Perl Interface to GrADS

From OpenGrads Wiki

The Perl interface to GrADS is an alternative method of scripting GrADS that can take advantage of the unique capabilities of Perl (http://en.wikipedia.org/wiki/Perl). Here are a few reasons for scripting GrADS in Perl:

Contents

- You are an experienced Perl (http://en.wikipedia.org/wiki/Perl) programmer new to GrADS and do not want to spend the time learning a new scripting language.

- You need some GrADS functionality inside your regular Perl script, say, parse the contents of a GrADS readable dataset or want to store your metadata in a MySQL database.
- You would like to transparently issue GraDS commands inside your cgi-bin (http://en.wikipedia.org/wiki/Common_Gateway_Interface) script and produce images for your dynamic website.
- You want to query your OpenDAP (http://www.opendap.org/index.html) server and figure out which is the latest forecast available before actually opening the dataset.
- You would like to use PerlTK (http://www.perltk.org/) or any other toolkit to write a Graphical User Interface for GrADS.
- Your script is getting too complex and you could use an object oriented approach to better organize and reuse your code.
- You would like to explore GrADS ability to slice and dice a meteorological dataset or OpenDAP URL, but prefer to use Perl Data Language (PDL) (http://pdl.perl.org/) to perform further analysis and visualization your dataset.

The Perl interface to GrADS, which is similar to the Python interface, enables full scripting capability for GrADS in Perl, and can be used together with the classic GrADS scripting language (http://grads.iges.org/grads/gadoc/script.html#intro). This interface comes in 2 flavors: 1) an object oriented module (`Grads.pm`), and 2) a more procedural module (`Grads::Gerl.pm`), built on top of Grads.pm, that has more of the look and feel of a classic GrADS script. For interactive work you may want to try `gadl`, a customization of the Perl Data Language (PDL) for GrADS.

In the remaining of this document I will assume that you have some familiarity with GrADS and Perl.

Getting Started

Requirements

You will need Perl (http://www.perl.org) 5 and GrADS (http://grads.iges.org/grads) Version 1.9.0 or later, or any OpenGRADS release. If you would like to use GrADS with the Perl Data Language (PDL) (http://pdl.perl.org) you will need to install this Perl module as well. These can be downloaded at their respective websites. These packages are available for most Linux distributions, MacOS X and Microsoft Windows, as well as in many flavors of Unix.

Downloading the software and sample datasets

The Perl GrADS modules can be downloaded from the OpenGrADS download (http://sourceforge.net/project/showfiles.php?group_id=161773&

package_id=256759) area at SourceForge.

For running the tutorials and examples below you need to download the sample data file model.nc (http://opengrads.org/test_data/model.nc).

Installation

This installs the usual way. Start by untarring the distribution:

```
% tar xvfz gerl-1xx.tar.gz
% cd gerl-1.xx
```

If you have admin privileges:

```
% perl Makefile.PL
% make
% make test
% make install  (needs admin privileges)
```

or else

```
% perl Makefile.PL PREFIX=$HOME   (for example)
% make
% make test
% make install
```

In this case, make sure to set the environment variable `PERL5LIB` to something like:

```
$HOME/perl5/site_perl/5.8.8
```

and that you have the following directory in your `PATH`

```
$HOME/local/bin
```

Grads or Gerl: Deciding which Interface is Right for You

In the remainder of this document we will use `Grads` and `Grads::Gerl` to denote the Perl modules of same name, and GrADS to refer to the GrADS application that these modules interface to.

There are two basic modules implementing the Perl interface to the GrADS application:

Grads

The module `Grads` implements an *object oriented interface* to GrADS by means of bi-directonal pipes. It starts GrADS, sends commands to it, parses its output and return codes, and provide high level interfaces to query GrADS properties and dimension environment.

Gerl

The module `Grads::Gerl` implements a *procedural interface* to GrADS. `Grads::Gerl` simply instantiates a single `Grads` object, and provides several functions that attempt to emulate the look and feel of the traditional GrADS application.

Let's examine an example comparing the two approaches. Consider a simple script that starts GrADS, opens a file and displays a variable using `Gerl`:

```
use Grads::Gerl;
grads { Bin=>"gradsnc", Window=>1 };
Open "model.nc";
display "ua;va";
quit;
```

Using the OO interface implemented in `Grads` this example would look like:

```
use Grads;
$ga = new Grads { Bin=>"gradsnc", Window=>1 };
$ga->Open "model.nc";
$ga->cmd "display "ua;va";
$ga = undef;
```

These are not that different, but as the TUTORIAL secions will illustrate, `Grads::Gerl`'s syntax is closer to the traditional GrADS command line, and provides a nice facility to evaluate a batch of GrADS commands with a convenient way for catching exceptions.

The wrapper script `gadl` goes a step further: it customizes the Perl Data Language (PDL) shell (`perldl`), automatically loading `Gerl` and providing a command line interface to GrADS. This combination provides a powerful and convenient environment for advanced geophysical data analysis and visualization. PDL complements GrADS quite nicely with a wealth of numerical methods and visualization tools. Here is how one would write the example above using `gadl`. From your OS command line you start `gadl`:

```
% gadl -nc
```

Then at the `gadl->` command line prompt enter

```
o model.nc
d ua;va
q
```

No semi-colons, no double quotes, and even additional shortcuts such as `o` for "Open". See the COMMAND LINE FILTER FOR PDL section in the manual page (http://opengrads.org/doc/perl/Gerl/) for a description of all the shorthands provided by `Grads::Gerl` when running under `perldl`.

The next two sections present a quick tutorial to modules `Grads` and `Grads::Gerl`. Consult the Manual Pages (http://opengrads.org/doc/) for a detailed description of these modules.

Tutorial I: Object Oriented Interface (Grads)

For running this tutorial you will need a sample GrADS dataset. Please download *model.nc* from

```
http://opengrads.org/sample_data.
```

If you are new to GrADS you may want to read the Tutorial on

```
http://grads.iges.org/grads/gadoc/tutorial.html
```

This document is not a GrADS tutorial but rather a tutorial of the Perl interface to GrADS.

In this tutorial we will use the `Data::Dumper` module to examine the contents of hashes returned by some of the methods. So, we start by using these 2 modules:

```
use  Grads;
use  Data::Dumper;
```

Let's create a `Grads` object by starting the *gradsnc* binary in landscape mode, with an active graphics window:

```
$ga = new Grads { Bin=>"gradsnc", Port=>0, Window=>1, };
```

The *cmd* method is used to send generic commands to GrADS, e.g.,

```
$rc = $ga->cmd("q config");
```

The return code `$rc` will be `0` if all went well, and non-zero if the particular GrADS command exited with an error.

Opening files

The *Open* method opens a GrADS dataset in any of the supported formats:

```
$fh = $ga->Open("wrong_file.nc") or
      warn ">>> Cannot open wrong_file.nc as expected, proceeding.\n";
```

In this particular case we fed it a bogus file name to force an error condition. Let's try again, this time with the *model.nc* file that you just downloaded:

```
$fh = $ga->Open("model.nc") or
  die "cannot open model.nc but it is needed for testing, aborting ";
```

Open returns a file handle `$fh` with useful metadata about the file just read. You can use *Dumper* for examining the contents of `$fh`:

```
print "\n>>> File opened: " . Dumper($fh) . "\n";
```

A slightly reformatted output follows:

```
 $fh = {
        'fid'      => '1',
        'bin'      => 'model.nc',
        'desc'     => 'model.nc',
        'title'    => '',
        'type'     => 'Gridded'
        'nvars'    => 8,
        'vars'     => [ 'ps','ts','pr','ua','va','zg','ta','hus' ],
        'var_levs' => [ '0',  '0', '0', '7', '7','7', '7', '7'   ],
        'nx'       => '72',
        'ny'       => '46',
        'nz'       => '7',
        'nt'       => '5',
      };
```

Querying the GrADS state

Similarly, the *Query* method returns a query handle with information about the particular GrADS property. Here are a few examples:

```
$qh = $ga->Query("time");
print "\n>>> Time handle: ";   print Dumper($qh);

$fh = $ga->Query("file");
```

```
print "\n>>> File handle: ";   print Dumper($fh);

$dh = $ga->Query("dims");
print "\n>>> Dim handle: ";    print Dumper($dh);
```

As of this writing only a handfull of GrADS `query` properties are implemented by the *Query* method, but this list is growing with each release. Be sure to contribute any extension you add. In the meantime, you can use the *rword* and *rline* methods to parse the output of native `query` command. Read on.

Parsing GrADS output, the traditional way

Traditionally, the built in GrADS scripting language (*gs*) includes functions `sublin` and `subwrd` to parse the lines and words within line of each GrADS command issued. To aid the conversion of *gs* scripts to Perl, we have included methods *rword* and *rline* which give access to words and lines in the GrADS output stream. The *rword*(i,j) method returns the *j*-th word in the *i*-th line of the GrADS command just issued, viz.

```
$ga->cmd("q config");
for $i ( 1...$ga->{nLines} ) {
    printf("RWORD %3d: ", $i);
    $j=1;  # starts from 1
    while ( $word=$ga->rword($i,$j) ) {
        print $word . " ";
        $j++;
    }
    print "\n";
    $i++;
}
```

To obtain a given output line, use the *rline* method:

```
print "RLINE  *3: " . $ga->rline(3) . "\n\n";
```

The Set method

The *Set* method is used to issue a batch of GrADS `set` commands, stored in an array. This is particularly useful to define a graphics context that can be reused prior to issuing each `display` command. Here is an example:

```
my @gc;
$ga->{Echo} = 1;
push @gc, "grads off";
push @gc, "gxout shaded";
$rc = $ga->Set(\@gc);
```

Interfacing to the Perl Data Language (PDL)

Method *Exp* allows you to export a GrAS variable into a Perl Data Language (PDL) object, commonly refered to as *piddles*:

```
$ps = $ga->Exp ps;
print "ps = $ps";
```

The output piddle `ps` is sometimes refered to as a *GrADS field* as it register a grid,

```
$grid = $ps->gethdr()
```

which contains information about the coordinate variables (longitude, latitude, vertical level and time), in addition to low level metadata for exchanging data with GrADS. (This is the same concept of *field* introduced by Earth-system Modeling Framework, ESMF). Alternatively, you can explicitly grab the `$grid` during export,

```
($ps,$grid) = $ga->Exp ps;
```

You can also import a piddle with the *Imp* method, provided one specifies the necessary `grid` metadata:

```
$logps = log($ps);
$rc = $ga->Imp logps, $logps, $grid;
$rc = $ga->display logps;
```

Of course, there is more than one way of doing this. Having a `$grid` you can use the piddle's `sethdr()` method to regsiter it:

```
$logps->sethdr($grid);
```

and then there is no need to explicitly pass `$grid` to method `Imp`:

```
$rc = $ga->Imp logps, $logps;
$rc = $ga->Display logps;
```

Given the appropriate metadata, one can also display a piddle in GrADS. If all one wants to do is to display a variable there is no need to import it first as we did above. In the previous example, you could display `$logps` directly:

```
$rc = $ga->Display $logps;
```

The current implementation requires that both `x` and `y` dimensions be varying. In addition, both `z` and `t` dimensions can be varying, individually or at the same time. Just like GrADS itself, `Display` cannot handle both `z` and `t` varying at the same time. Here is an example exporting a 4D variable:

```
$ga->cmd "set t 1 5";
$ga->cmd "set z 1 7";
$ua = Exp ua;
print $ua->dims;
```

Terminating your GrADS session

As usual in perl, the `Grads` destructor will be invoked when the object gets out of scope of when it is explitly undefined like this:

```
$ga = undef;
```

This will cause a `quit` to be sent to GrADS which in turn will end the connection.

Tutorial II: Procedural Interface (Gerl)

Since `Gerl` is based on the `Grads` module you are strongly encouraged to read the previous tutorial first.

For running this tutorial you will need the same sample GrADS dataset used in **Tutorial I**. Please download *model.nc* from

```
http://opengrads.org/sample_data.
```

If you are new to GrADS you may want to read the Tutorial on

```
http://grads.iges.org/grads/gadoc/tutorial.html
```

This document is not a GrADS tutorial but rather a tutorial of the Perl interface to GrADS.

```
In this example we will use the Data::Dumper module to examine the contents of hashes returned by some o
```

```
use  Grads::Gerl;
```

```
use  Data::Dumper;
```

To start the GrADS process we use the `grads` function:

```
$rc = grads { Bin=>"gradsnc", Window=>1 };
```

For this and subsequent function calls, the return code variable `$rc` will be zero if the command completed successfully and non-zero if an error occurred. Usually you would follow the command above with something like this:

```
die "cannot start grads" if ( $rc );
```

The function *ga_*() is used to send generic commands to GrADS, e.g.,

```
$rc = $ga "q config";
```

Opening files

The *Open*() function opens a GrADS dataset in any of the supported formats:

```
$fh = Open "wrong_file.nc" or
      warn ">>> Cannot open wrong_file.nc as expected, proceeding.\n";
```

In this particular case we fed it a bogus file name to force an error condition. Let's try again, this time with the *model.nc* file that you just downloaded:

```
$fh = Open "model.nc" or
  die "cannot open model.nc but it is needed for testing, aborting ";
```

Open() returns a file handle `$fh` with useful metadata about the file just read. You can use *Dumper* for examining the contents of `$fh`:

```
print "\n>>> File opened: " . Dumper($fh) . "\n";
```

A slightly reformatted output follows:

```
$fh = {
       'fid'      => '1',
       'bin'      => 'model.nc',
       'desc'     => 'model.nc',
       'title'    => '',
       'type'     => 'Gridded'
```

```
        'nvars'    => 8,
        'vars'     => [ 'ps','ts','pr','ua','va','zg','ta','hus' ],
        'var_levs' => [ '0',  '0', '0', '7', '7','7', '7', '7'   ],
        'nx'       => '72',
        'ny'       => '46',
        'nz'       => '7',
        'nt'       => '5',
      };
```

Basic syntax

Next we show examples of some simple grads commands. Notice how the word `print` is always escaped to avoid conflict with Perl's `print` command:

```
enable 'print gerl-test.gx';
set lev, 500;
set x, 1;
set gxout, shaded;
```

The use of commas in the previous block will come naturally to perl programmers, but may be confusing for someone coming from the classic Grads command line with no exposure to Perl. One of Perl's mottos is *There is more than one way to do it* (TIMTOWTDI, usually pronounced *Tim Toady*), and the first `set` command above could be written in a number of equivalent ways:

```
set("lev","500');
set(lev,500);     # this will not work is lev is defined somewhere
                  # a function name
set("lev 500");
set lev, 500;
set "lev 500";
ga_ "set lev 500";
```

However, the syntax

```
set lev 500
```

does not work because the arguments of `set` must be specified as a list. The `gadl` front-end to `perldl` removes this restriction as it provides a command line filter to special handle GrADS commands.

Like any Perl script, flow control and variable interpolation works as usual:

```
foreach $var ( ps, ta ) {
    foreach $t (1..5) {
        set t, $t;
        define tz, "ave($var,lon=0,lon=360)";
        display tz;
```

```
        draw title, "Zonal Means: $var at t=$t";
        printim "gerl-$var-$t.png";
        gaprint;  # print is already taken, so use gaprint instead
        clear;
    }
}
```

Executing native GrADS commands in a *here* document

A unique feature of `Gerl` compared to its sibiling `Grads` is that one can enter a batch of GrADS commands in the form of a *here* document as in this code fragment:

```
$rc = ga_ <<"EOF";
   set lev 700
   set lon 0 360
   set t 2
   d ua;va;sqrt(ua*ua+va*va)
   draw title Winds at 700 hPa
   printim gerl-winds.png
EOF
```

Querying the GrADS state

These commands are sent to GrADS, one at time, and in case of a non-zero error return `ga_`() stops execution and returns the first non-zero error code it encounters. This feature is particularly useful to embed existing GrADS code without the need to convert to Perl syntax. (A really useful feature would be a function that evaluates code fragments from the GrADS built in scripting language.)

Like the `Open`() function, *Query*() returns a query handle with information about the particular GrADS property. Here are a few examples:

```
$qh = Query "time";
print "\n>>> Time handle: ";   print Dumper($qh);

$qh = Query "file";
print "\n>>> Time handle: ";   print Dumper($qh);

$dh = Query "dims";
print "\n>>> Dim handle: ";    print Dumper($dh);
```

As of this writing only a handfull of GrADS `query` properties are implemented by *Query*(), but this list is growing with each release. Be sure to contribute any extension you add. In the meantime, you can use the *rword* and *rline* functions to parse the output of native `query` command. Read on.

Parsing GrADS output, the traditional way

Traditionally, the built in GrADS scripting language (*gs*) includes functions `sublin` and `subwrd` to parse the lines and words within line of each GrADS command issued. To aid the conversion of *gs* scripts to Perl, we have included functions *rword* and *rline* which give access to words and lines in the GrADS output stream. The *rword*(i,j) function returns the *j*-th word in the *i*-th line of the GrADS command just issued, viz.

```
ga_ "q config");
for $i ( 1...$Grads::Gerl::Ga->{nLines} ) {
    printf("RWORD %3d: ", $i);
    $j=1;  # starts from 1
    while ( $word=rword($i,$j) ) {
        print $word . " ";
        $j++;
    }
    print "\n";
    $i++;
}
```

To obtain a given output line, use the *rline* function:

```
print "RLINE  *3: " . rline(3) . "\n\n";
```

The Set command

The *Set*() function is used to issue a batch of GrADS `set` commands, stored in an array. This is particularly useful to define a graphics context that can be reused prior to issuing each `display` command. Here is an example:

```
my @gc;
$Grads::Gerl::Ga->{Echo} = 1;
push @gc, "grads off";
push @gc, "gxout shaded";
$rc = Set \@gc;
```

Interfacing to the Perl Data Language (PDL)

Function *Exp* allows you to export a GrAS variable into a Perl Data Language (PDL) object, commonly refered to as *piddles*:

```
 $ps = Exp ps;
 print "ps = $ps";
```

The output piddle `ps` is sometimes refered to as a *GrADS field* as it register a grid,

```
$grid = $ps->gethdr()
```

which contains information about the coordinate variables (longitude, latitude, vertical level and time), in addition to low level metadata for exchanging data with GrADS. (This is the same concept of *field* introduced by Earth-system Modeling Framework, ESMF). Alternatively, you can explicitly grab the `$grid` during export,

```
($ps,$grid) = Exp ps;
```

You can also import a piddle with the *Imp* function, provided one specifies the necessary `grid` metadata:

```
$logps = log($ps);
Imp logps, $logps, $grid;
display logps;
```

Of course, there is more than one way of doing this. Having a `$grid` you can use the piddle's `sethdr()` function to regsiter it:

```
$logps->sethdr($grid);
```

and then there is no need to explicitly pass `$grid` to function `Imp`:

```
$Imp logps, $logps;
Display logps;
```

Given the appropriate metadata, one can also display a piddle in GrADS. If all one wants to do is to display a variable there is no need to import it first as we did above. In the previous example, you could display `$logps` directly:

```
Display $logps;
```

The current implementation requires that both `x` and `y` dimensions be varying. In addition, both `z` and `t` dimensions can be varying, individually or at the same time. Just like GrADS itself, `Display` cannot handle both `z` and `t` varying at the same time. Here is an example exporting a 4D variable:

```
ga_ "set t 1 5";
ga_ "set z 1 7";
$ua = Exp ua;
```

```
print $ua->dims;
```

Terminating your GrADS session

To terminate your GrADS session just enter:

```
quit;
```

This will cause a `quit` to be sent to GrADS which in turn will end the connection.

Manual Pages

`Grads.pm`: Object Oriented Interface

Consult the Grads.pm Manual Page (http://opengrads.org/doc/perl/Grads/) translated from POD. Do not edit these pages, instead get the the source code `Grads.pm`, update the documentation there, and use `pod2html` to translate to html.

`Gerl.pm`: Procedural Interface

Consult the Gerl.pm Manual Page (http://opengrads.org/doc/perl/Gerl/) translated from POD. Do not edit these pages, instead get the the source code `Gerl.pm`, update the documentation there, and use `pod2html` to translate to html.

`Gadl`: An interactive shell based on PerlDL

Consult the Gadl Manual Page (http://opengrads.org/doc/perl/gadl) translated from POD. Do not edit these pages, instead get the the source code `gadl`, update the documentation there, and use `pod2html` to translate to html.

- This page was last modified on 7 December 2008, at 16:50.