Introduction to IDL

1 - Basics

Paulo Penteado
pp.penteado@gmail.com
http://www.ppenteado.net

IDL

- **I**nteractive **D**ata **L**anguage
- General Characteristics:
  - Created for data processing and visualization in Astronomy, Remote Sensing and Medical Imaging.
  - Array-centered: advanced support for vectorization in more than 1 dimension.
  - Can be used interactively or programmatically.
  - Extensive standard library for data processing and visualization.
  - Platform-independent.
  - **Not** free. Currently made by Exelis Vis.
  - http://www.exelisvis.com/ProductsServices/IDL.aspx

IDL

- Created in 1977, but still actively developed (version 8.4, with significant improvements, is from October 2014).
- Is a complete language.
- Has many modern features (objects, platform independence, interactive execution, vectorization, easy creation of graphics).
- Two main characteristics to classify programming languages:
  - Compiled x interpreted
  - Static types x dynamic types

Compiled x interpreted languages

- Modern languages are high level – source code close to how people communicate.
- All a computer understands is a binary code (lowest level).
- A program must be translated from the high level to the binary code.
- The translation can be done:
  - Once, before the program is used – compiled languages.
  - At every time the program is used (“live”) - interpreted languages.
- Some (usually) compiled languages: C, C++, Fortran.
- Some (usually) interpreted languages: R, Perl, bash.
- Languages either compiled or interpreted: IDL, Python
- Usually an IDL program is compiled on demand, the first time it is called in an IDL session (not automatically recompiled afterwards).
- IDL can also be used interactively (interpreted).

Static x dynamic types

- Most modern programs make use of variables.
- Each variable represents data of a certain type (integer, real, character, etc.).
- In a **statically typed** language, each variable has to be declared with a fixed type. Exs: C, C++, Fortran, Java.

Declaration examples (C, C++):

```
int number_of_days;
double temperature;
```

- In a **dynamically typed language**, a variable can come into existence at any time, and can change type / dimensions at any time in the program. Exs: IDL, Python, R, Perl.

Example (IDL):

```
a=17.9
a='some text'
```

a is now a real number (single precision / float)
a is now a string

Licenses and availability

- A full IDL is not free.
- License prices vary widely depending on country, status (commercial / academic / student), number of licenses bought, negotiation, license type (node-locked, network, etc), renewal status.
- The IDL Virtual Machine (VM) can be downloaded freely
  - Without a license, runs IDL for 7 minutes, with file writing disabled.
  - Some compiled IDL programs can be run with just the VM (there are restrictions).
  - With a full IDL, some programs can be compiled and packaged with the VM into a self-contained program that does not need licenses to run.

GDL / FL

- GNU Data Language
- A free, open-source implementation of the IDL language.
- Fully compatible with the IDL language up to IDL 7.1.
- Partially compatible with the IDL language elements introduced since IDL 8.0.
- Not everything in the IDL standard library has been implemented in GDL.
- http://gnudatalanguage.sourceforge.net/
- There is also another, lesser known, open source implementation, Fawlty Language (FL).
- Both GDL and FL are still active projects.

Libraries

- Though the IDL standard library is extensive (http://www.exelisvis.com/docs/routines-1.html), some commonly used functions were developed by others:
  - Coyote Library (David Fanning, http://www.idlcoyote.com/)
  - The IDL Astronomy User's Library (Wayne Landsman, http://idlastro.gsfc.nasa.gov/)
  - Michael Galloy's (http://michaelgalloy.com/)
  - Craig Markwardt's (http://www.physics.wisc.edu/~craigm/idl/)
  - Paulo Penteado's (http://www.ppenteado.net)

Library Installation

- Most of the time, installing a library means copying the files to some directory, then adding that directory to IDL's search path.
- IDL's search path is where it looks for programs when they are called.
- IDL looks in the path for a file with the same name of the program that was called (in lowercase letters).

Setting the search path:

- Through the workbench: Window / IDL -> Preferences -> Paths -> IDL Path
- Through the command line: pref_get / pref_set command. Ex:

```
IDL> path=pref_get('IDL_PATH')
IDL> print,path
<IDL_DEFAULT>:/software/idl/others/idlastro/pro:/software/pp_lib/src
IDL> path=path+':+/home/user/myidl/'
IDL> pref_set,'IDL_PATH',path,/commit
```

- Through an environment variable (IDL_PATH). If that variable exists, the path specified in the preferernces (Workbench / pref_set) is ignored.
- A path must always contain <IDL_DEFAULT>, usually as the first entry.
- Directories are separated by **:**, and a **+** before a directory measn to also include all subdirectories in it.

Using IDL

- From the command line (either a terminal, or the Workbench) the user can **interactively** run any commands, create variables, make plots, save files, ...

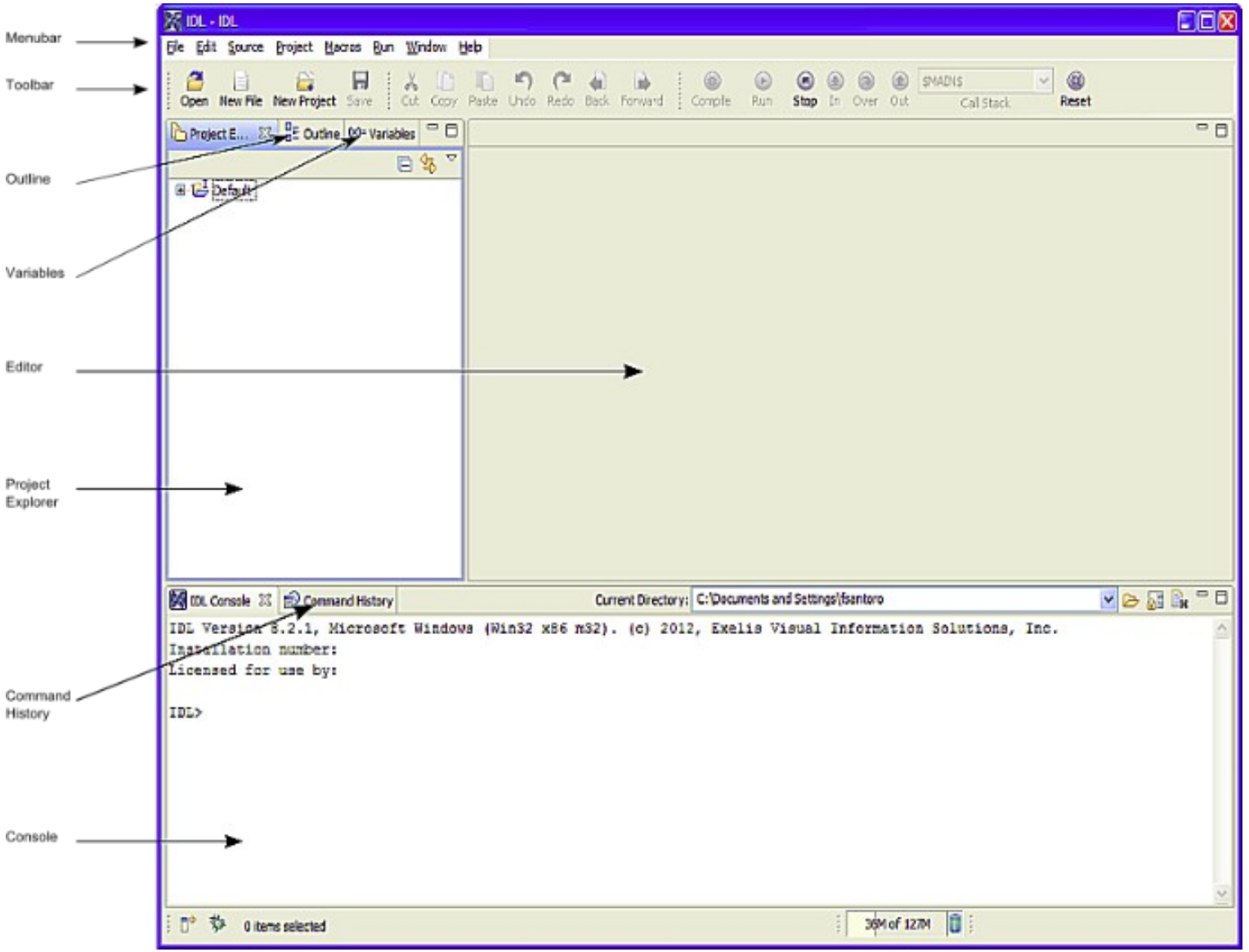
Using IDL

```
IDL> a=dindgen(100)/99d0
IDL> print,a
       0.0000000     0.010101010     0.020202020     0.030303030
0.040404040     0.050505051     0.060606061     0.070707071
     0.080808081     0.090909091      0.10101010      0.11111111

(...)

      0.80808081      0.81818182      0.82828283      0.83838384
0.84848485      0.85858586      0.86868687      0.87878788
      0.88888889      0.89898990      0.90909091      0.91919192
0.92929293      0.93939394      0.94949495      0.95959596
      0.96969697      0.97979798      0.98989899       1.0000000
IDL> b=cos(a*!dpi)
IDL> iplot,a,b
IDL> save,file='mydata.sav',a,b
```

The IDL Workbench

Started by clicking on some icon, or, from a terminal, with

`idlde`



(from http://www.exelisvis.com/docs/idldeoverview.html)

Language structure

- Source file types
  - Programs
    - Contain definitions of procedures, functions, and/or a main program.
  - Batch files
    - Contain a series of single-line statements, to be executed as if typing in the command line.
  - All use the .pro extension

Procedures and Functions

- A procedure is a routine that gets called and does things, usually using arguments/keywords for input/output.
- A function is like a procedure, but it returns a value (it can also use arguments and keywords for input and output).
- Examples:

```
IDL> print,17.2
      17.2000
IDL> myvariable=exp(3.2d0)
```

Procedure call (**`print`**)

Function call (**`exp`**)

Procedure and function definition

Put this into a file called **my_first_program.pro**

```
function my_function,arg
return,2.0*arg
end

pro my_first_program,argument1,argument2,keyword1=keyword1
;this code does some really boring, trivial stuff
argument2=argument1+my_function(3.0)+keyword1
end
```

Then,

```
IDL> my_first_program,1.0,a,keyword1=5.0
% Compiled module: MY_FIRST_PROGRAM.
IDL> print,a
      12.0000
```

Language elements

- Anything following a **;** is a comment.
  - Can be in the same line of code, or in a separate line.
- Statements usually are written one per line.
- Multiple statements can be put in the same line with an **&** separating them.
- A long statement can be broken into several lines by ending each line to be continued with a **$**.
- Variable/procedure/function names are case-insensitive.
- Text strings can be delimited by either single (**'**) or double quotes (**“**).
  - The string must be closed with the same type of quote mark used to open it.
- Function arguments are placed inside parenthesis.
- Arguments / keywords are separated by a **,**.

Language elements

- When a procedure or function is called, IDL looks for a file with the procedure/function name (ending in .pro) in the search path and the current directory.
  - When such a file is found, IDL looks for a function/procedure definition matching that name (in lower case).
  - The file is only read until IDL finds that function/procedure.
  - Any functions/procedures preceding the one being searched for get compiled.
  - Any functions/procedures after that are ignored.
  - If there is a .sav file with the name of the function/procedure, IDL will try to restore the function/procedure from that file (routines can be compiled and saved into .sav files).
  - IDL will not automatically recompile a routine. If you update a source code after it was compiled, it will only have effect if you ask IDL to compile it, or reset the session.

Language elements

- In the previous example, calling **`my_first_program`** will cause IDL to look for a file called **`my_first_program.pro`**.
- When that file is found, first `my_function` gets compiled, then **`my_first_program`** gets compiled.
- Just calling **`my_function`**, when t has not been compiled, will not find it, since IDL will look for a file called **`my_function.pro`**.
- After **`my_first_program.pro`** has been compiled, my_function can be used.
- To request IDL to compile the file, without running anything, use
  - **`.compile my_first_program`**
- Or reset the session, by either exiting and starting IDL, or with
  - **`.reset_session`** or **`.full_reset_session`**.

The 5 most used commands

- **exit**
- **`.full_reset_session`**
  - Almost the same as exiting and restarting IDL: erases variables, forgets all compiled routines, closes all graphics windows and open files, etc.
- **`?`**
  - Opens the IDL help in a web browser. If followed by a routine name (ex: **`?plot`**), opens the help in that routine's page.bles, among many other things. Exs:

The 5 most used commands

- **`print`**
  - Produces a text representation of one or more values (its arguments). Exs:

```
IDL> print,exp(2.7),cos(!dpi),' potato soup'
      14.8797      -1.0000000 potato soup
IDL> print,!dpi
       3.1415927
IDL> print,!dpi,format='(E22.15)'
 3.141592653589793E+00
```

The 5 most used commands

- **`help`**
  - Shows information about variables, among many other things. Exs:

```
IDL> a=12
IDL> b='salad'
IDL> c=[1,9,7]
IDL> help
% At $MAIN$
A               INT       =       12
B               STRING    = 'salad'
C               INT       = Array[3]
IDL> help,b
B               STRING    = 'salad'
```

Operators

=	Assignment
+ - * /	Basic math
+	Concatenation of strings, lists and hashes (**`a='some '+'string'`**)
**	Exponentiation
mod	Modulo (**`5 mod 2`** is 1)
++ --	Increment / decrement by one
*	Pointer dereference
eq ne	Equal to, not equal to
gt lt	Greater than, Less than
ge le	Greater than or equal to, less than or equal to
# ##	Matrix product
.	Method invocation / field access
->	Method invocation
and or not	Bitwise operators
&& || ~	Logical operators
>	The larger of the two. Ex: **`3 > 4`** is 4.
<	The smaller of the two. Ex: **`3 < 4`** is 3.

Compound assignment

- An operator combined with **=**.
- Ex: **+=**:
  - **a+=9** means **a=a+9**
- The same idea for other compound operators:
  - **-=**, ***=**, **/=**, etc.

Control structures

- **if .. then .. else**

```
IDL> if (a gt 7) then print,'a is greater than 7' else print,'not'
a is greater than 7

if (a gt 9) then begin
  b=78
  a=a-b
endif else begin
  b=0
  c=cos(17.9)
endelse
```

Control structures

- **`case`**

```
x=5
CASE x OF
   1: PRINT, 'one'
   2: PRINT, 'two'
   3: PRINT, 'three'
   4: PRINT, 'four'
   ELSE: BEGIN
      PRINT, 'You entered: ', x
      PRINT, 'Please enter a value between 1 and 4'
      END
```

(from http://www.exelisvis.com/docs/case.html#Addition)

Control structures

- **switch**

```
PRO ex_switch, x
   SWITCH x OF
      1: PRINT, 'one'
      2: PRINT, 'two'
      3: PRINT, 'three'
      4: BEGIN
         PRINT, 'four'
         BREAK
        END
      ELSE: BEGIN
         PRINT, 'You entered: ', x
         PRINT, 'Please enter a value between 1 and 4'
         END
   ENDSWITCH
END

IDL>ex_switch, 2
two
three
four
```

Control structures

- **`for`** loops

```
IDL> for i=3,7 do print,i
       3
       4
       5
       6
       7
IDL> for i=3,7,2 do print,i
       3
       5
       7
```

Control structures

- **`for`** loops

```
for i=3,7 do begin
  a=sqrt(i)
  print,i,a
endfor
```

- **`foreach`** loops will be discussed after we talk about arrays, lists and hashes.

Control structures

- **`while .. do`**
- If the condition is true, keeps repeating the statement / block until it becomes false.

```
a=9
while (a lt 13) do begin
  a=a+0.8
  print,a
endwhile
```

Control structures

- **`repeat .. until`**
- Repeats a statement or block until the condition becomes true. The statement / block is executed at least once.

```
A = 1

B = 10

REPEAT A = A * 2 UNTIL A GT B


REPEAT BEGIN
   A = A * 2
ENDREP UNTIL A GT B
```

(from http://www.exelisvis.com/docs/REPEAT___UNTIL.html)

A few more elements

- **`break`**
  - When inside a loop, program execution jumps to the next line after the end of the loop.
- **`continue`**
  - When inside a loop, program execution goes to the beginning of the loop, for the next iteration (if there is one).
- **`stop`**
  - Interrupts a program execution. The command line will be at the point the program was stopped, with all the variables acessible.

A few more elements

- `? :`
  - Ternary conditional assignment operator.
  - Example:

```
a=b gt 9 ? 10 : 20
```

is a way of saying

```
if (b gt 9) then a =10 else a=20.
```

Logical values

- Before IDL 8.4, there was no boolean (true/false) type.
- Variables / expressions of any type can be interpreted for true/false:
  - If the type is not integer: zero or null values (empty string, null pointer, null object) are false; everything else is true.
  - If the type is integer, it depends on a compilation flag:
    - By default, integers are interpreted bitwise: even numbers are false, odd numbers are true.
    - If using '**`compile_opt logical_predicate`**', zero is false, anything else is true – as most other languages do.

Batch files

- If you put a series of single statements into a .pro file, it can be executed as if they were being typed on the command line:
  - **`@my_batch_file`**
  - This will run everything in **`my_batch_file.pro`**.
  - The batch file cannot contain blocks (begin .. end) or routine definitions.

Journal files

- Create a log file that is also a batch file:

```
IDL> journal,'myjournal.pro'
IDL> a=12.7
IDL> print,a-cos(1.8)
      12.9272
IDL> journal
```

- Result (**`myjournal.pro`**):

```
; IDL Version 8.4 (linux x86_64 m64)
; Journal File
; Date: Thu Jan 15 06:17:37 2015

a=12.7
print,a-cos(1.8)
;      12.9272
```

Main programs

- A `.pro` file can contain a main program.
  - Written just like a procedure, but without arguments/keywords, without the declaration line (“`pro myprocedure,arg1,arg2,...`”)
  - Executed with “**`.run myprogram`**”, which will look for a main program in the file **`myprogram.pro`**.
  - At the end, all the program's variables are still accessible from the command line.
  - The file can contain routine definitions.

Main programs

- Example – file called **`my_main_program.pro`**:

```
function my_function,arg
return,2.0*arg
end

pro
my_first_program,argument1,argument2,keyword1=keyword1
;this code does some really boring, trivial stuff
argument2=argument1+my_function(3.0)+keyword1
end

my_first_program,1.8,c,keyword1=-18.9
end
```

Main programs

- When we run that file:

```
IDL> .run my_main_program
% Compiled module: MY_FUNCTION.
% Compiled module: MY_FIRST_PROGRAM.
% Compiled module: $MAIN$.
IDL> help
% At $MAIN$                      1
/homec/penteado/cpc/cpcc/new/my_main_program.pro
C               FLOAT     =      -11.1000
Compiled Procedures:
    $MAIN$  MY_FIRST_PROGRAM

Compiled Functions:
        MY_FUNCTION
```

Debugger

- Integrated into the Workbench, allows easy inspection of the state of the program, while the program is in the middle of its execution.
- The user can run the program line by line, inspecting the values of variables, even making plots with them.
- The user can easily see how the program got to that line in the source code (which routine called which routine, at what line).
- Makes debugging much easier and faster than filling the program with print statements.
- The live demo shows this better.

Some references

- The IDL Newsgroup
  - https://groups.google.com/forum/#!forum/comp.lang.idl-pvwave
- Modern IDL
  - Book by Michael Galloy, the best reference. Kept updated with each new IDL release.
  - http://modernidl.idldev.com/
- IDL Help online
  - http://www.exelisvis.com/docs/routines-1.html
- Coyote's Guide to IDL Programming (by David Fanning)
  - http://www.idlcoyote.com/
- The IDL Workbench video tutorial
  - https://www.youtube.com/watch?v=TTeZbFWy8YI
- This file
  - http://www.ppenteado.net/idl/intro