dy together indicate the slope along which the text is drawn

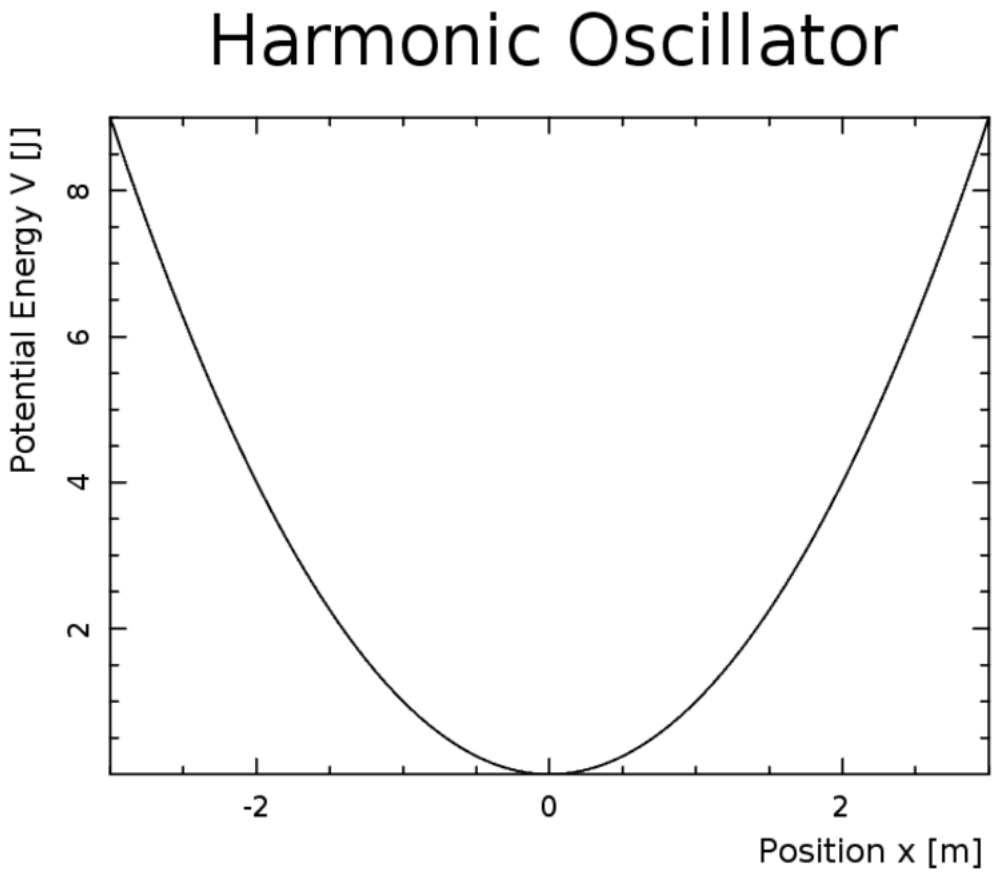
$just indicates the location of the reference point of the string.

0 means the reference point is the string's left edge; 1 indicates the right edge

TEXTPOSITION 3 argument form

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
        DEV => 'pscairo',
        FILE => 'text1.eps');

my $x = zeroes(100)->xlinvals(-3,3);
my $y = $x**2;
$pl->xyplot($x, $y);

$pl->setparm(CHARSIZE => 1.2);
# x label on the lower right
$pl->text('Position x [m]',
        TEXTPOSITION => ['b', 3, 1, 1]);
```

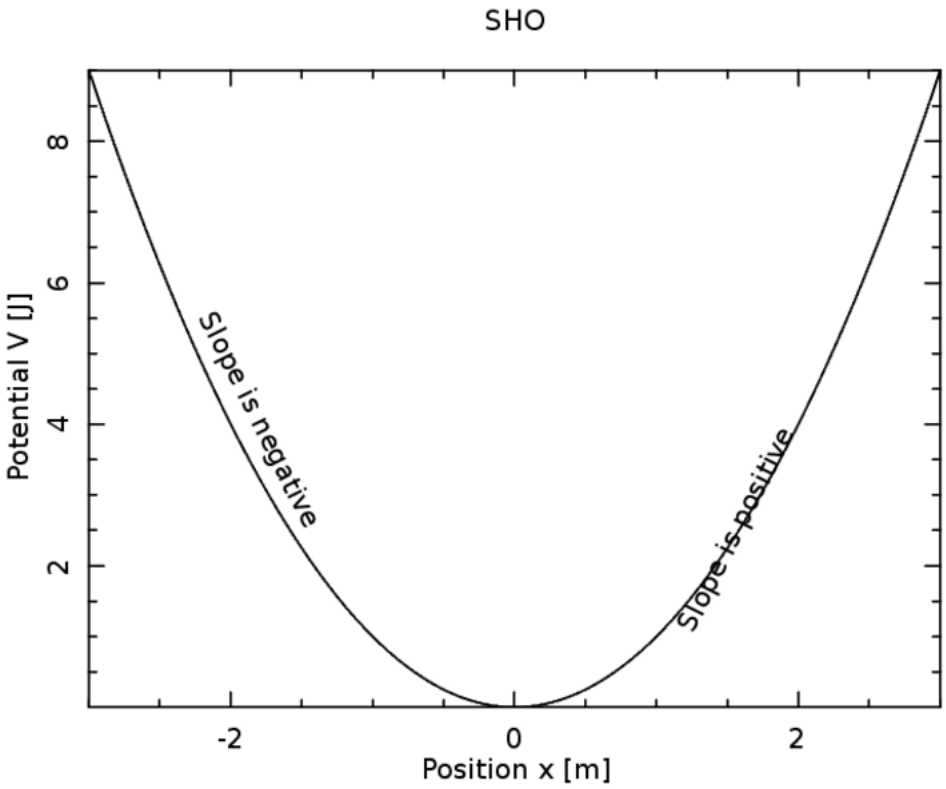
```
# y label on the upper left
$pl->text('Potential Energy V [J]',
         TEXTPOSITION => ['l', 3.5, 1, 1]);

# title at the center top
$pl->text('Harmonic Oscillator',
         CHARSIZE => 2.5,
         TEXTPOSITION => ['t', 1.5, 0.5, 0.5]);

$pl->close;
```



TEXTPOSITION 4 argument form

```
#!/usr/bin/perl
use strict;
use warnings;
use PDL;
use aliased 'PDL::Graphics::PLplot';

my $pl = PLplot->new(
       DEV => 'pscairo',
       FILE => 'text2.eps');

# Plot a quadratic
```

```
my $x = zeroes(100)->xlinvals(-3,3);
my $y = $x**2;

$pl->xyplot($x, $y, TITLE => 'SHO',
       XLAB => 'Position x [m]',
       YLAB => 'Potential V [J]');

# annotate negative slope at (-2, 4)
$pl->text('Slope is negative',
         TEXTPOSITION => [-1.8, 4.1, 1, -4, 0.5]);

# annotate positive slope at (2, 4)
$pl->text('Slope is positive',
         TEXTPOSITION => [1.9, 3.9, 10, 40, 1]);

$pl->close;
```

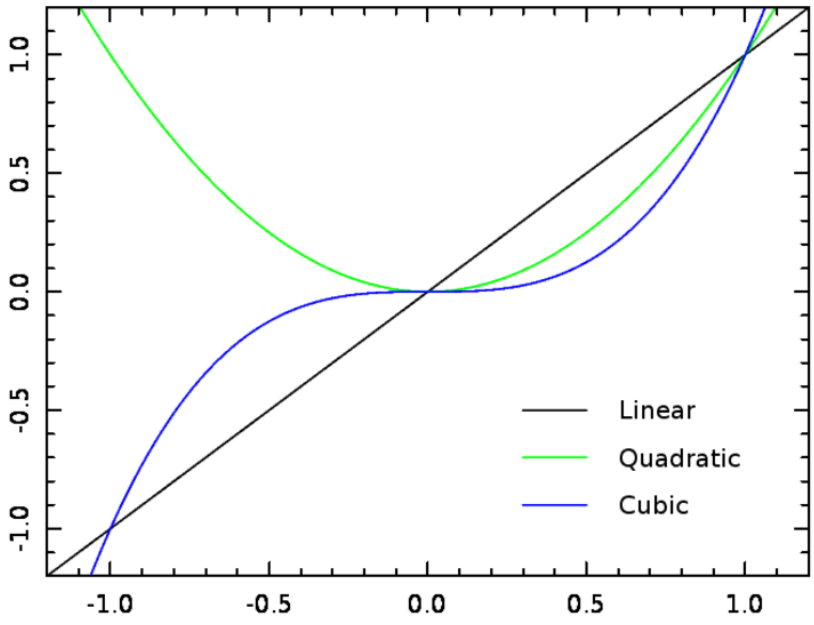


Legends

PLplot does not have a command to create legends. We must make them ourselves. Legends are only necessary when plotting discrete data sets. If possible, use color keys instead of constructing legends by hand.

```
#!/usr/bin/perl
use strict;
use warnings;
```

```
use PDL;
use PDL::Graphics::PLplot;

my $pl = PDL::Graphics::PLplot->new(
              DEV => 'pscairo',
              FILE => 'legend.eps');

my $x = zeroes(100)->xlinvals(-1.2, 1.2);
my @colors = qw(BLACK GREEN BLUE);
my @labels = qw(Linear Quadratic Cubic);
my $legend_x = pdl(0.3, 0.5);
my $legend_y = -0.5;

# Plot linear, quadratic, and cubic curves with a legend
 for my $i (0..2) {

 $pl->xyplot($x, $x**($i+1), COLOR => $colors[$i]);

 $pl->xyplot($legend_x, pdl($legend_y, $legend_y),
        COLOR => $colors[$i]);

 $pl->text($labels[$i], COLOR => 'BLACK',
        TEXTPOSITION => [0.6, $legend_y, 1, 0, 0]);

 $legend_y -= 0.2;
}

$pl->close;
```



3D Graphics with OpenGL

Introduction

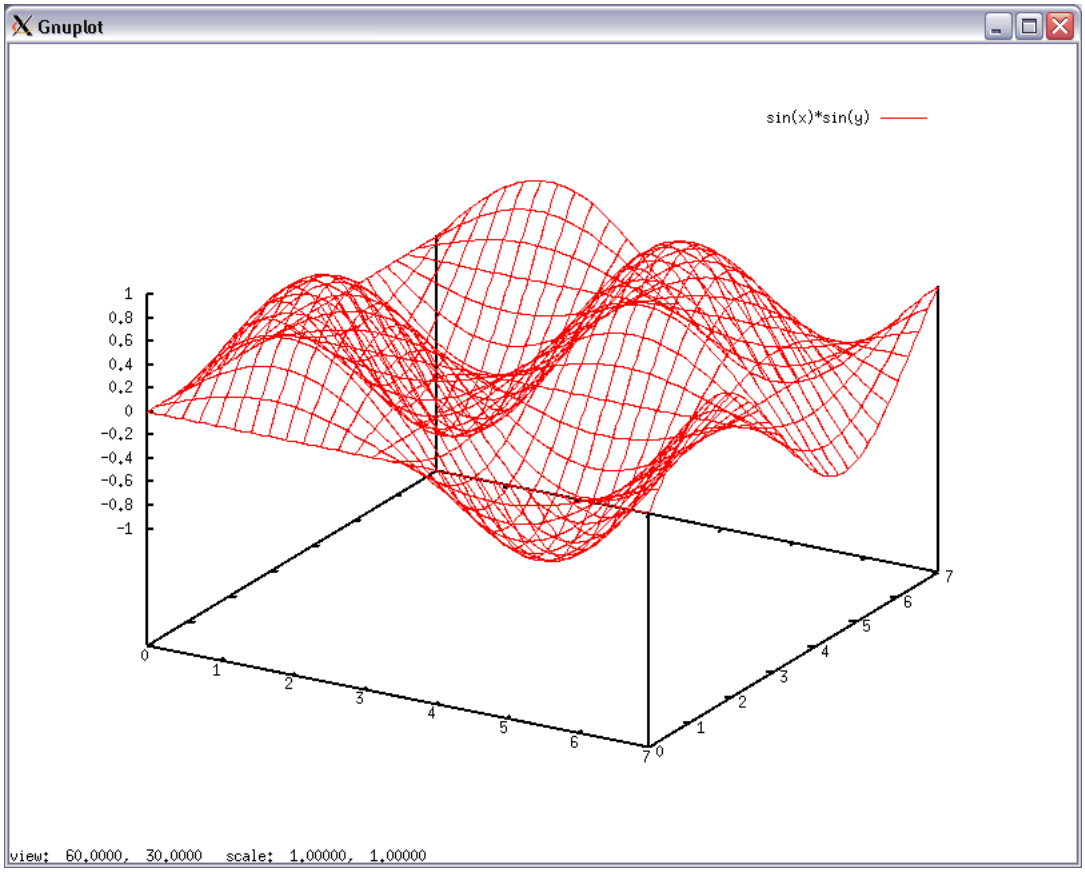


Figure 3.1: A 3D surface graph plotted using gnuplot, using the commands:

```
set isosamples 30; splot [0:7] [0:7] sin(x)*sin(y).
```

There are lots of programs that let you plot so-called 3D surface graphs, such as the one shown in Fig. 3.1. 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.

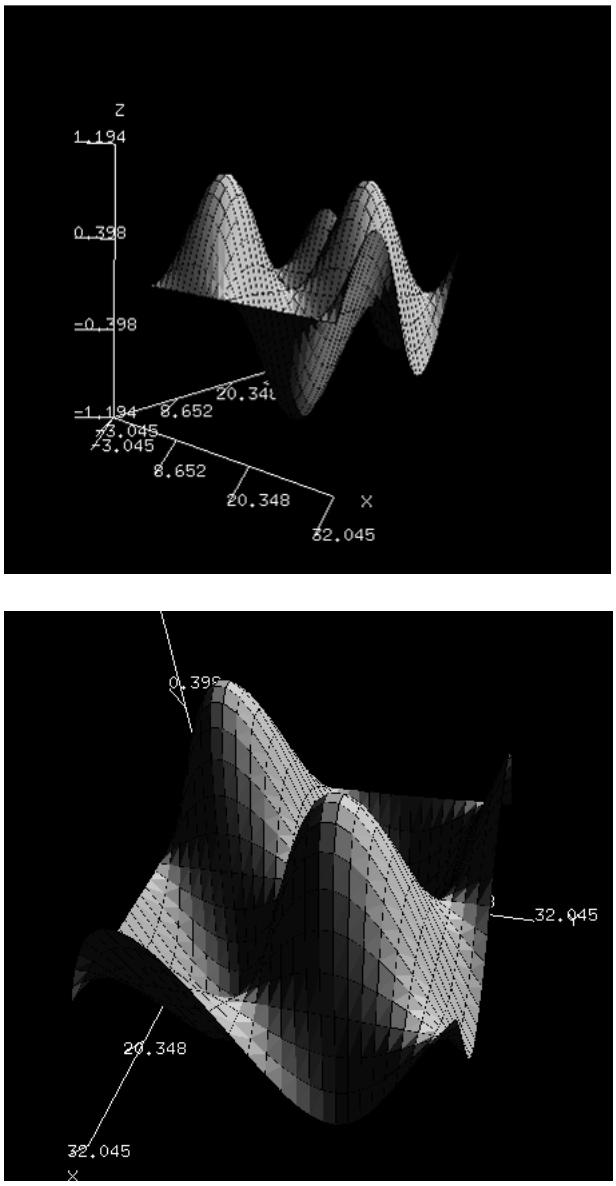


Figure 3.2: The same 3D surface, plotted using the PDL::TriD module. The two different images were obtained literally by grabbing the image in the window opened by PDL and dragging it with the mouse to rotate.

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

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

This should produce a new window with the image seen in Figure 3.2. 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 graphics library. (See the Perl Data Language web site at

http://pdl.perl.org for information on installing and using PDL.)

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.

```
pdl> use PDL::Graphics::TriD;
```

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

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

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

```
pdl> 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,

```
pdl> 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. (See chapter [chap_slice] for some 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.

Parametric Graphics

We alluded in the introduction that allowing

```
pdl> imag3d $piddle;
```

could be ambiguous and should be written

```
pdl> 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

```
pdl> imag3d $piddle;
```

is in fact legal code---if and only if 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 shape `[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`.

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;
use PDL::Graphics::TriD;
use PDL::NiceSlice;

$torus = zeroes(3, 60, 20);

$x = $torus((0));
$y = $torus((1));
$z = $torus((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 { $torus(($_)) } 0..2;
```

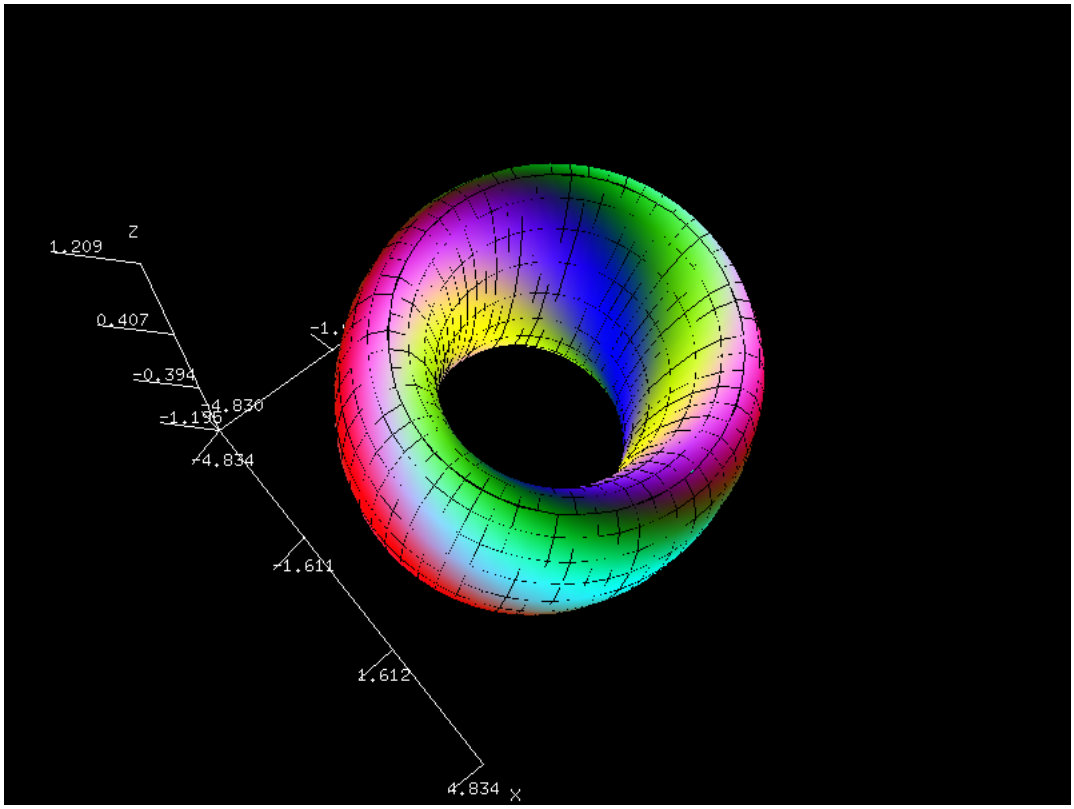
Next, we color the torus. Let's put stripes on it:

```
$r = (1 + sin(2*$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.

```
$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 here'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_ns [$x, $y, $z], [$r, $g, $b];
```

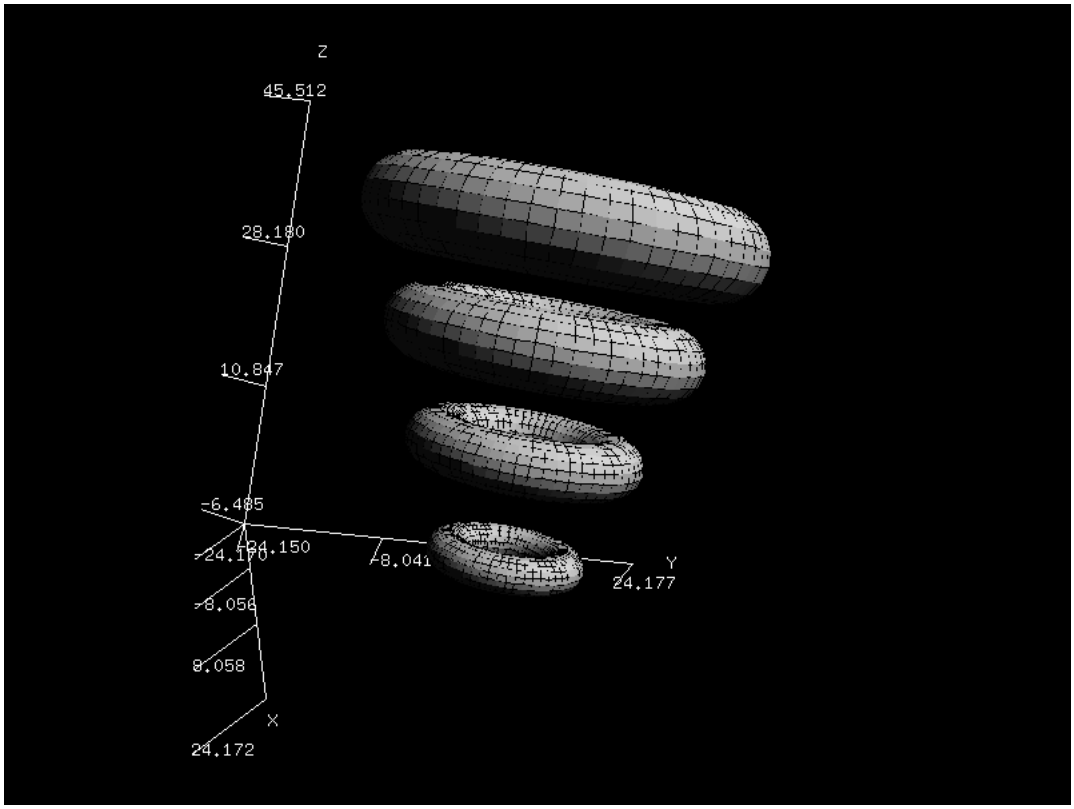
which will produce the same results. Also, we could concatenate the RGB piddles to form a single `[3,60,20]` piddle that could be used without square brackets:

```
$rgb = cat($r,$g,$b)->mv(-1,0);  # $rgb is [3,60,20]
imag3d_ns $torus, $rgb;
```

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 shape `[3,n_t,n_u,...]` where the three periods in the end indicate the beginning of the extra parameters.

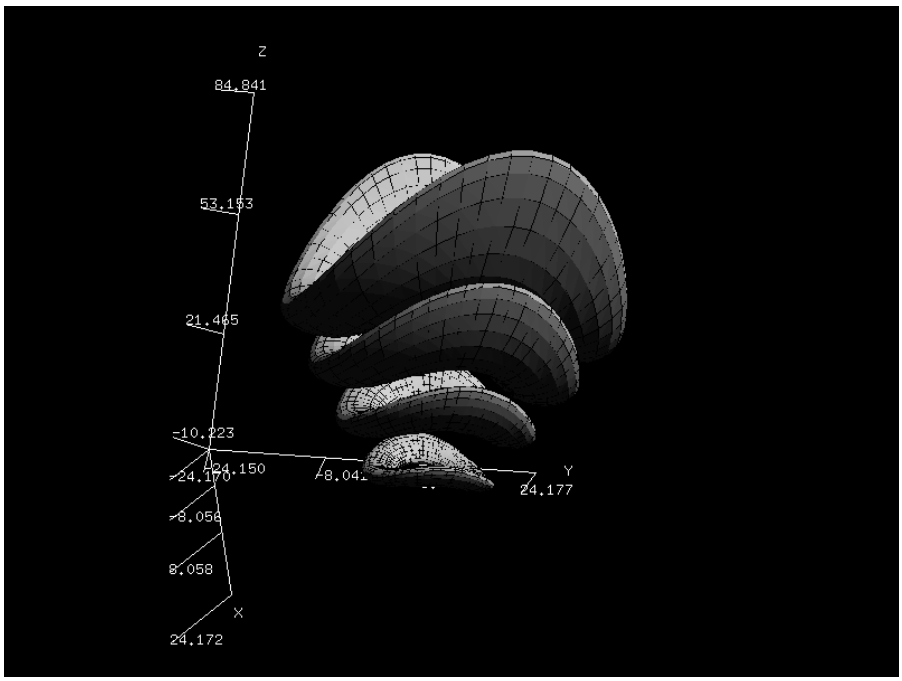
For example, we can plot a family of shrinking toruses by adding an extra dimension into `$torus`:

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



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

```
$x = $cone(0);
$cone(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.

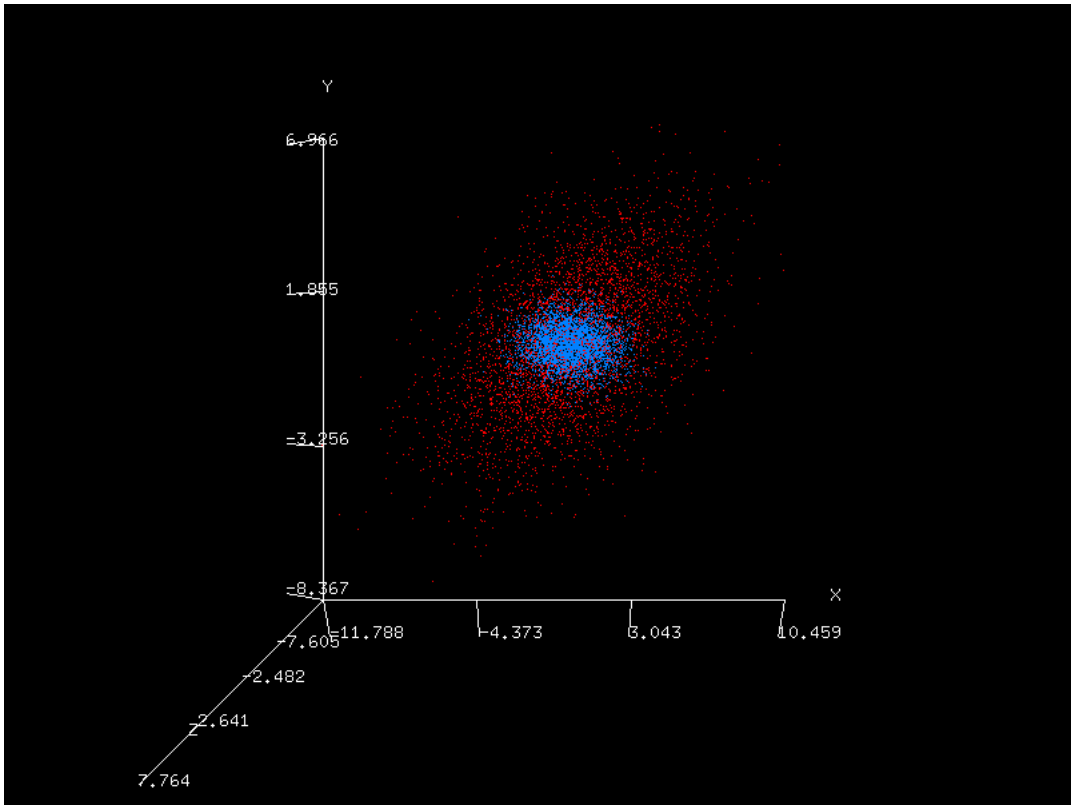
Of course, since PDL tries to follow "simple things simple, complicated things possible", it is possible to override the default context.

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];
```

And the result:



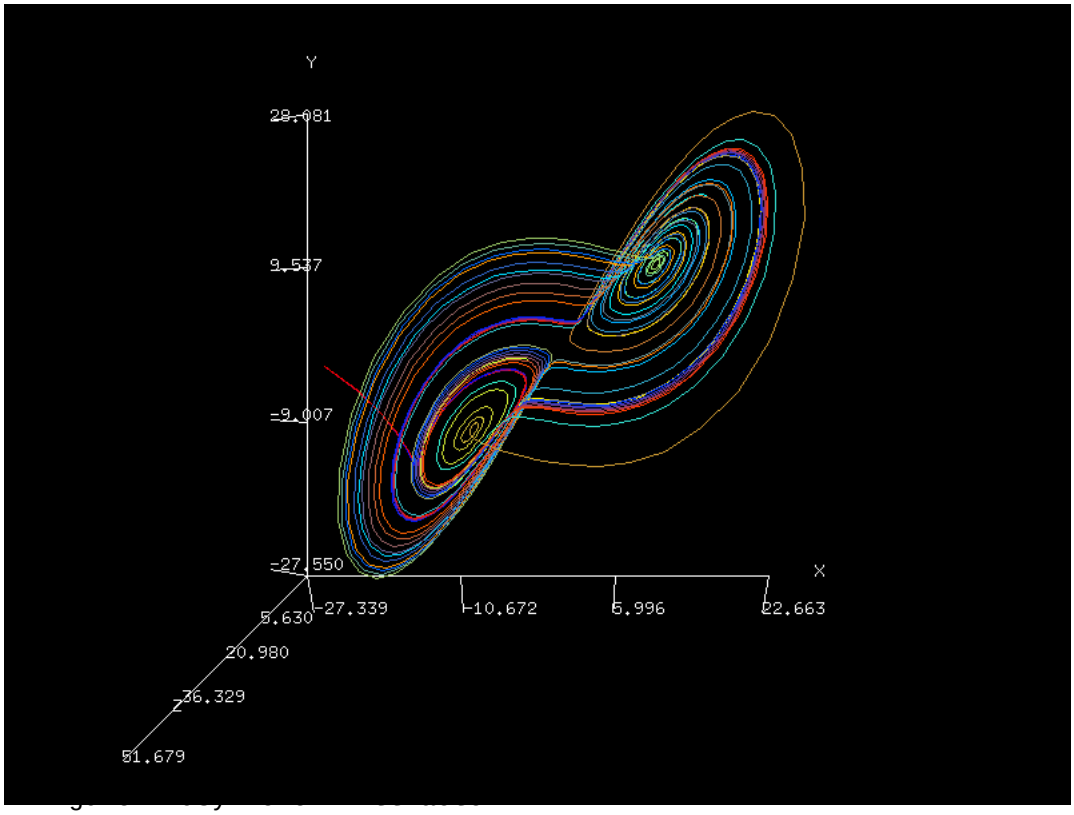
A lot of fun things can be done with points but we'll go into that later.

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

```
dx
--    =    sigma (y - x)
dt

dy
--    =    (r - z) x  -  y
dt

dz
--    =    (y - b) z
dt
```

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.

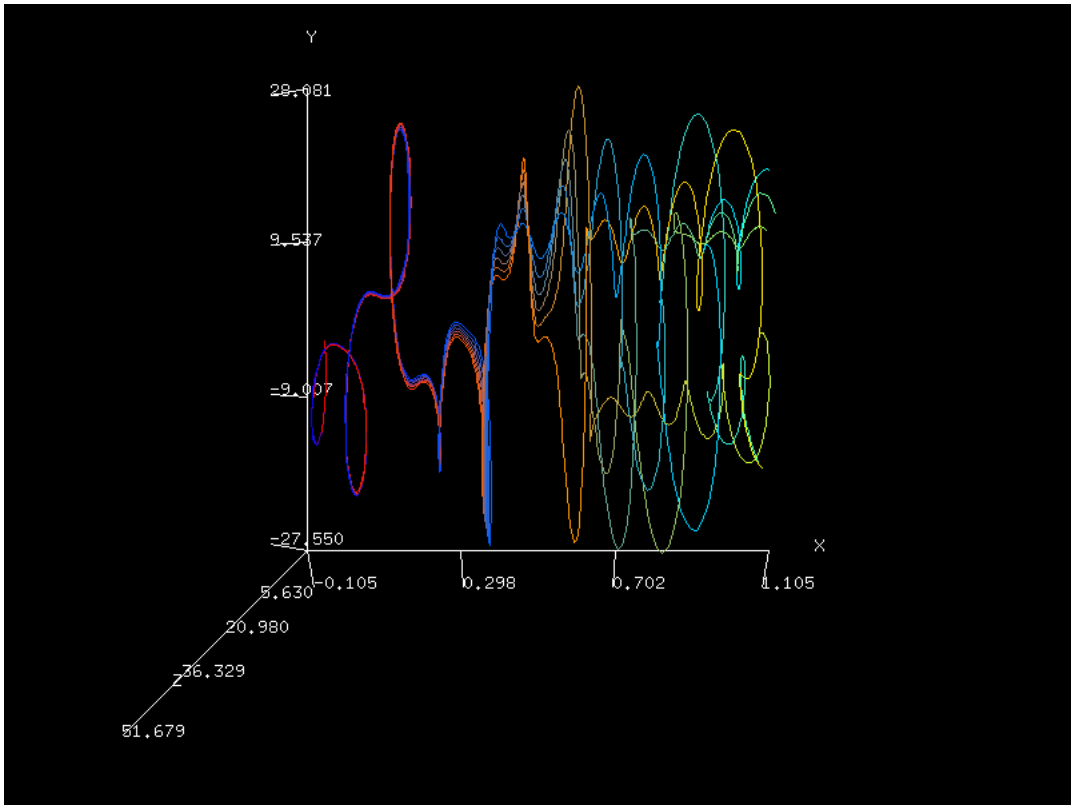
```
use PDL::Graphics::TriD;
$n = 500;
$nstart = 0;
$nc = 6;
$delta = 0.015;
# $x = pdl(1, 1, 1, 1, 1);
# $y = pdl(1, 1, 1, 1, 1);
# $z = pdl(1, 1.01, 1.02, 1.03, 1.04);
$xs = zeroes($n, $nc);
$ys = zeroes($n, $nc);
$zs = zeroes($n, $nc);
$x = -23 * ones($nc);
$y = -2 * ones($nc);
$z = 20 * ones($nc)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

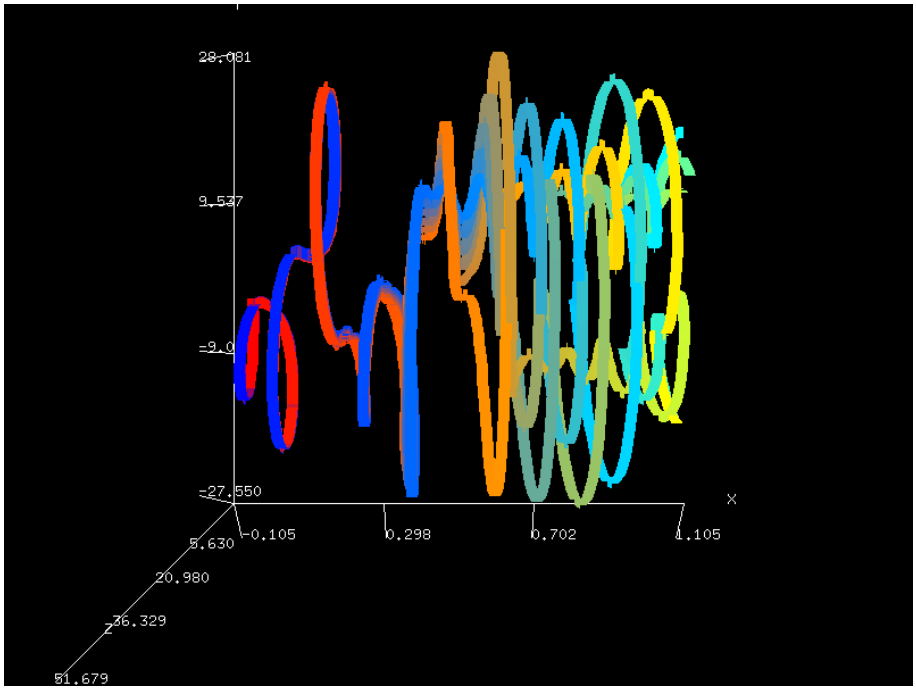
This yields a much clearer plot of the chaotic behavior when the lines diverge with time.



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. There is also a command `mesh3d` similar to the `imag3d` 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 <-- 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.

More than one Image

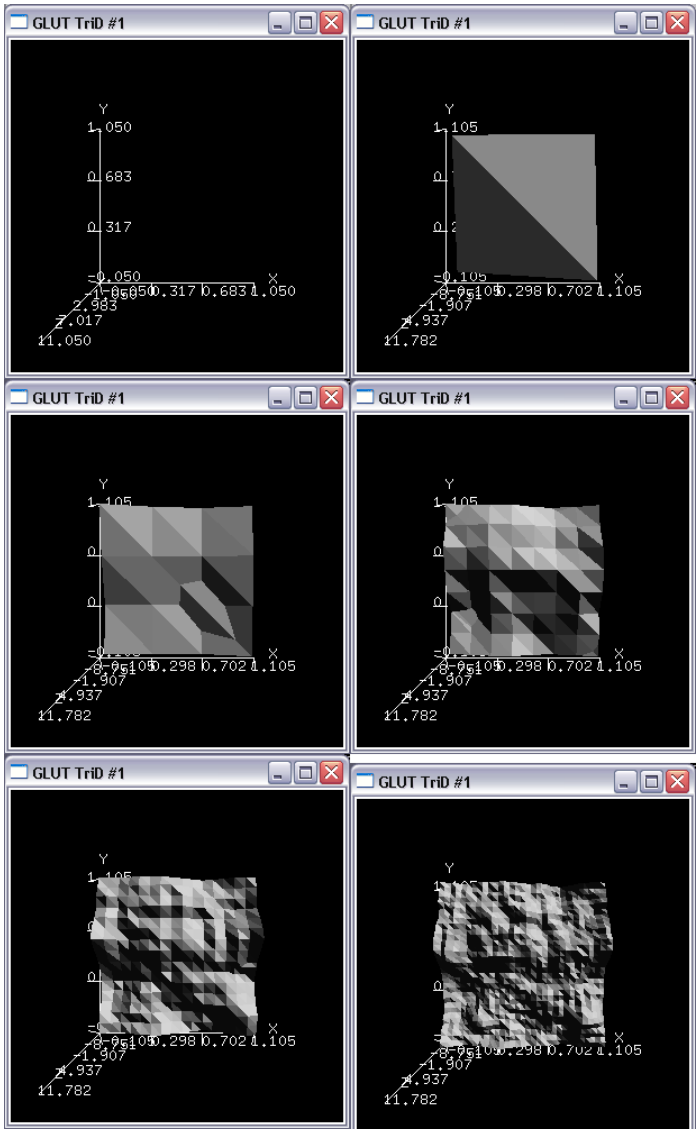
If you have used the PDL PGPLOT interface for plotting multiple graphs then TriD is not going to surprise you: the commands `hold3d` and `release3d` work just like their PGPLOT counterparts. Before going further, however, let me remind you that for many plots, it is not necessary to explicitly plot several 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).

```
use PDL; # XXX FIX - LOOKS BAD.
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 5x5 kernel to smooth our data at each step by convolution. That's the numerical part, now here is the sequence of images created:



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. 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);
}
```

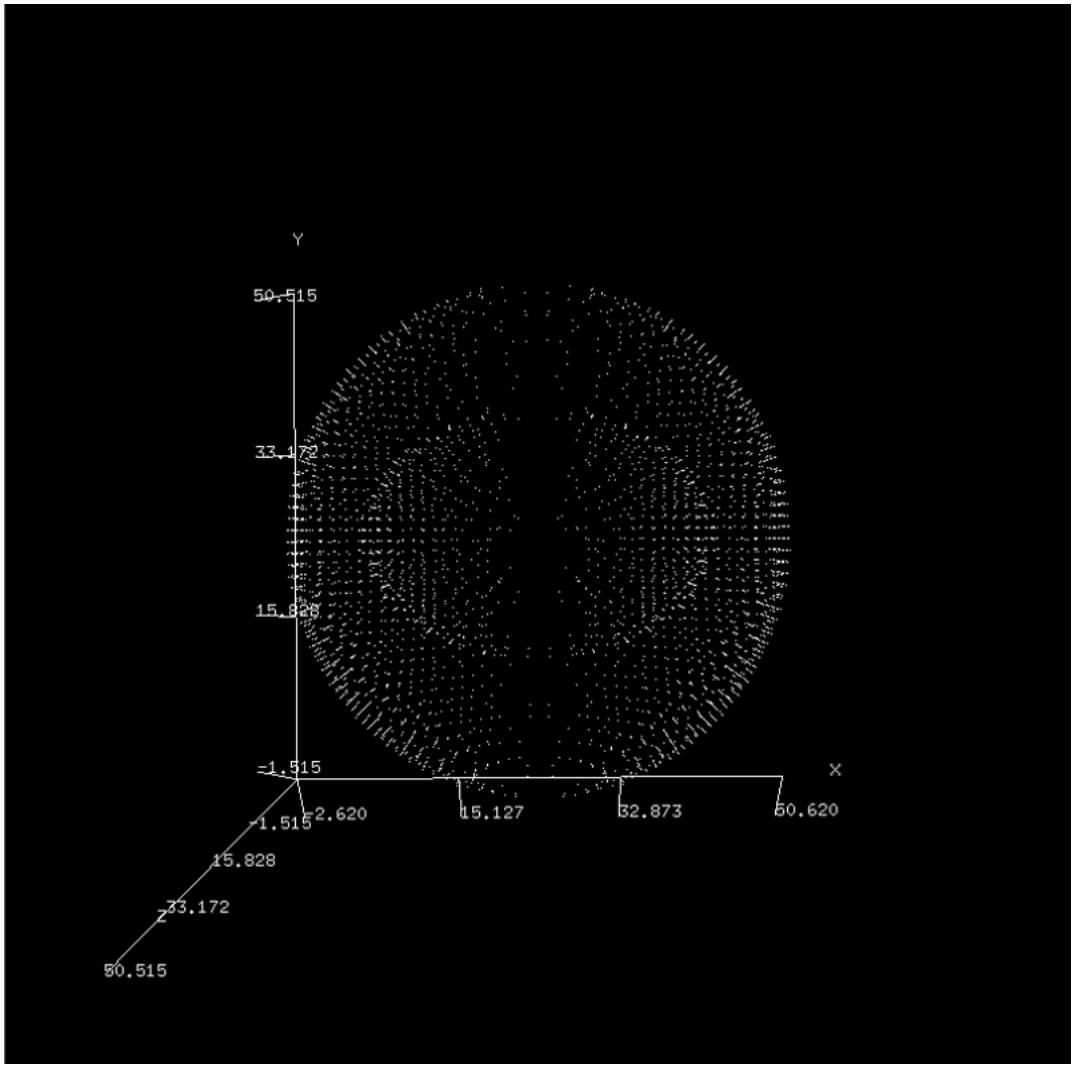
Note here that we can't use the += operator for `$y` since below we use the same piddle for the three coordinates (with a simple dummy transformation). Now, we want to picture approximately where the function crosses zero, but since there are two separate zero surfaces we can't just use an algorithm 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 50x50x50 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(0:-2) != $sign(1:-1));
points3d whichND($df);  # for PDL-2.4.10
```

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.

NOTE: For PDL-2.4.9 and earlier, you'll need to use `points3d [ whichND($df) ];` since previous to PDL-2.4.10 `whichND` returned a list of piddles in list context. That behavior is now deprecated.

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:

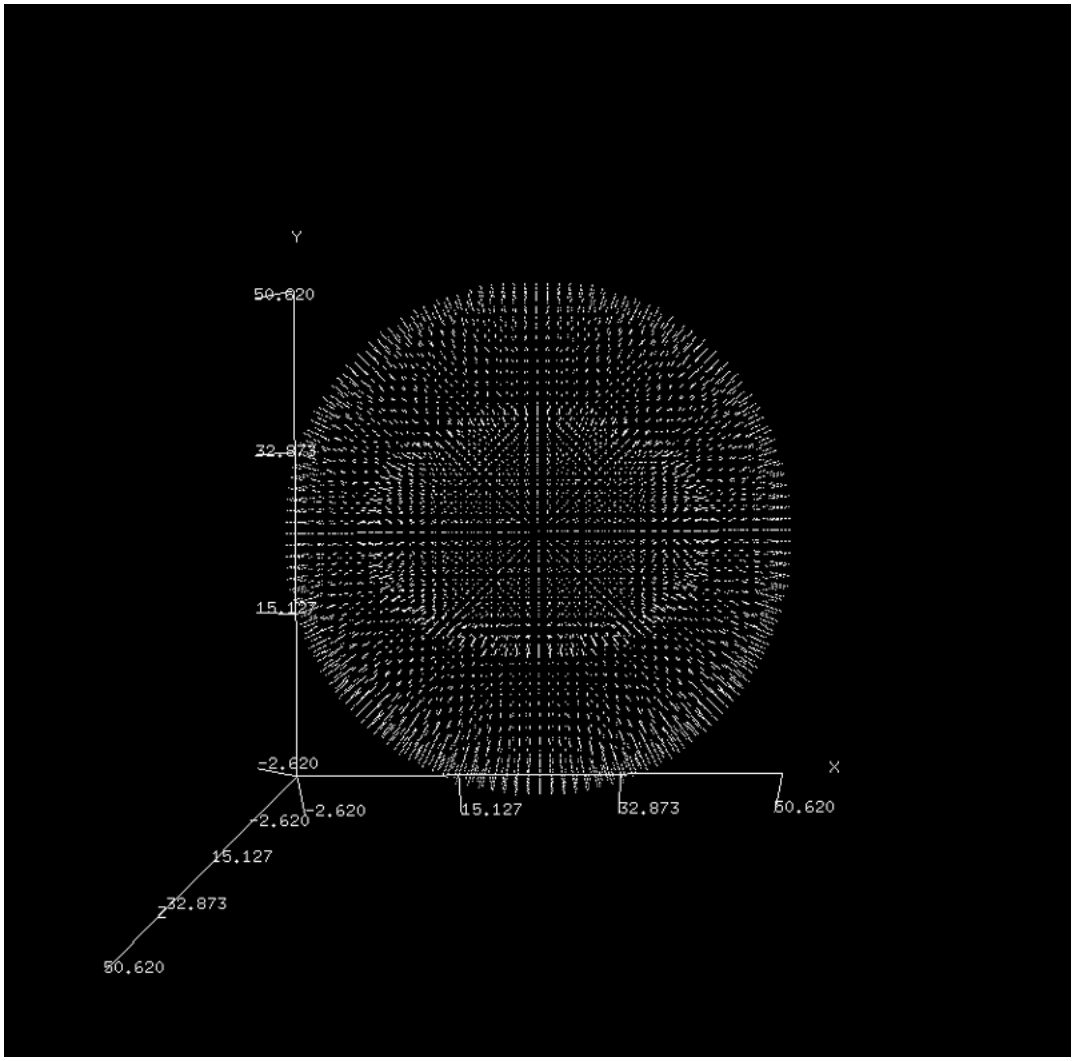
```
$a = $sign;
foreach (1,2,4) {
   $t=($a(0:-2)<<$_);
   $t+=$a(1:-1);
   $a=$t->mv(0,2);
}
points3d [whichND(($a != 0) & ($a != 255))];
```

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.

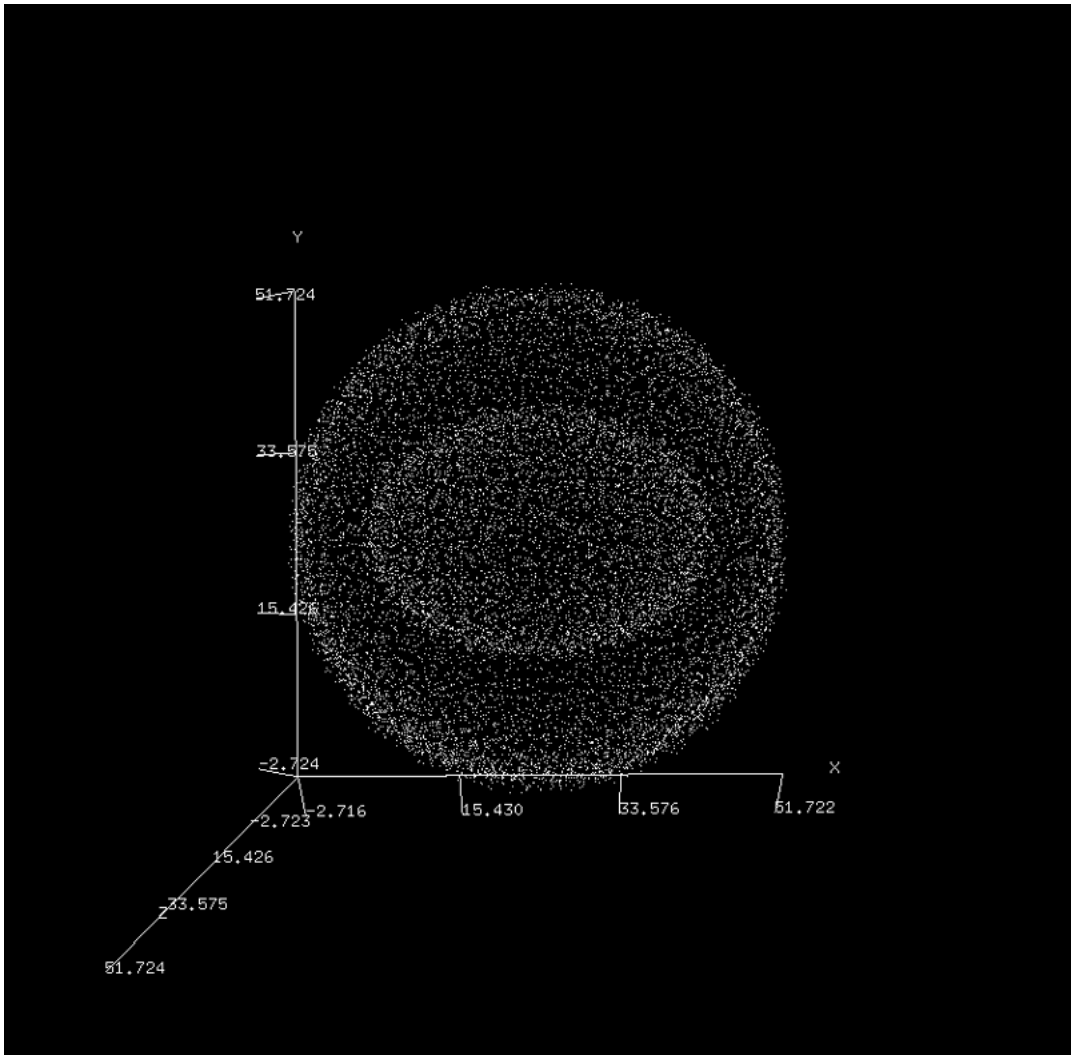
Choosing the shifts to be 1,2,4 is the key: this way after the first round, the piddle contains values 0,1,2,3 after the second it contains 0...15 and after the third, 0..255. None of the shifts shift anything on top of each other so the plus operation could be replaced with a bitwise or.

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)))
```



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* this topic away and move to the next one.

```
   * Keep It Simple, Stupid
=head1 The PDL Preprocessor
```

The PDL PreProcessor, or PDL::PP, is PDL's secret weapon. With PDL::PP, you can quickly and easily implement new "primitive" compiled C-language PDL functions that follow the PDL threading rules, without having to write tedious loops or glue code. You can write simple computations with zero or more active dimensions (see *PDL::Book::Threading*), write functions that contain a mix of Perl and compiled code, and/or generate output PDLs that remain linked to the source PDL in trivial or nontrivial ways.

The PDL::PP module is a preprocessor that accepts a metalanguage ("PP") and emits both Perl and XS code. PDL::PP is not generally invoked directly by you, the coder, at run time -- it is invoked as part of your module's build process (via *ExtUtils::MakeMaker* or *Module::Build*) or by Inline::Pdlpp as part of inline compilation of snippets of PP. I will use the latter case throughout this documentation as it allows me to give full copy-and-paste examples.

Note that the vast majority of these examples are tested and should work by simply pasting them

directly into a text editor. The only correction you will need to make is to ensure that the __END__ and __Pdlpp__ markers are flush against the left edge, i.e. there are no spaces before the underscores.

After reading this introduction, you should have a firm grasp on the basics of using PDL::PP and the full documentation in the *PDL::PP man page* should be fairly easy to follow.

Basics

In this section I discuss the basics of writing PP code using `pp_def`. I will use *Inline::Pdlpp* for all of my examples, including this first one. If you need help getting *Inline::Pdlpp* to work, see Appendix A.

The contents of the *Inline::Pdlpp* is no more than a Perl script that calls special functions defined in the *PDL::PP* module. The final result of this Perl script are a Perl module (.pm file) and a Perl extension (.xs file). The latter gets expanded to C code and compiled to produce XSUBs that ultimately end up as methods in the PDL package.

`pp_def` accepts a collection of parameters that describe both the way the new method should interact with the threading engine (e.g. its dimensional signature and which data types it should support natively), and also the code for the core of the method.

First Example

Let's begin with a variation on the canonical Hello World.

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';
my $a = sequence(10);
$a->printout;

__END__

__Pdlpp__

pp_def('printout',
    Pars => 'a()',
    Code => q{
        printf("%f\n", $a());
    },
);
```

If you run that script, after a short pause you should see output that looks like this:

```
> perl my_script.pl
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
```

During that pause, Inlne took the text below the __Pdlpp__ marker and sent it off to *Inline::Pdlpp*,

which generated a source file and a Makefile. Inline took it from there, compiling the function and then loading the newly compiled module into your current Perl interpreter. That module defined the function `PDL::printout`, which the script ran a couple of lines below the `use Inline 'Pdlpp'`. The cool part about Inline is that it caches the result of that build process and only rebuilds if you change the part below the __Pdlpp__ marker. You can freely play with the Perl part of the file and it will use the same cached Pdlpp code. Now that you understand what Inline did, let's take a closer look at how I actually defined the `printout` function.

PDL::PP is a Perl module that you use to **generate** the XS and Perl code for your PDL functions. This means that everything below the __Pdlpp__ marker is actually a plain Perl script, except that you don't need to `use PDL::PP` because *Inline::Pdlpp* took care of that for you.

In order to generate your XS code, you call one of the many functions defined in PDL::PP. All of these are discussed in the PDL::PP documentation, and in this chapter I will focus entirely on PDL::PP's workhorse: `pp_def`. In the above example, the code of interest is this:

```
pp_def('printout',
    Pars => 'a()',
    Code => q{
        printf("%f\n", $a());
    },
);
```

The first argument to `pp_def` is the name of the function you want to create. After that, you pass a number of key/value pairs to tell PDL::PP precisely what sort of function you are trying to create. The bare minimum for a normal computational function (as opposed to a slice function, for which there is sadly no documentation) is the `Pars` key and the `Code` key.

The `Pars` key specifies the **piddle** arguments for your function. It accepts a simple Perl string with the argument names and dimensions, delimited by semicolons. In the example I only use a single argument, but you can specify multiple input and output arguments, and you can even restrict (that is, force a coercion in) their data types. Note that the parentheses that follow the a are important and cannot be omitted. They might make the statement look like a function, but we'll see soon why they are important.

The `Code` key specifies a Perl string with a quasi-C block of code that I am going to call PP code. This Perl string gets thoroughly transformed by PDL::PP and combined with other keys to produce the XS (and eventually C) code for your function. You can think of PP code as being regular C code with a few special macros and notations. The first example already demonstrates one such notation: to access the value in a piddle, you must prefix the name with a dollar-sign and you must postfix it with parentheses. In the next section we'll see just what sort of arguments you can put in those parentheses.

Best Practice: Use q{ } for Code Sections

When creating a string for the Code key (as well as the BadCode, BackCode, and BadBackCode keys), I strongly recommend that you use Perl's q quote operator with curly braces as delimiters, as I have used in the examples so far. Perl offers many ways to quote long blocks of text. Your first impulse may be to simply use normal Perl quotes like so:

```
Code => ' printf("%f\n", $a()); ',
```

For longer lines, you would probably pull out the ever-useful heredoc:

```
Code => <<EOCode,

    printf("%f\n", $a());
```

```
EOCode
```

I have two reasons for recommending Perl's q operator. First, it makes your Code section look like a code block:

```
Code => q{
    printf("%f\n", $a());
}
```

Second, PDL::PP's error reporting is not the greatest, and if you miss a curly brace, Perl's **interpreter** will catch it as a problem. This is not the case with the other delimiters. In this example, I forgot to include a closing brace:

```
Code => <<'EOCode',
    printf("Starting\n");

    for(i = 0; i < $SIZE(n); ++i) {
        printf("%d: %f\n", i, $a(n => i));

    printf("All done\n");
EOCode
```

The C compiler will croak on the above example with an error that is likely to be obscure and only tangentially helpful. However, Perl will catch this typo at compile time if you use `q{ }`:

```
Code => q{
    printf("Starting\n");

    for(i = 0; i < $SIZE(n); ++i) {
        printf("%d: %f\n", i, $a(n => i));

    printf("All done\n");
},
```

Also note that I do not recommend using the `qq` quoting operator. Almost all the PDL::PP code strings delimit piddles using dollar-signs (like `$a( )` above) and you must escape each one of these unless you want Perl to interpolate a variable for you. Obviously `qq` has its uses occasionally, but in general I recommend sticking almost exclusively with `q`.

Let's now expand the example so that the function takes two arguments. Replace the original `pp_def` with this slightly more interesting code:

```
pp_def('printout_sum',
    Pars => 'a(); b()',
    Code => q{
        printf("%f + %f = %f\n", $a(), $b(), $a() + $b());
    },
);
```

Change the line that reads

```
$a->printout;
```

to the following two lines:

```
my $b = $a->random;
$a->printout_sum($b);
```

and you should get output that looks like this:

```
> perl two-args.pl
0.000000 + 0.690920 = 0.690920
1.000000 + 0.907612 = 1.907612
2.000000 + 0.479112 = 2.479112
3.000000 + 0.421556 = 3.421556
4.000000 + 0.431388 = 4.431388
5.000000 + 0.022563 = 5.022563
6.000000 + 0.014719 = 6.014719
7.000000 + 0.354457 = 7.354457
8.000000 + 0.705733 = 8.705733
9.000000 + 0.827809 = 9.827809
```

The differences between this and the previous example are not complicated but deserve some discussion. A cosmetic difference is that I have used a different name for the function, but a more substantial difference is that the function now takes two arguments, `a()` and `b()`, as specified by the `Pars` key. The `Code` block makes use of these two piddles, printing out the two and their sum. Notice that I access the value in a with the expression `$a()`, and the value in b with `$b()`. Also notice that I can use those values in an arithmetic expression.

Returning Values

The examples I have used have all demonstrated their behavior by printing out their results to STDOUT. If you are like me, you will be glad to know that you can use printfs throughout your PP code when it comes time to debug, but these functions would be far more useful if they returned piddles with the calculated results. Fortunately, PDL::PP functions are really just C functions in disguise, and ultimately the data are passed around in C arrays, essentially by reference. This means that you can modify incoming piddles in-place. For example, this function increments a piddle:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';
my $a = sequence(10);
print "a is initially $a\n";
$a->my_inc;
print "a is now $a\n";


__END__
__Pdlpp__
pp_def('my_inc',
    Pars => 'a()',
    Code => q{
        $a()++;
    },
);
```

When I run that, I get this output:

```
a is initially [0 1 2 3 4 5 6 7 8 9]
a is now [1 2 3 4 5 6 7 8 9 10]
```

If you want to modify a piddle in-place, PDL provides multiple mechanisms for handling this, depending on what you are trying to accomplish. In particular, there are ways to handle the `inplace`

flag for a given piddle. But I'm getting a bit ahead of myself. Generally speaking, you shouldn't modify a piddle in-place: you should return a result instead. To do this, you simply mark the argument in the `Pars` key with the `[o]` qualifier. Here, I show how to return two arguments:

```
pp_def('my_sum_and_diff',
    Pars => 'left(); right(); [o] sum(); [o] diff()',
    Code => q{
        $sum() = $left() + $right();
        $diff() = $left() - $right();
    },
);
```

This function takes `$left` and `$right` as input arguments (in that order) and it outputs `$sum` and `$diff` (also in that order, as a Perl list). For example, we could run the above pp-code with Perl code like this:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';
my $left = sequence(10);
my $right = $left->random;

my ($sum, $diff) = $left->my_sum_and_diff($right);

print "Left:  $left\n";
print "Right: $right\n";
print "Sum:   $sum\n";
print "Diff:  $diff\n";
```

The functions defined using `pp_def` actually allow for you to pass in the output piddles as arguments, but I'll explore that in one of the exercises rather than boring you with more details.

Exercise Set 1

So far I have shown you how to write basic PP code that prints values to the screen or returns values. The great thing about PDL::PP is that this code actually allows for two different calling conventions, and it Does What You Mean when you give it all manner of piddles. Rather than bore you to death with more prose, I am going to give you a couple of exercises. Solutions to these exercises are in Appendix B.

1. Slices

   Working with `printout_sum`, replace `$b` with a **slice** from some other piddle. Does it do what you expect?

2. Threading

   With `printout_sum`, what if you replace `$b` with a two-dimensional piddle that is thread-compatible with `$a`? Try to guess the order of the output that you'll get before running the example. Did you guess correctly?

3. Orthogonal Piddles

   What if `$a` has dimensions M and `$b` has dimensions (1, N) with `printout_sum`? What about `my_sum_and_diff`?

4. Varying Input Order

The PP code that I present puts all the output piddles at the end of the `Pars` section. What happens if you move them to the beginning of the section instead of the end?

5. Supplying Outputs in the Function Call

   You can call `pp_defined` functions by supplying all the arguments to the function. For example, instead of calling `my_sum_and_diff` like this:

   ```
   # No output piddles in function call
   my ($sum, $diff) = $left->my_sum_and_diff($right);
   ```

   you can call it like this:

   ```
   # All in function call, both outputs null
   my ($sum, $diff) = (PDL::null, PDL::null);
   $left->my_sum_and_diff($right, $sum, $diff);
   ```

   What is the return value of this sort of invocation? How does the function call change if you alter the `Pars` order? There's a good reason for this capability, can you guess why PDL lets you do this?

Higher Dimensional Functions

So far I have shown how to write rudimentary functions that accept zero-dimensional piddles. In this section, I will explain how to write functions that accept higher-dimensional data.

Specifying Dimensions and Using Explicit Looping

Exercises 1.2 and 1.3 demonstrate that PDL::PP automatically loops over the values in a piddle for you. What if you want to do some sort of aggregate behavior, such as computing the sum of all the values in a piddle? This requires more fine-grained control of the code over which PDL::PP loops.

Our discussion begins by looking more closely at the `Pars` key. When you have a parameter list like `'input(); [o] output()'`, you are telling PDL::PP that you want it to present the data from the input and output piddles as scalars. The code you specify in the `Code` key gets wrapped by a couple of C `for` loops that loop through higher dimensions, something that we call *threading*. There are many calculations you cannot do with this simplistic representation of the data, such as write a Fourier Transform, matrix-matrix multiplication, or a cumulative sum. For these, you need PDL::PP to represent your data as vectors or matrices.

Note: I am about to cover some material that makes sense once you get it, but which is very easy to mis-interpret. Pay close attention!

To tell PDL::PP that you want it to represent the data as a vector, you specify a *dimension name* in the `Pars` key, such as

```
Pars => 'input(n); [o] sum()'
```

Notice that I have put something within the parentheses of the input piddle, n. That means that I want PDL::PP to represent the input as a vector with one dimension and I am going to refer to its (single) dimension by the name n. Then, to access the third element of that vector, you would write `$input(n => 2)`. (Element access uses zero-offsets, just like Perl and C array access.) To sum all the values in the vector and store the result in the output variable, you could use a C for-loop like so:

```
int i;
$sum() = 0;
for (i = 0; i < $SIZE(n); i++) {
    $sum() += $input(n => i);
}
```

Here, `$SIZE(n)` is a PDL::PP macro that returns the length of the vector (or more precisely, the size of the dimension that we have called n).

Best practice: optimize for clarity when using $SIZE

When I first encountered the `$SIZE` PDL::PP macro, I assumed it produced slow code. It turns out that it replaces itself with a direct variable access, which is quite fast. As a general rule regarding `$SIZE`, optimize for clarity. The only exception is that, as of this writing, you **cannot** use `$SIZE` within a direct memory access, as I discuss next.

Wart: no parenthesized expressions within direct memory access

Due to a current limitation in PDL::PP, you cannot use parenthesized expressions within a memory access. For example, this will fail to compile and will throw a most obscure error:

```
$sum() += $input(n => (i-1));
```

The reason is that the parser isn't a real parser: it's just a series of regular expressions. It takes everything up until the first closing parenthesis and doesn't realize that you put `i-1` in parentheses. This means that these also fail:

```
$sum() += $input(n => calculate_offset(i));
$sum() += $input(n => $SIZE(n)-1);
```

You can use expressions that do not involve parentheses, even expressions involving arithmetic, so you can achieve the same ends with these work-arounds:

```
long calc_off = calculate_offset(i);
$sum() += $input(n => calc_off);

long N = $SIZE(n);
$sum() += $input(n => N-1);
```

I intend to improve this soon so that at least parenthesized expressions will work in memory access statements. However, fixing access statement parsing to allow `$SIZE(n)` may require a more substantial overhaul of the parser and may not happen any time soon. Sorry.

PDL::PP also provides a convenient short-hand for this sort of loop:

```
$sum() = 0;
loop (n) %{
    $sum() += $input();
%}
```

Here, I declare a PDL::PP loop block. Standard blocks in C (and in Perl) are delimited with curly braces, but the loop block is delimited with `%{` and `%}`. You end up with code that is functionally identical to the previous method for writing this sum, but you can use fewer keystrokes to do it.

Putting this all together, here is a complete example that performs a sum over a vector:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';
my $a = sequence(10);
print "a is $a and its sumover is "
    , $a->my_sumover, "\n";

my $b = sequence(3, 5);
```

```
print "b is $b and its sumover is "
    , $b->my_sumover, "\n";

__END__

__Pdlpp__

pp_def('my_sumover',
    Pars => 'input(n); [o] sum()',
    Code => q{
        $sum() = 0;
        loop (n) %{
            $sum() += $input();
        %}
    }
);
```

That gives the following output:

```
a is [0 1 2 3 4 5 6 7 8 9] and its sumover is 45
b is
[
 [ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]
 [12 13 14]
]
 and its sumover is [3 12 21 30 39]
```

As the calculation on $a shows, when you perform the calculation on a one-dimensional piddle, it returns a single result with the sum of all the elements. The calculation on $b treats each row as a vector and performs the calculation on each row.

Matrix-Matrix Multiplication

Let's look at another example, matrix-matrix multiplication. (You remember how to do matrix-matrix multiplication, right? No? Brush-up on *http://en.wikipedia.org/wiki/Matrix_multiplication*.) How would we write such an algorithm using PDL::PP? First, the `Pars` section needs to indicate what sort of input and output piddles we want to handle. The length of the row of the first matrix has to be equal to the length of the column of the second matrix. The output matrix will have as many rows as the second matrix, and as many columns as the first matrix. Second, we need to loop over the entire output dimensions. Altogether, my first guess at this function looked like this:

```
pp_def('my_matrix_mult',
    Pars => 'left(n,m); right(m,p); [o] output(n,p)',
    Code => q{
        loop (n) %{
            loop (p) %{
                loop (m) %{
                    $output() = $left() * $right();
                %}
            %}
        %}
    },
```

```
);
```

"Wait," you say, "That's it? It's that simple?" Yep. Once you figure out the relationship of the dimension sizes, the threading engine just Does What You Mean. (As you'll see, I got the dimensions wrong, but it'll be a quick fix.) You can run that with this Perl code:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';
my $left = sequence(2,4);
my $right = sequence(4, 5);
print "$left times $right is ", $left->my_matrix_mult($right);
```

and that gives this output:

```
[
 [0 1]
 [2 3]
 [4 5]
 [6 7]
]
 times
[
 [ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]
 [16 17 18 19]
]
 is
[
 [ 18  21]
 [ 42  49]
 [ 66  77]
 [ 90 105]
 [114 133]
]
```

Oops! You can see that PDL considers the first argument to the number of columns, not the number of rows! I'll let you fix that in an exercise.

Threadloops

PDL::PP also has the `threadloop` construct, which lets you declare the code over which PDL should thread, and the code that should come before and after the thread loop. Here's a simple example demonstrating the `threadloop` construct in conjunction with the `loop` construct:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';


# Run the code on a 2x4 matrix:
sequence(2,4)->my_print_rows;
```

```
# Run the code on a 3x4x5 matrix:
sequence(3,4,5)->my_print_rows;

__END__

__Pdlpp__

pp_def('my_print_rows',
    Pars => 'in(n)',
    Code => q{
        printf("About to start printing rows\n");
        int row_counter = 0;
        threadloop %{
            printf("  Row %3d: ", row_counter);
            loop(n) %{
                printf("%f, ", $in());
            %}
            printf("\n");
            row_counter++;
        %}
        printf("All done!\n");
    },
);
```

A snippet of that output looks like this:

```
About to start printing rows
  Row   0: 0.000000, 1.000000,
  Row   1: 2.000000, 3.000000,
  Row   2: 4.000000, 5.000000,
  Row   3: 6.000000, 7.000000,
All done!
About to start printing rows
  Row   0: 0.000000, 1.000000, 2.000000,
  Row   1: 3.000000, 4.000000, 5.000000,
  ...
  Row  19: 57.000000, 58.000000, 59.000000,
All done!
```

There are two important aspects to remember about threadloops. First, you must not put anything between the `threadloop` and the `%{` except white space. For example:

```
/* ok */
threadloop %{

/* ok */
threadloop
    %{

/* BAD */
threadloop /* outer loop */ %{
```

As you can see, the parser for the PDL PreProcessor is not terribly sophisticated. It's mostly a pile of

regular expressions, and 99% of the time, it does exactly what you need.

Another potential area of confusion can arise if you have something that looks like a threadloop in your code, but you've commented it out:

```
...
#if 0  /* skip this for now */

threadloop %{
 printf("  Row %3d: ", row_counter);
 loop(n) %{
  printf("%f, ", $in());
 %}
 printf("\n");
 row_counter++;
%}

#endif /* skipped block of code */
...
```

The problem is that if you do not indicate where the threadloop is supposed to go, PDL wraps all your code in the threadloop logic. However, if PDL sees what looks like a threadloop block, it assumes that you want to be more precise about where the threadloop logic goes. In fact, it even inserts the threadloop logic where you indicated it was to go, but this will eventually get discarded by the C preprocessor thanks to the `#if 0` block. This means that the code that contains the `loop(n)` block, below the `#endif`, does not have the threadloop logic that it needs to do its job, and you will get erroneous results.

The easiest fix for this? In addition to commenting out the blocks, put something (anything) between the text `threadloop` and the percent block `%{`. As already discussed, this will always prevent PDL from identifying the threadloop, which is what you need it to temporarily do in this case.

It may seem that threadloops are bad things to be avoided, but threadloops are particularly useful if you are writing a function that needs access to a system resource that is costly to allocate with each iteration. For that sort of operation, you allocate it before entering the threadloop and de-allocate it after leaving:

```
Code => q{
    /* allocate system resource */
    threadloop %{
        /* use system resource */
    %}
    /* Free system resource */
},
```

They are also handy if you need to perform a particularly expensive calculation once each time the function is invoked.

A Complex Example

To put this all together, I am going to consider writing a PDL::PP function that computes the first numerical derivative of a time series. You can read about finite difference formulas here: *http://en.wikipedia.org/wiki/Numerical_differentiation*. Normally, finite difference formulas result in a numerical derivative with one less point than the original time series. Since I have not discussed how to set a return dimension with a calculated size, I'm going to use a slightly modified numerical derivative. The derivatives associated with the first and last points will be calculated using the right

and left finite differences, respectively, whereas the points in the middle will be calculated using a centered-difference formula. I'll run this function on the sine wave and compare the results with the actual derivative of the sine wave, which is the cosine wave. I've marked a couple of points in the code for the discussion that follows.

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

# Create some sine data:
my $h = 0.3;
my $sine = sin(sequence(10) * $h);
my $derivative = $sine->my_first_derivative($h);
my $cosine = cos(sequence(10) * $h);

print "The difference between the computed and actual derivative:\n"
   , $derivative - $cosine, "\n";

__END__

__Pdlpp__

pp_def('my_first_derivative',
    Pars => 't_series(n); step(); [o] derivative(n)',
    Code => q{
        int N = $SIZE(n);
        threadloop %{
            /* Derivative for i = 0 */
            $derivative(n => 0)
                = ($t_series(n => 1) - $t_series(n => 0))
                    / $step();
            /* Derivatives for 1 <= i <= N-2 */
            /* (Point 1) */
            loop (n) %{
                /* Skip the first and last elements (Point 2) */
                if (n == 0 || n == N - 1) {
                    /* (Point 3) */
                    continue;

                }
                /* (Points 4 and 5) */
                $derivative()
                    = ($t_series(n => n+1) - $t_series(n => n-1))
                        / 2.0 / $step();
            %}
            /* Derivative for i = N-1 */
            $derivative(n => N-1)
                = ($t_series(n => N-1) - $t_series(n => N-2))
                    / $step();
        %}
    },
);
```

The output on my machine looks like this:

The difference between the computed and actual derivative:

```
[-0.014932644 -0.0142657   -0.012324443 -0.0092822807 -0.0054109595
 -0.0010562935 0.0033927281 0.0075386874 0.011011238   0.077127808 ]
```

These differences are fairly small, four times smaller than the (fairly large) step size. And if I decrease the size of $h by 2, these errors should get smaller by a factor of 4 except at the endpoints. Not bad.

But what we really care about is the code, which uses a number of tricks I haven't discussed yet. Let's run through each point in turn.

point 1, a sub-optimal example

> The code within this loop does `not` actually compute results for all indices from zero to N-1. As such, I should use a for loop that starts from 1 and runs to N-2. I dislike it when bad examples are used for pedagogical reasons, but that's what I'm going to do here. Sorry.

point 2, a useful register

> The actual C code that gets generated by the `loop` construct creates a register variable called n within the scope of the loop block. Thus, we can access the current value of n from within the loop by simply using that value in our code. I do that in this `if` statement and in the memory accesses later.

point 3, C looping commands

> The `loop` construct creates a bona fide `for` loop, so you can use `break` and `continue`, just like in a real C `for` loop.

point 4, explicit dimension values within a loop block

> When we `loop` over n, it saves you keystrokes in your memory access by making it unnecessary to specify n. This is exploited when I say `$derivative()` without specifying a value for n. However, we can override that value for n within the loop by explicitly specifying it, which is what I do with `$t_series(n =` n-2)>.

point 5: which n?

> Look closely at the access statements for `$t_series`:
>
> ```
> $t_series(n => n-1)
> ```
>
> PDL::PP parses this as
>
> ```
> $ <pars-variable-name> ( <dimension-name> => <value>,
>                          <dimension-name> => <value>,
>                          ...
>                        )
> ```
>
> and replaces it with a direct array access statement. In this statement, the n on the left side of the fat comma (the =>) is the name of the dimension. The n on the right side of the fat comma is part of a C expression and is not touched by PDL::PP. That means that the n on the right side refers to the C variable n. This makes two uses of the same token, n, which can be a bit confusing. I'm not suggesting that this is a best practice, but it is a possible practice which may be useful to you. So now you know.

In the above section I have explained how to use `loop` and `threadloop` to control how PDL::PP presents data to your code, and to control which sections of code PDL::PP threads over. I have also shown you how to access specific memory locations when you have vector representations of your data.

Exercise Set 2

1. Matrix Multiplication, Fixed

   I noted above that my code for the matrix multiplication is incorrect and I explained why. Changing nothing more than the `Pars` section, fix this code so that it performs proper matrix multiplication.

2. Threading Engine Tricks

   The function `my_sumover` uses a `loop` construct, so it only operates on individual rows. What if you wanted to perform the sum an entire matrix? Using Perl level operations, find a way to manipulate the incoming piddle so that you can call `my_sumover` to get the sum over the entire matrix. Bonus points if the same technique works for higher dimensional piddles.

3. Cumulative Sum

   Modify `my_sumover` to create a function, `my_cumulative_sum`, which returns the cumulative sum for each row. By this I mean that it would take the input such as (1, 2, 3, 4) and return (1, 3, 6, 10), so that each element of the output corresponds to the sum of all the row's elements up to that point.

4. Full Cumulative Sum

   Take your code for `my_cumulative_sum` and modify it so that it returns the cumulative sum over the entire piddle, regardless of the piddle's dimension. Your resulting code should not have any `loop` constructs.

Tips

These are a couple of things I have learned which help me make effective use of PDL::PP, but which did not sensibly fit elsewhere.

Best Practice: use pp_line_numbers

PDL::PP includes a brand new function in PDL 2.4.10 called `pp_line_numbers`. This function takes two arguments: a number and a string. The number should indicate the actual line in your Perl source file at which the string starts, and the function causes `#line` directives to be inserted into the string. This is `ENORMOUSLY` helpful when you have a syntax error. Without it, the syntax error is reported as coming from a given line in your XS file, but with it the error is reported as coming from your own source file.

I will illustrate this with an example that gave me great trouble while I was preparing this text:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

# Run the code on a 2x4 matrix:
sequence(2,4)->my_print_rows;

__END__

__Pdlpp__

pp_def('my_print_rows',
    Pars => 'in(n)',
    Code => q{
        printf("About to start printing rows\n");
        int row_counter = 0;
        threadloop %{
```

```
            printf("  Row %3d: ", row_counter);
            loop(n) %{
                printf("%f, ", $in())
            %}
            printf("\n");
            row_counter++;
        %}
        printf("All done!\n");
    },
);
```

Notice what's missing? The semicolon at the end of the `printf` is missing. Unfortunately, the error output of this example (contained in *_Inline/build/bad_error_reporting_pl_8328/out.make*) borders on useless:

```
 bad_error_reporting_pl_4420.xs: In function
'pdl_my_print_rows_readdata':
 bad_error_reporting_pl_4420.xs:177: warning: format '%f' expects
type 'double', but argument 2 has type 'int'
 bad_error_reporting_pl_4420.xs:177: warning: format '%f' expects
type 'double', but argument 2 has type 'int'
 bad_error_reporting_pl_4420.xs:178: error: expected ';' before '}'
token
 bad_error_reporting_pl_4420.xs:222: warning: format '%f' expects
type 'double', but argument 2 has type 'int'
 bad_error_reporting_pl_4420.xs:222: warning: format '%f' expects
type 'double', but argument 2 has type 'int'
 bad_error_reporting_pl_4420.xs:223: error: expected ';' before '}'
token
 bad_error_reporting_pl_4420.xs:267: warning: format '%f' expects
type 'double', but argument 2 has type 'int'
 bad_error_reporting_pl_4420.xs:267: warning: format '%f' expects
type 'double', but argument 2 has type 'int'
 bad_error_reporting_pl_4420.xs:268: error: expected ';' before '}'
token
 bad_error_reporting_pl_4420.xs:312: warning: format '%f' expects
type 'double', but argument 2 has type 'PDL_Long'
 bad_error_reporting_pl_4420.xs:312: warning: format '%f' expects
type 'double', but argument 2 has type 'PDL_Long'
 bad_error_reporting_pl_4420.xs:313: error: expected ';' before '}'
token
 bad_error_reporting_pl_4420.xs:357: warning: format '%f' expects
type 'double', but argument 2 has type 'PDL_LongLong'
 bad_error_reporting_pl_4420.xs:357: warning: format '%f' expects
type 'double', but argument 2 has type 'PDL_LongLong'
 bad_error_reporting_pl_4420.xs:358: error: expected ';' before '}'
token
 bad_error_reporting_pl_4420.xs:403: error: expected ';' before '}'
token
 bad_error_reporting_pl_4420.xs:448: error: expected ';' before '}'
token
```

If you're a seasoned C programmer, you'll recognize the warning: it arises because PDL::PP creates a branches of code for each data type that PDL supports, so using the `%f` type is not correct. (The correct way to handle this is to use the `$T` macro.) That's not our problem,

though. The issue is the expected semicolon error. For a small function, you can probably just scan through the code and look for a missing semicolon, but when you are working on a much larger set of PP code, having the line number of the error would be **much** more useful. You accomplish that by using the `pp_line_numbers` function, which adds `#line` directives into your code so that errors get reported on the correct lines. Here is a slightly doctored version to illustrate the issue. (Note that the text `#line 1 ...` must be flush against the left margin, just like the `__END__` and `__Pdlpp__` markers, or Perl won't realize that you are trying to tell it about line numbers and things will be reported incorrectly.)

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

# Run the code on a 2x4 matrix:
sequence(2,4)->my_print_rows;

__END__

__Pdlpp__
#line 1 "my-inline-work"
                  # This is reported as line 1
pp_def('my_print_rows',
    Pars => 'in(n)',
    Code => pp_line_numbers(__LINE__, q{
        /* This line is reported as line 5
         * Thanks to pp_line_numbers */
        printf("About to start printing rows\n");
        int row_counter = 0;
        threadloop %{
            printf("  Row %3d: ", row_counter);
            loop(n) %{
                printf("%f, ", $in())
            %}
            printf("\n");
            row_counter++;
        %}
        printf("All done!\n");
        /* This is line 18 */
    }),
);     # This is reported as line 20
```

Apart from a couple of comments to indicate the line counting, I introduced two modifications: I added a `#line` directive at the top of the Pdlpp section and I wrapped the `Code` section in a call to `pp_line_numbers`. (The `#line` directive is only necessary when using *Inline::Pdlpp*, and is not necessary in a .pd file.) Now the error output gives the line of the closing bracket that reports the missing semicolon:

```
 my-inline-work: In function 'pdl_my_print_rows_readdata':
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'int'
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'int'
 my-inline-work:13: error: expected ';' before '}' token
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'int'
```

```
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'int'
 my-inline-work:13: error: expected ';' before '}' token
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'int'
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'int'
 my-inline-work:13: error: expected ';' before '}' token
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'PDL_Long'
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'PDL_Long'
 my-inline-work:13: error: expected ';' before '}' token
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'PDL_LongLong'
 my-inline-work:12: warning: format '%f' expects type 'double', but
argument 2 has type 'PDL_LongLong'
 my-inline-work:13: error: expected ';' before '}' token
 my-inline-work:13: error: expected ';' before '}' token
 my-inline-work:13: error: expected ';' before '}' token
```

All the errors are reported as occurring on line 13, immediately directing your eye to where the problem lies. This lets you fix your problem and get on with your work.

Sometimes PDL::PP's parser croaks on invalid input. Sometimes it doesn't. For those times when you when you feed PDL::PP bad code and the error reporting leaves you scratching your head, consider wrapping your code in a `pp_line_numbers` call.

Wart: /* */ doesn't always work; use #if 0

Note: This issue has been addressed in the git copy of PDL as of April 23, 2012. It will make its way onto CPAN with the release of PDL v2.4.11, slated for spring or summer of 2012.

Until the latest fixes, some of XS code that PDL::PP generates includes C-style comments indicating what they do. This is useful when you find yourself digging into the generated XS code as it helps you get your bearings. However, it can also break a relatively common use of comments. (With the latest work, the commentary is still present, but they use a preprocessor trick so that they don't break C-style comments anymore.)

When there is a logic bug in my code I find it helpful to reduce the complexity of the code and comment-out sections at a time until I get an output that makes sense.

Here's an example. I am trying to print out the values in a piddle, but I have mistakenly used `\r` instead of `\n` in my `printf` statement. On some systems, nothing will get sent to *STDOUT* because IO operations are buffered, and I am left with a function that appears to print nothing when it gets called. (The last value may get printed when the buffer fills, or when the program terminates. Either way, it's very confusing.) So, I tried to comment out the confusing print behavior and replace with something foolproof:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

# Run the code on a 2x4 matrix:
sequence(2,4)->my_printout;

__END__
```

```
__Pdlpp__
#line 1 "my-printout-pdlpp"
pp_def('my_printout',
    Pars => 'in()',
    Code => pp_line_numbers(__LINE__, q{
        printf("This piddle contains:\n");
        threadloop %{
            /* grr, not working
            printf("  %f\r", $in());
            */
            printf("  Here\n");
        %}
    }),
);
```

This *should* work without a hitch. Unfortunately, this gives me these errors:

```
my-printout-pdlpp: In function 'pdl_my_printout_readdata':
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
my-printout-pdlpp:7: error: expected statement before ')' token
my-printout-pdlpp:8: error: expected expression before '/' token
```

(Got different line numbers? Be sure to put remove all spaces before `#line 1 "my-printout-pdlpp"`.) Lines seven and eight are these:

```
            printf("  %f\r", $in());
            */
```

Perplexed? You bet. I just *commented out some code*, how could I possibly have introduced a compile error? Using `pp_line_numbers`, I know which lines in my code caused the C compiler to choke, but I'm even more confused as to why it choked there.

The problem is that the memory access, `$in()`, gets replaced with a chunk of C code that includes the comment `/* ACCESS() */`. As C comments do not nest, this leads to some *very* wrong code. A different approach that achieves the same end is to use `#if 0`, a common technique among C programmers for cutting out blocks of code:

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

# Run the code on a 2x4 matrix:
sequence(2,4)->my_printout;

__END__
```

```
__Pdlpp__
#line 1 "my-printout-pdlpp"
pp_def('my_printout',
    Pars => 'in()',
    Code => pp_line_numbers(__LINE__, q{
        printf("This piddle contains:\n");
        threadloop %{
          #if 0
            printf("  %f\r", $in());
          #endif
            printf("  Here\n");
        %}
    }),
);
```

PDL::PP will still merrily fiddle with the stuff between the `#if 0` and `#endif`, but the C preprocessor will get rid of it before it actually tries to compile the code. Now the code at least runs and printouts the expected dumb results:

```
This piddle contains:
  Here
  Here
  Here
  Here
  Here
  Here
  Here
  Here
```

Hopefully this gives me enough to find that errant `\r`.

Recap

In this chapter, I've covered the very basics of using PDL::PP to write fast, versatile code. I have covered much less material than I had hoped, and I hope to expand this chapter in the coming months. Nonetheless, I hope and believe it will serve as a good starting point for learning PDL::PP, and I expect it will give you enough to dig into the PDL::PP documentation.

Good luck, and happy piddling!

Appendix A: Installing Inline::Pdlpp

The PDL installation always installs *Inline::Pdlpp*, but that does not mean it works for you because *Inline* is not actually a prerequisite for PDL. The good news is that once you have installed *Inline*, *Inline::Pdlpp* will work automatically.

To begin, you will need to have access to the C compiler that compiled your copy of Perl. On Mac and Linux, this amounts to ensuring that the developer tools that contain `gcc` are installed on your system. On Windows, this will depend on your flavor of Perl. I personally have excellent experience working with Strawberry Perl, which ships with a working C compiler, but you can also work with Visual C or Cygwin. If you run into trouble, contact the PDL mailing list for help.

If you are on Linux, you can probably install *Inline* using your package manager. If you are not on Linux or you do not have administrative privileges, you will have to install *Inline* using CPAN. To do this, enter the following commands at your console:

```
> cpan Inline
```

This will likely ask you a few questions during the installation, so do not walk away to get a cup of coffee and expect it to be done.

Once that's installed, you should be ready to work with the examples.

Appendix B: Solutions to Exercises

Excercise Set 1

1. Slices

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';
use PDL::NiceSlice;

# Create $a
my $a = sequence(5);
print "a is $a\n";

# Create $b as a five-element slice from a sequence:
my $idx = pdl(1, 2, 7, 4, 8);
my $b = sequence(20)->index($idx);
print "b is $b\n";

print "printout_sum(a, b) says:\n";
$a->printout_sum($b);

no PDL::NiceSlice;

__END__

__Pdlpp__
pp_def('printout_sum',
    Pars => 'a(); b()',
    Code => q{
        printf("%f + %f = %f\n", $a(), $b(), $a() + $b());
    },
);
```

2. Threading

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

my $a = sequence(5);
print "a is $a\n";
my $b = sequence(5,3);
print "b is $b\n";

print "a + b = ", $a + $b, "\n";

print "printout_sum(a, b) says:\n";
$a->printout_sum($b);
```

```
__END__

__Pdlpp__
pp_def('printout_sum',
    Pars => 'a(); b()',
    Code => q{
        printf("%f + %f = %f\n", $a(), $b(), $a() + $b());
    },
);
```

3. Orthogonal Piddles

```
use strict;
use warnings;
use PDL;
use Inline 'Pdlpp';

my $a = sequence(5);
print "a is $a\n";
my $b = sequence(1,3);
print "b is $b\n";

print "a + b = ", $a + $b, "\n";

print "printout_sum(a, b) says:\n";
$a->printout_sum($b);

__END__

__Pdlpp__
pp_def('printout_sum',
    Pars => 'a(); b()',
    Code => q{
        printf("%f + %f = %f\n", $a(), $b(), $a() + $b());
    },
);
```

4. Varying Input Order

Different input order would be like this:

```
Pars => '[o] sum(); left(); [o] diff(); right()';
Pars => '[o] sum(); [o] diff(); left(); right()';
```

The only consistency here is that `sum` always comes before `diff`, and `left` always comes before right.

5. Supplying Outputs in the Function Call

For a `Pars` key like this:

```
Pars => 'left(); right(); [o] sum(); [o] diff()';
```

You can call the function like this:

```
my ($sum, $diff) = $left->my_sum_and_diff($right);

my ($sum, $diff);
```

```
$left->my_sum_and_diff($right
    , ($sum = PDL::null), ($diff = PDL::null));

my $sum = $left->zeroes;
my $diff = PDL::null;
$left->my_sum_and_diff($right, $sum, $diff);
```

For the latter calling convention, the function returns nothing (rather than `$sum` and `$diff`). When you supply a null piddle (as in the middle example) or you call the function with the input piddles only (as in the first example), PDL will allocate memory for you. As demonstrated with the last example, you can supply a pre-allocated piddle, in which case PDL will **not** allocate memory for you. This can be a performance issue when you regularly call functions

Exercise Set 2

1. Matrix Multiplication, Fixed

   The corrected `Pars` section should look like this:

   ```
   Pars => 'left(m,n); right(p,m); [o] output(n,p)',
   ```

2. Threading Engine Tricks

   The key is to use `clump(-1)`:

   ```
   my $matrix = sequence(2,4);
   my $result = $matrix->clump(-1)->my_sumover;
   ```

3. Cumulative Sum

   ```
   use strict;
   use warnings;
   use PDL;
   use Inline 'Pdlpp';
   my $a = sequence(10);
   print "Cumulative sum for a:\n";
   print $a->my_cumulative_sum;
   my $b = grandom(10,3);
   print "\nCumulative sum for b:\n";
   print $b->my_cumulative_sum;

   __END__

   __Pdlpp__

   pp_def('my_cumulative_sum',
       Pars => 'input(n); [o] output(n)',
       Code => q{
           double cumulative_sum;
           threadloop %{
               cumulative_sum = 0.0;
               loop (n) %{
                   cumulative_sum += $input();
                   $output() = cumulative_sum;
               %}
           %}
       }
   );
   ```

4. Full Cumulative Sum

```
pp_def('my_full_cumulative_sum',
    Pars => 'input(); [o] output()',
    Code => q{
        double cumulative_sum = 0.0;
        threadloop %{
            cumulative_sum += $input();
            $output() = cumulative_sum;
        %}
    }
);
```

The Beginnings of PDL

"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 accumulated 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 properties 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 commercial 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 (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/*). The ease with which I could then create charts and graphs, for my papers, was refreshing.

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

The case for a high-level approach

We've all been there. You know how you want to analyze your data. You need to Fourier transform it, take the square root, multiply by a high-pass filter and sum up all the high frequency 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 separate 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 commercial software such as `IDL` ('Image Data Language' from Research Systems Inc.*http://www.rsinc.com*), `MATLAB` (from The Mathworks, Inc. *http://www.mathworks.com*) and the public domain program `Octave` *http://www.octave.org*. These implement special-purpose high-level languages where data is 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.

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. Although 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 customizable. 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 institutes 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 customization.

Finally there *is* also the issue of cost. Equivalent commercial packages cost several thousand dollars per workstation. We are not anti-commercial, 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.

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? {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 rationalizations'!)

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.

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 proprietary 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 (*http://numpy.scipy.org/*). 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 recognize of course that much of this is just a matter of preference. NumPy and SciPy have grown into a well supported set of modules, so if you are into Python, go on and use them!

Several separate sources of material have been used to make this 2012 version of the PDL Book. The biggest source of material has been "PDL - Scientific Programming in Perl", written in 2001 and added to over the past decade by Karl Glazebrook, Christian Soeller, Tuomas J. Lukka, Marc Lehmann, Jarle Brinchmann, Doug Hunt, John Cerney, Robin Williams and Tim Pickering, and several chapters written by Craig DeForest from 2009.

The original source was written in LaTeX and LyX, which allowed embedding of figures in the document. However, this has presented a small hurdle for other authours to add their own material. With this in mind, Matthew Kenworthy set about converting one chapter of the PDL book into POD to see what could be done, and the result didn't look too bad at all. Although this may seem to be a step back from the finer formatting of LaTeX, tags can be included in POD so that the basic documentation is readable at the command line, and there are enough filters to provide clean output in HTML and PDF formats.

Several other people have carried out conversion of the original book and figures into POD, and others have also contributed original new material for the PDL Book.

In alphabetical order, we have:

Joel Berger

Craig DeForest

Karl Glazebrook

Matthew Kenworthy

David Mertens

Chris Marshall

Joe Milosch

Creating.pod:

Section 2.4 from PDL LyX book by Craig DeForest POD-ed by M. Kenworthy

Functions:

Written by Matthew Kenworthy 2011

PGPLOT.pod:

Original text from "PDL - Scientific Programming in Perl" (2001) Chap. 4

Authors: Karl Glazebrook, Marc Lehmann, John Cerney, Christian Soeller, Jarle Brinchmann, Robin Williams, Christopher Marshall, Tuomas J. Lukka, Doug Hunt, Tim Pickering.

Modified to LyX by Chris Marshall for PDL 2.4.3, December 2006.

Converted to POD format by Matthew Kenworthy, May 2010.

PLplot:

Joe Milosch, also known as zentara on perlmonks, assembled this document. David Merten, wrote most of the section and examples on Object Oriented usage, which were taken from his slide show on PLplot.

David Merten's slide show on **PDL::Graphics-PLplot**. His very informative slide show can be downloaded or viewed at *http://www.slideshare.net/dcmertens/p-lplot-talk*

PP.pod:

David Mertens <dcmertens.perl@gmail.com>

Piddle.pod:

Original text from "PDL - Scientific Programming in Perl" (2001) Chap. 1

Authors: Karl Glazebrook, Marc Lehmann, John Cerney, Christian Soeller, Jarle Brinchmann, Robin Williams, Christopher Marshall, Tuomas J. Lukka, Doug Hunt, Tim Pickering.

Modified to LyX by Chris Marshall for PDL 2.4.3, December 2006.

Converted to POD format by Mike Burns, May 2010.