KIS
Kiepenheuer-Institut
für Sonnenphysik
Stiftung des öffentlichen Rechts des Landes Baden-Württemberg
Mitglied der Leibniz-Gemeinschaft

Manual for the
astrophysical lab course

Introduction to IDL

1 Introduction

IDL (Interactive Data Language) is a program language widely used in astrophysics, mainly for data analysis.

Since IDL is a commercial software you will not be able to use it on your private computer. There is a freeware version called GDL (Gnu Data Language) with the same syntax. GDL is always some years behind the commercial original and it also lacks some features like saving data or exporting figures. For this tutorial GDL is sufficient. We recommend to download and install the software and to do the exercises in this manual before the astrophysical lab course. Especially if you do not have any knowledge in programming this will make the lab course easier for you. If you do already know a language, maybe C, C^{++} or Python, it should be an easy task anyway.

2 GDL compatible part

2.1 Using IDl as a calculator

You can perform all sorts of mathematical operations in the IDL prompt. Type in `print, 2 + 2` and hit `ENTER`.

2.1.1 Data types and variables

You can assign values to variables. Variables always start with letter. IDL is not case sensitive. For example:

```
a = 3
print, a
b = 4
c = a * b
```

operator	operation
+	addition
−	subtraction
*	multiplication
/	division
^	potentiation
mod	modular
##	matrix multiplication

Tabelle 1: Mathematical operators in IDL.

!h

data type	purpose	example
integer	whole number	1
long integer	whole number	1L
floating point value	real number	1.0
double	real number	1D
complex	complex number	complex(1.0, 2.4)
string	words	'hello'

Tabelle 2: This table shows some of the datatypes supported by IDL

```
print, c
```

Variables, like `a`, `c` and `c` in the example above always have a data type. IDL supports the common data types most program languages do. Some are listed in table 2.

Be cautious if you operate with integers. Try something like `print, 3 / 2` and compare the result to `print, 3.0 / 2.0`. To be on the save side you should operate with floats during the lab course.

In IDL you do not have to declare the type of a variable. It is possible to assign an other type to an existing variable. For example you can do something like

```
a = 'a'
a = 1
print, a
```

This can lead to errors. Especially if you loose track on whether a variable is a float or an integer.

!h

function	purpose
n_elements(arg)	returns the length of an array
sqrt(arg)	square root operation
min(arg)	return the minimum of an array
max(arg)	return the maximum of an array
mean(arg)	return the mean of an array
total(arg)	sum all entries of an array
fltarr(arg)	creates an array of zeros of specified length
findgen(arg)	creates an array of natural numbers of specified length

Tabelle 3: Some useful arithmetic functions implemented in IDL.

2.1.2 Arrays

You can perform the basic arithmetic operations not only on single numbers but allso on arrays of data. For example, type:

```
a = [1,2,3]
print, a
print, a + a
print, 2 * a
```

Arrays can be multidimensional `b = [[1,2],[3,4]]` creates a 2×2 matrix. We will handle images as large two dimensional arrays and curves as one dimensional arrays.

2.1.3 Functions

IDL has a lot of so called functions implemented. Most mathematical operations are in implemented as functions. All functions have in common that they return a value. Some functions take one or multiple arguments. Some useful basic functions are listed in table 3.

Example:

```
a = sqrt(5.0)
print, a
```

2.1.4 Procedures

Procedures are similar to functions. One difference is that procedures do not need to have a return value. If procedures are called with one or more arguments, those are passed to them separated by commas. You already know the `print` procedure. Other useful procedures are `help` and `statist` to gain information about a variable. We will use the `plot` and the `tvscl` procedures to display our data.

!h

command	meaning
eq	equal
gt	greater than
ge	greater equal
lt	less then
le	less equal
not	negates the statement

Tabelle 4: Logical operators in IDL

Example:

```
x = findgen(1000)
y = (x-300.)^2
plot, x, y
```

Some functions and procedures can be passed a lot of arguments. See the exelis web page for further information on procedures.

2.1.5 Writing your own programs

IDL procedures and functions are written in `ASCII` files named `.pro`. To write a program just create a file call, e.g. `myprogram.pro` and open it. Do not use capital letters in file names, otherwise IDL does not compile it automatically. The beginning of a program must contain `pro program_name`, at the end the `end` statement is needed.

Example:

```
pro myprogram
print, 'hello world'
end
```

Save this program and type its name in the IDL prompt. The program is executed. If you change something in the program and save it while IDL is still open you have to compile. You can do so by typing `.r program_name` int the prompt. The program has either to be located in the directory where you started IDL or the directory where the program is located has to be added to the IDL_PATH variable for IDL to find the program. You can do everything inside a program that you can do in the command line. For example define variables, execute calculation or call other procedures and functions.

2.1.6 Comparison

2.1.7 If clause

You can compare variables in IDL with the logical operators listed in table 4. The if clause lets you execute parts of your program if certain conditions are fulfilled.

Example:

```
a = 1 b = 1 if a eq b then print, 'hello'
```

Everything after the 'then' is executed if the condition is true. If you want to execute multiple lines you can use the `begin` and `endif` statements. They function like brackets in many other languages do.

```
if a eq b then begin
print, 'hello'
endif
```

2.1.8 Else clause

You may optionally add an `else` in case you want to execute something else when the if clause is not fulfilled.

```
if a eq b then begin
print, 'hello'
endif else begin
print, 'hola'
endelse
```

2.1.9 For loop

The for loop is useful if you want to run the same section of a program multiple times.

You can execute it in one line:

```
for i=0,9 do print, i
```

Or you can also repeat multiple lines of your program:

```
array = findgen(10)
for i=0,n_elements(array)-1 do begin
array[i] = 0
endfor
```

The if else and for commands should be sufficient for the lab course. There are others like `while` and `case` as well.

2.1.10 Ploting

Use the `plot` command to display curves.

Minimal example:

```
x = findgen(100)
y = x^2
plot, x, y
```

You can also change colors and overplot other curves.

```
x = findgen(100)
y = x^2
plot, x, y, xtitle = 'xaxis', ytitle = 'yaxis'
loadct, 2 ; load a different color table
oplot, x, y+300, color = 100
```

There are a lot other commands which you can use to tweak your figures.

2.1.11 Displaying data arrays

Two dimensional arrays are displayed with `tvscl`. We will use this command to display images. It scales the array between its minimum and maximum.

```
image = dist(200) ; create an image for demonstration purposes tvscl, image
```

2.1.12 Working with arrays

If you have a large image you may not be interested in the whole array. You can address subsections as follows:

```
array = dist(1000)
subimage = array[300:599,100:799]
tvscl, subimage
```

- Note that IDL starts counting with zero.

You can also slice a multidimensional array:

```
line = reform(subimage[100,*])
plot, line
```

$*$ addresses the whole dimension. The `reform` function reduces the dimensionality (IDL is sometimes a little weird).

2.1.13 The `stop` command

The `stop` command is a very useful debugging tool, the program stops where you put it and jumps to the command line. You can address all defined variables there. You can also change their values. With `.c` (for continue) you can proceed again.

2.1.14 Comments

You can comment in IDL with the hashtag symbol.

```
; this line will not be executed.
```

2.2 Optional tasks

The Fibonacci series
The Fibonacci series $f(n)$ is defined as:

$$\begin{aligned} f(1) &= 1 \\ f(2) &= 1 \\ f(n) &= f(n-2) + f(n-1) \quad \text{for} \quad n > 2 \end{aligned}$$

- Write a function that calculates the nth element of the Fibonacci series.
- Plot the Fibonacci series up to the 100th element.

Prime Numbers

- Write a program that checks whether a natural is a prime number.
- Write a program that counts the number of prime numbers in a given interval.

3 GDL incompatible part

This part uses functions which are not supported by the freeware version. You need the commercial IDL.

3.1 Reading data

First, the file containing the data has to be specified. Then the data in this file can be read into a variable.

```
path='/Pfad 1/Pfad 2/.../Bildordner' file='test.fits' raw = readfits(path+file)
```

The variable `raw` is multidimensional. Its structure can be determined with:

```
s = size(raw, /dim) print, s
```

3.2 Saving and restoring data

If one uses exclussively IDL saving into an idl `.sav` file is useful. Otherwise there are also ways to write into `ASCII` files.

```
data = findgen(100) save, data, filename = 'test.sav'
```

IDL could be closed now. In a new session the variable can be restored via:

3.3 Exporting figures

Alternatively to plotting data into the x-window the `.eps` format can be chosen aswell. Be careful and double check your plot. x-window and eps figures do not look the same.

```
x = findgen(10)
set_plot, 'ps'
device, filename = 'test.eps', /encapsulated, /color
plot, x
device, /close
set_plot, 'x'
```

One might want to convert the result into `.pdf` later from the command line. To do this type `epstopdf text.eps` in the prompt after creating the file.

Two dimensional arrays can be saved as `.png`.

```
image = dist(300)
write_png, 'test.png', bytscl(image)
```

3.4 Fitting curves

Parabola fit
There are multiple function to fit curves to data in IDL. For a parabola fit `poly_fit` is useful.

```
y = spectrum[x0:x1]
y = [13., 9., 7.,3., 2.,3., 5., 8., 12.]
x = findgen(n_elements(y)
p = reform(poly_fit(x,y,2))
fit = p[0] + p[1] x + p[2] x2̂
plot, x, y
loadct, 2
oplot, x, fit, color = 100
```

Fitting arbitrary functions using curvefit
You can use IDL to fit any analytical function to your data. The following example shows how to fit a sine curve.

$$f(x) = a\sin(bx + c) + d \tag{1}$$

Calculate the partial derivatives with respect to the free parameters:

$$\frac{\mathrm{d}f}{\mathrm{d}a} = \sin(bx) \tag{2}$$
$$\frac{\mathrm{d}f}{\mathrm{d}b} = ax\cos(bx + c) \tag{3}$$
$$\frac{\mathrm{d}f}{\mathrm{d}c} = a\cos(bx + c) \tag{4}$$
$$\frac{\mathrm{d}f}{\mathrm{d}d} = 1.0 \tag{5}$$

Now we write a program which creates the curve we want to fit. Create a file and name it `sine_function.pro`. Write the following program:

```
pro sine_function, x, param, f, pder
f = param[0] * sin(param[1]*x + param[2]) + param[3]
if n_params() ge 4 then $
   pder = [[sin(param[1]*x + param[2])], $
           [param[0]*x*cos(param[1]*x+param[2])], $
           [param[0]*cos(param[1]*x+param[2])], $
           [replicate(1.0, n_elements(x))]]
end
```

It contains the defentition of the function. When it is called dith four parameters it returns the partial derivatives aswell. Compile the function. Now we create sample data to fit:

```
xdata = findgen(1000)
ydata = sin(xdata/1000.*2.*!pi)*5.+2.+randomn(seed, 1000)
plot, xdata, ydata
;We define equal weights and call curve fit:
weights = replicate(1., n_elements(xdata))
;Initial guess parameters:
a = [3.,0.007,0.,0.]
yfit = curvefit(xdata, ydata, weights, a, sigma, function_name='sine_function')
loadct, 2, /silent
oplot, xdata, ydata, color=100
```

3.5 Data Analysis

We recommend to analyze your data within an IDL procedure. Create a `.pro` file for each lab experiment.

```
pro analysis
# load some data here
#do some calculations
#plot some figures
stop
end
```

This way you can iteratively change and adjust your program, save it, recompile it and run it again. Always leave a `stop` at the end so you can do some testing. Analyzing data this way is way more convenient than loading everything from the command line.