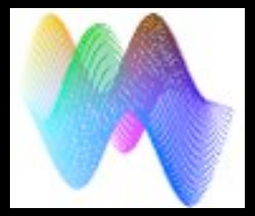
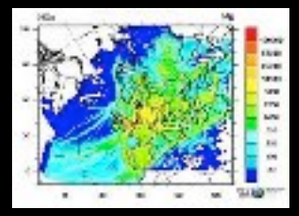
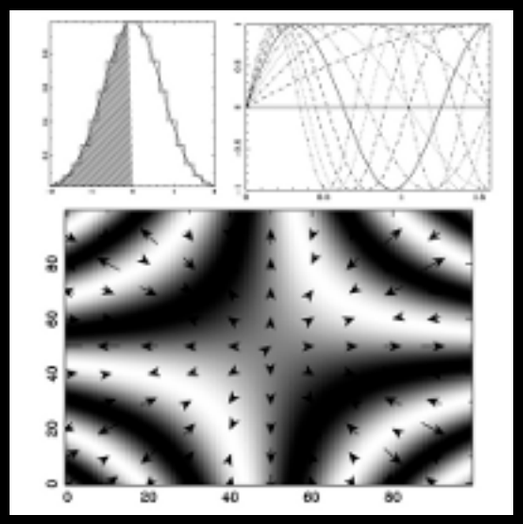




The Perl Data Language (PDL) – A short intro

Karl Glazebrook



What is PDL?

– Array processing oriented language
– Multiple datatypes (ints, floats, doubles...)
– Arrays are stored in C-friendly compact memory blocks
– C-speed for array processing
– Lots of numerical oriented functions

```
$r = rvals(2000,2000);

$sin = 10*sin(0.02*$r);

imag $sin/max($sin)+grandom($r)*0.1;
```

DEMO

History

Some key features

High-level expressive coding

2D graphics uses familiar PGPLOT (support for multiple 2D+3D graphics libraries)

Deep vectorization

Access to all of Perl (CPAN libraries) – text processing, DB/SQL support, WWW interfaces,...

Excellent FITS & astro support

Fast

Free

Easy extension with C code (inline!)

Deep vectorization

```
$x = random(1000000);
$median = medover($x); # 0D answer

$x = random(100000,200);
$median = medover($x); # 1D answer (200 elements)
$median = medover($x->mv(1,0)); # 1D answer (100000)
```

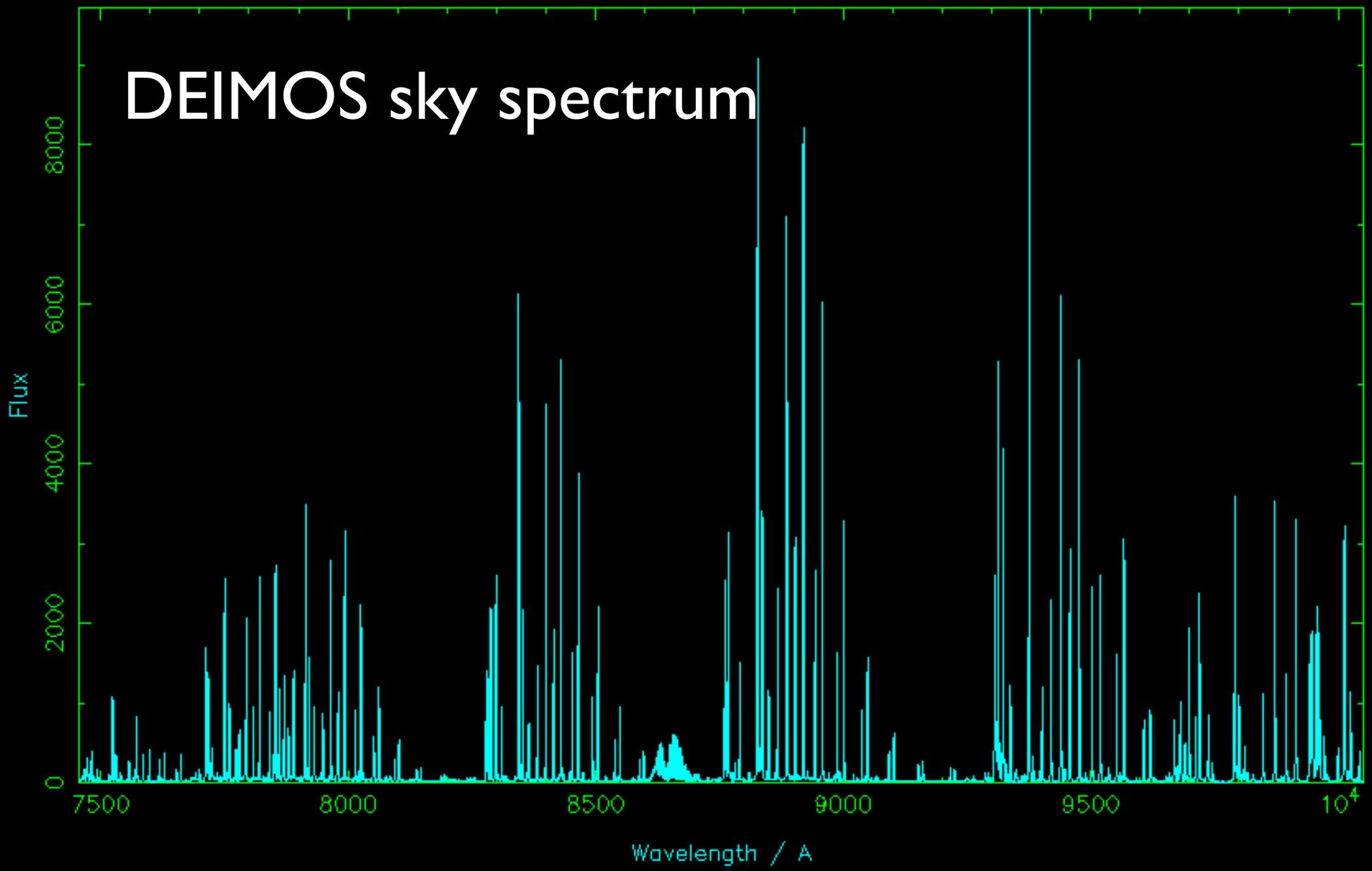
All vector functions operate at C speed and automatically '*thread*' over extra dimensions

Lots of functions for slicing, dicing, clumping, indexing and generally mixing up dimensions

```
$x = random(10, 2000,2000,2);
$median = medover($x->clump(1,2)->mv(1,0)); # answer is 10x2
```

Real World Example

Something I did recently...



Code

```
($w,$f) = rcols 'cooper-skyspec.dat';

# Smooth with a series of gaussians

$fwhm =  10**(sequence(20)/10) * 2;
$r = rvals(500);
$gauss = exp(-0.5*( $r/($fwhm->dummy(0)/2.35) )**2) ;
$fsm = conv1d($f, $gauss);
imag $fsm;
```

Log sequence of FWHMs

Tmp column vector

```
# Calculate average brightness of the darkest 80%

$fsm_norm = $fsm / sumover($fsm)->dummy(0); # Normalise each spectrum
$sort = qsort($fsm_norm);
$i80 = int( 0.8 * $sort->getdim(0)) ; # 80%ile pixel on x axis
$brightness = average( $sort(0:$i80,:) );

line $fwhm, $brightness/min($brightness); # Plot
pglab 'FWHM / A', 'Relative brightness of darkest 80%','';
```

DEMO

More non-trivial examples

- Karl – 2 component SFH stellar mass fitting code
- Sánchez – R3D – IFU reduction package
- Benson – semi-analytic model analysis
- Karl – GDDS data reduction prototyping
- Kenworthy – SPIRAL1 datacube processing
- de Forest – post-processing of magnetohydrodynamical solar simulations

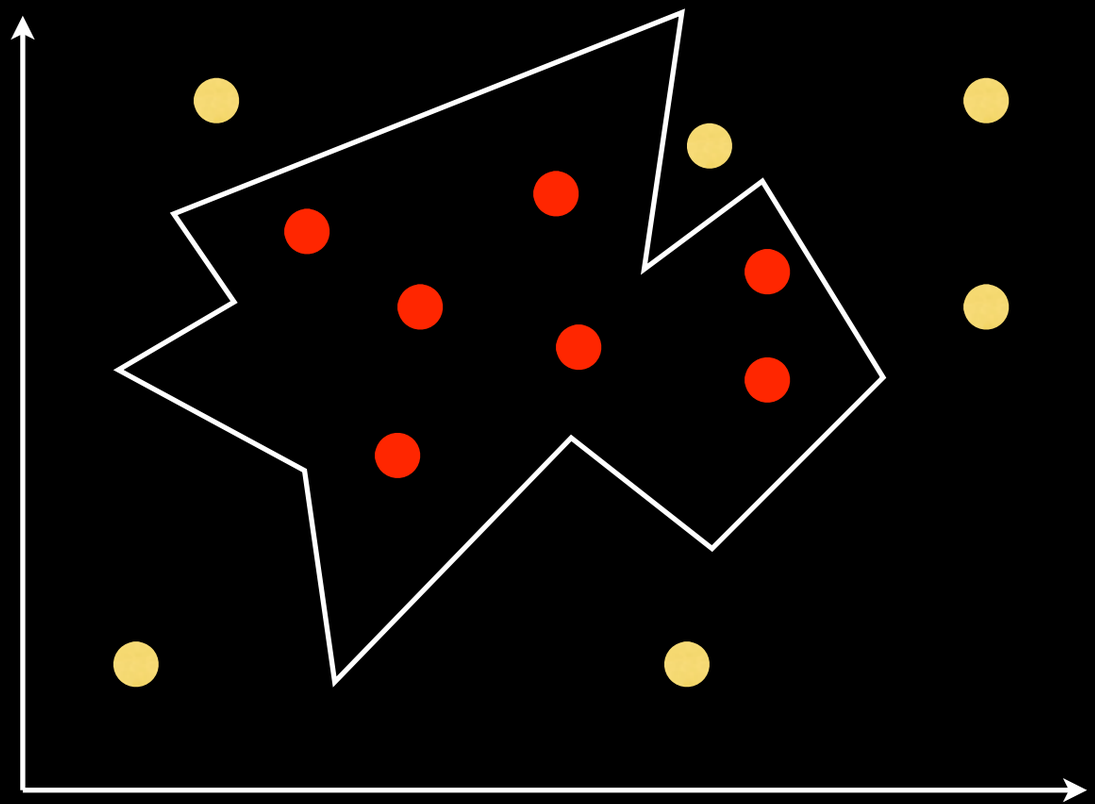
PDL vs IDL

- No licensing headaches
- Perl is a *real* language, does not break at 'edge cases', proper support for modern programming styles
- Modular: namespaces/modular extensions/ easy to add in your own C/F77 code
- Speeds are comparable for simple examples but PDL has better vectorization
- Excellent for non-numeric language tasks

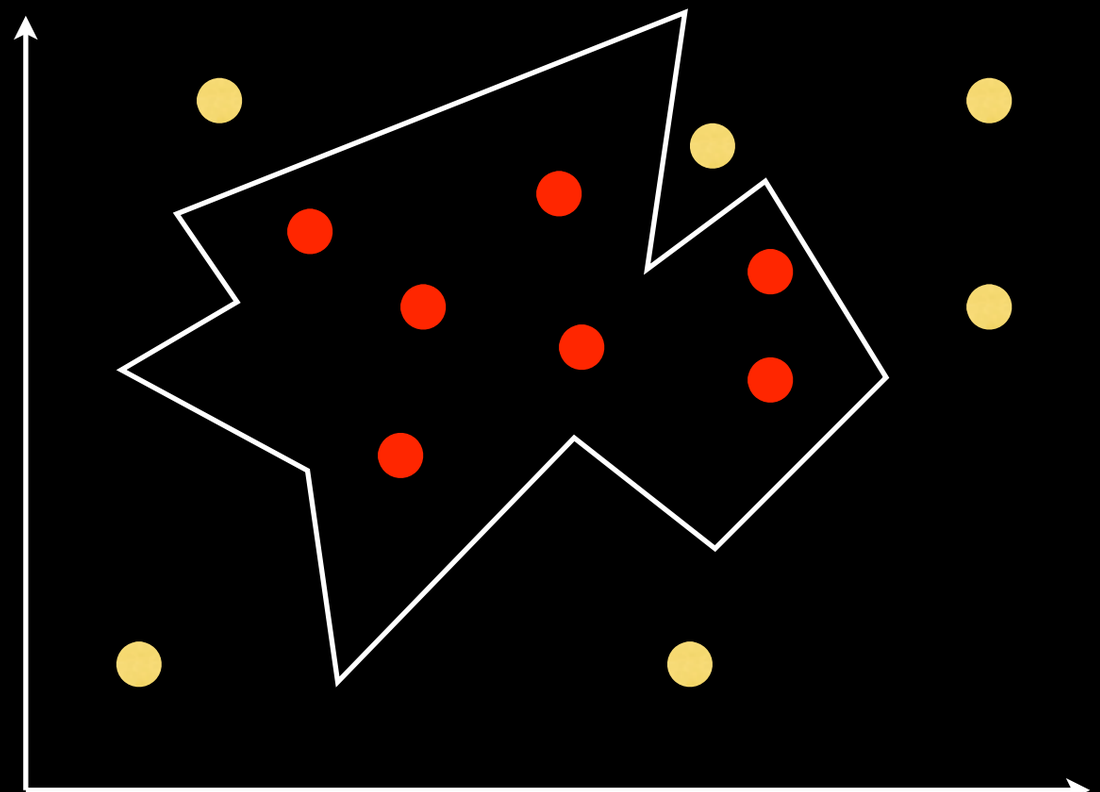
* No multidimensional wildcarding/threading -- once you use one "*" in a dimension, you're done(!)
* No null sets -- so operations like where() (their equivalent of our which()) always require checking -- in the null case it returns -1;
* No rich ND operations
* Interpolation is inconsistent (pixel-centered vs. corner centered for different types of interpolation)
* No heterogeneous arrays/lists -- the best you can do is a "data cube"
* No hashes -- IDL "structures" are absolutely wretched. Anonymous structures can be mocked up to work sort of like hashes, but they use a linear search through a set of string tags -- so you have to recopy the whole structure if you add/delete a tag(!!); and searching is linear(!!).
* No type promotion -- loops fail on the 32,768th iteration by default.
* Awful string handling
* Ghastly widget sets, if you're into that kind of thing (e.g. Perl's Tk is much, much easier to use)
* Nothing remotely like PDL::Transform
* Hideous handling of booleans (the low-order bit of an integer is treated as the boolean value, so 2 is false)
* No hierarchical namespace
* It's ****ing broken out of the box. (I recently installed IDL 8 to run some instrument-provided calibration code. It wouldn't install because a relative path in the install script didn't agree with the structure of their tarball.)
* The owners have taken active steps to prevent data compatibility (sorry to say, our PDL::IO::IDL module is responsible for the plaintext legalese warning at the top of all recent IDL .SAV files).

– Craig DeForest, SWRI

Example: the ‘points in polygon problem’



Example: the ‘points in polygon problem’



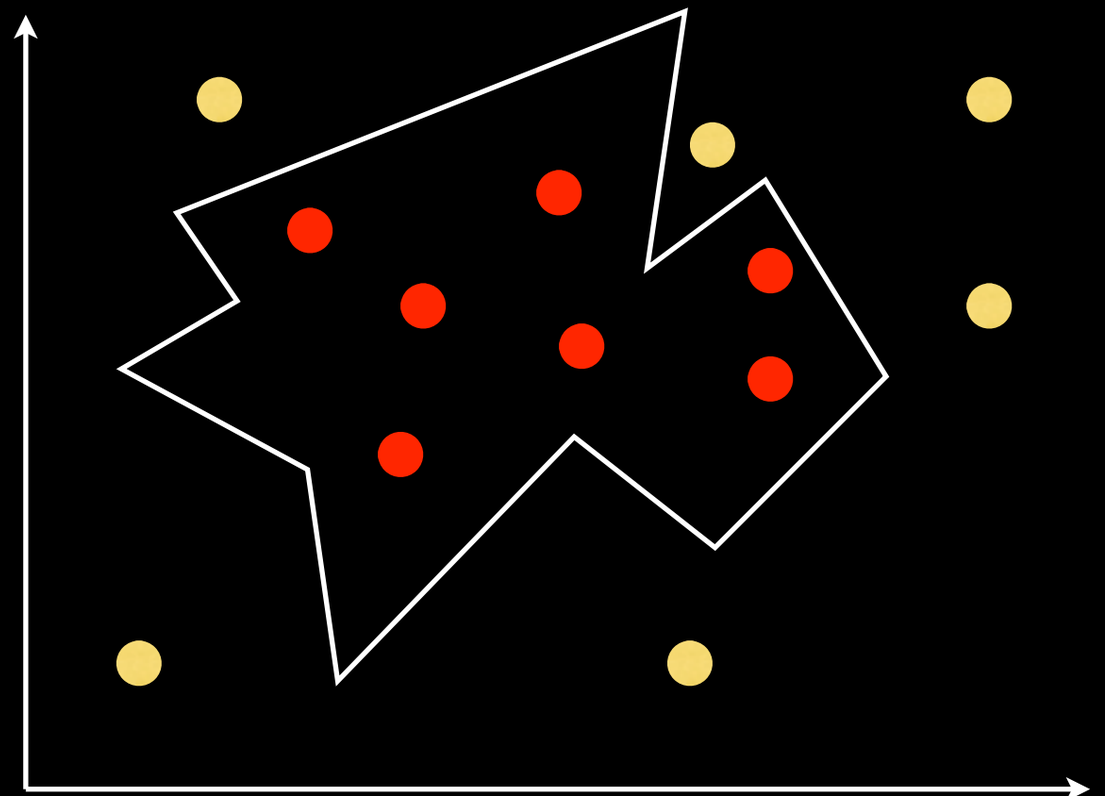
C code

Here is the code, for reference. Excluding lines with only braces, there are only *7 lines* of code.

```
int pnpoly(int nvert, float *vertx, float *verty, float testx, float testy)
{
  int i, j, c = 0;
  for (i = 0, j = nvert-1; i < nvert; j = i++) {
    if ( ((verty[i]>testy) != (verty[j]>testy)) &&
         (testx < (vertx[j]-vertx[i]) * (testy-verty[i]) / (verty[j]-verty[i]) + vertx[i]) )
       c = !c;
  }
  return c;
}
```

Argument	Meaning
nvert	Number of vertices in the polygon. Whether to repeat the first vertex at the end is discussed below.
vertx, verty	Arrays containing the x- and y-coordinates of the polygon's vertices.
testx, testy	X- and y-coordinate of the test point.

Example: the ‘points in polygon problem’



PDL code

```
sub pnpoly{
   my($tx, $ty, $vertx, $verty) = @_; # ALL vectors!
   my $testx = $tx->dummy(0);
   my $testy = $ty->dummy(0);
   my $vertxj = $vertx->rotate(1);
   my $vertyj = $verty->rotate(1);
   $c =    sumover(    (($verty>$testy) != ($vertyj>$testy)) &
             ($testx < ($vertxj-$vertx) * ($testy-$verty) /
             ($vertyj-$verty) + $vertx)
             ) %2;
  return $c; # Vector of 1's and 0's
}
```

Example: the 'points in polygon problem'

IDL code

```
FUNCTION Inside, x, y, px, py

;  x - The x coordinate of the point.
;  y - The y coordinate of the point.
; px - The x coordinates of the polygon.
; py - The y coordinates of the polygon.
;
; The return value of the function is 1 if the point is inside the
; polygon and 0 if it is outside the polygon.

    sx = Size(px)
    sy = Size(py)
    IF (sx[0] EQ 1) THEN NX=sx[1] ELSE RETURN, -1     ; Error if px not a vector
    IF (sy[0] EQ 1) THEN NY=sy[1] ELSE RETURN, -1     ; Error if py not a vector
    IF (NX EQ NY) THEN N = NX ELSE RETURN, -1         ; Incompatible dimensions

    tmp_px = [px, px[0]]                              ; Close Polygon in x
    tmp_py = [py, py[0]]                              ; Close Polygon in y

    i = indgen(N)                                     ; Counter (0:NX-1)
    ip = indgen(N)+1                                  ; Counter (1:nx)

    X1 = tmp_px(i)  - x
    Y1 = tmp_py(i)  - y
    X2 = tmp_px(ip) - x
    Y2 = tmp_py(ip) - y

    dp = X1*X2 + Y1*Y2                                ; Dot-product
    cp = X1*Y2 - Y1*X2                                ; Cross-product
    theta = Atan(cp,dp)

    IF (Abs(Total(theta)) GT !PI) THEN RETURN, 1 ELSE RETURN, 0
END
```

PDL vs SciPy

(or Perl vs Python)

Python: more of a 'bondage & discipline' language – style is coerced (e.g. indents!), everything is an object.

Perl: free form expression, variety of styles, more rope to hang yourself

SciPy: adopted by STScI, IRAF

PDL: faster, easier to extend with your own vector code

Two cool things

Warning: deep Nerd territory!

Inline:PDLpp

Automatically vectorized C extensions

```
use Inline Pdlpp; # the actual code is in the __Pdlpp__ block below
$a = 10+sequence 10; $b = random(10); $c=sequence(10)*20;
print $a->myfunc($b,$c),"\n";

$x = random(1000,2000);
$y = $x->tcumul;  # Output is a vector of length 2000

__DATA__
__Pdlpp__
pp_def('myfunc',
        Pars => 'a(); b(); c(); [o] o()',
        Code => '$o() = $a()*$b() + $c();',
       );

pp_def('tcumul',
        Pars => 'in(n);[o] mul()',
        Code => '$mul() = 1;
                 loop(n) %{
                   $mul() *= $in();
                 %}',
);
# end example script
```

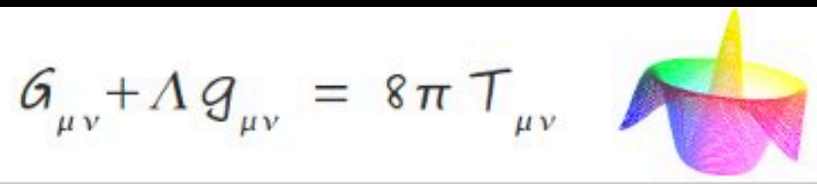
PDL::ParallelCPU

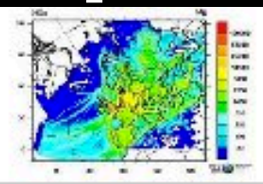
(Experimental)

```
# Set target of 4 parallel pthreads to create, with a lower limit of
#  5Meg elements for splitting processing into parallel pthreads.
set_autopthread_targ(4);
set_autopthread_size(5);

$a = zeroes(5000,5000); # Create 25Meg element array

$b = $a + 5; # Processing will be split up into multiple pthreads

# Get the actual number of pthreads for the last
#  processing operation.
$actualPthreads = get_autopthread_actual();
```

Where to start: pdl.perl.org

Where to start: pdl.perl.org





Google™ Custom Search Search

About PDL
- Home
- Get PDL
- Screenshots
- PDL Users
- Mailing Lists

Documentation
- FAQ
- Tutorials
- Modules
- Course
- Index

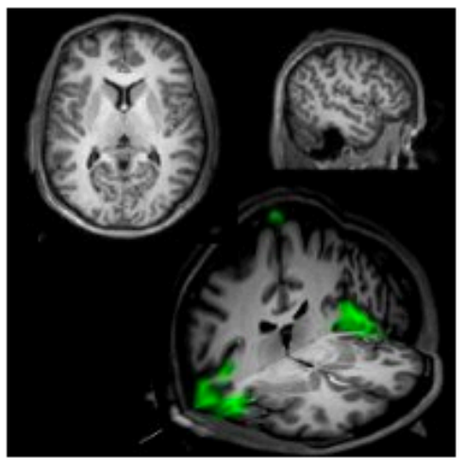
Resources
- Demos
- Presentations
- PDL Wiki
- External Libs

Development
- Get Started
- Browse Git
- Bugs
- Patches
- Credits

sourceforge

Perl Data Language

Scientific computing with Perl



PDL ("Perl Data Language") gives standard Perl the ability to *compactly* store and *speedily* manipulate the large N-dimensional data arrays which are the bread and butter of scientific computing.

PDL turns Perl in to a free, array-oriented, numerical language similar to (but, we believe, better than) such commerical packages as IDL and MatLab. One can write simple perl expressions to manipulate entire numerical arrays all at once. Simple interactive shells, `pdl2` and `perldl`, are provided for use from the command line along with the `PDL` module for use in Perl scripts.

Recent News

2011-08-13 Support for large (>2GB) pdls in PDL

2011-05-21 Multi-Core support available in git and on CPAN

2011-05-11 SciPDL Package for Mac OS X released

2011-04-11 PDL 2.4.9 released

2011-04-06 inSCIght Scientific Computing Podcast about PDL

Where to start: pdl.perl.org

About PDL
- Home
- Get PDL
- Screenshots
- PDL Users
- Mailing Lists

Documentation
- FAQ
- Tutorials
- Modules
- Course
- Index

Resources
- Demos
- Presentations
- PDL Wiki
- External Libs

Development
- Get Started
- Browse Git
- Bugs
- Patches
- Credits

sourceforge

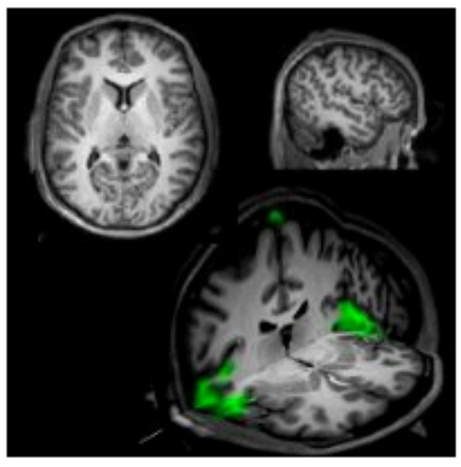
Perl Data Language

Scientific computing with Perl

get... PDL

get... Docs

get... Help



PDL ("Perl Data Language") gives standard Perl the ability to *compactly* store and *speedily* manipulate the large N-dimensional data arrays which are the bread and butter of scientific computing.

PDL turns Perl in to a free, array-oriented, numerical language similar to (but, we believe, better than) such commerical packages as IDL and MatLab. One can write simple perl expressions to manipulate entire numerical arrays all at once. Simple interactive shells, `pdl2` and `perldl`, are provided for use from the command line along with the `PDL` module for use in Perl scripts.

Recent News

2011-08-13 Support for large (>2GB) pdls in PDL

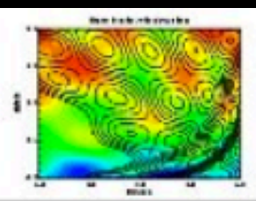
2011-05-21 Multi-Core support available in git and on CPAN

2011-05-11 SciPDL Package for Mac OS X released

2011-04-11 PDL 2.4.9 released

2011-04-06 inSCIght Scientific Computing Podcast about PDL

$$x_1 = x_0 - \frac{y_0}{f'(x_0)}$$



Google™ Custom Search Search

About PDL
Home
Get PDL
Screenshots
PDL Users
Mailing Lists

Documentation
FAQ
Tutorials
Modules
Course
Index

Resources
Demos
Presentations
PDL Wiki
External Libs

Development
Get Started
Browse Git
Bugs
Patches
Credits

sourceforge

Install PDL

New to Perl?
No worries! If you are using Linux or Mac OS X, you should already have Perl installed. If you are a Windows user, you can use Strawberry Perl or Active Perl.

I am looking for ...

- (•) The easiest possible install.
- ( ) The latest version of PDL.
- ( ) A customized installation.

Pre-Built Binaries

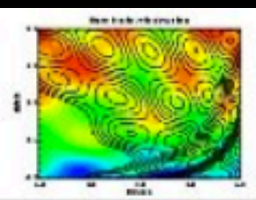
My platform is ...

- ( ) Windows.
- ( ) Ubuntu / Debian.
- ( ) Fedora.
- (•) Mac OS X.
- ( ) Mandriva.
- ( ) OpenSUSE.

Easiest install - Mac O X

Install "SciPDL" binary.

$$x_1 = x_0 - \frac{y_0}{f'(x_0)}$$



Google™ Custom Search Search

About PDL
Home
Get PDL
Screenshots
PDL Users
Mailing Lists

Documentation
FAQ
Tutorials
Modules
Course
Index

Resources
Demos
Presentations
PDL Wiki
External Libs

Development
Get Started
Browse Git
Bugs
Patches
Credits

sourceforge

Install PDL

New to
No worrie
installed.

I am looking

- The easiest p
- The latest ver
- A customized

Easiest

Install "Sc