IDL visualization macros

Gábor Tóth
University of Michigan

October 4, 2020

Contents

1 Introduction

This document describes the use of the IDL macros in the **share/IDL/General/** directory. These macros were originally developed for the Versatile Advection Code, and modified and improved for BATS-R-US. Since the macros are written in a rather general manner, they can be used to visualize and analyze all kinds of model or observational data as long as it can be read. For example the plot files from the PWOM, IPIC3D2, FLEKS and ALTOR of the SWMF can also be visualized and analyzed with these macros. In addition, simple ASCII files (satellite data, trajectory files, etc.) as well as various lookup tables (for radiative cooling, equation of state etc.) can be read and visualized. The plot data modified or created in IDL can be written into files in the same formats.

2 IDL path and startup file

It is necessary to let IDL know about the existence of the macros. You can define the search path for IDL, for example

```
setenv IDL_PATH "${HOME}/SWMF/share/IDL/General:<IDL_DEFAULT>"
```

for the csh or tcsh shell. You can also make IDL to read the **idlrc** file automatically upon start up with

```
setenv IDL_STARTUP idlrc
```

These environment settings can be put into your ~/.login or ~/.cshrc files. The above is valid for the csh and tcsh UNIX shells.

For other UNIX shells (bash, ksh, zsh), use

```
export IDL_PATH="${HOME}/BATSRUS/share/IDL/General:<IDL_DEFAULT>"
export IDL_STARTUP=idlrc
```

in the ~/.profile or similar file.

3 Running IDL

If the IDL_PATH and the IDL_STARTUP variables are set, simply start IDL from the directory where the ***.out** IDL plot files and the ***.log** logfiles are, e.g.

```
cd run/IO2
idl
```

If IDL_STARTUP is not set, type

```
@idlrc
```

at the IDL> prompt, so that the commands in the **idlrc** file are executed: the procedures in **procedures.pro, funcdef.pro** and **vector.pro** are compiled and the script **set_defaults** is executed to set some common block variables to their default values. You can customize the startup of IDL by editing **idlrc**, e.g. you can compile your own IDL procedures.

If an error occurs, the code usually returns to the main level, so one can fix the settings and try again. The corresponds to the default setting

```
onerror=2
```

For debugging purposes, it may be useful to set

```
onerror=0
```

so that variables can be written out inside the macro where the error occurred. In this case, or in general if you get trapped by an error inside some IDL routine, typing

```
retall
```

will return to the main level. To exit IDL type

```
exit
```

The alternative 'quit' command is also defined for convenience.

4 Reading a snapshot with read_data

To read a snapshot from a file, type at the "IDL>" prompt of IDL

```
read_data
```

The procedure will prompt you for the **filename**, and it determines the **filetypes** and **npictinfiles** (the number of snapshots in the file) automatically. Then it asks for the frame-number **npict** (1, 2,... npictinfiles) of the snapshot to be read from the file. When **npictinfiles=1**, the frame number is set to 1 automatically:

```
filename(s)   ? example1.out
filetype(s)   = binary
npictinfile(s)=      1
npict=       1
```

The header of the file is read and echoed on the screen. A typical result:

```
filename   = cut.outs
filetype   = ascii
headline   = km Mp/cc km/s km/s km/s nT nT nT nPa
it         =       77
time       =       60.360603
```

```
gencoord   =        0
ndim       =        1
neqpar     =        5
nw         =        8
nx         =      256
parameters =       1000. 3. 1.66667  0. 0.
coord names= x
var   names= Rho Ux Uy Uz Bx By Bz P
param names= xSI r g cuty cutz
Read x and w
GRID (PLOT_DATA)
               LONG      = Array[256]
```

At the end, the **x** and **w** variables (containing the coordinates and the plot variables respectively) are read from the file. Note that IDL, unlike FORTRAN, starts indexing from 0 instead of 1. The **GRID** index array is useful for defining cuts of the computational domain for plotting, see details in section 11.

After reading the data you can do whatever you want with **x**, **w**, and all the other variables **headline, it, time, gencoord, ndim, neqpar, nw, nx, eqpar, variables** defined by the header. You can use the **plot_data** procedure (see section 8 to get some sophisticated plots or you can use any of the IDL procedures directly to examine and/or plot the data, e.g. for 1D data

```
print,time,it
print,nx
print,variables
plot,x,w(*,0),xtitle='X [km]',ytitle='Density'
```

while for unstructured 2D data, for example, use

```
contour,w(*,*,2),x(*,*,0),x(*,*,1),/fill,nlevel=30,/irr
oplot,x(*,*,0),x(*,*,1),psym=1,color=0
```

The first command produces a color plot of the 3rd variable (index 2). The second command will show the grid points. Another way to plot the grid points is to use the **plotgrid** procedure as

```
plotgrid,x
```

For structured 2D grid, you can plot the grid lines connecting the grid points with

```
plotgrid,x,/lines
```

5 Transformation of non-regular grid

The file may contain data on a generalized or unstructured 2D grid. This is signaled by a negative **ndim** in the plotfile and by the variable **gencoord=1**

in IDL. A **generalized grid** has the same topology as a regular grid but the coordinates are not Cartesian. It is a *continuouus* distortion of the original grid. On the other hand, **an unstructured grid** has the grid points in an arbitrary order, therefore the second and third elements of the **nx** array are 1. The AMR grids of BATS-R-US and AMRVAC are unstructured, while VAC can produce generalized grids.

The default behavior is to leave the grid in its original form. This choice saves the time of transformation, and uses the original grid and variables, but only some of the plotting modes are available for generalized and unstructured grids: contour, contfill, contlabel, contbar, vector and stream. Others (e.g. surface, tv, tvbar, velovect) are not. To allow all plotting modes (and cuts across the grid), the data has to be transformed (interpolated) onto a regular grid. To achieve that select the 'regular' transformation with

```
transform (r=regular/p=polar/u=unpolar/n=none)=none ? r
```

The size of the regular grid can be given by setting the **nxreg** array

```
nxreg=[100,100]
```

If **nxreg** is not set, then the size of the regular grid will be asked when the data is read with **read_data** or **animate_data**:

```
Generalized coordinates, dimensions for regular grid
nxreg(0) (use negative sign to limit x)=      100
nxreg(1) (use negative sign to limit y)=      100
```

With these settings the original **w** array is interpolated to a 100×100 **wreg** array and the coordinates for the regular grid **xreg** are also determined. You can plot the first variable, usually density, in **wreg** the same way as before

```
surface,wreg(*,*,0)
```

It is possible to restrict the transformation to a rectangular part of the original 2D data by setting the **xreglimits** array

```
xreglimits=[-15, -10, 30.5, 10]
```

Note that the order of the elements is xmin, ymin, xmax, ymax. Another way to do this is adding a negative sign when prompted for nxreg(0) and nxreg(1), e.g.

```
nxreg(0) (use negative sign to limit x)? -100
xreglimits(0) (xmin)? -15.
xreglimits(2) (xmax)? 30.5
nxreg(1) (use negative sign to limit y)? -50
xreglimits(1) (ymin)? -10.
xreglimits(3) (ymax)? 10.
```

Now the 100×50 **xreg** array is limited to the range [-15.,30.5] in x, and [-10.,10.] in y, and this is where the **wreg** array is defined. To return to the default behaviour, which is plotting the whole computational domain, set

```
xreglimits=0
```

If the transformation or the transformation parameters are changed, the **read_data** or **animate_data** procedures, which read data from the disk, will calculate **wreg** with the new tranformation settings as expected. The **plot_data** procedure does not do the transformation by default to speed things up. If the transformation parameters are changed set

```
dotransform='y'
```

to force **plot_data** to redo the transformation as necessary. After that you can return to dotransform='n' to save the time of transformation.

6 Other grid transformations

The **transform** parameter can be used to perform various grid transformations by setting it to one of the following values

```
n, none    - no tranformation
r, regular - interpolate onto a regular Cartesian grid (previous section)
p, polar   - convert from Cartesian X-Y(-Z) to polar R-Phi(-Z) coordinates
u, unpolar - convert from polar R-Phi(-Z) to Cartesian X-Y(-Z) coordinates
s, sphere  - convert from Cartesian X-Y-Z to spherical R-Theta-Phi coordinates
m, my      - perform the transformation in the {\bf do_my_transform} procedure
```

The **polar, unpolar, sphere** options are standard geometric transformations of the coordinates **x** and the vector variables in **w** to **xreg** and **wreg**. The angles have to be in radians. The number of vector variables is read into **nvector** and the indexes of the first components of these vector variables are read into the **vectors** array.

The **transform='my'** option allows to define an arbitrary grid/data transformation that has to be implemented into a **do_my_transform.pro** procedure that needs to be compiled as

```
.r do_my_transform
```

The following simple example converts the second and third coordinates of 3D data from radians to degrees:

```
do_my_transform,ifile,variables,x,w,xreg,wreg,usereg
   if max(x(*,*,*,1:2)) lt 10.0 then $
      x(*,*,*,1:2) = x(*,*,*,1:2)*180/!pi
end
```

The if statement prevents accidental multiple transformations by **plot_data** or converting coordinates of a file that uses degrees to start with. As the number of arguments indicate, this feature can be used for much more complicated transformations too.

7 Comparison of data

You can read snapshots from up to 10 files for purposes of comparison. Simply give the filenames separated by spaces:

```
set_default_values
read_data
filename(s)   ? example1.out example2.out
filetype(s)   = real4 real4
npictinfile(s)=      21     10
npict?  2
```

This will read the second snapshots from 'example1.out' and 'example2.out'. You may also use wild card characters

```
  *  ?  [ ]
```

that are recognized by the Unix 'ls' command, e.g.

```
IDL> filename='example[12].out'
IDL> filename='example?.out'
IDL> filename='exampl*.out'
```

Note that if any wild card character is used then the order of the files will be alphabetical.

After reading the files with read_data, the coordinates and the conservative variables will be put into **x0, x1** and **w0, w1** respectively, however, the header information, which is printed for each file onto the screen, will be overwritten by the last file read, in this case it will belong to **example2.out**. The generic **x, w** arrays will also be filled by the data read from the last file, and this is what **plot_data** plots. If the files contained data on non-regular grid, and transform='regular' is set, the data will be interpolated into the arrays **wreg0** and **wreg1**. To compare the two data sets run

```
IDL> compare,w0,w1
iw max(|w1-w2|)/max(|w1|+|w2|) sum(|w1-w2|)/sum(|w1|+|w2|)
       0     0.018272938   0.00017745799
       1      0.24608387     0.017349624
       2      0.14307581     0.016188008
       3 wsum=0
       4     0.022624079   0.00022667312
       5     0.014965503   0.00022646554
       6     0.018034518   0.00020733169
       7 wsum=0
```

or add the wnames array (an optional argument) to get

```
IDL> compare,w0,w1,wnames
iw max(|w1-w2|)/max(|w1|+|w2|) sum(|w1-w2|)/sum(|w1|+|w2|)
```

```
rho      0.018272938   0.00017745799
mx       0.24608387     0.017349624
my       0.14307581     0.016188008
mz wsum=0
e     0.022624079   0.00022667312
bx     0.014965503   0.00022646554
by     0.018034518   0.00020733169
bz wsum=0
```

The comparison shows the maximum difference divided by the sum of maximum absolute values and the sum of absolute differences divided by the sum of absolute values for each variable in **w0** and **w1**. If a variable is zero everywhere both in **w0** and **w1**, the **wsum=0** message is shown. You can compare arbitrary 1, 2, 3 and 4 dimensional arrays as long as they have the same size. The last dimension is interpreted as the variable index **iw**. E.g. you could compare two cuts of `wreg` with

```
compare,wreg(0:20,2,*),wreg(0:20,3,*)}
```

or you can check if the data read from two files have the same grid

```
compare,x0,x1
```

If the two files have different resolutions, the **coarsen** function can be used. For example, if two solutions were obtained on 100 x 50 and 300 x 150 grids, then

```
rholow =w0(*,*,0)
rhohigh=coarsen(w1(*,*,0), 3)
print,total(abs(rholow-rhohigh))/100/50
```

will give the average deviation in density. The coarsening is done in finite volume sense, i.e. the fine cells within the coarsened cell are averaged out. The **coarsen** function works properly for uniform Cartesian grids only. Pointwise values can be compared with the use of the **triplet** and **quadruplet** functions (see section 11).

One can also visualize the difference between two data files with the **plot_data** procedure (see next section) by setting

```
w=w1-w0
```

if the two files use the same grid, or

```
wreg=wreg1-wreg0
```

if the files use different unstructured grids but they are interpolated onto the same regular grid.

8 Plotting data with plot_data

Once the data is read by **read_data** or **animate_data** you can plot functions of **w** with

```
plot_data
```

You will see some plotting parameters with their current values:

```
======= CURRENT PLOTTING PARAMETERS ================
ax,az=  30, 30, contourlevel= 30, velvector= 200, velspeed (0..5)= 5
multiplot= 0 (default), axistype (coord/cells)=coord, fixaspect=1
bottomline=3, headerline=0
```

The plots are normally shown in physical coordinates, i.e. **axistype='coord'**, but the axes can also run in cell indices if **axistype='cells'** is set (that works for structured grid only!). If **fixaspect=1** or -1 the aspect ratio of the plot will be the same as the true aspect ratio of the two axes, while **fixaspect=0** allows the aspect ratio to adjust so that the plot fits into the plotting window tightly. The **fixaspect=-1** setting preserves the aspect ratio but it allows adjusting the spacing between rows and columns of subplots independent of the shape of the plotting window (this can result in large margins). The variables **bottomline** and **headerline** control the number of values shown at the bottom from **time, it, nx** and at the top from **headline, nx**. You can change these values explicitly (e.g. **bottomline=0**), or change their default values in procedure **set_default_values** in **procedures.pro**. See sections 10 and 11 for more detail.

Now, you will be prompted for the name of function(s) and the corresponding plotting mode(s):

```
======= PLOTTING PARAMETERS ========================
wnames                      =  rho ux uy uz p bx by bz
func(s) (e.g. rho p m1;m2) ? rho uy
2D scalar: shade/surface/contour/contlabel/contfill/contbar/tv/tvbar
2D polar : polar/polarlabel/polarfill/polarbar
2D vector: stream/stream2/vector/velovect/ovelovect
plotmode(s)                 ? default
plottitle(s) (e.g. B [G];J)=default
autorange(s) (y/n)          =y
GRID            INT       = Array(50, 50)
```

The function(s) to be plotted are determined by the **func** string parameter, which is a list of function names separated by spaces. The number of functions **nfunc** is thus determined by the number of function names listed in **func**.

For each function you may set the plotting mode with the **plotmode** string. If you give fewer plotting mode(s) than **nfunc**, the rest of the functions will use the last plotting mode given, in the above example **default**, which is 'contbar' for scalars and 'streamover' for vector valued functions. This padding rule is

used for all the arrays described by strings. See section 10 for more details on plotting modes.

The **plottitle** parameter is usually set to **'default'** which means that the function name is used for the title, but you can set it explicitly, e.g. **plottitle='Density;Momentum'**. Here the separator character is a semicolon, thus the titles may contain spaces. No titles are produced if **plottitle=' '** is set.

For each function you may set the plotting range by hand or let IDL calculate the minimum and maximum by itself. This is defined by the **autorange** string parameter, which is a list of 'y' and 'n' characters, each referring to the respective function. If you set 'n' for any of the variables, the **fmin** and **fmax** arrays have to be set, e.g.

```
fmin=[1. ,-1.]
fmax=[1.1, 1.]
```

IDL remembers the previous setting and uses it, unless the number of functions are changed. You can always set fmin=0, fmax=0, and let IDL prompt you for the values.

9 Function names in string func

The function names listed in the **func** string can be any of the variable names listed in the string array **wnames**, which is read from the header of the file, or any of the function name strings shown in the **functiondef** array at the beginning of **funcdef.pro** (see section 16), or any expression using the standard variable, coordinate and scalar parameter names and various constants:

```
x y z r
rho p ux uy uz uu u bx by bz bb b
xSI gamma gammae Mi Me Qi Qe clight rbody
mu0 mu0A c0 op0 oc0 rg0 di0 ld0
```

Here "uu" and "bb" are the velocity and magnetic field squared, while "u" and "b" are the velocity and magnetic field magnitudes, respectively. Note that "x ... b" are arrays, while "gamma" is a scalar. For example the maximum Alfvén speed could be given as **func='b/sqrt(rho*mu0A)'**, but this is already defined in funcdef.pro as 'calfven'. However, for a multifluid data file, one can use a different density, e.g.

```
func='b/sqrt({OpRho}*mu0A)'
```

where mu0A is the vacuum permeability together with the unit conversion factors. Note that the OpRho variable (density of O+ ion) is not among the standard arrays, but it can still be used by enclosing it with the curly brackets.

You may combine two function names with the **;** character representing two components of a vector, e.g. **ux;uy** or **bx;bz**, which can either be plotted as a vectorfield by the **velovect**, **vector** and **arrow** plot modes, or as streamlines

or fieldlines, using the **stream** plotting modes. For other plotting modes the absolute value $\sqrt{ux^2 + uy^2}$ is plotted. You can also put a minus sign in front of any function or variable name, which will simply multiply the value of the rest of the string by -1. For example '-Ti' plots (-1)*temperature of ions. This is just a shorthand for the general syntax '-Ti'.

10 Plotting modes in string plotmode

There are many plotting modes available. These can be listed in the **plotmode** string for each function separated by spaces. If the number of plotting modes is less than the number of functions, the last plotting mode is applied for the rest of the functions.

For 1D plots the following plotting modes are available:

Plotmode	Horizontal axis	Vertical axis
plot	linear	linear
plot_io	linear	logarithmic
plot_oi	logarithmic	linear
plot_oo	logarithmic	logarithmic

The default value for plotmode is 'plot', which uses linear axes. The names of these plotting modes are identical with the corresponding IDL procedures.

For 2D data there are many more possibilities. For scalar functions the main plotting modes are

Plotmode	Parameters	Description
contour	contourlevel	contourlines
contlabel	contourlevel	contourlines labeled by value
contfill	contourlevel	levels colored by value
contbar	contourlevel	levels colored by value plus colorbar
polar		polar contous plot with guessed angle unit
polarrad		polar contour plot with angles in radians
polardeg		polar contour plot with angles in degrees
polarhour		polar contour plot with angles in hours
polarlabel	contourlevel	polar contour plot with labels
polarfill	contourlevel	polar plot colored by value
polarbar	contourlevel	polar plot colored by value plus colorbar
tv		grid cells colored by value
tvbar		grid cells colored by value plus colorbar
surface	ax az	surface mesh, height proportional to value
shade	ax az	shaded surface, height proportional to value

The parameters **ax** and **az** define the viewing angle, while the **contourlevel** parameter determines the number of contourlevels. The 'tv','tvbar','surface' and

'shade' plotting modes can be used for Cartesian grids only (or grids transformed to Cartesian).

For functions with 2 components (e.g. 'bx;bz') the following plotting modes are available

```
Plotmode  Parameters  Description
-----------------------------------------------------------------
stream    velvector   stream/fieldlines at random/selected points
          velpos
          velrandom

arrow     velvector   arrows of fixed length at random/selected positions
          velpos
          velrandom

vector    velvector   vectors at random/selected positions
          velpos
          velspeed
          velrandom

velovect              vectors at every grid point

ovelovect             vectors at every grid point (for overplot)
```

The **velvector** parameter determines the number of arrows or stream/fieldlines shown. By default the position of arrows/streamlines is random. The positions can be fixed with the **velpos** array (see section 11 for details). During an animation the arrows can move from their initial position parallel to the local velocity at a speed proportional to the magnitude of the velocity and the **velspeed** parameter. The maximum value is the default `velspeed=5`, while `velspeed=0` does not allow the arrows to move. When the arrows move, it may be necessary to reinitialize them with a random position periodically, otherwise the arrows may converge to a small part of the domain. Setting `velrandom` to a small positive integer value (e.g. 5) will reinitialize the position of every 5th vector every 5 picture of the animation.

The 'velovect' and 'ovelovect' plotting modes can be used for Cartesian grids only.

The following options can be added to any of the above plotting modes:

```
Option  Description
-------------------------------------------------------------------
log     show the 10 based logarithm of the function if it is positive
over    overplot the previous function
noaxis  do not show the axes
#ctNNN  use color table NNN (NNN is an integer from 0 to 999)
#cNNN   use color NNN       (NNN is an integer from 0 to 255)
max     plot maximum of the last "nplotstore" snapshots
```

```
mean    plot average of the last "nplotstore" snapshots
```

For 2D plots there are a few more options

```
Option  Description
------------------------------------------------------------------------
irr     force triangulation for irregular (non-Cartesian) grid
grid    show grid points with plus signs
mesh    show the mesh (lines connecting grid points)
           only for structured grid
white   draw vectors, stream lines and the grid/mesh with white color even
           for postscript output
body    show the spherical body with radius rBody
           at the origin as a black circle
map     draw a world map under the plot
usa     draw the USA map under the plot
```

Note that it makes no sense to overplot the grid for the `surface` plotting mode, on the other hand `plotmode='shademesh'` will plot the shaded surface together with the mesh of 'surface'. The radius of the body **rBody** is usueally read from the scalar parameter named 'r' or 'rbody' in the data file, or it can be set manually.

Here is an example for some more complex plotmode strings:

```
plotmode='contbargridlog streamwhiteoverbody'
```

will show the 10 based logarithm of the first scalar quantity with a color bar and the grid points, and overplot the second vector quantity with white streamlines and a black body at the origin.

For any of the colored plotting modes ('shade', 'contfill', 'contbar', 'polarfill', 'polarbar', 'tv', and 'tvbar') the colortable can be changed by one of the

```
xloadct
loadct,3
makect,'red'
```

commands. The `xloadct` and `loadct` commands are part of IDL, while the `makect` procedure is defined in **procedures.pro**. When no argument is given for `loadct` or `makect`, all the available color tables are listed and the choice can be made interactively. If multiple color tables are needed, they can be loaded with the #ctNNN option. The color can be set with the #cNNN option.

Many characteristics of the plots can be adjusted with system variables. Here is a partial list of these:

```
system variable     description
-------------------------------------------------------------------------
!p.charsize         overall character size
!p.psym             symbols instead of lines in 1D plot
!p.symsize          symbol size
```

```
!p.thick            thickness of lines in plots
!p.linestyle        line style in plots (0=solid, 1=dotted, 2=dashed...)
!x.title            title of the X axis
!x.charsize         X axis character size
!x.thick            thickness of the lines forming the X axis
!x.ticks            number of X axis tick marks
!x.tickv            positions of X axis tick marks
!x.tickname         strings at X axis tick marks
!x.minor            number of minor tickmarks
```

The Y and Z axes are affected by the analogous !y. and !z. variables.

11 Plotting part of the domain

It is possible to plot a part of the simulation domain. One way that works for both structured and unstructured grids is to set the global system variables

```
!x.range=[-10.,10.]
!y.range=[-20.,-5.]
```

This will work well for 'flat' plotting modes, like 'contour', 'contfill', etc. For unstructured grids which are not transformed to a regular grid, it works for all the available plotting modes. To switch back to the default maximum range, use

```
!x.range=0
!y.range=0
```

When the data is transfored to a regular grids, the domain of transformation can be limited by the **xreglimits** array as described in section 5.

For structured grids, there is a further option of limiting or coarsening the plot domain, and/or reducing the dimensionality of the plot. The **cut** index array selects some part of the function(s) determined by **func**. This is done *after* any grid transformation and *after* the functions are calculated so that derivatives can be properly taken by the **funcdef** function. The **grid** index array is defined to help to construct the **cut** array, e.g. if the grid size is 100 times 100:

```
cut=grid(*,50:*)
plot_data
```

will show the upper half of the domain. To eliminate the edges use

```
cut=grid(1:98,1:98)
```

A cross section of the domain can be plotted by reducing the number of dimensions of **cut** relative to **grid**:

```
cut=grid(*,50)
```

will produce 1D plots of the cross section along the midline parallel to the first coordinate axis. For a 2D cut of 3D data, use for example

```
cut=grid(*,50,*)
```

The effect of the **cut** array can be switched off by

```
cut=0
```

or by running **set_default_values**.

The **triplet** function provides an easy way to set the **cut** index array to represent a coarser grid. This is particularly useful for the **velovect** plotting mode, which tends to draw too many tiny arrows. The **triplet** function can have 3, 6, 9, or 12 parameters depending on the number of dimensions, and each triplet describes a subset of the indices in the given direction. The three elements are the minimum, maximum, and stride (like in Fortran 90), e.g.

```
filename='example2.out'
npict=10
read_data
func='ux;uy'
plotmode='velovect'
cut=triplet(0,49,2, 33,66,1)
plot_data
```

will show every second cell in the **x** direction and the middle third in the **y** direction. Note that the maximum index value should be the actual grid size-1 except for the last dimension, otherwise the indices will not be correct. This problem can be solved by the use of the **quadruplet** function, which has four parameters per dimension: size, minimum, maximum, and stride. To show a coarse $20*20$ grid from the top left $40*40$ part of the $50*50$ grid use

```
cut=quadruplet(50,0,39,2, 50,10,49,2)
plot_data
```

Another way to show velocity vectors at fixed positions is the use of the "vector" plotting mode after setting the number of vectors **velvector** and the array of positions **velpos(velvector,2)** containing the X and Y coordinates for each vector. In principle **velpos** can be defined as an arbitrary set of points. For a simple coarsening of the original grid points, the triplet and quadruplet functions can be used again:

```
cut=grid(0:39,10:49)
velvector=20*13 & velpos=fltarr(velvector,2)
velpos(*,*)=x(quadruplet(50,1,39,2, 50,11,49,3, 2,0,1,1))
plot_data

cut=0
plotmode='vector'
```

```
velvector=25*25 & velpos=fltarr(velvector,2)
velpos(*,*)=x(triplet(0,49,2, 0,49,2, 0,1,1))
plot_data
```

Note that the last 0,1,1 triplet and 2,0,1,1 quadruplet correspond to the second dimension of **velpos** that always runs from 0 (X coordinate) to 1 (Y coordinate). This approach is very useful when the velocity vectors are shown together with plots of other functions that should not be coarsened. To use random positions again, set **velpos=0**. The **velpos** array can also be used to position streamlines for plotmode='stream' and 'stream2'.

Finally the **rcut** parameter can be used to cut out a circle around the origin. While the **body** modifier in the plotting mode simply covers the circle, the **rcut** removes all the data inside that radius, so it has no effect on the range of values.

12 Multiplot

The number and arrangement of subplots is automatically set based on the number of files and and the number of functions. The default arrangement be can overriden by setting the **multiplot** array. The most complicated use has 4 elements

```
multiplot=[2,3,0,2]
```

gives 2 by 3 subplots filled in row-wise (3rd element is 0) starting with the 3rd subplot (4th element is 2). If the third element is 1, the subplots are filled in column-wise. If the 4th element is not given

```
multiplot=[2,3,0]
```

the plotting starts at the top left subplot, as usual. Even simpler cases are handled by setting multiplot to a scalar value. Setting **multiplot=3** is identical with **multiplot=[3,1,0]** (a single row of plots), **multiplot=-3** is identical with **multiplot=[1,3,1]** (a single column of plots), **multiplot=-1** is a column of subplots based on the number functions and files, while **multiplot=0** gives the default behaviour (the number of rows and columns is about the same and filled in row-wise).

The spacing between subplots can be adjusted with the **plot_spacex** and **plot_spacey** variables. The spacing is measured in character size (which depends on !p.charsize). The default values are 3 for both, which is usually sufficient to show the axis labels. Setting both to zero (and also fixaspect=0 or -1) can result in tigthly packed plots.

To save space, the axis labels and axis titles are normally only shown for the subplots are at the leftmost column or the bottom row. Setting (some of) the **showxtitle, showytitle, showxaxis, showyaxis** parameters to 1 can be used to force the X and/or Y axes and their title plotted for all subplots.

Functions can be plot on top of each other by setting the **multiplot** array such that the number of subplots is smaller than the number of functions times the number of files.

To overplot functions, the most convenient approach is to use the 'over' option in the plotmode string, for example

```
func='rho bx;by'
plotmode='contbar streamover'
plottitle='rho;B'
plot_data
func='p ux;uy'
plottitle='p;U'
plotmode='contfill arrowover'
plot_data
```

The number of functions and the number of subplots can be any combination. In 1D plots, the line style is varied for the different functions, so the curves can be distinguished.

13 Plotting another snapshot

If you type

```
read_data
plot_data
```

again, the data will be read and plotted again without any questions asked, since IDL remembers the previous settings.

If you want to read another frame, say the second, from the same file, type

```
npict=2
read_data
```

You can change the **func** and **plotmode** variables the same way:

```
func='rho p'
plotmode='contour surface'
plot_data
```

Note that we did not need to reread the data. Other variables, all listed in the common blocks at the beginning of the **procedues.pro** file, can be set similarly. If you set

```
doask=1
```

the macros will ask for all the parameters to be confirmed by a simple RETURN, or to be changed by typing a new value. Set **doask=0** to get the default behaviour, which is no confirmation asked. To overplot previous plots without erasing the screen, set

```
noerase=1
```

You can return to the default settings for all parameters by calling the

```
set_default_values
```

procedure.

14 Animation and plotting with animate_data

This general procedure can plot, save into image and video file(s), or animate (using IDL's Xinteranimate) different functions of data read from one or more files. If a single snapshot is read, the plot is drawn without animation. In essence, **animate_data** combines **read_data** and **plot_data** for any number of files and any number of snapshots.

```
animate_data
```

will first prompt you for **filename(s)** unless already given. Animating more than one input files in parallel is most useful for comparing simulations with the same or very similar physics using different methods or grid resolution. It is a good idea to save snapshots at the same *physical* time into the data files. By default (if multiplot=0), the functions corresponding to the files will be plot columnwise with the leftmost column belonging to the first file. The headlines and the grid sizes will be shown in for each file separately above the corresponding columns if **headerline=2** is set.

The function(s) to be animated and the plotting mode(s) for the functions are determined by the same **func, plotmode**, and **plottitle** strings as for **plot_data**. To plot different functions, plot modes, and/or plot titles per file, set the **func_file, plotmode_file** and/or **plottitle_file** string arrays with as many elements as the number of files. If any part of the **autorange** string is set to **'y'**, the data file(s) will be read twice: first for setting the common range(s) for all the snapshots and the second time for plotting. If **autorange='n'** the file(s) will only be read once. Here is an example showing two cuts of a 3D simulation together:

```
filename='y=0.outs z=0.outs'
plottitle_file=['Density and velocity in y=0 plane', 'Density and U in z=0 plane']
func_file=['rho ux;uz', 'rho ux;uy']
plotmode_file=['contfill streamover', 'contbar streamover']
showxtitle=1
showytitle=1
animate_data
```

The **showxtitle** and **showytitle** logicals control if the axis labels are shown for each subplot or not. The default is to show the X axis for the bottom row only and the Y axis for the left column only. When the axes vary from file to file, it is best to show the axis labels for each subplot.

The number of snapshots to be animated is limited by the end of file(s) and/or by the **npictmax** parameter. With a formula

```
npict=min( npictmax, min( 1 + (npictinfiles-firstpict)/dpict ) )
```

The animation runs from **firstpict**, every **dpict**-th picture is plotted and the total number of animated frames is at most **npictmax**. If **firstpict** and **dpict** are scalars, the same values are applied for all the files, but it is also possible to use different values for each file by setting array values, e.g. for two files

```
firstpict=[5,9]
dpict    =[1,2]
```

will plot every frame starting from the 5th in the first file, and every second frame from the 9th in the second file.

The **multiplot** array can be used to get some really interesting effects in **animate_data**: the data from multiple files can be overplotted for comparison purposes. Probably it is a good idea to compare 1D slices rather than full 2D plots, e.g.

```
filename='example[12].out'
func='rho ux'
cut=grid(*,25)
multiplot=2
animate_data
```

will overplot density and velocity read from the two files. The lines belonging to the two data files are distinguished by the different line styles. For a 2D comparison, one could use

```
filename='example[12].out'
func='rho ux'
plotmode_file=['contfill', 'contour']
multiplot=2
animate_data
```

Timeseries can also be produced easily with **multiplot**.

```
filename='example2.out'
func='rho ux;uy'
plotmode='contfill vectorover'
npictmax=6
multiplot=[3,2,0]
bottomline=1
animate_data
```

will show the first 6 snapshots of density with overplot velocity vectors in a single plot. Now the time is shown for each plot individually, and setting **bottomline=1** limits the time stamp to the most essential information, time. The **bottomline=2** shows the iteration number and the time, while **bottomline=3** also shows the grid size. This information can be customized by setting **bottomline** to a string valued expression, e.g.

```
bottomline='"Time="+stime+", Grid="+snx+", Iteration="+sit'
```

where **stime, snx, sit** are formatted strings of time, iteration number and grid size. An even more complicated example is to use expressions and string formatting explicitly, e.g.

```
bottomline='"Time="+string(time*1e9,format="(f5.1)")+" ns"'
```

that will show time in units of nanoseconds.

An alternative approach (that works with animate_data only) is to set the **timetitle** string to format the time and show it as the plot title. For example

```
timetitle='("t=",f8.1,"s")'
```

will show the time as **t=240000.0s**. The time units and the initial time (offset) can be set with **timetitleunit** and **timetitlestart**. For example

```
timetitleunit=3600.0
timetitlestart=60.0
timetitle='(``t='',f5.2,''h'')'
```

will show the time as **t= 6.67h**. Note that timetitleunit is relative to the time units used in the data file (e.g. seconds) while the offset is given in the time units defined by timetitleunit.

Set timetitle to an empty string and/or timetitleunit and timetitlestart to zero to return to the default behavior. If **npict*nfile*nplot** is greater than the number of subplots defined by **multiplot**, an animation is done. Type

```
multiplot=0
```

to return to default behavior, which is one snapshot per plot.

Even after exiting from Xinteranimate, the animation can be repeated again without rereading the data file(s) by typing

```
xinteranimate,/keep_pixmaps
```

Sometimes it is interesting to visualize the difference of two runs, e.g. to visualize deviations from the initial state, or from a steady state. This can be achieved by setting the **wsubtract** array, which will be subtracted from **w** for each snapshot. Note that the subtraction is done for the original variables in **w**, so derived quantities should not be plotted. Set **wsubtract=0** to switch off the subtraction.

The **timediff=1** setting can be used to calculate the time derivative by subtracting the previous plot data stored as **wprev** from the current one and dividing by the elapsed time **time-timeprev**. Again, derived quantities may not be meaningful. Set **timediff=0** to switch off this feature.

Note that the **wsubtract** and **timediff** features are only used by "animate_data", since a single snapshot can be easily manipulated explicitly, e.g. **w=w1-w0**, before plotting with **plot_data**.

Even more access to the animated data is provided by the variables found in the **plot_store** common block. Setting **nplotstore** to the number of snapshots to be stored, for example

```
nplotstore = npictmax
animate_data
help, plotstore, timestore
```

will produce the four dimensional **plotstore(n,nplotstore,nfunc,nfile)** array. The first index is for the discrete points in the plotted functions in the same order as in the **w** or **wreg** array. The second index is for the snapshot (time), the third is for the plot function (for multiple functions), and the last is the file index (for multiple files). The times of the snapshots are saved into the **timestore(nplotstore,nfilestore)** array.

Note that if the plot mode contain 'max' or 'mean' option, the maximum or mean will be calculated for the last nplotstore snapshots, but the plotstore array will still contain the original function values.

15 Slicing structured 3D data

For visualizing 3D data, plot_data or animate_data can be used after a 1 or 2D **cut** array has been defined. Alternatively slices of a single snapshot (read by read_data) can be animated by

```
slice_data
```

The 3D data is cut along **slicedir**, e.g. for cuts parallel to the X-Y plane, set

```
slicedir (1, 2, or 3)? 3
```

If the grid size is e.g. 50∗100∗60, then there are 60 slices to plot. The number of animated slices can be reduced: at most **nslicemax** slices are shown starting from **firstslice**, and only every **dslice**-th slice is shown. The plots can be further reduced by setting the **cut** array, however, now indices in cut refer to a single slice. The **grid2d** index array (generated by the first slice_data, in this case it is a 50∗100 array) can be used, e.g.

```
cut=grid2d(*,30:70)
```

For **plotmode='vector'** the vectors are not advected with the flow (i.e. velspeed=0) since it does not make sense for the slices.

The x and w arrays are overwritten with the 2D cuts during the slicing, and only restored to the 3D arrays at the end. If the slicing failed for any reason, use

```
slice_data_restore
```

to restore the arrays.

16 Function definitions in funcdef

The plot functions are set by the **funcdef.pro** procedure. This may be customized by the user. If it is modified, it should be recompiled before being used:

```
.r funcdef
```

Any function of the variables in **w**, the coordinates in **x**, the scalar parameters in **eqpar** can be defined in the **funcdef.pro** file. The names of these variables are defined by the **variables** string array. Further information is provided by various unit conversion constants set by the **set_units** procedure that is usually called internally when the data file is read, but it can also be called directly, for example:

```
set_units,'PIC',distunit=0.01,Mion=16,Melectron=0.16
fixunits=1
```

where the first argument sets the unit system ('SI', 'CGS', 'NORMALIZED', 'PIC', 'PLANETARY', or 'SOLAR') which is normally guessed from the **headerline** variable of the data file; the distance unit is given in meters (usually set by the xSI scalar parameter in the data file); finally Mion and Melectron set the ion and electron masses in AMU that may be defined by the Mi and Me scalar parameters. All these arguments are optional. The **fixunits=1** ensures that the settings are not changed when (re)reading a data file.

Once the units are properly set, many functions can be derived from the basic variables. Here is a list of the currently defined functions (for vectors, only the X components are listed for sake of brevity):

```
Function name   Meaning
-------------------------------------------------------------
Ax              vector potential X component
mx              momentum X component
mxB             Boris momentum X component
Ex              electric field X component -(u x B)_x
j               current density (from jx, jy, jz)
jpx             particle current density X component
jppar           particle current density parallel with B
jpperp          particle current density perpendicular to B
jpxbx           Lorentz force X component from particle current
divbxy          div(B) in 2D
divb1xy         div(B1) in 2D
uHx             Hall velocity X component (jx/ne)
uH              Hall velocity
uex             electron velocity X component (ux-uHx)
ue              electron velocity
e               energy density p/(gamma-1)+0.5*(rho*uu + bb)
pbeta           plasma beta: p/(B^2/(2*mu0))
s               entropy: p/rho^gamma
ni              ion number density
ne              electron number density
qtot            total charge density
Ti              ion temperature: p/(n*k)
Te              electonr temperature: pe/(n*k)
uth             ion thermal speed
```

```
uthe            electron thermal speed
csound          sound speed: sqrt(gamma*p_thermal/rho)
cslowx          slow magnetosonic speed along X dimension
calfvenx        Alfven speed along X dimension: bx/sqrt(rho)
calfven         maximum of Alfven speed: |B|/sqrt(rho)
cfastx          fast magnetosonic speed along X dimension
cfast           maximum of fast speed: sqrt(csound^2+calfven^2)
machx           Mach number:  ux/csound
mach            Mach number: |u|/csound
Mslowx          slow Mach number along X dimension: ux/cslowx
Malfvenx        Alfven Mach number along X dim: ux/calfvenx
Malfven         maximum Alfven Mach number: |u|/calfven
Mfastx          fast Mach number along X dimension: ux/cfastx
Mfast           maximum fast Mach number: |u|/cfast
omegapi         ion plasma frequency
omegape         electron plasma frequency
omegaci         ion gyro frequency
omegace         electron gyro frequency
rgyro           gyro radius
rgSI            gyro radius in SI
rgyroe          electron gyro radius
rgeSI           electron gyro radius in SI
dinertial       inertial length
diSI            ion inertial length in SI
skindepth       electron skin depth
deSI            electron skin depth in SI
ldebye          Debye length
ldSI            Debye length in SI
```

17 Reading logfiles with read_log_data

One or more (at most ten) logfiles can be read by

```
read_log_data
```

which reads data from the file(s) determined by the **logfilename** parameter. This can be a space separated list of file names and/or it may include wild card characters. The data in the logfile(s) is put into the **wlog** (**wlog1, wlog2 ...**) real arrays, while the names of the variables are put into the **wlognames** (**wlognames1, wlognames2, ...**) string arrays. If the logfile contains the variable names 't' or 'time', 'hour' or 'hours', 'yr mo dy hr mn sc ms' or 'year month day hour min sec msec', then the time is calculated and stored in the **logtime** (**logtime1, logtime2, ...**) 1D real arrays. The time unit is defined by the **timeunit** string (possible values are **'h', 'm', 's', 'millisec', 'microsec', 'ns'**); the default units are hours. If no time variables are found, the **logtime** array is set to the row index.

After running **read_log_data**, the **wlog(nrow,ncol)** real array contains the rows and columns of the logfile, the **wlognames(ncol)** string array the names and the **logtime(nrow)** array the times. A simple example is

```
read_log_data
logfilename(s) =log_n020001.log
logfile  =log_n020001.log
headline =Volume averages, fluxes, etc
  wlog(*, 0)= it
  wlog(*, 1)= t
  wlog(*, 2)= dt
  wlog(*, 3)= rho
  wlog(*, 4)= mx
  wlog(*, 5)= my
  wlog(*, 6)= mz
  wlog(*, 7)= p
  wlog(*, 8)= bx
  wlog(*, 9)= by
  wlog(*,10)= bz
  wlog(*,11)= pmin
  wlog(*,12)= pmax
Number of recorded timesteps: nt=      1000
Setting logtime
```

You can use the IDL plotting procedures directly to visualize the data, e.g.

```
plot,logtime,wlog(*,3),xtitle='hour',ytitle='rho_mean'
```

checks the global mass conservation.

The **plot_log_data** procedure described next can be used to get much more sophisticated plots.

18 Plotting logfile data with plot_log_data

Once the data in the logfile(s) have been read with **read_log_data**, it can be easily visualized with the **plot_log_data** procedure. It will prompt for the names of the log functions that is a space separated list of a subset of the strings in the wlognames array:

```
plot_log_data
logfunc(s)     ? mx pmin pmax
```

To change the list of functions simply change the **logfunc** string. The spacing around the subplots (which are alway arranged vertically) can be set with the **log_spacex** and **log_spacey** constants given in character size (default values are 5 for both).

The time range of the plot can be set with the 2 element **xrange** array, while the vertical plot range for the individual functions can be set by the 2 by nfunc element **yranges** array. For example

```
xrange=[10,20]
yranges=[[-10,10],[0,0.1],[0,100]]
```

The default plot title, the X title (time) and the Y titles (log functions) can be modified by setting the **title** and **xtitle** strings and the **ytitles** string array, respectively. For example

```
title='Simulation Results'
xtitle='Hours from October 29, 2003'
ytitles=['m!DX!N','P!Dmin!N','P!Dmax!N']
```

To have no title at all, set these variables to empty strings. For the default titles, set the variables to 0.

For multiple logfiles the plot_log_data procedure will overplot the data. By default the lines belonging to the different data files are distinguished by color, but it is possible to use different line styles or symbols by setting the **linestyles** or **symbols** arrays. For example

```
colors=[255,255]
linestyles=[1,2]
symbols=[-4,-5]
```

will show a dotted and a dashed line with the same default color (usually white or black) and with diamonds and trianlges at the data points, respectively.

The time coordinates can be shifted (towards the negative direction) by setting the **timeshifts** array which should have **nlogfile** elements if set. Set timeshifts=0 to get the default behavior.

It is possible to add legends to the plot by defining the **legendpos** array that contains the **xmin, xmax, ymin, ymax** coordinates in normalized (0 to 1) coordinates. The legends consist of horizontal lines and/or symbols extending from xmin to xmax with the same colors, line types, and symbols as used for the data. Each horizontal line/symbol is followed by a string that is either the name of the corresponding logfile, or the corresponding element of the **legends** string array consisting of **nlog** strings. For example

```
legendpos=[0.1,0.12,0.5,0.7]
```

will draw horizontal lines from 0.1 to 0.12 with vertical coordinates starting from 0.7 all the way down to 0.5. For symbols it is better to use a single point, e.g.:

```
legendpos=[0.11,0.11,0.5,0.7]
```

The horizontal lines or symbols will be foolowed by the file names by default. Different legend strings can be given as

```
legends=['run1 with no AMR', $
         'run2 with 1 level of AMR', $
         'run2 with 2 levels of AMR']
```

The legends can be switched off with

```
legendpos=0
```

The data can be smoothed with the **smooths** array. The smoothing width can be defined for each logfile separately. A value less than 2 means that the data is not smoothed for that file. For example

```
smooths=[100,0]
```

will smooth the data from the first file only with a 100 point wide stencil.

Fourier transformation can also be performed by setting

```
dofft=1
```

In this case the horizontal axis will be the frequency, and the vertical axis will be the power spectrum of the log functions defined in **logfunc**. The **xrange** array can be used to select the relevant frequency range.

19 Reading and plotting logfile data with read_data and plot_data

While the **read_log_data** and **plot_log_data** procedures provide easy and flexible ways to plot data in logfiles, they do not allow to combine various columns into new functions. One can work around this by reading the data with **read_log_data** first, then manipulate the **wlog** array(s) directly, and then plot the modified data with **plot_log_data**.

An alternative approach is to use the **read_data** procedure to read the logfile data into the **x** and **w** arrays:

```
read_data
filename(s)   =*.log
filetype(s)   = log
npictinfile(s)=       1
npict=       1
headline  =Volume averages, fluxes, etc
ndim      = 1, neqpar= 0, nw=13
it        =       0, time=      0.00000
nx        =     1182
eqpar     =      0.00000
variables = hour it t dt rho mx my mz bx by bz e Pmin Pmax
Read x and w
GRID            LONG      = Array[1182]
```

The file type **log** is recognized from the filename extension which has to be **'.log'** or **'.sat'**. After reading the logfile, the **x** array contains the **logtime** in hours (currently the timeunit is not adjustable in this approach), while the **w** array will contain the various log functions. Now the logfile data can be plot as standard 1D data with **plot_data**.

20 Saving plots into postscript and graphics files

In IDL printing a plot is possible through Postscript files using the **set_device** and **close_device** procedures.

```
set_device,'myfile.eps'
loadct,3
plot_data
close_device
```

The first optional argument of the **set_device** procedure is the filename. If it is not given, the default filename 'idl.ps' is used. There are several keyword arguments too. Layout is set by `/port` for portrait, `/land` for landscape (this is the default, but in some cases it is needed) and `/square` for a square shaped figure. The `xratio` and `yratio` options can be used to shrink figure relative to the page size (default values are 1). The `/eps` option selects encapsulated postscript format (which is also default if the file extension is .eps). The `psfont=12` argument can be used to select a specific font.

With no arguments, the **close_device** procedure simply closes the postscript device, and opens the 'X' device. If the optional `/pdf` argument is present, the output PostScript or EPS file is converted to PDF using either the default `ps2pdf` program or the programname given as a value, e.g. `pdf='convert'`. If the `/delete` argument is present as well, the original PS or EPS file is removed.

You can use animate_data instead of plot_data (e.g. for multiple files or for time series) in combination with `set_device` and `close_device`, but make sure that only one plot is produced by setting **npictmax=1**, and use **firstpict** to select the snapshot. Here is an example showing many of the optional arguments:

```
npictmax=1
firstpict=12
set_device,'figure2.eps', /port, xratio=0.8, psfont=8
animate_data
close_device,pdf='convert',/delete
```

This will produce a 'figure2.pdf' in portrait format with Helvetica font, using the `convert` program to convert to PDF.

To save all frames of an animation into a series of Postscript files, do not use **set_device** but set

```
savemovie='ps'
```

This will produce files **Movie/0001.ps,Movie/0002.ps,...** in the Movie directory, which is created automatically if it does not exist. The PostScript files are best suited for printing. You can also save the frames in PNG, TIFF, JPEG or BMP formats, e.g. by setting

```
savemovie='png'
```

The frames can be put together into a movie by some program like **mpeg_encode** or ImageMagick's **convert**, or Apple's **QuicktimePro7**, but there is a simpler approach. To save the animation into a video file directly, use the 'mov', 'avi' or 'mp4' format. The name of the video file can be set with 'videofile' (default name is 'movie') and the number of frames per second with 'videorate' (default is 10), for example

```
savemovie='mp4'
videofile='waterwaves'
videorate=24
animate_data
```

will create 'waterwaves.mp4' file with 20 frames per second.

21 IDL scripts and procedures

All the IDL commands can be collected into a script file, for example

```
IDL_mine/myfig.pro
```

which can be run from IDL by

```
@myfig
```

This is a convenient way to store the commands for producing complicated figures. An example can be found in **EXAMPLE.pro**. There are some restrictions on scripts, however. Loops and other multi-line structures cannot be used in a script. If loops are needed, simply add an **end** statement to the end, and call it as

```
.r myfig
```

For even more complicated cases a true procedure can be written. The common variables can be imported through common blocks. This can be called as any other procedure, for example

```
my_fig, inputfile='test.out', outputfile='test.eps'
```