i386-linux-thread-multUserGContributed Perl i386-linux-thread-multi::PGPLOT(3)

NAME

PGPLOT - allow subroutines in the PGPLOT graphics library to be called from Perl.

SYNOPSIS

```
 use PGPLOT;

 pgbegin(0,"/xserve",1,1);
 pgenv(1,10,1,10,0,0);
 pglabel('X','Y','My plot');
 pgpoint(7,[2..8],[2..8],17);

 # etc...

 pgend;
```

DESCRIPTION

Originally developed in the olden days of Perl4 (when it was known as 'pgperl' due to the necessity of making a special perl executable) PGPLOT is now a dynamically loadable perl module which interfaces to the FORTRAN graphics library of the same name.

PGPLOT, originally developed as a FORTRAN library, is now available with C bindings (which the Perl module uses), though a FORTRAN compiler is still required to build it.

For every PGPLOT C/FORTRAN function the module provides an equivalent Perl function with the same arguments. Thus the user of the module should refer to the PGPLOT manual to learn all about how to use PGPLOT and for the complete list of available functions. This manual comes with the PGPLOT distribution and is also available at the WWW address:

http://astro.caltech.edu/~tjp/pgplot/

Also refer to the extensive set of test scripts ("test*.p") included in the module distribution for examples of usage of all kinds of PGPLOT routines.

How the FORTRAN/C function calls map on to Perl calls is detailed below.

ARGUMENT MAPPING - SIMPLE NUMBERS AND ARRAYS

This is more or less as you might expect - use Perl scalars and Perl arrays in place of FORTRAN/C variables and arrays.

Any FORTRAN REAL/INTEGER/CHARACTER* scalar variable maps to a Perl scalar (Perl doesn't care about the differences between strings and numbers and ints and floats).

Thus you can say:

To draw a line to point (42,$x):

```
 pgdraw(42,$x);
```

To plot 10 points with data in Perl arrays @x and @y with plot symbol no. 17. Note the Perl arrays are passed by reference:

```
 pgpoint(10, \@x, \@y, 17);
```

You can also use the old Perl4 style:

```
 pgpoint(10, *x, *y, 17);
```

but this is deprecated in Perl5.

Label the axes:

```
 pglabel("X axis", "Data units", $label);
```

Draw ONE point, see how when "N=1" "pgpoint()" can take a scalar as well as a array argument:

```
  pgpoint(1, $x, $y, 17);
```

ARGUMENT MAPPING - IMAGES AND 2D ARRAYS

Many of the PGPLOT commands (e.g. "pggray") take 2D arrays as arguments. Several schemes are provided to allow efficient use from Perl:

1. Simply pass a reference to a 2D array, e.g:

```
  # Create 2D array

  $x=[];
  for($i=0; $i<128; $i++) {
     for($j=0; $j<128; $j++) {
       $$x[$i][$j] = sqrt($i*$j);
     }
  }
  pggray( $x, 128, 128, ...);
```

2. Pass a reference to a 1D array:

```
  @x=();
  for($i=0; $i<128; $i++) {
     for($j=0; $j<128; $j++) {
       $x[$i][$j] = sqrt($i*$j);
     }
  }
  pggray( \@x, 128, 128, ...);
```

   Here @x is a 1D array of 1D arrays. (Confused? - see perldata(1)). Alternatively @x could be a flat 1D array with 128x128 elements, 2D routines such as "pggray()" etc. are programmed to do the right thing as long as the number of elements match.

3. If your image data is packed in raw binary form into a character string you can simply pass the raw string. e.g.:

```
   read(IMG, $img, 32768);
   pggray($img, $xsize, $ysize, ...);
```

   Here the "read()" function reads the binary data from a file and the "pggray()" function displays it as a grey-scale image.

   This saves unpacking the image data in to a potentially very large 2D perl array. However the types must match. The string must be packed as a "f*" for example to use "pggray". This is intended as a short-cut for sophisticated users. Even more sophisticated users will want to download the "PDL" module which provides a wealth of functions for manipulating binary data.

   PLEASE NOTE: As PGPLOT is a Fortran library it expects it's images to be be stored in row order. Thus a 1D list is interpreted as a sequence of rows end to end. Perl is similar to C in that 2D arrays are arrays of pointers thus images end up stored in column order.

Thus using perl multidimensional arrays the coordinate ($i,$j) should be stored in $img[$j][$i] for things to work as expected, e.g:

```
$img = [];
for $j (0..$nx-1) for $i (0..$ny-1) {
   $$img[$j][$i] = whatever();
}}
pggray($$img, $nx, $ny, ...);
```

Also PGPLOT displays coordinate (0,0) at the bottom left (this is natural as the subroutine library was written by an astronomer!).

ARGUMENT MAPPING - FUNCTION NAMES

Some PGPLOT functions (e.g. "pgfunx") take functions as callback arguments. In Perl simply pass a subroutine reference or a name, e.g.:

```
 # Anonymous code reference:

 pgfunx(sub{ sqrt($_[0]) },  500, 0, 10, 0);

 # Pass by ref:

 sub foo {
   my $x=shift;
   return sin(4*$x);
 }

 pgfuny(\&foo, 360, 0, 2*$pi, 0);

 # Pass by name:

 pgfuny("foo", 360, 0, 2*$pi, 0);
```

ARGUMENT MAPPING - GENERAL HANDLING OF BINARY DATA

In addition to the implicit rules mentioned above PGPLOT now provides a scheme for explictly handling binary data in all routines.

If your scalar variable (e.g. $x) holds binary data (i.e. ’packed’) then simply pass PGPLOT a reference to it (e.g. "\$x"). Thus one can say:

```
   read(MYDATA, $wavelens, $n*4);
   read(MYDATA, $spectrum, $n*4);
   pgline($n, \$wavelens, \$spectrum);
```

This is very efficient as we can be sure the data never gets copied and will always be interpreted as binary.

Again see the "PDL" module for sophisticated manipulation of binary data. "PDL" takes great advantage of these facilities.

Be VERY careful binary data is of the right size or your segments might get violated.