Appendix B

A brief introduction to IDL/GDL

Although the exercises may be done in any true programming language that the student is familiar with (e.g. C, C++, F77, F90, F95, PASCAL, ...), we will give support mainly in a simple programming environment called IDL (Interactive Data Language). Since IDL is proprietary software, and licenses are extremely expensive, we will make use of an open-source clone of IDL, called GDL (Gnu Data Language). This project is still under development, so it will not have the complete functionality of IDL. However, it will be sufficient for making small programs and making plots.

The main advantages of using GDL for the exercises compared to many other programming languages are:

1. The programming language is *interpreter-based*, meaning that you do not compile a program first and then run it from the command-line, but you go into GDL, get a GDL-prompt and you can do command-line calculations. The prompt is so-to-speak the main routine, and any command you type is immediately executed. It is therefore an *interactive* programming language. At the same time you can also run a program (which is almost - but not 100% - the same as a series of command-line statements bound together in a file or in a subroutine).

2. You can plot the results of your programs immediately, without having to write results, read them into GNUPLOT or something similar.

3. If a program stops due to a STOP statement or an error, then the prompts is stuck right there in the subroutine where the stop or error took place. So you can type "print, a" to see what variable a contains, even if this is a local variable in a subroutine. By deliberately putting in STOP statements into a program that contains an unfixed bug you can debug the code. A simple ".c" is sufficient to continue the program after that stop statement. This is the ultimate debugging tool!

4. The GDL programming language is very easy (although it has its nasty unlogical things, too): it is very similar to the BASIC programming language.

5. It has the possibility to handle large arrays at once, with a single command. This requires a bit of exercise, but can be very handy.

The disadvantages are:

1. GDL is slow as a programming language. For multi-D hydrodynamics modeling it is utterly inadequate. But we shall use it as an *interface* for external Fortran programs. In GDL we will set up the problem, write a set of input files for the Fortran program, then we run the Fortran program externally from GDL, and at the end we read the output files of the Fortran program into GDL for plotting. This will be done for all more complex problems involving multi-D hydrodynamics.

2. GDL is still an experimental package, so sometimes things that work brilliantly in IDL do not work in GDL. We will have to see when this takes place, although this presumably does not take place for the basic things we want to do.

B.1 Getting started

Let us make a few command-line exercises. First we go into GDL by typing:

```
> gdl
```

in the prompt (the > is the shell prompt). We then get a GDL prompt:

```
gdl>
```

From now on whenever there is a "gdl>" we mean that this is the prompt, and the text behind it is what we type in. Any next lines are usually the result of what we type. So now type

```
gdl> print,'Hello world'
Hello world
```

to get the famous sentence. We can define a variable:

```
gdl> a=7
gdl> print,a
 7
```

By typing a=7 instead of a=7. we tell GDL that a is an integer and not a floating-point variable. So of we do

```
gdl> a=7
gdl> print,a/2
 3
```

But now this:

```
gdl> a=7.
gdl> print,a/2
 3.5
```

Note that there is a difference between real (*8) floating points and doubleprecision (*16) floating points:

```
gdl> a=7.e0
gdl> print,sqrt(a),format='(E29.22)'
 2.64575123786926269531 25E+00
gdl> a=7.d0
gdl> print,sqrt(a),format='(E29.22)'
 2.64575131106459071617 11E+00
```

and notice the way that formatting works. To find out what type a variable has we can type

```
gdl> help,a
<Expression>    DOUBLE    =        7.0000000
```

Now type

```
gdl> a=fltarr(10)
gdl> print,a
      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000
```

This is an array of numbers. We can do

```
gdl> a=dblarr(10)
gdl> for i=0,9 do a[i]=i
gdl> print,a
      0.00000      1.00000      2.00000      3.00000
      4.00000      5.00000      6.00000      7.00000
      8.00000      9.00000
```

There is a powerful shortcut for this:

```
gdl> a=dindgen(10)
gdl> print,a
      0.00000      1.00000      2.00000      3.00000
      4.00000      5.00000      6.00000      7.00000
      8.00000      9.00000
gdl> help,a
A               DOUBLE    = Array[10]
```

Note that dblarr has a (*8) float variant called fltarr, and dindgen has findgen. Now type

```
gdl> a = a+6
gdl> print,a
       6.0000000       7.0000000       8.0000000       9.0000000
       10.000000       11.000000       12.000000       13.000000
       14.000000       15.000000
gdl> b=dindgen(10)
gdl> print,a-b
       6.0000000       6.0000000       6.0000000       6.0000000
       6.0000000       6.0000000       6.0000000       6.0000000
       6.0000000       6.0000000
```

This shows the ease with which one can manipulate data in GDL.

Now for some plotting

```
gdl> nx=100
gdl> xmin=-2*!pi
gdl> xmax=2*!pi
gdl> x=xmin+(xmax-xmin)*dindgen(nx)/(nx-1.d0)
gdl> y=sin(x)
gdl> plot,x,y
```

This should produce a nice sinus curve. Now try

```
gdl> plot,x,y,psym=6
gdl> plot,x,y,psym=-6
gdl> plot,x,y,psym=3
gdl> plot,x,y,line=2
gdl> window,2
gdl> plot,y,x
gdl> wset,0
gdl> plot,y,x
```

Now with titles and stuff

```
gdl> plot,x,y,xtitle='x',ytitle='y',title='A nice plot'
```

We can also constrain the plotting domain

```
gdl> plot,x,y,xrange=[-4,4]
```

Sometimes GDL thinks a bit too much, and does things not the way we want. We can force it to:

```
gdl> plot,x,y,yrange=[-1.2,1.2]
gdl> plot,x,y,yrange=[-1.2,1.2],ystyle=1
```

Color is a bit more complex in IDL. First we have to choose if we want to have full control over the RGB colors, or if we want to have a color-map (in which 256 colors are pre-chosen and you only have to give a number 0 to 255 to choose which color).

```
gdl> rc=1L
gdl> gc=256L
gdl> bc=256L*256L
gdl> device,decomposed=1
gdl> plot,x,y,color=120*rc,80*gc,250*bc
gdl> device,decomposed=0
gdl> loadct,3
% LOADCT: Loading table RED TEMPERATURE
gdl> plot,x,y,color=160
```

The 256L (the L) stands for long-integer. We can also do this

```
gdl> device,decomposed=0
gdl> loadct,12
% LOADCT: Loading table 16 LEVEL
gdl> plot,x,y,color=160
gdl> plot,x,y,/nodata,color=255
gdl> oplot,x,y,color=140
gdl> oplot,x,y*0.4,color=20
```

We can also plot to postscript files.

```
gdl> set_plot,'ps'
gdl> plot,x,y
gdl> device,/close
gdl> set_plot,'x'
```

This plots to the file idl.ps. Note that this time the plot is on a white background...
CAREFUL: The device,/close is crucual!
CAREFUL: It is useful to set the plot back to 'x' after any 'ps' plotting, because otherwise one may wonder why one does not see anything on the screen!
We can also directly specify the file name:

```
gdl> set_plot,'ps'
gdl> device,file='myplot.ps'
gdl> plot,x,y
gdl> device,/close
gdl> set_plot,'x'
```

In postscript, color works a bit different from x. We have, for lines, only color tables, and we must explicitly tell the device to use color and how 'true' the color should be:

```
gdl> set_plot,'ps'
gdl> device,file='myplot.ps',/color,bits_per_pixel=24
gdl> loadct,12
gdl> plot,x,y,color=160
gdl> device,/close
gdl> set_plot,'x'
```

Opening and writing files

```
gdl> a=dindgen(10)
gdl> openw,1,'mydata.out'
gdl> printf,1,a
gdl> close,1
```

Reading this file back in:

```
gdl> a=dindgen(10)
gdl> openr,1,'mydata.out'
gdl> readf,1,a
gdl> close,1
gdl> print,a
```

Notice the openr instead of openw here.

Finally let us create a *structure*, which is nothing else than a package of variables put together into a single entity. It is often the most useful way to transfer heterogeneous sets of data in one go.

```
gdl> a=1.5
gdl> b='Hello'
gdl> c=1
gdl> d=[4.,5.,7.]
gdl> s={a:a,b:b,c:c,d:d}
gdl> print, s.a
    1.5000000
gdl> print, s.b+'. This is a full sentence.'
 Hello. This is a full sentence.
```

B.2 Programming in GDL

The above commands can be grouped into a subroutine or a program. In such modules we can even use more kinds of constructions. Let us make a little program (to show that it is a file, it is marked here with <filename>, but of course this is NOT the first line of that file.)

```
<myprog.pro>
a=1.d0
for i=1,10 do begin
   a=sqrt(a+1.)
endfor
print,a
end
```

Now to run this program we type

```
gdl> .r myprog.pro
     1.6180286
```

We can also have if-statements:

```
<myprog.pro>
a=1.d0
for i=1,10 do begin
   a=sqrt(a+1.)
   if a lt 1.6 then print,'yes!'
endfor
print,a
end
```

or

```
<myprog.pro>
a=1.d0
```

```
for i=1,10 do begin
   a=sqrt(a+1.)
   if a lt 1.6 then begin
       print,'yes!'
       print,'  once more... Yes!'
   endif else begin
       print,'No...'
   endelse
endfor
print,a
end
```