

**M1 Astrophysique**  
*Introduction à IDL*  
*pour le traitement de données astro*

---

**Stéphane Erard**

# Références

---

## Supports de cours IDL en ligne

- S. Erard** <http://www.lesia.obspm.fr/perso/stephane-erard/idl/idl.html>  
(ou site UFE: <http://ufe.obspm.fr/offreFP/coursIDL/>)
- J. Abouharham** <http://ufe.obspm.fr/offreFP/coursIDL/>
- J. M. Malherbe** (cours M1, sur site UFE ?)

## En ligne aussi (avec beaucoup d'autres):

- Tutoriels IDL** [https://www.l3harrisgeospatial.com/docs/using\\_idl\\_home.html](https://www.l3harrisgeospatial.com/docs/using_idl_home.html)  
(très orientés vers les nouveautés, pas le plus efficace)
- R. O'Connell** <https://rwoconne.github.io/rwoclass/ast511/IDLexercises/index.html>  
(orienté astro)

## IDL astronomy user's library (NASA)

<https://idlastro.gsfc.nasa.gov/homepage.html>

## En bref...

---

- **Langage de traitement de données standard en astro**  
(développé dans un labo NASA au début des années 80)
- **Capacités graphiques très développées**  
(uniques au début des années 90)
- **Système multiplateforme (Mac / Unix / Linux / Windows)**  
Permet le travail collaboratif, le développement sur projets
- **Orienté images / visualisation**  
Fonctionnement interactif  $\Leftrightarrow$  langage interprété  
A eu la réputation d'être lent côté calcul (plus très vrai)
- **Alternatives :**  
Matlab (calcul + visu)  
C, C++, fortran (calculs rapides)  
python (pas forcément plus simple, gratuit)  
**\*\* GDL (clone open source, en développement) \*\***

# A savoir

---

## **C'est un langage de programmation...**

**Très orienté usage interactif**

- **Moderne:**

**On ne programme pas comme en C / fortran 77 / Basic / Pascal...**

**Syntaxe très condensée, les calculs sont vectorisés:**

- **Toutes les variables se manipulent comme des scalaires**
- **On fait le moins de boucles possibles**
- **IDL 8.x intègre une syntaxe ~ python**

- **Orienté graphique:**

**Très puissantes fonctions d'affichage toutes faites**

- **De nombreuses bibliothèques publiques existent**

**Utiliser les bonnes...**

- **Coûteux !**

**On utilise GDL à l'Ecole Doctorale: il y a des limitations  
(émulation d'IDL 7)**

# Lancement

## Deux solutions sous IDL:

- Ligne de commande au terminal (Unix / Linux / Mac)
- Environnement idlde (tous systèmes)

## Sous GDL:

- Ligne de commande au terminal  
(pour Windows, voir la doc)

The screenshot displays the IDL environment on a Mac. On the left, a Terminal window shows the execution of the 'idl' command, displaying version information and environment variables. The main window is the IDL Workbench IDE, showing a project named 'diskmap.pro'. The IDE interface includes a toolbar, a project tree, a code editor with a syntax-highlighted script, a console window showing the output of the script, and a command line at the bottom. The console output shows the execution of various IDL commands and the resulting plot of a sine wave. A progress window at the bottom right indicates that the IDL startup process has completed.

```
[pandora:~] serard% idl
IDL Version 7.0.8, Mac OS X (darwin x86_64 m64). (c) 2009, ITT Visual Information Solutions
Installation number:
Licensed for use by:

% Compiled module: PATH_SEP.
% Compiled module: ASTROLIB.
% ASTROLIB: Astronomy Library system variables have been added
% Compiled module: ISMDEF.
% $MAIN$: Fichiers de cartographie planetaire dans !CARTES_DIR
% $MAIN$: Exemples doc dans !EX_DIR
Ajout des chemins pour les exemples
% $MAIN$: Spectres PDS dans !PDS_SLIB
% $MAIN$: Spectres dans !SPEC_DIR
% $MAIN$: Images Ceres dans !Ceres_dir
% $MAIN$: Spectres PDS dans !VEX_DIR

IDL> help
% At $MAIN$
SEPAR      STRING      = ':'
Compiled Procedures:
  $MAIN$  ASTROLIB  ISMDEF

Compiled Functions:
  PATH_SEP

IDL> ? plot
% ONLINE_HELP: Starting the online help browser.
IDL> x=findgen(200)
IDL> plot, x
IDL> x2=findgen(200)*3.6
IDL> y=sin(x2/180.*pi)
IDL> plot, x2, y
IDL> plot, x2, y, title = 'Fonction sinus'
IDL>
IDL> plots, [0,300], [0,0]
IDL> oplot, cos(x2/180.*pi) /2.
IDL> xyouts, 300, 0.5, 'Legende'
IDL>
IDL> plot, x, y, title = 'Fonction sinus', psym=2
IDL> save, filename='bidon.idl', X2, Y
IDL> $pwd
/Users/serard
IDL> print, 10+3
13
```

# Utilisation

---

## Mode interactif (TP1)

On tape des instructions successives sur la ligne de commande  
=> Pas de bloc d'instructions, tout doit tenir sur une seule ligne  
Exécutées quand on appuie sur ENTER  
On peut appeler des routines avec paramètres

Exploration de données  
Calculs rapides...

## Séquences d'instructions

On écrit des séquences dans un fichier qu'on appelle en ligne  
=> entièrement exécutées en série

Appel par @script

## Programmation (TP2)

On écrit des routines, elles s'appellent entre elles  
=> acceptent et retournent des arguments et paramètres  
=> les blocs d'instructions permettent de structurer  
=> il faut compiler avant exécution

Traitements récurrents  
ou routiniers  
Traitements complexes



# Routines

---

## Procédures

Syntaxe d'appel:

**procedure, argument1, argument2, ... motclef1=val1, motclef2=val2**

**Arguments : peuvent être une expression ou une variable  
servent à la fois en entrée et en sortie (les variables peuvent être modifiées)**

**Mot-clefs : prédéfinis dans la routine  
la valeur peut être une expression ou une variable**

Exemple-type: **plot, x, y**

## Fonctions

Syntaxe d'appel:

**resultat = fonction(argument1, argument2, ... motclef1=val1, motclef2=val2)**

**En principe, renvoient seulement un résultat, affecté à une variable  
Dans IDL, les arguments fonctionnent comme ceux des procédures**

Exemple-type: **a = exp(x)**



# Routines

---

**procedure, argument1, argument2, ... motclef1=val1, motclef2=val2**  
**resultat = fonction(argument1, argument2, ... motclef1=val1, motclef2=val2)**

**Mot-clefs : optionnels par définition**

la valeur (obligatoire) peut être une constante, une expression ou une variable

Exemple: `plot, x, color=100`

la syntaxe `/motclef` signifie `motclef = 1` (utilisée pour les options binaires)

Exemple: `plot, r, theta, /polar`

le motclef peut être abrégé (forme minimum non ambiguë)

Exemple: `plot, x, col=100`

**Arguments : paramètres, servent à la fois en entrée et en sortie**

**=> tous les arguments *variables* peuvent être modifiés en sortie**

**Tous les paramètres n'ont pas à être fournis**

**Leur signification peut changer selon leur nombre**

Exemple: `plot, x` => affiche le vecteur x en fonction du numéro d'indice

`plot, x, y` => affiche y en fonction de x

**Résultat : paramètre de sortie pour une fonction**

**type fixé par la fonction**

**Toutes les routines de base ont un fonctionnement simple par défaut, modifiable en ajoutant des paramètres et mots-clefs (plot...)**

# Valeurs

---

## Constantes

Valeurs représentées directement

Exemples : 3 3. 2.2E3 3.45d 2L "abcd" [1,2]

## Variables

Stockent des valeurs, désignées par un nom (commençant par une lettre)

**Les noms doivent être différents des noms de routines/instructions**

Exemples: a x X1sdfsdfre45668o000

## Expressions

Valeurs calculées immédiatement – non stockées, recalculées à chaque fois

Exemples : 3 3.+2 2+a cos(a) exp(x+2.)/y

Affectation d'une valeur à une variable:

On évalue l'expression et on la stocke dans la variable:

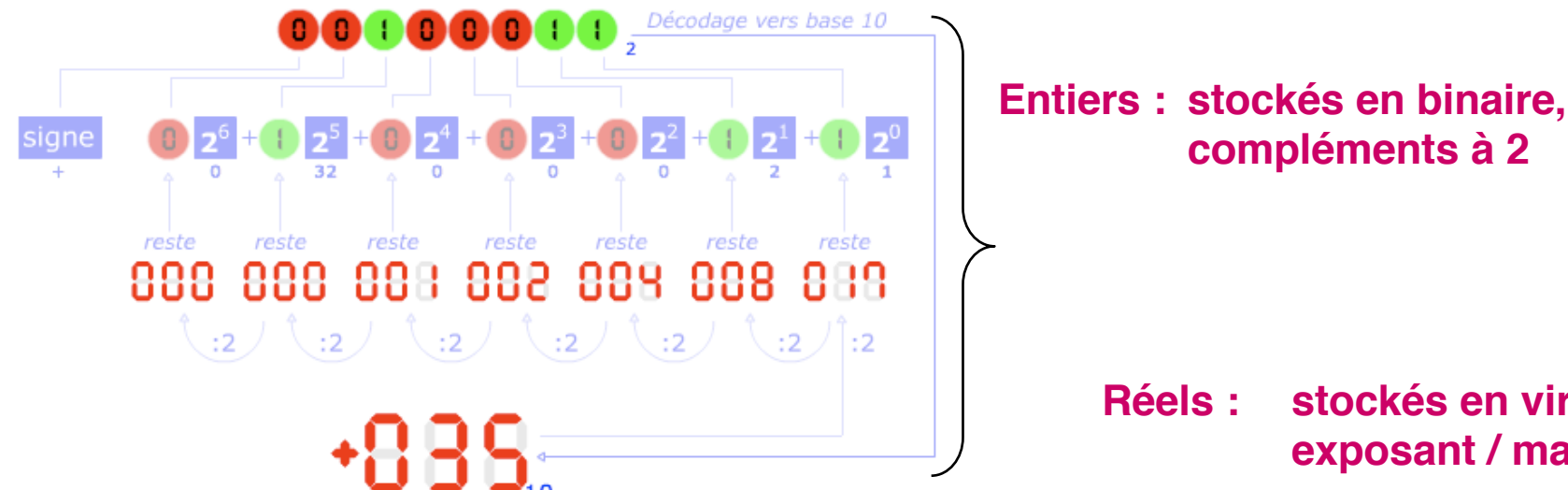
**Variable = < expression >**

# Types de variables

Les variables et expressions sont stockées dans un espace en mémoire, et évaluées par la machine. Ceci impose :

- ⇒ Des limites en valeurs (min / max)
- ⇒ Des limites en précision (nombre de chiffres significatifs pour les réels)
- ⇒ Des limites de calcul (erreurs d'arrondi, cumulées)

Il existe différents types de variables, liés au mode de représentation en mémoire



Codage par virgule flottante : 1 0 0 0 0 0 0 x 2<sup>000</sup>

caché

# Types de variables

Les variables et expressions ont toutes un type, lié au mode de représentation en mémoire  
Le type est défini au moment de l'affectation, généralement de façon implicite

**Le type d'une variable peut changer en cours de traitement**

Types principaux	Taille de stockage	Exemples	Limites
entiers (courts)	16 bits (2 octets)	3	-32768 à 32767
entiers non-signés	16 bits	uint(3)	0 à 65535
entiers longs	32 bits	3L 50000	$-(2^{31})$ à $2^{31}-1$
Réels (flottants)	32 bits	3. 3e-4	$\pm 10^{38}$ 7 chiffres significatifs
Réels double précision	64 bits	3.2d 3d-4	$\pm 10^{308}$ 14 chiffres significatifs
Complexes	2 x 64 bits	complex(3.2,2)	$(\pm 10^{38}, \pm 10^{38})$
Chaînes	Ajustable	"abcd" 'abcd'	pas de caractères spéciaux

# Scalaires et tableaux

Les variables et expressions ont également une dimension tensorielle

La dimension est définie au moment de la déclaration, généralement de façon explicite

**Les dimensions d'une variable peuvent être redéfinies en cours de traitement**

Types principaux	Exemples	Valeurs
scalaires	3, etc...	3
vecteurs (en ligne!)	[ 3 , 2 ]	(3, 2)
vecteurs-colonne	transpose( [ 3,2 ] ) [ [3] , [2] ]	$\begin{pmatrix} 3 \\ 2 \end{pmatrix}$
Tableaux	[ [2,3] , [4,2] ]	$\begin{pmatrix} 2 & 3 \\ 4 & 2 \end{pmatrix}$

Jusqu'à 8 dimensions possibles

La notation est transposée par rapport à l'écriture matricielle: (X,Y)

Définition explicite du type et des dimensions : `image = ftarr(250, 340)`

Propriétés (type + valeur [scalaire] ou dimensions [tableaux] ) : `help`  
`help, <variable>`

# Opérateurs

Les opérations numériques peuvent s'effectuer entre variables ou constantes de types et dimensions variables

Les opérateurs ne mélangent jamais les types numériques et les chaînes

Opérateurs	Exemples	Commentaires
numériques	+ - / * ^ mod	tous types sauf chaînes
matriciels	# ##	multiplication matricielle: ##
logiques	AND OR NOT &&    ~	identiques si appliqués à des valeurs booléennes (0 ou 1)
comparaisons	GT LT GE LE EQ NE	retournent toujours une valeur vraie (1) ou fausse (0)
seuillages	> <	retournent le plus grand / petit des deux arguments

priorité décroissante

Les parenthèses s'utilisent de la façon habituelle

Seule opération sur les chaînes: concaténation

print, 'et' + chaîne1

# Opérateurs, calcul

---

Type d'une expression déterminé par celui des variables et constantes utilisées

**Le résultat d'un calcul peut excéder le type des variables !**

```
print, 10/3  
print, 32000 * 2
```

**=> A vérifier soi-même!**

En cas de doute, utiliser des entiers longs  
ou des réels

```
print, 32000 * 2L  
print, 32000 * 2.
```

## Codage des réels en norme IEEE

Il existe des valeurs spéciales :

$\pm\text{Inf}$	(résultat infini)	1. / 0.
NaN	(résultat non-numérique)	0. / 0.

**=> Message d'erreur en cas de dépassement (overflow)**

## Affichage

Calculer une opération n'affiche pas le résultat pour autant  
**=> affichage sur le terminal**

```
print, 10/3  
print, 32000 * 2
```

# Manipulation de tableaux

---

## Indexation

**Adressage avec des indices entre [ ]**

**Le premier élément est toujours noté 0 (pas 1),  
le dernier est donc N-1**

```
image = fltarr(340, 440)
print, image[0,0]
print, image[339, 439]
```

**La notation : désigne un intervalle d'indices**

```
tv, image[150:200,150:250]
```

**La notation \* représente toute une dimension  
... ou le dernier indice d'une dimension**

```
plot, image[150,*]
plot, image[150,100:*]
```

**Les tableaux multidimensionnels comprennent  
des listes d'indices monodimensionnels dans  
le sens Y=0, puis Y=1...**

```
plot, image
plot, image[*]
```

```
plot, image[* ,1]
plot, image[340:340+349]
```

**Un tableau peut indiquer un autre tableau**

```
a = [3,4,5,10]
print, image[a]
print, image[ a[3] ]
```

**On peut indiquer une expression en tableau**

```
print, ( exp( image ) )[3]
```



# Manipulation de tableaux

---

## Propriétés

**Nombre de dimensions**

```
print, size(image, /N_dim)
```

**Nb d'éléments dans chaque dimension**

```
print, size(image, /dim)
```

**Nb total d'éléments**

```
print, size(image, /N_elements)
```

**Type numérique** (codé)

```
print, size(image, /type)
```

**Initialisation avec liste d'indices (part de 0...)**

```
liste = indgen(250)
```

**Vecteur/tableau constant**

```
a = replicate( cst, 6)
```

**Changement de dimensions**

**(le nb d'éléments est préservé,**

**+ la dernière dimension est éliminée si elle est dégénérée)**

```
b = reform(a, 2, 3)
```

**Test sur les éléments d'un tableau**

=> renvoie une liste monodimensionnelle  
d'éléments vérifiant la condition

**Permet de supprimer la plupart des boucles!**

**=> accélère énormément l'exécution**

```
Liste1 = where(image GT 40.)
```

```
im1 = image
```

```
im1[Liste1]=alog(image[Liste1]-40)+40
```

# Manipulation de tableaux

---

## Opérations

avec `TAB = fctarr(N,M)`

`cst * TAB`      => appliqué à chaque elt de tableau : (N,M)

a = 3

vecteur = [2.,4.,6.,2.,4.,10.]

help, a \* vecteur

`TAB1 + TAB2`

=> opération elt par elt si les dimensions sont identiques: (N,M)

=> réduit à la partie commune

si les dimensions sont différentes: (min(N1,N2), min(M1,M2))

`TAB1 ## TAB2`

= `TAB2 # TAB1`

=> Multiplication matricielle: (N2, M1)

Erreur si  $N1 \neq M2$

**Attention à la transposition!**

`[vect1, vect2]`

allonge le vecteur: (N1+N2)

`[[vect1], [vect2]]`

empile des vecteurs de même dimension: (N,2)

`[mat1, mat2]`

allonge les lignes de la matrice: (N1+N2, M)

`[[mat1], [mat2]]`

allonge les colonnes de la matrice: (N, M1+M2)

# Manipulation de tableaux

---

## Fonctions de calcul

min(TAB)

max(TAB)

mean(TAB)

stddev(TAB)

=> min, max, moyenne

=> renvoie l'écart-type

tvsc1, TAB > 10

tv, bytsc1(TAB)

=> le seuillage est très utile pour afficher des images

=> rééchelonne TAB entre 0 et 255 (surtout pour affichage)

total(TAB)

total(TAB, 1)

total(TAB, 2)

=> renvoie la somme des éléments

=> renvoie la somme en colonnes (première dimension)

=> renvoie la somme en ligne (deuxième dimension)

A \* B

total(A \* A)

=> produit scalaire

=> module carré

# Manipulation de tableaux

---

## Modification d'images

avec **TAB = filtarr(N,M)**

transpose(TAB)

shift(TAB,d1, d2)

rotate(TAB, N)

rebin(TAB,N1, M1)

=> **décalage circulaire de d1 et d2 pixels dans chaque direction**

=> **rotation par pas de 90° , rotation+transposition pour N≥4**

=> **redimensionne d'un coefficient entier dans chaque direction  
(rééchantillonnage)**

rot(TAB,angle)

congrid(TAB, N1,M1)

=> **rotation continue**

=> **redimensionne d'un coefficient arbitraire**

Interpolation, plusieurs options disponibles

rot(TAB,angle, Coef, X0, Y0)

=> **(rotation + redim + décalage) simultanés**

Minimise les artefacts / arrondis

smooth(TAB, N)

median(TAB, N)

=> **Lissage par moyenne mobile N (impaire)**

=> **Lissage par médiane de largeur N (impaire)**

Méthode standard de réduction de bruit

# Chaînes de caractères

---

**Valeurs constantes introduites entre " ou '**

**Peuvent être utilisées à l'affichage**

**— seul l'ascii strict peut être affiché au terminal  
(pas de caractères spéciaux: é, ç « ...)**

**Seule opération sur les chaînes: concaténation**

**Fonctions sur chaînes: str\***

**S'utilisent souvent en tableaux**

```
ch1 = "abcd"
```

```
ch2 = 'abcd'
```

```
print, 'et'
```

```
xyouts, x, y, chaine
```

```
print, 'et' + chaine1
```

```
print, strlowercase(chaine)
```

```
print, strmid(ch, 'a')
```

```
chaine=['tonique', 'tierce', 'quinte']
```

# Affichages

---

## Graphiques

plot, x                   => vecteur en fonction du numéro d'indice  
plot, x, y               => y en fonction de x  
oplot, x, y2             => superpose à un graphique existant

## Options sur graphiques (plot)

**Type de graphique:**               Nsum=3   /NoData   /polar

**Mise en page:**               title="chaine"   xtitle = "vecteur X"   xrange=[0,1]   /iso   /Ynozero  
                                  thick=2   Linestyle = 3   Psym=4   color = 100   Xstyle=1

**Unités:**   /device                En pixels depuis bas/gauche écran  
              /data                 Selon les axes  
              /normal               De 0 à 1 depuis bas / gauche écran dans la fenêtre  
              /centi ou /inches    Dispo pour certaines routines

## Affichages complémentaires:

Errplot, x, E1, E2  
xyouts, x, y, "chaine"  
legend (dans ASTRON)

# Affichages

## Graphiques multiples

`!P.multi = [0, 2, 2, 0, 1]` variable de contrôle graphique

- `!P.multi(0)` : Nb de graphiques restant
- `!P.multi(1)` : Nb de graphiques en largeur
- `!P.multi(2)` : Nb de graphiques en hauteur
- `!P.multi(3)` : Nb de graphiques en Z
- `!P.multi(4)` : ordre d'affichage ligne/colonne

`plot, x, y`  
`contour, ima`  
`!P.multi = 0`

} => passent au graphique suivant au lieu d'effacer la fenêtre  
réinitialisation

## Caractères / polices

	Taille	Graisse	Type de police (Hershey / device / TrueType)
mots-clefs de plot,...	<code>charsize = 1.5</code>	<code>charthick = 2</code>	<code>font = -1/0/1</code>
variables système	<code>!P.charsize = 1.5</code>	<code>!P.charthick = 2</code>	<code>!P.font=-1/0/1</code>

**Formatage:** Les codes `!<n>` inclus dans les chaînes modifient le format courant (voir détails dans la doc IDL)

<code>!3</code> : standard	}	!	<code>!1</code> : indice
<code>!4</code> : grec			<code>!E</code> : exposant
<code>!6</code> : épais, romain			<code>!N</code> : niveau de base
<code>!7</code> : épais, grec			<code>!C</code> : à la ligne
<code>!8</code> : italiques			<code>!X</code> : première police
			<code>!9</code> : math et symboles

**Symboles latex:** Installer/utiliser la bibliothèque `TextToIDL`

`x tit = textoidl('Wavelength (\mum)')`  
`plot, lambda, x, Xtitle = x tit`

# Formatage de texte

## Exemples de la doc IDL

Lower<sup>Exponent</sup><sub>Index</sub> Normal<sup>Exp Up</sup><sub>Ind Down</sub> Above  
Below

Lower<sup>Exponent</sup><sub>Index</sub> Normal<sup>Exp Up</sup><sub>Ind Down</sub> Above  
Below

Hershey (en haut): font = -1  
TrueType (en bas): font = 1

XYOUTS, 0.1, 0.3, SIZE=3, '!LLower!S!EExponent!R!!Index!N Normal' + \$  
'!S!EExp!R!!Ind!N!S!U Up!R!D Down!N!S!A Above!R!B Below'

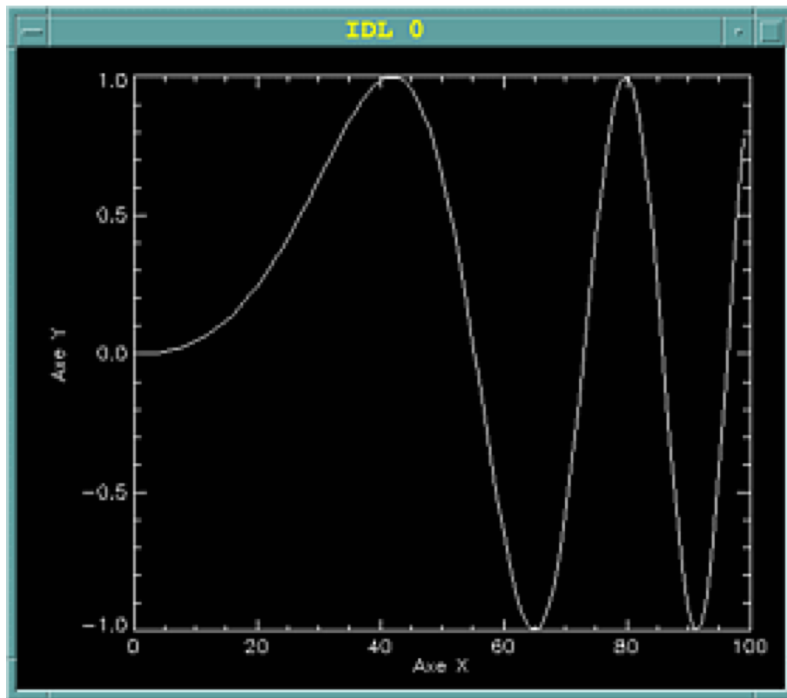
$$\int_p^x \rho_i U_i^2 dx$$

Hershey: font = -1

XYOUTS, 0, .2, '!M!S!A!E!8x!R!B!lp!N !7q!!i!N!8U!S!E2!R!!i!Ndx', \$  
SIZE = 3, /NORMAL

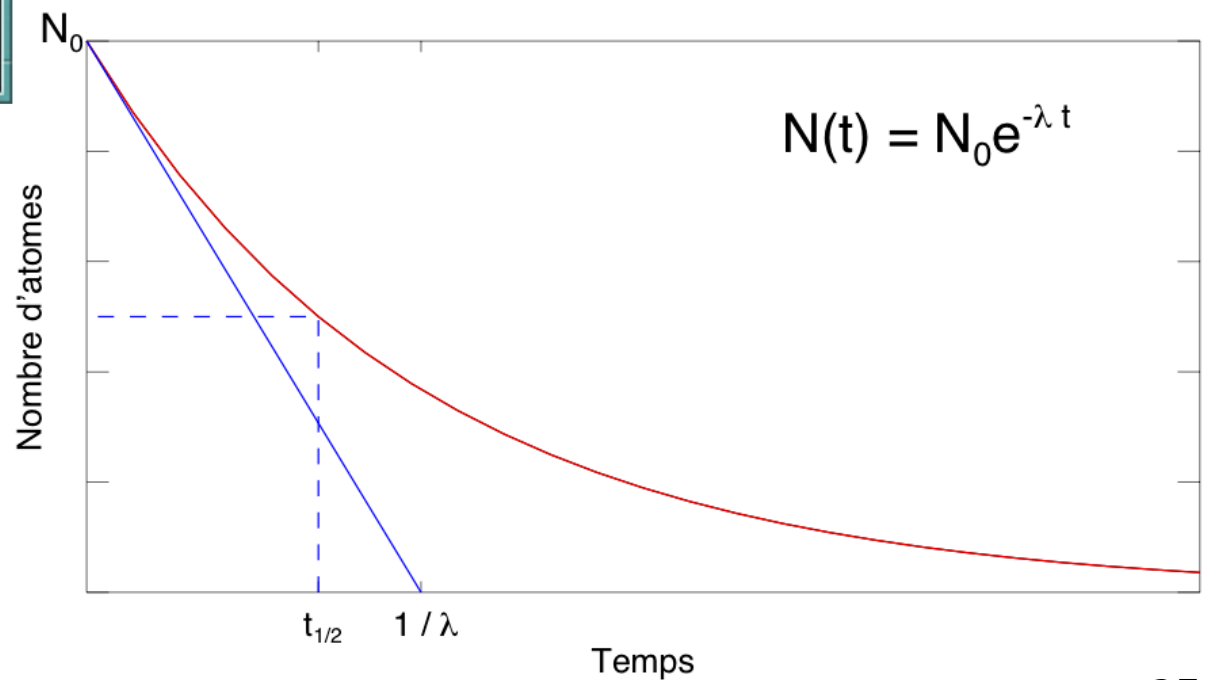


# Graphiques



Version immédiate, toute moche mais pratique

Version plus présentable, avec un peu de travail



# Affichages

## Images

Mode couleur standard : 256 couleurs indicées (niveaux de gris ou couleurs arbitraires)

`tv, ima` ⇒ Affiche les niveaux tels quels (valeurs entières arrondies)

`tvsc1, ima` ⇒ Rééchelonne d'abord entre 0 et 255

Affichage dans le coin bas / gauche par défaut

Ligne 0 en bas

Table de couleur par défaut : 256 gris de plus en plus clairs

`tv, ima, 0`

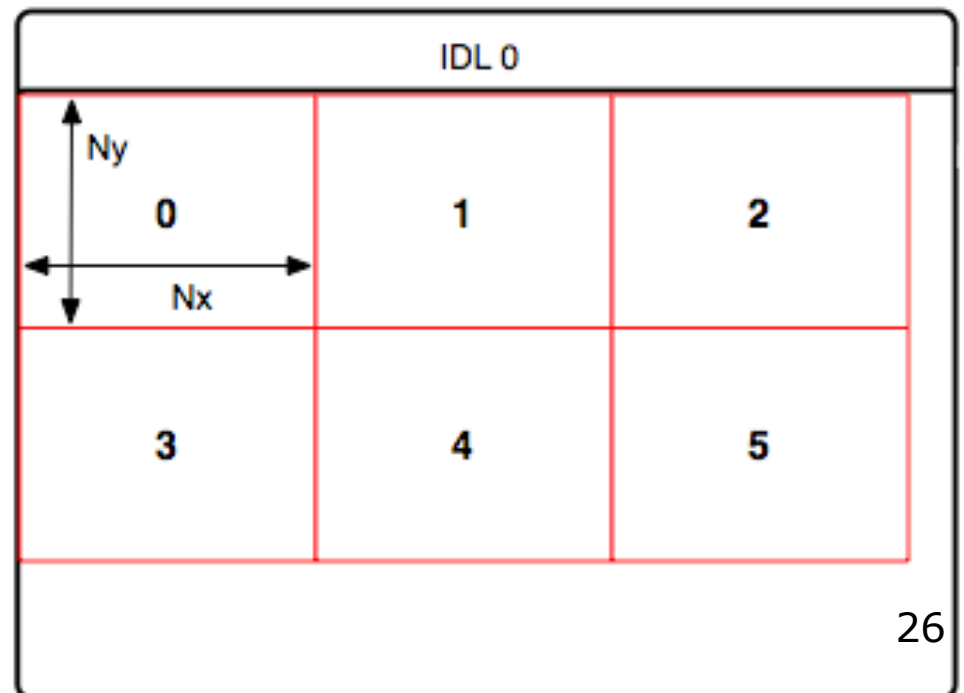
⇒ affichage en position 0 (haut / gauche)

`loadct, N`

Sélectionne une table de couleurs prédéfinie

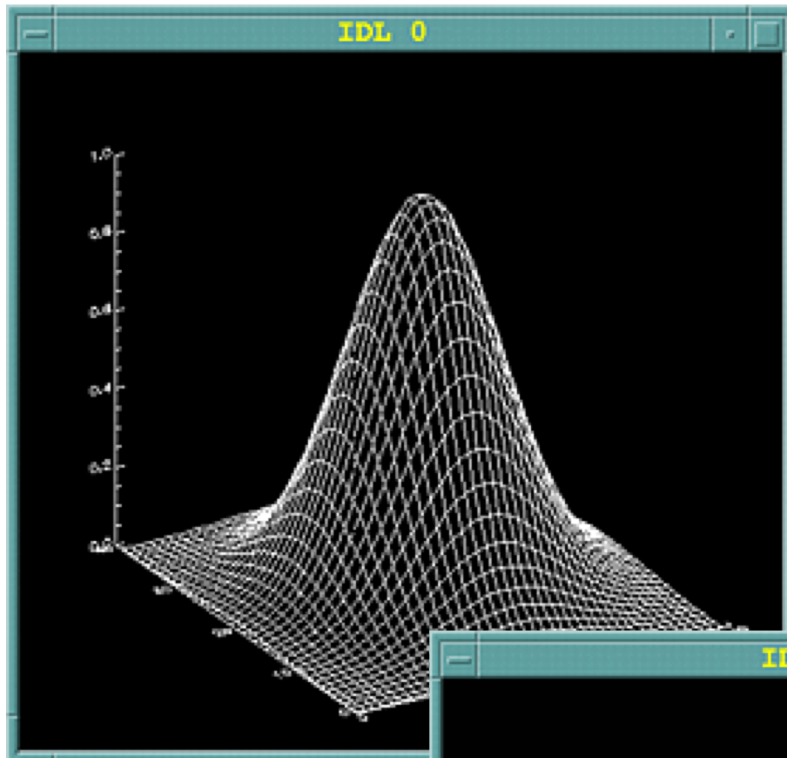
$N = 0$  à 40

(0 ⇒ 256 gris)

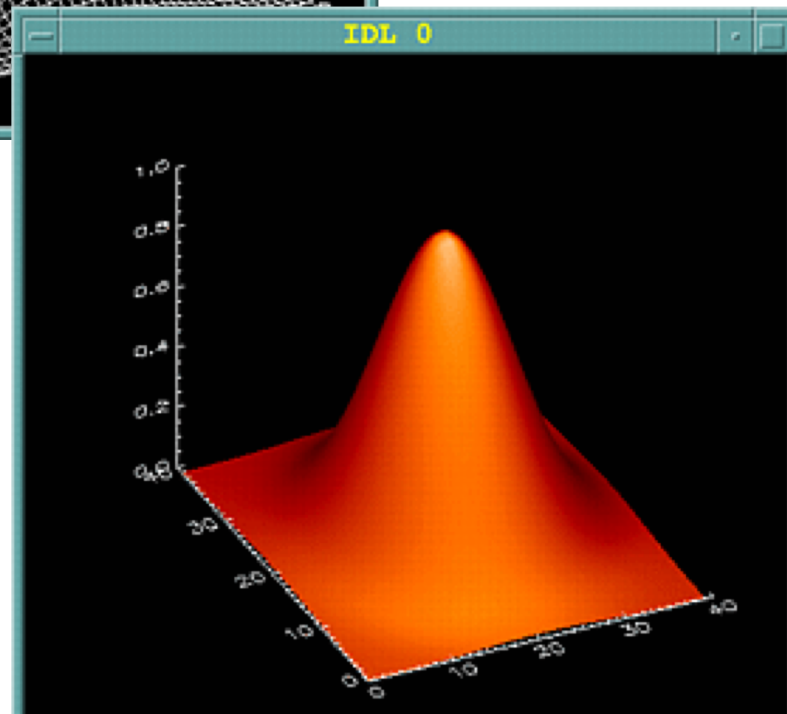




# Graphiques / Images

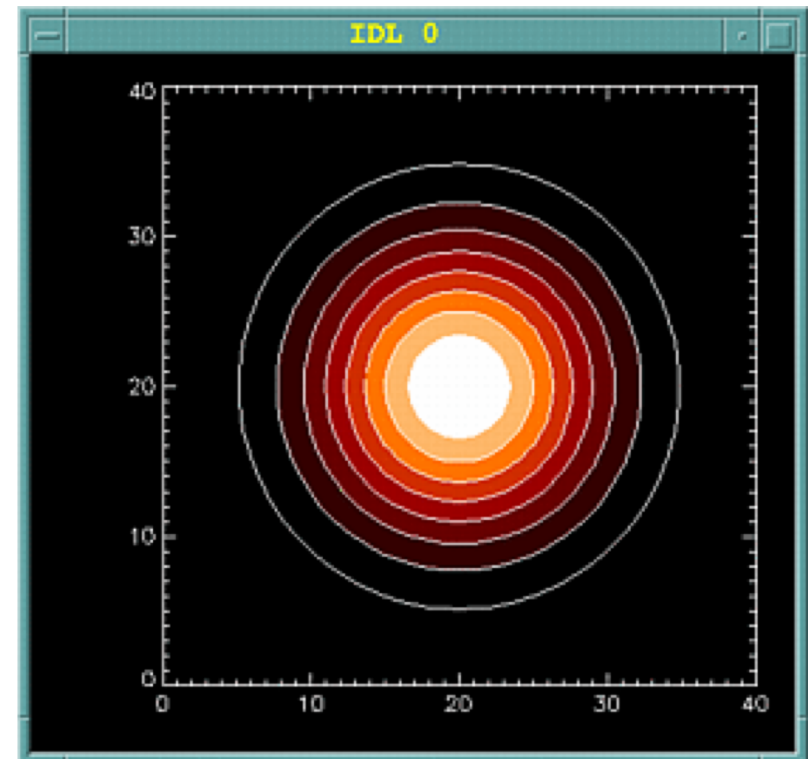


Surface



Shade\_surf

Contour



# Affichages

---

## Fenêtrage

**Pilote graphique par défaut = celui du terminal (Xwindow ou Windows)**

**Ouvre une nouvelle fenêtre**

(dimensions en pixels optionnelles, ID = [0,31])

window, xs=Nx, ys=Ny, ID

**Gère automatiquement les n° de fenêtre**

(=> ID = [32,127])

window, xs=Nx, ys=Ny, /free

**Met la fenêtre ID au premier plan**

wshow, ID

**Sélectionne la fenêtre ID pour affichage**

wset, ID

**Efface le contenu**

erase

**Ferme une fenêtre (contenu perdu)**

wdelete, ID (ou avec souris)

## Précautions en début de session

**Gestion des contenus par le système  
+ utilise les tables de couleur indexées**

device, retain=2, decompose=0



# Installation / démarrage

---

- IDL et GDL utilisent**
- un noyau de routines précompilées (écrites en C)  
plot print polyfillV convol FFT...
  - des routines écrites en IDL et compilées dynamiquement  
image\_cont read\_image butterworth...
  - des bibliothèques de routines externes qu'on peut ajouter  
readfits centroid + affichages divers

## Bibliothèques principales

ASTRON (Nasa), JHUAPL, ATV... (ne marchent pas toutes sous GDL)

## Installation

- Copier dans un répertoire accessible (~/.IDL/userlib/ ou dans le système)
- Ajouter ce répertoire dans la variable !path  
defsysv, '!LIB\_DIR', "~/.IDL/userlib"  
!PATH = !PATH + Path\_sep(/search) + EXPAND\_PATH('+'+!LIB\_DIR)

## Configurations de démarrage

On peut définir un fichier script IDL qui sera exécuté à chaque démarrage  
=> définition du !path, de variables personnelles, préférences graphiques...

# Ecriture des routines

---

Suite de commandes structurées effectuant une tâche.

## Procédures

Pro libelulle, <liste de paramètres>

<instructions>

end

gdl> libelulle, <parametres>

## Fonctions

Function papillon, <liste de paramètres>

<instructions>

return, <resultat>

end

gdl> result = papillon(<parametres>)

## Compilation

Traduit le programme en code

Compiler (à faire après chaque modification)

=> doit être dans le !path ou dans le répertoire courant

Exécuter (s'utilise comme une routine IDL normale)

gdl> .r libelulle

gdl> cd, '<directory>'

<appel>



# Écriture des routines

---

## Correspondance entre syntaxes d'appel et d'écriture

### Appel

### Routine

#### Arguments: identifiés par position

```
libelulle, param1, param2  
print, param1, param2
```

```
Pro libelulle, arg1, arg2  
  arg1 = 12.  
  arg2 = 24.  
end
```

#### Mots-clefs: identifiés par leur nom Correspond à une variable dans la routine

```
libelulle, etiquette = parametre  
print, parametre
```

```
Pro libelulle, etiquette = Variable_motclef1  
  Variable_motclef1 = 12.  
end
```

#### On utilise généralement le même nom pour le mot-clef et la variable

```
Pro libelulle, motclef1 = motclef1  
  motclef1 = 12.  
end
```

# Écriture des routines

---

## Paramètres de sortie (procédures et fonctions)

### Appel

### Routine

**Arguments: tous sont modifiés dans la routine**  
On récupère les nouvelles valeurs en sortie  
ssi les paramètres d'appel sont des variables

```
param1 = 1  
param2 = 10  
libelulle, param1, param2  
print, param1, param2
```

```
Pro libelulle, arg1, arg2  
arg1 = arg1 + 12.  
arg2 = arg2 + arg1  
end
```

On peut utiliser des constantes à l'appel  
mais les valeurs calculées dans la  
routine ne sont pas transmises

```
libelulle, 23, 12  
?
```

# Ecriture des routines

---

## Test des paramètres d'appel

### Combien d'arguments ?

(sans compter les mots-clefs)

N\_params()

### Mot-clef actif ?

= 1 si initialisé à une valeur ≠ 0

keyword\_set(<motclef>)

### Mot-clef présent à l'appel ?

= 0 si absent, nb d'elt si présent

N\_elements(<valeur\_motclef>)

### Dimensions des arguments

size(arg)

## Passage d'arguments en cascade

Tartufo, a, b, thick=5, color=3, /cause

```
PRO tartufo, a, b, _extra=pipo, color=c, cause=cause
if keyword_set(cause) then print, 'blah blah blah'
plot, a, b, _extra=pipo, color=c+10
; Color est intercepté puis transmis explicitement,
; Cause est réservé à la routine
; les autres sont stockés dans _extra et transmises
end
```

# Ecriture des routines

## Mise au point

Ecrire, compiler, lancer

```
.r libelulle  
libelulle, param1, param2  
% Compiled module: DIST.
```

## Si ça plante...

=> Lire le message d'erreur

```
% Variable is undefined: N.  
% Error occurred at: DIST 59 /Applications/itt/idl708/lib/dist.pro  
% $MAIN$  
% Execution halted at: $MAIN$
```

message d'erreur

routine et ligne de code où s'est produit l'erreur

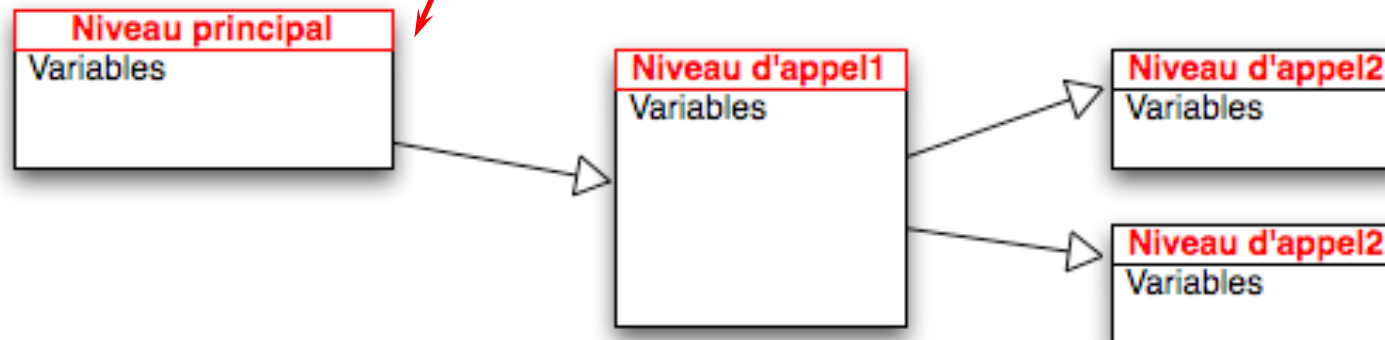
Liste toutes les variables  
du niveau actif !

niveau de routine où on est bloqué

```
help  
help, arg
```

Revenir au niveau principal

```
retall
```



# Mise au point des routines: checklist

---

## Mise au point

Ecrire, compiler, lancer

```
.r libelulle  
libelulle, param1, param2  
% Compiled module: DIST.
```

## Si ça plante...

=> Lire le message d'erreur

- Vérifier les variables, les expressions types et valeurs, dimensions: help, print, plot, tv...
- Revenir au niveau principal si besoin
- Corriger dans un éditeur + sauver
- Recompiler
- Relancer



**Vérifier le résultat !**

# Instructions de structure

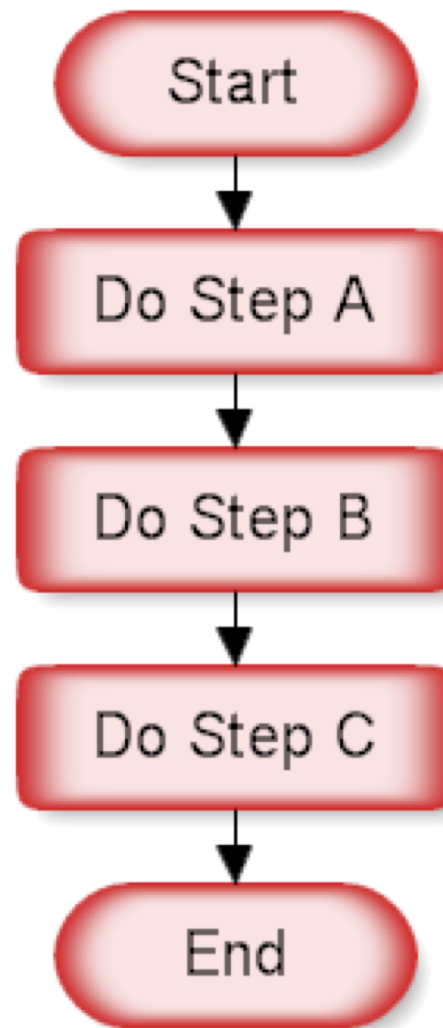
---

Les instructions permettent de procéder à des traitements selon la valeur de certaines variables. Elles déterminent le déroulement d'un programme.

## Déroulement linéaire

C'est le déroulement typique d'une session interactive

IDL utilise des instructions standard pour modifier ce déroulement



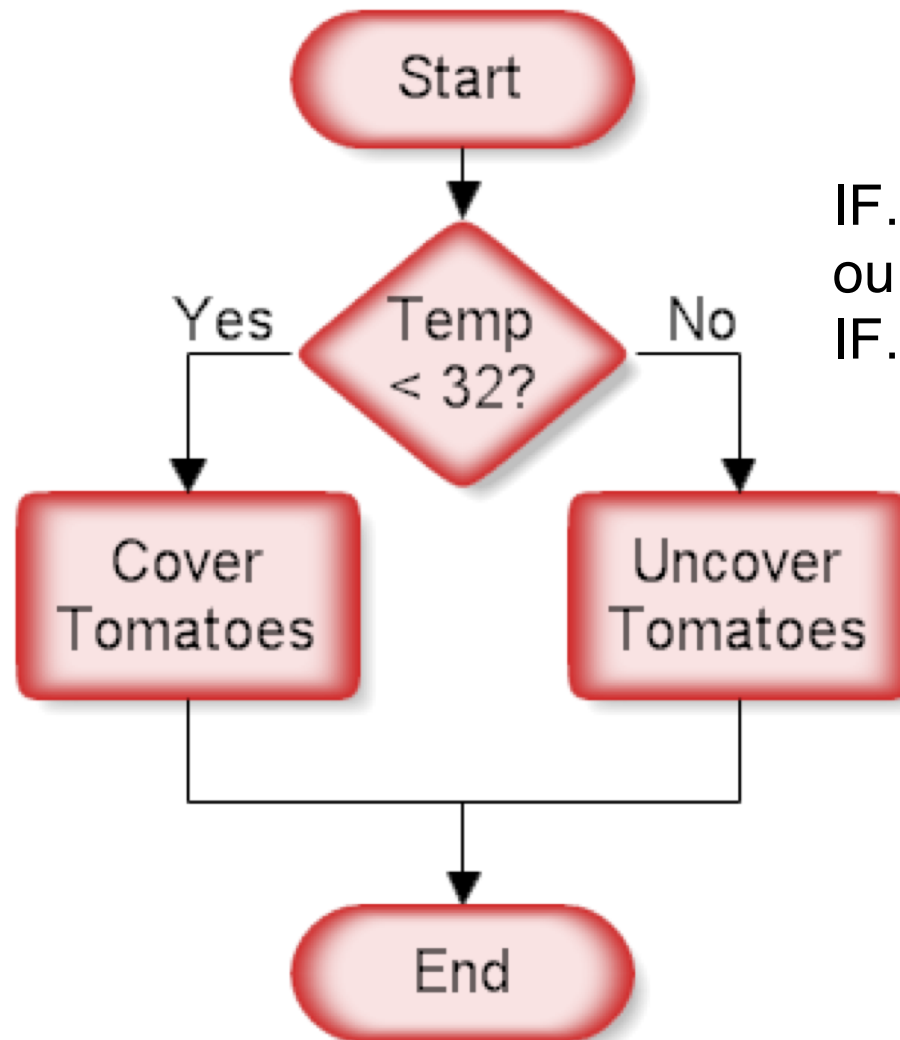
# Instructions de structure

---

On peut vouloir effectuer un traitement différent selon les conditions

On teste alors une condition logique, et on exécute des séries d'instructions différentes

## Tests



IF... THEN...

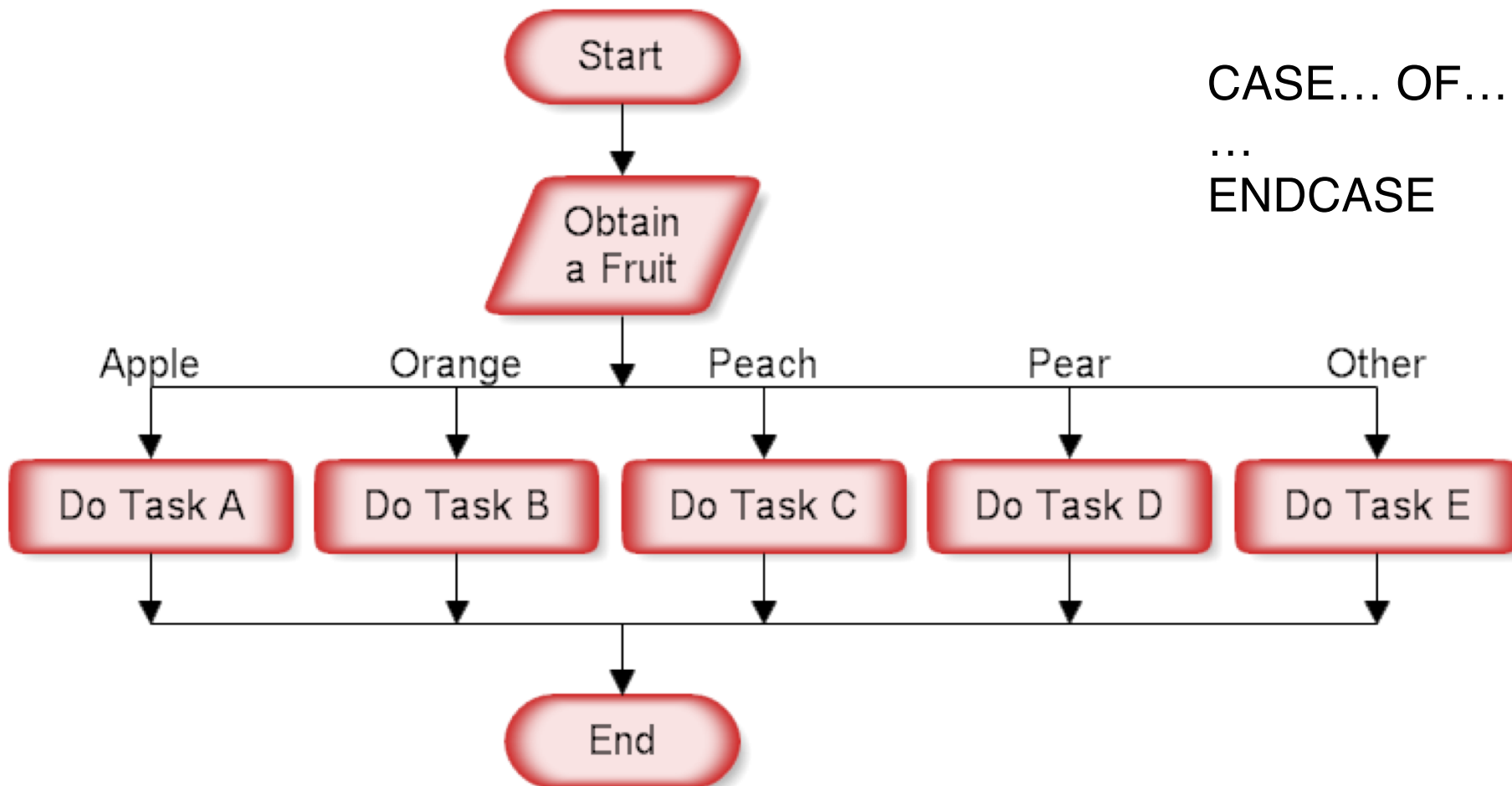
ou

IF... THEN... ELSE...

# Instructions de structure

On peut vouloir effectuer plusieurs traitements différents selon une condition  
=> cascade de IF (difficile à relire)

## Tests multivariés



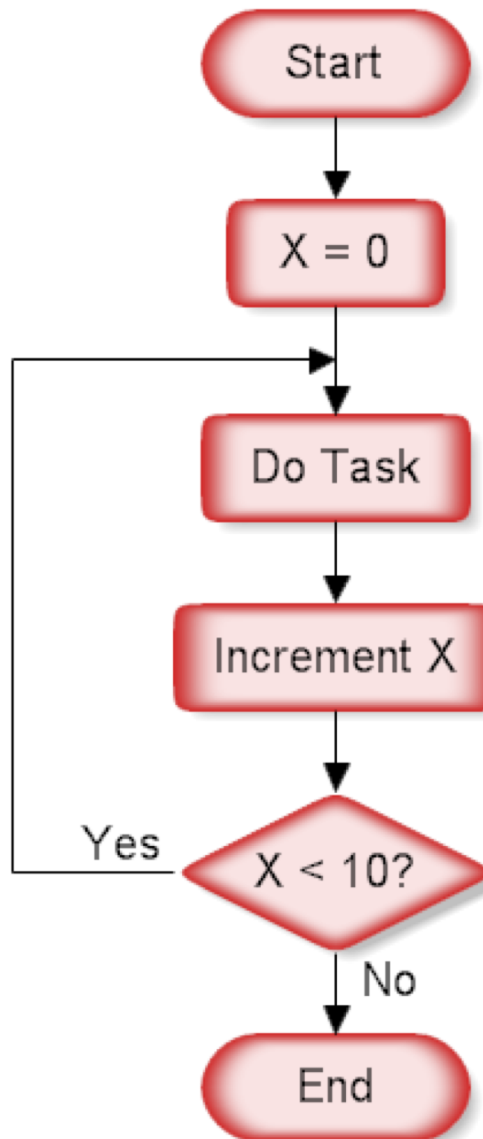


# Instructions de structure

On peut vouloir effectuer un traitement un certain nombre de fois, éventuellement sur des variables différentes

⇒ Utilise un compteur, géré automatiquement

## Boucles



FOR... DO

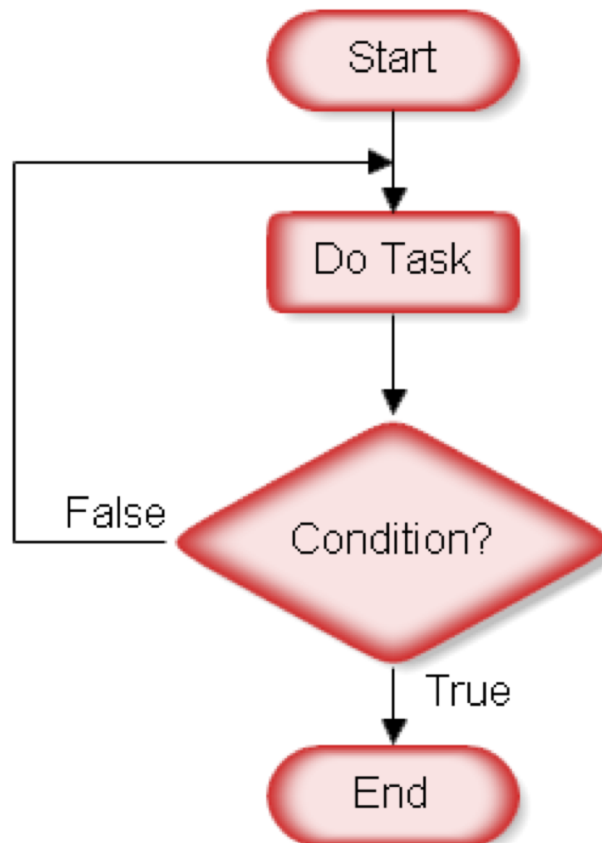
# Instructions de structure

Le compteur peut être remplacé par une condition logique, évaluée avant ou après le traitement — pas de compteur dans ce cas

## Boucles conditionnelles

### Repeat Until Loop

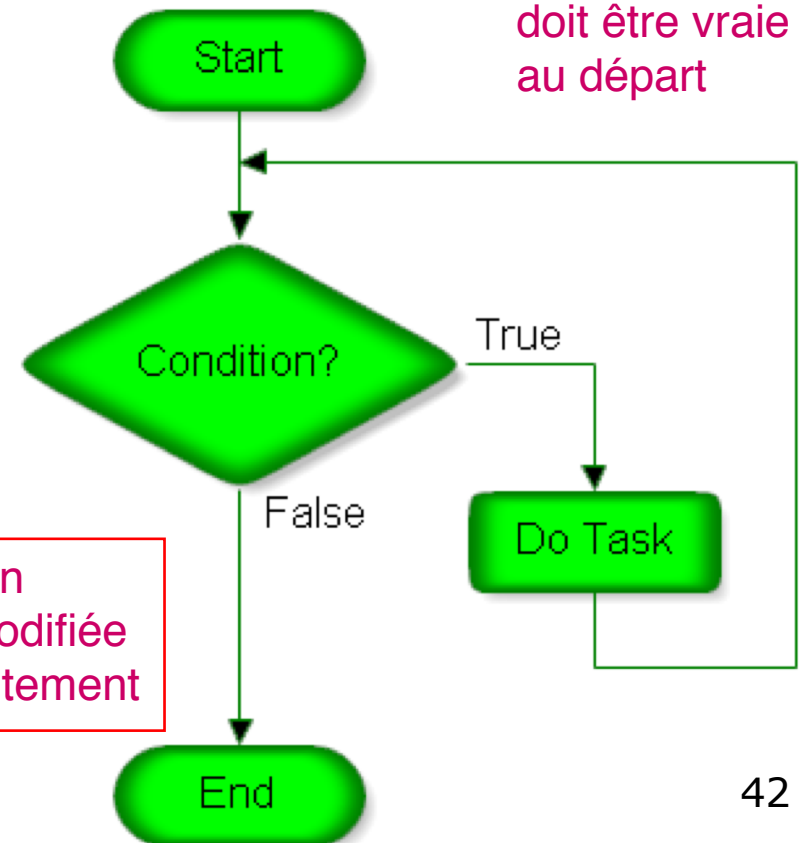
Exécutée au moins une fois



La condition doit être modifiée dans le traitement

### WHILE... DO Loop

La condition doit être vraie au départ



## Instructions de structure

---

Les instructions permettent de procéder à des traitements selon la valeurs de certaines variables. Elles déterminent le déroulement d'un programme.

Type	Exemples	Commentaires
boucles	FOR... DO... WHILE... DO... REPEAT... UNTIL...	A ne pas utiliser pour traiter des éléments de tableau ! (en général, utiliser where)
tests	IF... THEN... IF... THEN... ELSE	
branchements	CASE... OF SWITCH... OF	Simplifient les séries de tests complexes
sauts, ruptures de traitement	GOTO BREAK CONTINUE	A éviter...

# Instructions de structure, syntaxe

## Sur une seule ligne (avec une seule instruction)

```
FOR i = iDeb, iFin DO print, i
```

**Boucle**

```
IF a GT 10 THEN print, a
```

**Test**

**Seules disponibles  
en mode interactif**

## Plusieurs instructions / plusieurs lignes

```
FOR i = iDeb, iFin DO BEGIN  
  print, i  
  print, 2*i  
ENDFOR
```

**Bloc d'instruction  
BEGIN**

...  
**END<type>** (ou END)

```
IF a GT 10 THEN BEGIN  
  print, a  
  print, a/2  
ENDIF
```

```
IF a GT 10 THEN BEGIN  
  print, a  
ENDIF ELSE BEGIN  
  print, a/2  
ENDELSE
```

```
CASE <expression> OF BEGIN  
<result1>: <instruction1>  
<result3>: <instruction2>  
ELSE: <instruction3>  
ENDCASE
```

# Lecture/écriture de fichiers

---

## Formats d'image standards (gif, tiff, jpeg, png...)

Cette routine gère tous les formats courants:

```
ima = read_image('fichier.ext')
```

Le fichier doit être dans le répertoire courant  
ou on doit donner le chemin du fichier  
**avec la syntaxe système**

```
cd, '~/machin/data'
```

```
ima = read_image('~/machin/data/fichier.ext')
```

On écrit un fichier en précisant le format

```
write_image, 'fichier.ext', 'PNG', ima
```

Dans les deux cas on peut inclure/relire  
la table de couleurs des images 8 bits

```
ima = read_image('fichier.ext', R, G, B)  
tvlct, r, g, b
```

Ecrire d'un chemin indépendamment de l'OS

```
chemin = filepath(nomF,root='bla',sub=['a','b'])
```

## Format d'archive IDL (XDR)

Binaire indépendant de la plateforme  
sauve des variables avec leurs noms

```
restore, 'fichier.ext'  
save, file = 'fichier.ext', <liste de variables>
```

# Lecture/écriture de fichiers

## Format FITS (images astro)

Dans la bibliothèque ASTRON

ima = readfits('fichier.fits')  
writefits, 'fichier.fits', ima

The screenshot shows the SAOImage ds9 interface. On the left, a control panel for 'J13gd.fits' includes fields for 'Ceres14\_J', 'X', 'Y', 'X', 'Y', 'Zoom' (1.000), and 'Angle' (0.000). Below these are buttons for 'affichage', 'fenêtre', 'bin', 'zoom', 'scale', 'couleur', 'ouvrir', 'save image', 'entête', and 'configurer la palette'. The main window displays a circular astronomical image of Ceres. Overlaid on this is a terminal window showing the FITS header for 'J13gd.fits'.

```
SIMPLE = T / Written by IDL: Sat May 20 14:44:22 2006
BITPIX = -32 / # of bits per pix value
NAXIS = 2 / # of axes in data array
NAXIS1 = 512 / Number of positions along axis 1
NAXIS2 = 512 / Number of positions along axis 2
ORIGIN = 'ESO-PARANAL' / European Southern Observatory
DATE = '2004-01-14T02:56:51.3012' / Date this file was written
EXPTIME = 0.400 / Integration time
MJD-OBS = 53018.12245533 / Obs start
DATE-OBS = '2004-01-14T02:56:20.1404' / Observing date
ORIGFILE = 'NACO_0160.fits' / Original File Name
INSTRUME = 'NAOS+CONICA' / Instrument used
TELESCOP = 'ESO-VLT-U4' / ESO <TEL>
RA = 110.210352 / 07:20:50.4 RA (J2000) pointing
DEC = 30.56260 / 30:33:45.3 DEC (J2000) pointing
EQUINOX = 2000. / Standard FK5
RADECSYS = 'FK5' / FK5
LST = 20783.213 / 05:46:23.213 LST
UTC = 10577.000 / 02:56:17.000 UTC
OBSERVER = 'FORNI' / Name of observer
PI-COI = 'ERARD' / Name(s) of proposer(s)
ALARM = '' / Active alarm(s), if any.
OBJECT = 'Ceres14_J' / Target description
AIRMASS = 1.97200 / Averaged air mass
CRVAL1 = 110.21035 / Coordinate at reference pixel in <axis
CRVAL2 = 30.56260 / Coordinate at reference pixel in <axis
CRPIX1 = 548.0 / Ref pixel in <axis direction>
CRPIX2 = 509.9 / Ref pixel in <axis direction>
CDELTA1 = -0.000003683 / Increment in <axis direction>
CDELTA2 = 0.000003683 / Increment in <axis direction>
CTYPE1 = 'RA---TAN' / Coordinate system of <axis direction>
CTYPE2 = 'DEC--TAN' / Coordinate system of <axis direction>
CD1_1 = -3.68333E-06 / Translation matrix element
CD1_2 = 0.00000E+00 / Translation matrix element
CD2_1 = 0.00000E+00 / Translation matrix element
CD2_2 = 3.68333E-06 / Translation matrix element
EXTEND = T / FITS Extension may be present
HIERARCH ESO OBS DID = 'ESO-VLT-DIC.OBS-1.7' / OBS Dictionary
HIERARCH ESO OBS OBSERVER = 'UNKNOWN' / Observer Name
HIERARCH ESO OBS EXECUTE = 1084 / Expected execution time
```

**En-tête (ascii)**  
**mot-clef = valeur**  
...  
**Données1 (binaires)**  
&@aptart!ç!àà#1#ùm^  
**Données2 (binaires)**  
&@aptart!ç!àà#1#ùm^

# Lecture/écriture de fichiers

---

## Format FITS (images astro)

### Lecture de l'en-tête avec les données

=> tableau de chaînes, on peut chercher dedans

```
ima = readfits('fichier.fits', header)
```

### Extraction de la valeur correspondant à un mot-clef

```
tp_integ=sxpar(header, 'EXPTIME')
```

### Ajout d'un mot-clef et d'une valeur associée

dans une variable

```
sxaddpar, header, 'COMMENT2', com2
```

### Écriture d'un fichier et de son en-tête

(les premiers mots-clefs sont obligatoires dans le standard FITS, mais la routine ne vérifie pas l'intégrité de l'en-tête)

```
writefits, 'fichier2.fits', data, header
```

# Lecture/écriture de fichiers

---

## Ouvrir un fichier

### Associe un fichier à une unité logique

openr / openw / openu: lecture / écriture / les deux (update)

/get\_lun: gère automatiquement l'unité logique

/comp : compresse/décompresse (gzip)

openr, unit, 'fichier.ext', /get\_lun

## Lire/écrire le contenu

### fichiers binaires

readu, unit, <liste de variable>

writeu, unit, <liste de variable>

### fichiers ascii

printf, unit, <liste de variable>

readf, unit, <liste de variable>

### Formats explicites:

#### ceux du fortran et du C

printf, unit, format='("incidence : ",F6.2,"° ")',inc

## Fermer le fichier

+ libérer l'unité logique

close, unit

Free\_lun, unit



# Impression de graphiques

---

**De loin la solution la plus simple:**

**Fonctions de copie d'écran dans la bibliothèque JHUAPL**

Fournir le nom de fichier

+ éventuellement les paramètres du format

pngscreen, 'fichier.png'  
jpegscreen, 'fichier.jpeg'  
gifscreen, 'fichier.gif'  
tiffscreen, 'fichier.tiff'

**=> stockage bitmap (pixélisé)**

**Solution élégante et propre : PostScript**

**=> stockage vectorisé, redimensionnable, avec polices propres et lisibles**  
(pour publications)

# Graphiques PostScript

---

## PS géré comme un pilote graphique, pas comme un type de fichiers

Sélectionne le pilote PS

Détermine le nom de fichier

pour une version 256 couleurs:

```
set_plot, 'ps'
```

```
device, filename='fichier.ps'
```

```
device, filename='fichier.ps', /color, bits=8
```

*(instructions graphiques habituelles)*

Ferme le fichier

Retour au pilote X11

```
device, /close
```

```
set_plot, 'X'
```

## Différence majeure avec le pilote X11

10000 pixels/cm => images en taille réduite,  
...mais pixels redimensionnables

```
tv, ima, xsize=XX, ysize=YY
```

Fastidieux pour les images...

Voir routines PSON / PSOFF dans biblio PIP

**Problème possible avec certaines éditions de GDL** : utiliser le pilote svg à la place de ps 50

# Vade mecum

---

## Routines/syntaxe

? <fonction> ( #<fonction> sous GDL )

## Variables

help  
print

## Graphiques

plot + oplot  
contour (2D)  
surface (2D)  
tv + tvscl (2D ou 3D)

## Tableaux

where + adressage par liste d'indice  
=> jamais de boucle sur des elt de tableau

## Contrôle d'exécution

.compile / .com  
.run / .r  
.continue / .c  
return (-1 niveau)  
retall (vers niveau principal)

## Routines

- faire un en-tête décrivant le fonctionnement
- mettre des commentaires (utiles) !

**Toujours avoir un regard critique sur le résultat :  
est-ce crédible ? est-ce ce qu'on attend ? est-ce que ça fait ce qu'on veut ?**

## Ce qu'IDL a de plus (que GDL)

---

### **Mode « fonctions graphiques »**

Graphiques vectoriels sur écran, manipulables à la souris, depuis la version 8

### **Programmation objets**

Lourd, mais fonctions puissantes et rapides, bibliothèque très étendue

### **Compatibilité python**

Appels IDL ↔ python, Jupyter notebooks ([ex](#))

### **Création d'interfaces utilisateur (widget)**

Pas trop pratique, sert surtout pour développer des outils partagés

### **Editeur intégré**

On peut vivre sans pour le traitement scientifique, mais pratique pour des développements importants