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`.

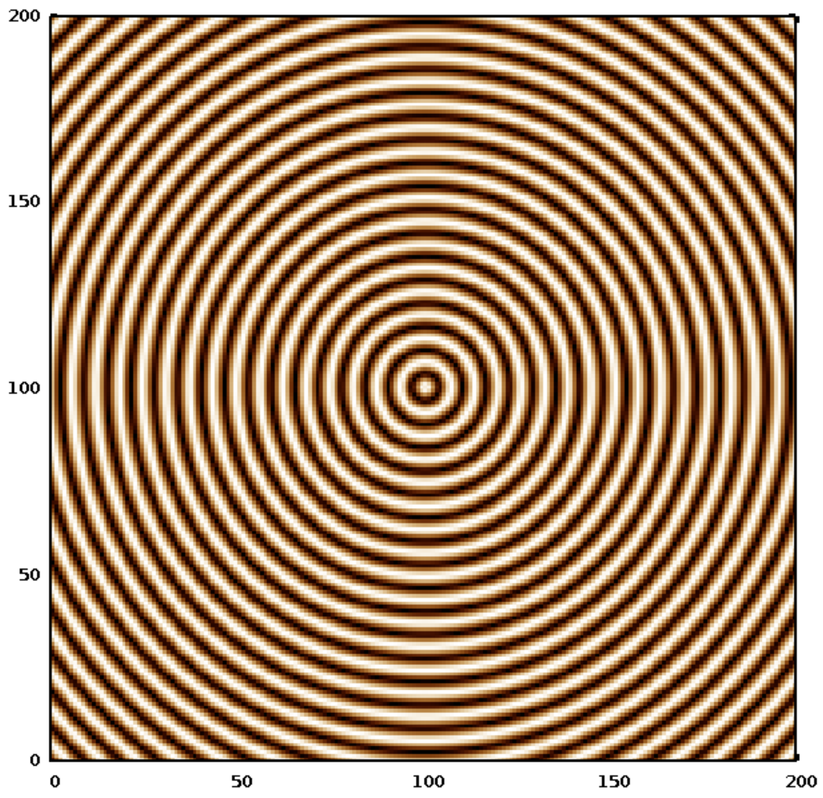
```
myhost% perldl
perlDL shell v1.357
 PDL comes with ABSOLUTELY NO WARRANTY. For details, see the file
 'COPYING' in the PDL distribution. This is free software and you
 are welcome to redistribute it under certain conditions, see
 the same file for details.
ReadLines, NiceSlice, MultiLines  enabled
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.

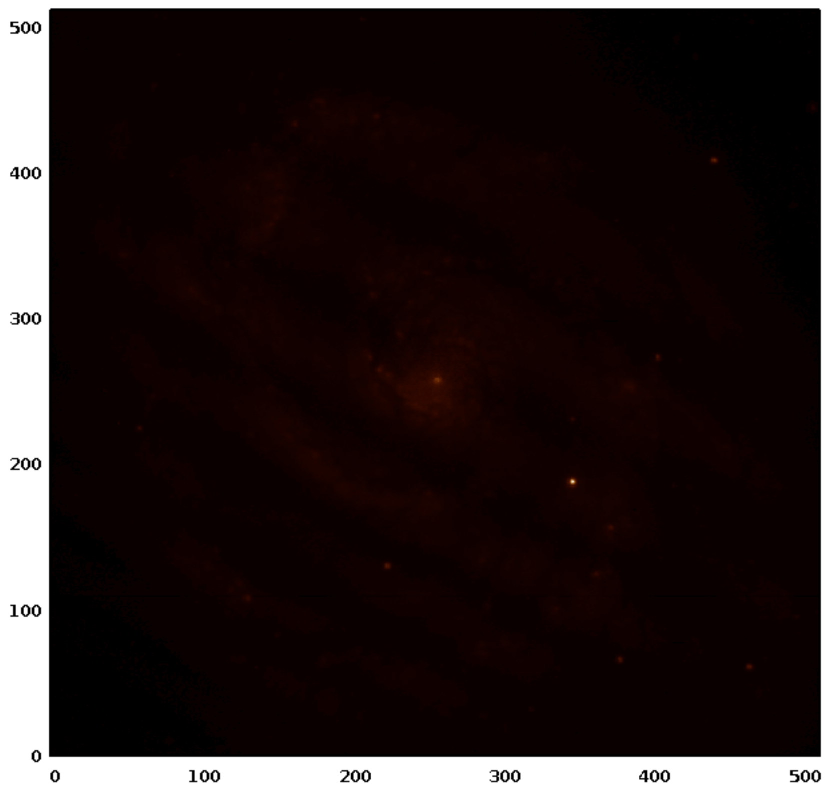
Before we start seriously playing around with M51 it is worth noting that we can also say:

```
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

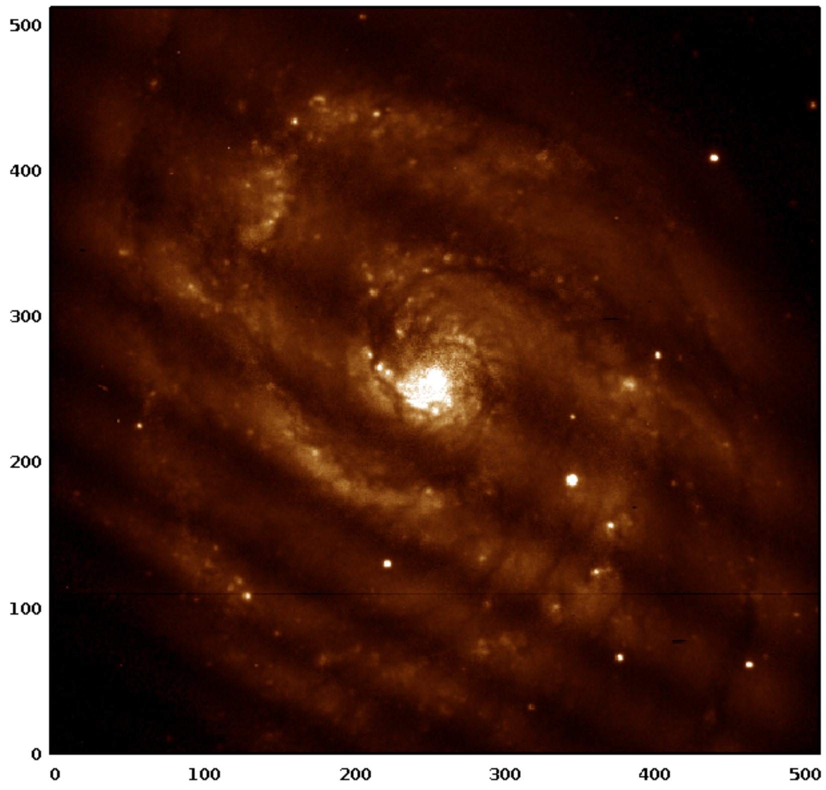
Let's look at M51:

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

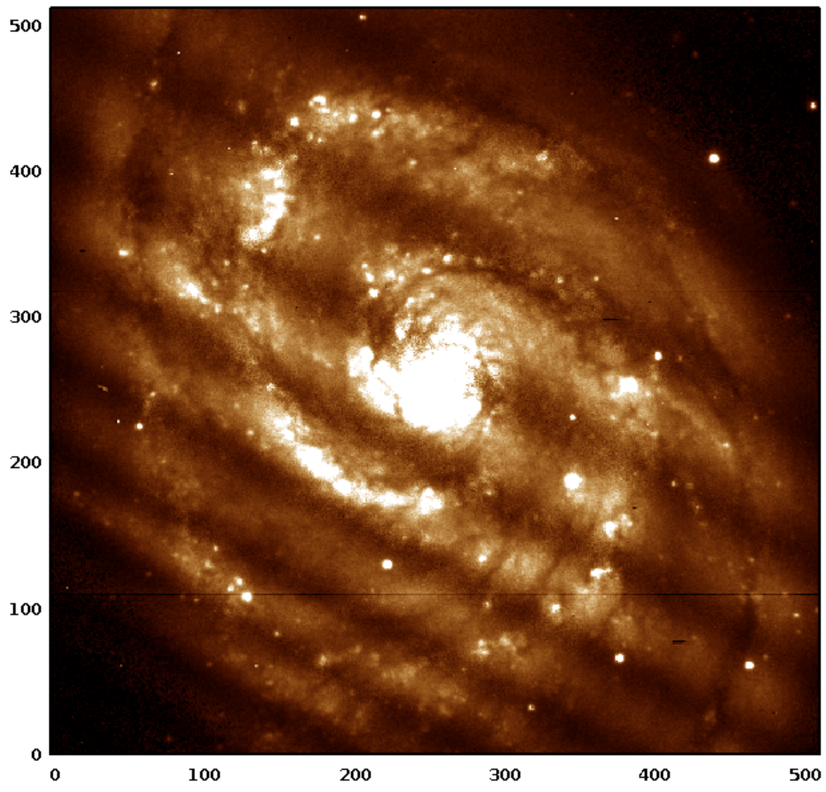


```
Figure of the raw image C<m51_raw.fits> shown with
progressively greater contrast using the C<imag> command.
```

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:

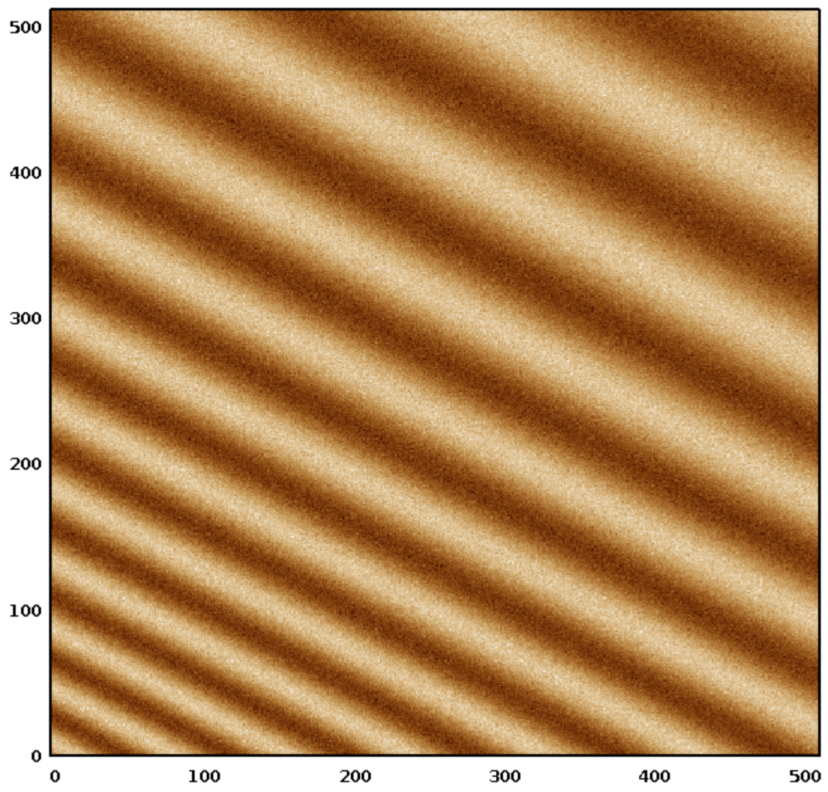


Figure: The 'flatfield' image showing the detector sensitivity of the raw data.

```
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:

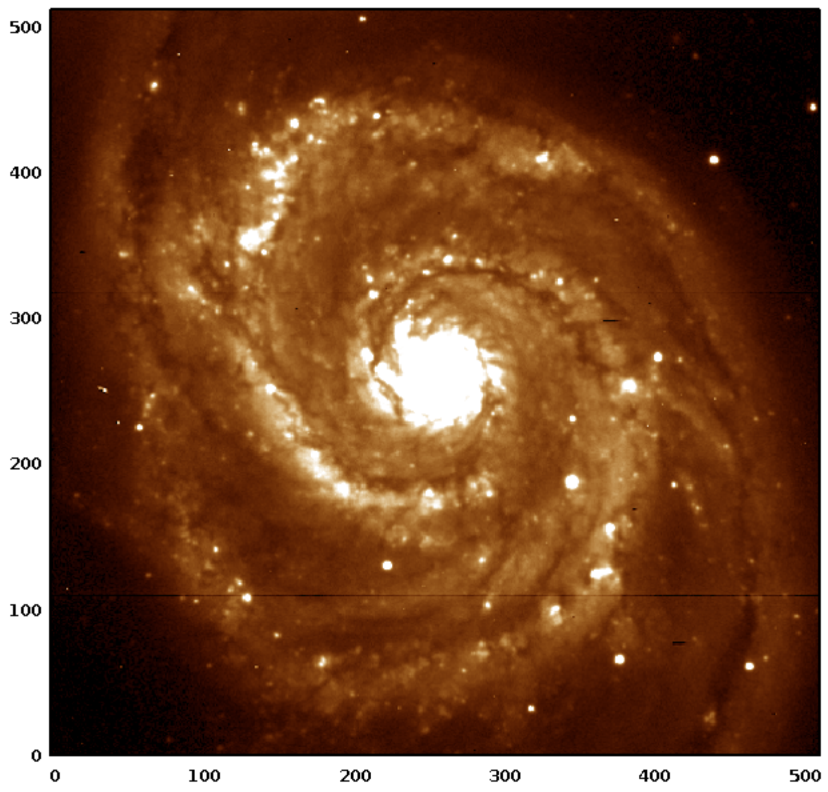


Figure: The M51 image corrected for the flatfield.

```
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
```

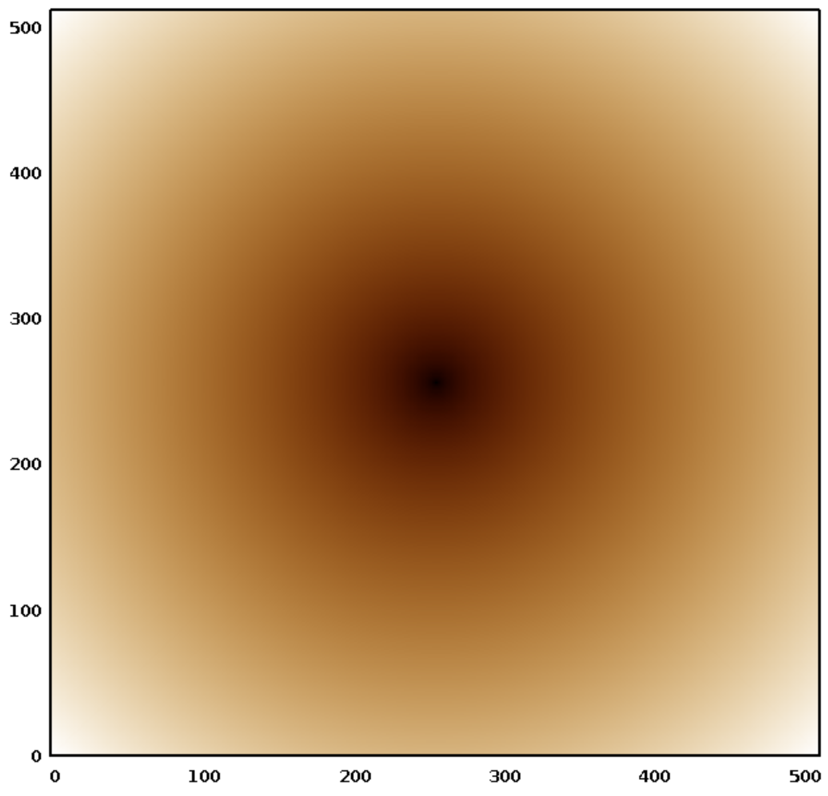
This is also about 200 solar luminosities, (Note we have switched to using `p` as a shorthand for `print` - which only works in the `pdl` and `pdl2` shells) which gives 4 billion solar luminosities for the whole galaxy.

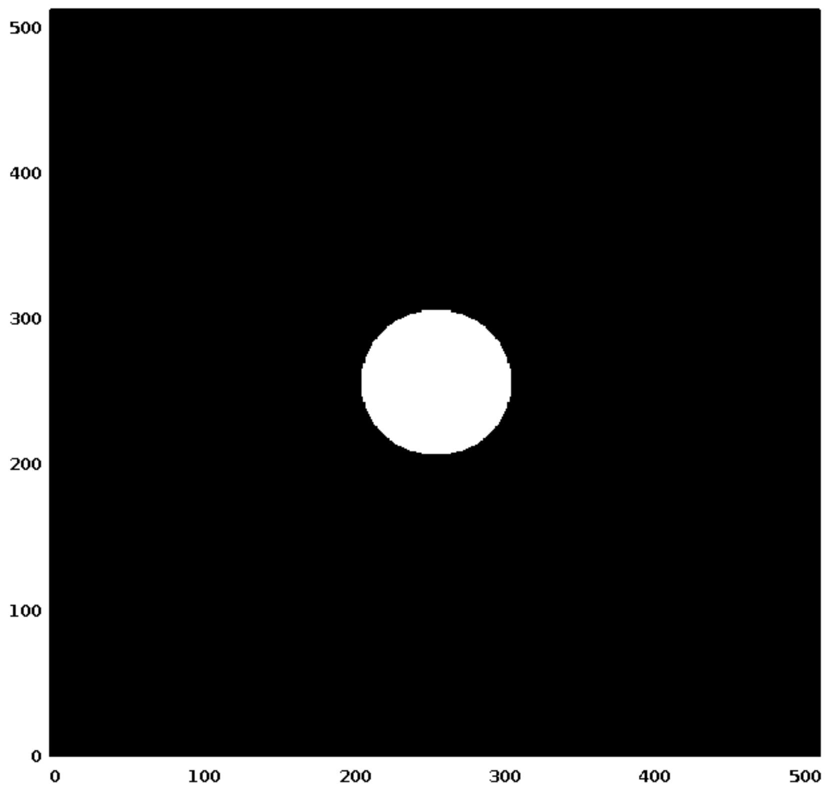
OK we do not need PDL for this simple arithmetic, let's get back to computations that involve the whole image. How can we get the sum of a piece of an image, e.g. near the centre? Well in PDL there is more than one way to do it (Perl aficionados call this phenomenon TIMTOWTDI). In this case, because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

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

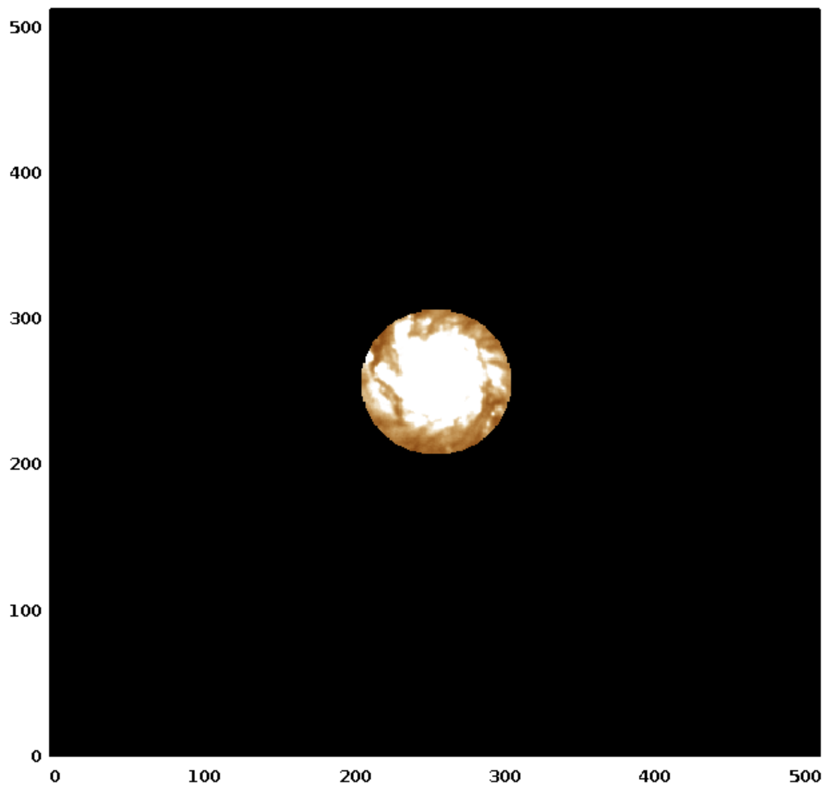
Remember `rvals`? It replaces all the pixels in an image with its distance from the centre. We can turn this into a *mask* with a simple operation like:

```
  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

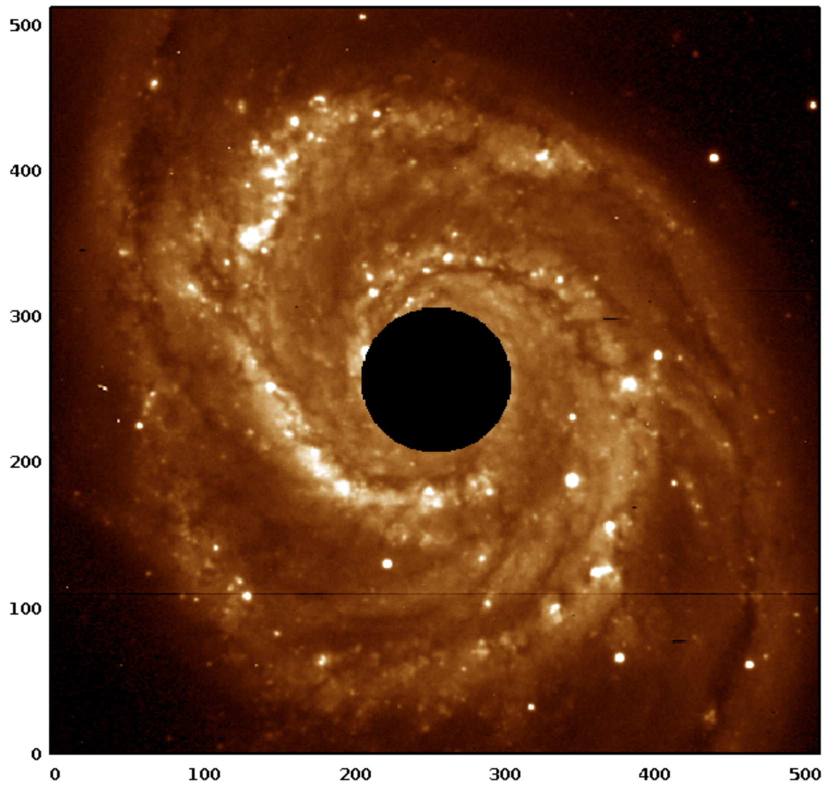


Figure: Using `rvals` to generate a mask image to isolate the galaxy bulge and disk. Top row: radial gradient image `$r`, and radial gradient masked with less than operator `$r < 50`. Bottom row: Bulge and disk of the galaxy.

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!

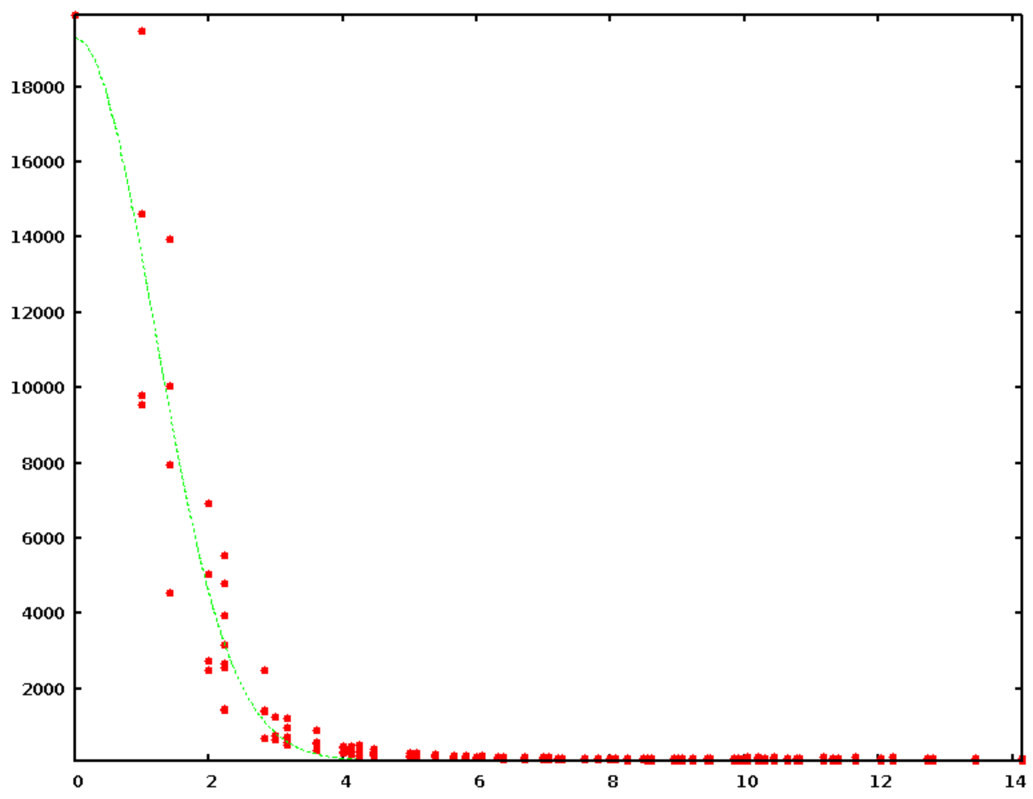


Figure: Radial light profile of the bright star with fitted curve.

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.

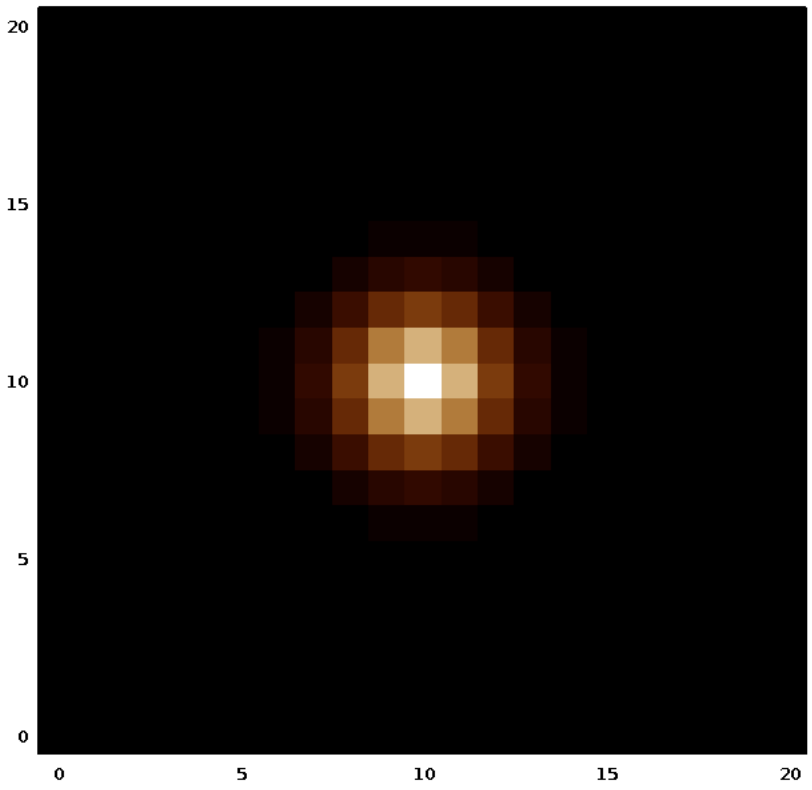
```
pdl> hold; # This command stops new plots starting new pages
pdl> line $rrr, $fit, {Colour=>2} ; # Line plot
```

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);



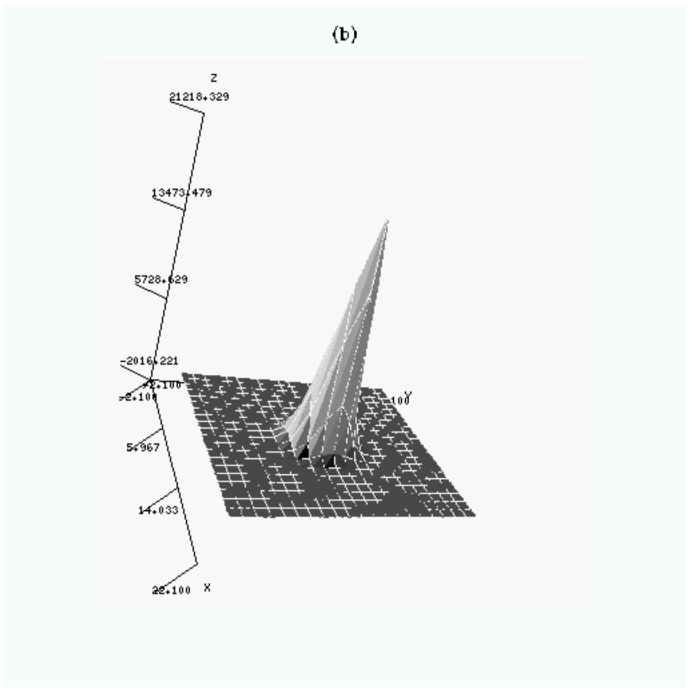


Figure: Two different views of the 2D simulated Point Spread Function.

```
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