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. Aboudarham http://ufe.obspm.fr/offreFP/coursIDL/

J. M. Malherbe (cours M1, sur site UFE ?)

En ligne aussi (avec beaucoup d'autres):

Tutoriels IDL http://www.exelisvis.com/docs/using_idl_home.html
(très orientés vers les nouveautés, pas le plus efficace)

R. O'Connell http://www.astro.virginia.edu/class/oconnell/astr511/IDLguide.html
(orienté astro)

IDRIS http://www.idris.fr/ada/ada-idl-doc.html
(en français)

IDL astronomy user's library (NASA)

http://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 <=> langage interprété
  A la réputation d'être lent côté calcul (plus très vrai)

- **Alternatives :**
  Matlab (calcul + visu)
  C, C++, fortran (calculs rapides)
  **** 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 / basic / Pascal…
  Syntaxe très condensée, les calculs sont vectorisés:
  - Toutes les variables se manipulent comme des scalaires
  - On fait le moins de boucles possibles

- **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 quelques limitations

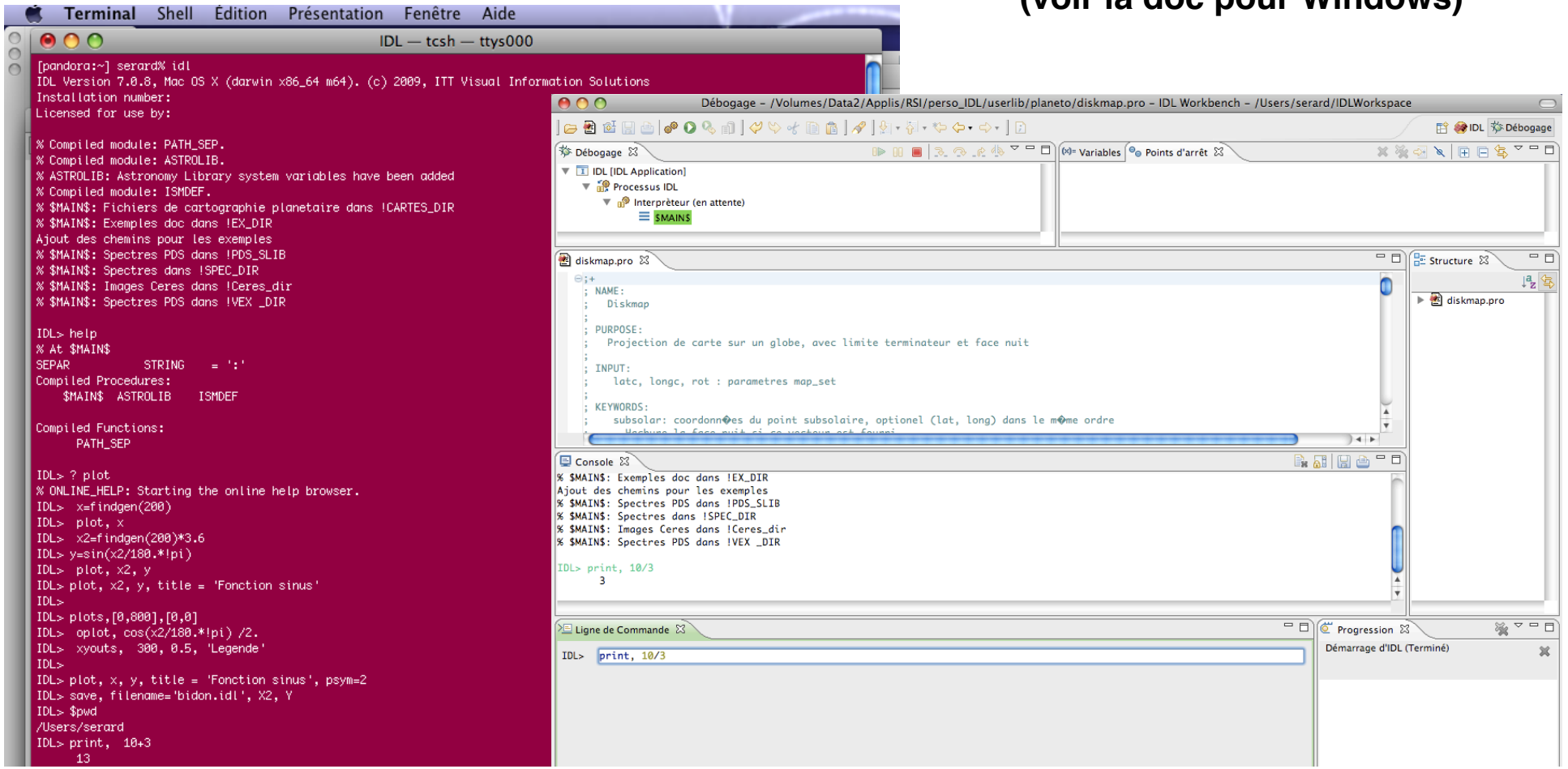
Lancement

Deux solutions sous IDL:

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

Sous GDL:

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



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

Eléments de langage

Routines

Demandent d'effectuer une action: **procédures : font quelque chose**
fonctions : retournent un résultat

On peut écrire les siennes ou installer des bibliothèques utilisateur

Commandes

Structurent le déroulement du programme, par ex: **boucles**
tests
branchements…

Instructions de contrôle

Contrôlent l'exécution des routines (pas dans une routine)
Commencent toutes par . **.run** **+ retall, help, etc**
.comp
.cont

Ecriture

Insensible à la casse sauf dans les noms de fichiers !
Caractères spéciaux : **& chaîne deux instructions sur la même ligne**
$ caractère de continuation en fin de ligne
; commentaire

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 X1sdfsdfrer45668o000

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:

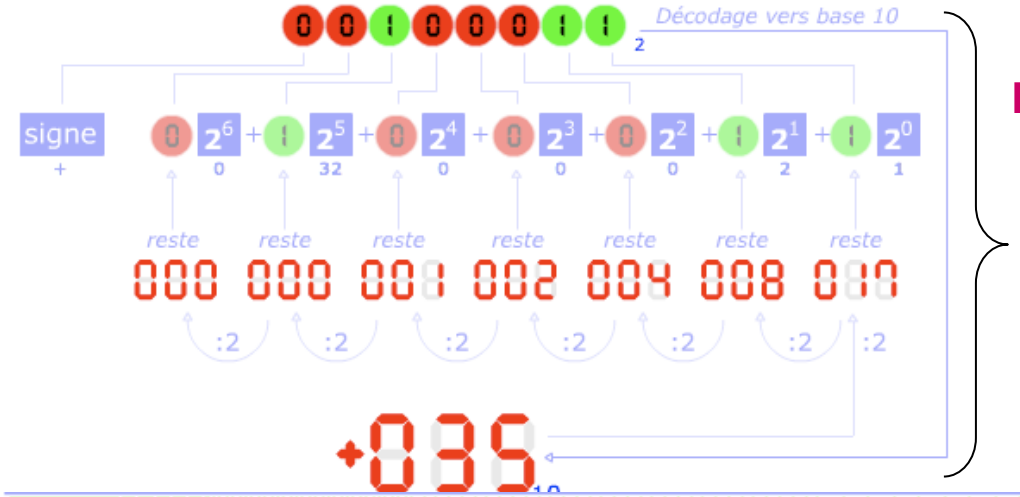
On évalue l'expression et on la stocke dans la variable:

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



Entiers : stockés en binaire, compléments à 2

Réels : stockés en virgule flottante, exposant / mantisse

http://www.arcanapercipio.com/lessons/codage_binaire_des_nombres/codage_binaire_des_nombres.html

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 (colonne!)	[3 , 2]	$\begin{pmatrix} 3 \\ 2 \end{pmatrix}$
vecteurs-lignes	transpose([3,2]) [[3] , [2]]	$(3, 2)$
Tableaux	[[2,4] , [3,2]]	$\begin{pmatrix} 2 & 3 \\ 4 & 2 \end{pmatrix}$

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 = fltarr(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' + chaine1

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 :

±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 indicer un autre tableau

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

On peut indicer 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 = fltarr(N,M)

a = 3
vecteur = [2.,4.,6.,2.,4.,10.]
help, a * vecteur

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

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) **Attention à la transposition!**
Erreur si N1 ≠ M2

[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)	**=> min, max, moyenne**
mean(TAB)	
stddev(TAB)	**=> renvoie l'écart-type**
tvscl, TAB > 10	**=> le seuillage est très utile pour afficher des images**
tv, bytscl(TAB)	**=> rééchelonne TAB entre 0 et 255 (surtout pour affichage)**
total(TAB)	**=> renvoie la somme des éléments**
total(TAB, 1)	**=> renvoie la somme en colonnes (première dimension)**
total(TAB, 2)	**=> renvoie la somme en ligne (deuxième dimension)**
A * B	**=> produit scalaire**
total(A * A)	**=> module carré**

Manipulation de tableaux

Modification d'images

avec TAB = fltarr(N,M)

transpose(TAB)
shift(TAB,d1, d2) **=> décalage circulaire de d1 et d2 pixels dans chaque direction**
rotate(TAB, N) **=> rotation par pas de 90°, rotation+transposition pour N≥4**
rebin(TAB,N1, M1) **=> redimensionne d'un coefficient entier dans chaque direction (rééchantillonnage)**

rot(TAB,angle) **=> rotation continue**
congrid(TAB, N1,M1) **=> 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) **=> Lissage par moyenne mobile N (impaire)**
median(TAB, N) **=> Lissage par médiane de largeur N (impaire)**
Méthode standard de réduction de bruit

Chaînes de caractères

Valeurs constantes introduites entre " ou ’

```
ch1 = "abcd"
ch2 = 'abcd'
```

Peuvent être utilisées à l’affichage
— seul l’ascii strict peut être affiché au terminal
(pas de caractères spéciaux: é, ç « …)

```
print, 'et'
xyouts, x, y, chaine
```

Seule opération sur les chaînes: concaténation

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

Fonctions sur chaînes: str*

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

S’utilisent souvent en tableaux

```
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 (Herschey / device / TrueType)
mots-clefs de plot,…	charsize = 1.5	charthick = 2	font = -1/0/1
variables système	!P.charsize = 1.5	!P.charthick = 2	!P.font=-1/0/1

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

- !3 : standard
- !4 : grec
- !6 : épais, romain
- !7 : épais, grec
- !8 : italiques

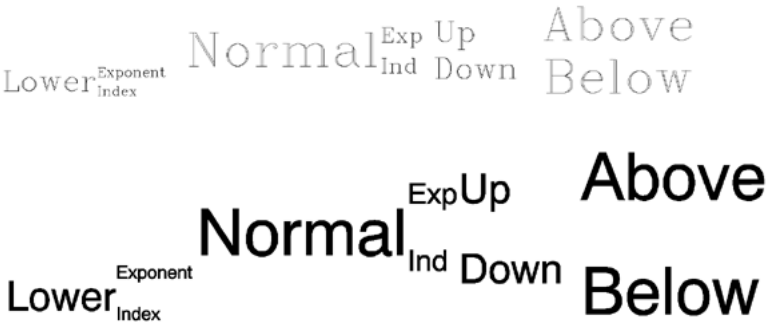
- !I : indice
- !E : exposant
- !N : niveau de base
- !C : à la ligne
- !X : première police
- !9 : math et symboles

Symboles latex: Installer/utiliser la bibliothèque TexToIDL

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

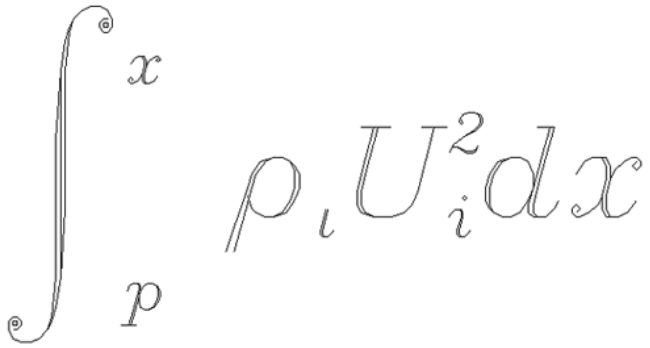
Formatage de texte

Exemples de la doc IDL



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

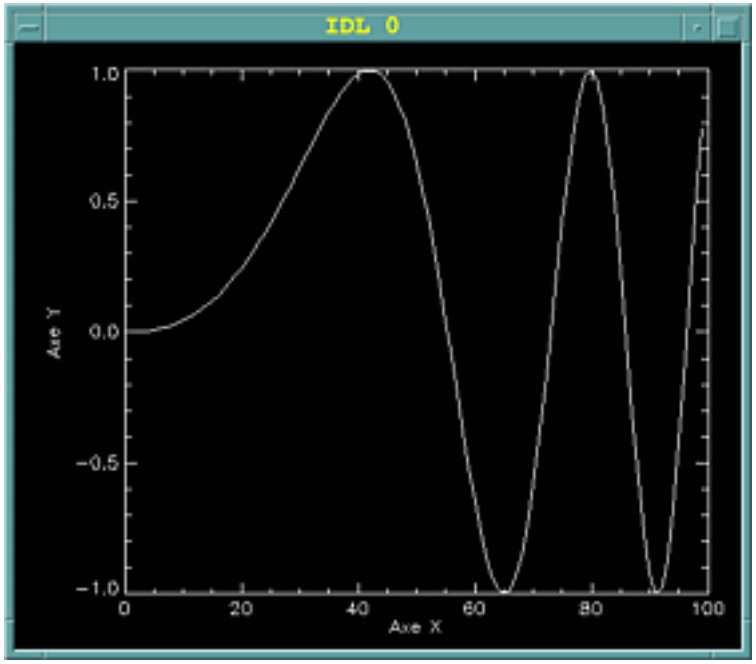
```
XYOUTS, 0.1, 0.3, SIZE=3, '!LLower!S!EExponent!R!IIndex!N Normal' + $
   '!S!EExp!R!IInd!N!S!U Up!R!D Down!N!S!A Above!R!B Below'
```



Hershey: font = -1

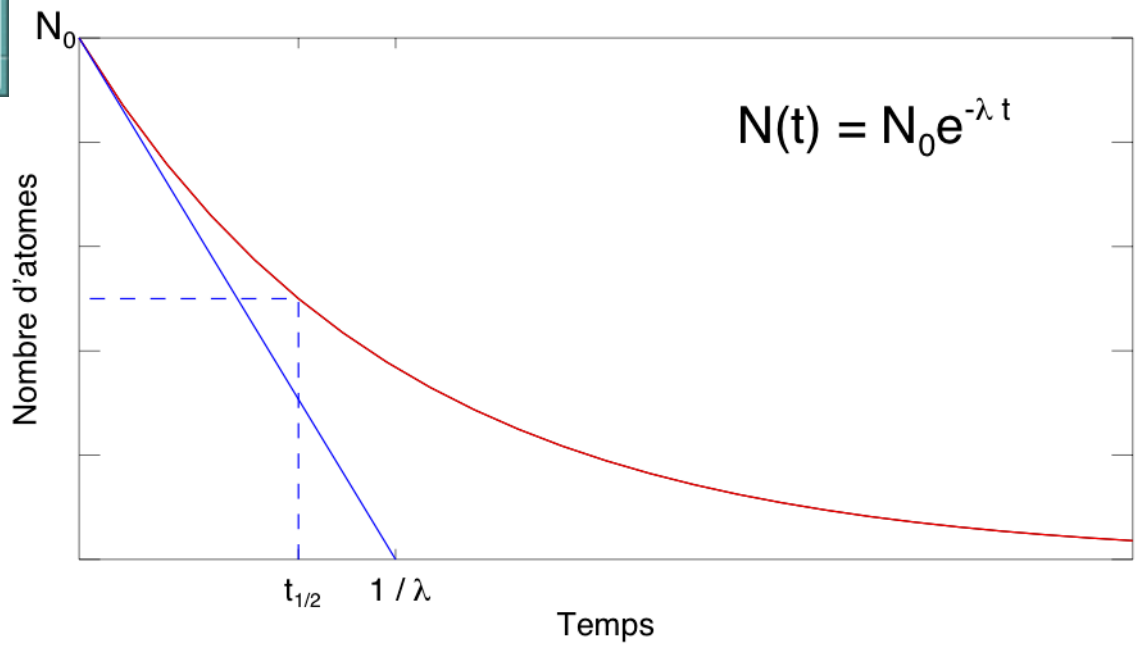
```
XYOUTS, 0, .2, '!MI!S!A!E!8x!R!B!Ip!N !7q!Ii!N!8U!S!E2!R!Ii!Ndx', $
  SIZE = 3, /NORMAL
```

Graphiques



Version immédiate, toute moche mais pratique

Version plus présentable,
avec un peu de travail



Affichages

Images

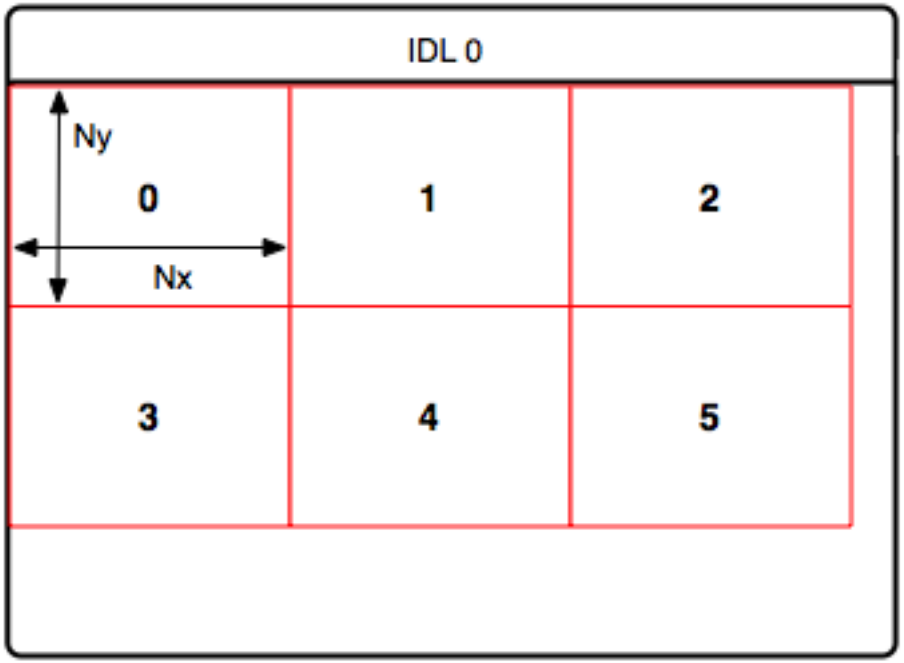
Mode couleur standard : 256 couleurs indicées (niveaux de gris ou couleurs arbitraires)

tv, ima **=> Affiche les niveaux tels quels (valeurs entières arrondies)**
tvscl, 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 prédéfine
N = 0 à 40
(0 => 256 gris)



Affichages

Images, suite

contour, ima	**=> Affiche les isocontours (réglable sur options)**
surface, ima	**=> Affiche en fils de fer (réglable sur options)**
atv, ima	**Module d'examen d'images (similaire à DS9, bibliothèque externe)**

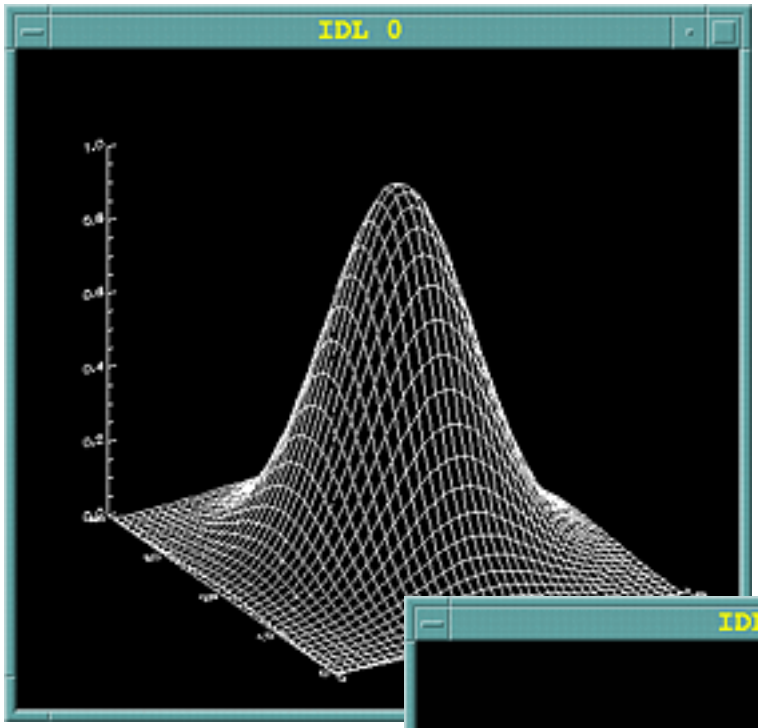
Mode couleur réels : quantités de R,V,B codées sur 256 niveaux chacune
Adapté aux images 24 bits (jpeg…) ou au compositage d'images astronomiques

tv, ima24, true = 1	**affiche une image 24 bits selon le type de stockage (1 à 3)**
tv, ima, channel = 1	**affiche une image 8 bits dans un plan couleur (0 à 3)**

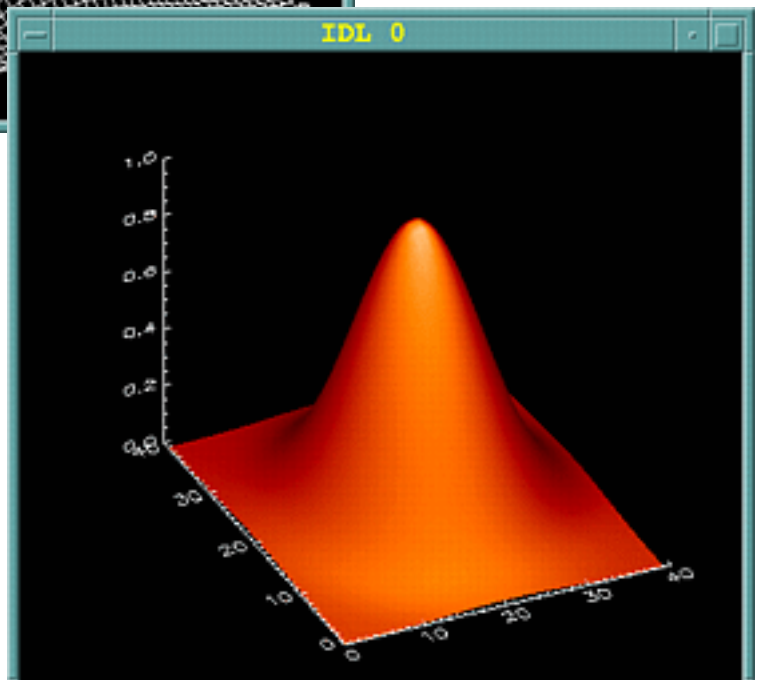
Lecture écran (limitée sous GDL!)

rdpix, A	**affiche au terminal des valeurs pointées à la souris sur une image**
rdplot,A, /print, /full	**affiche au terminal des valeurs pointées à la souris sur un graphique (dans ASTRON)**
profiles, ima	**Affiche des profils de l'image**
ima2 = tvrd()	**Lit le contenu de la fenêtre (pas l'image d'origine!) dans ima2**
b = profile(image)	**Extrait un profil entre deux points**

Graphiques / Images

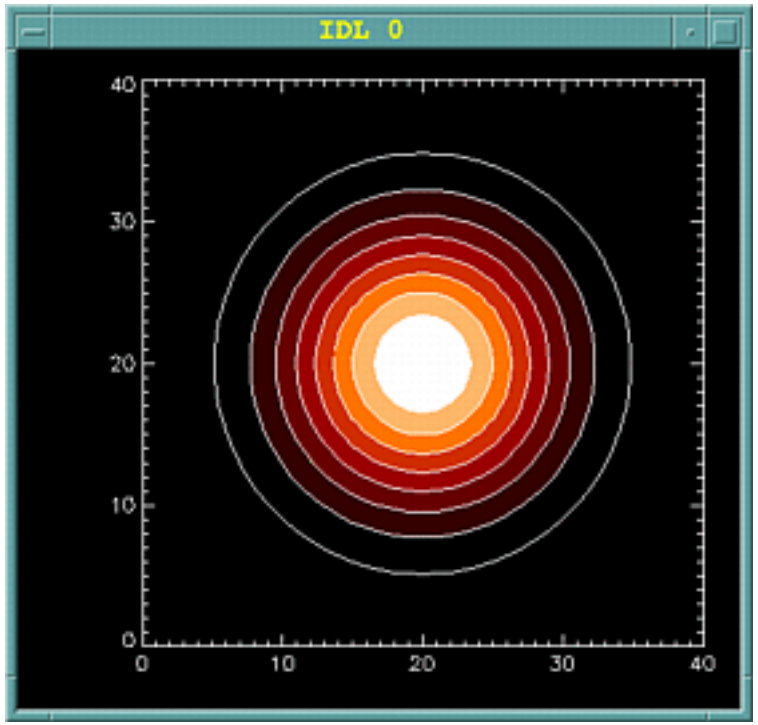


Surface



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>
```

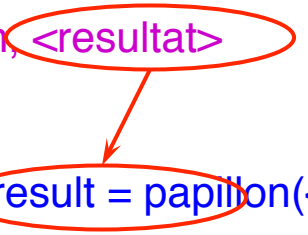
Fonctions

```
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 réperoire courant
Exécuter (s'utilise comme une routine IDL normale)

```
gdl> .r libelulle
gdl> cd, '<directory>'
<appel>
```

Ecriture 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

Ecriture 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

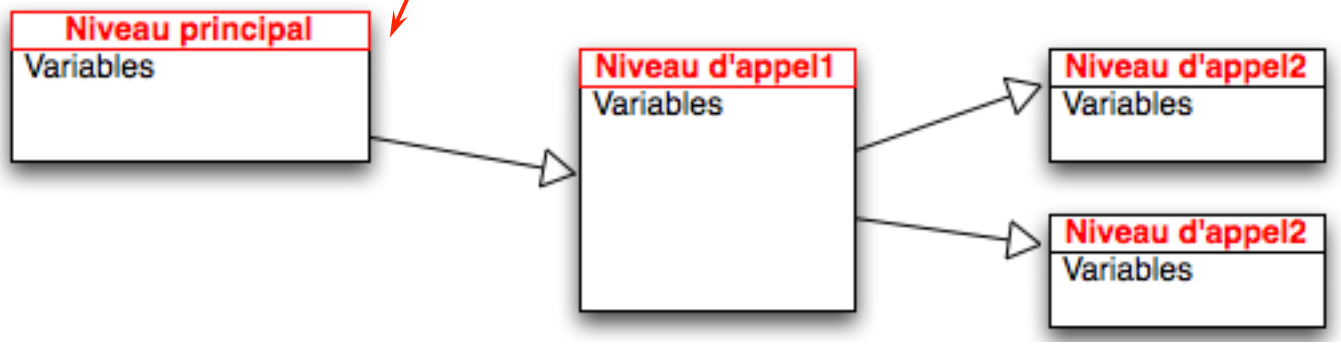
niveau de routine où on est bloqué

Liste toutes les variables
du niveau actif !

help
help, arg

Revenir au niveau principal

retall



Mise au point des routines: checklist

Mise au point

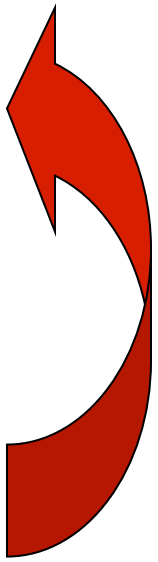
Ecrire, compiler, lancer

```
.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 !

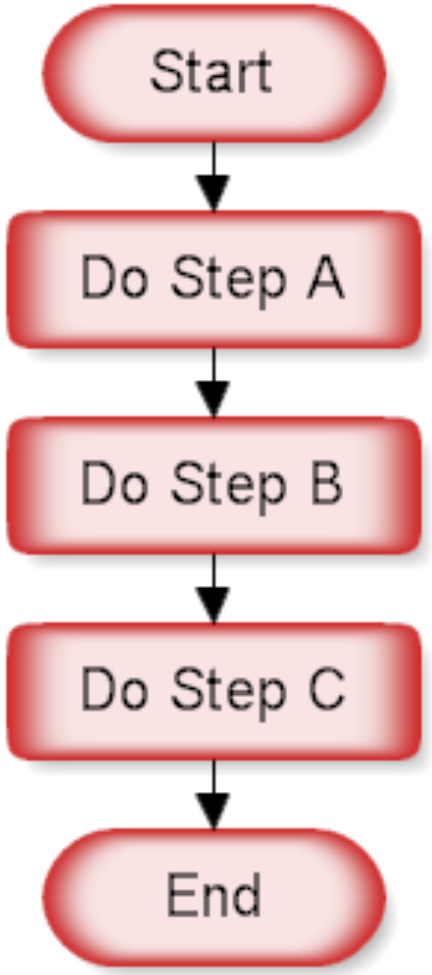
Instructions de structure

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

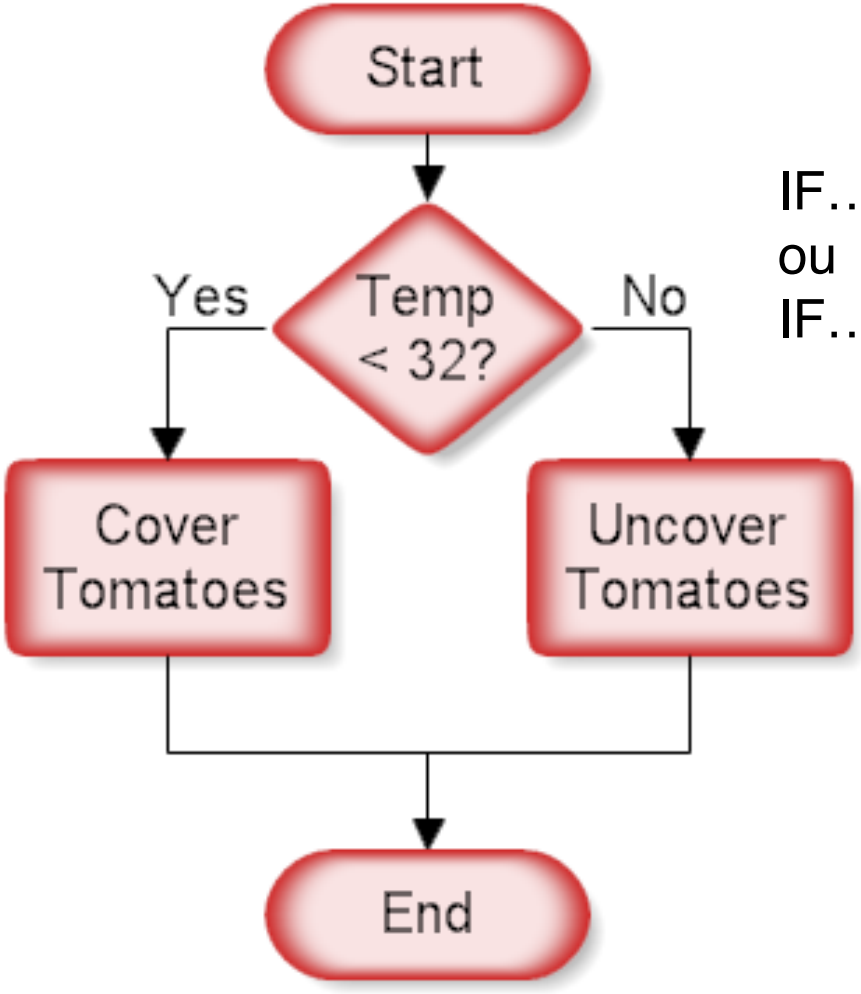
IDL utilise des instructions standard pour modifier ce déroulement



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

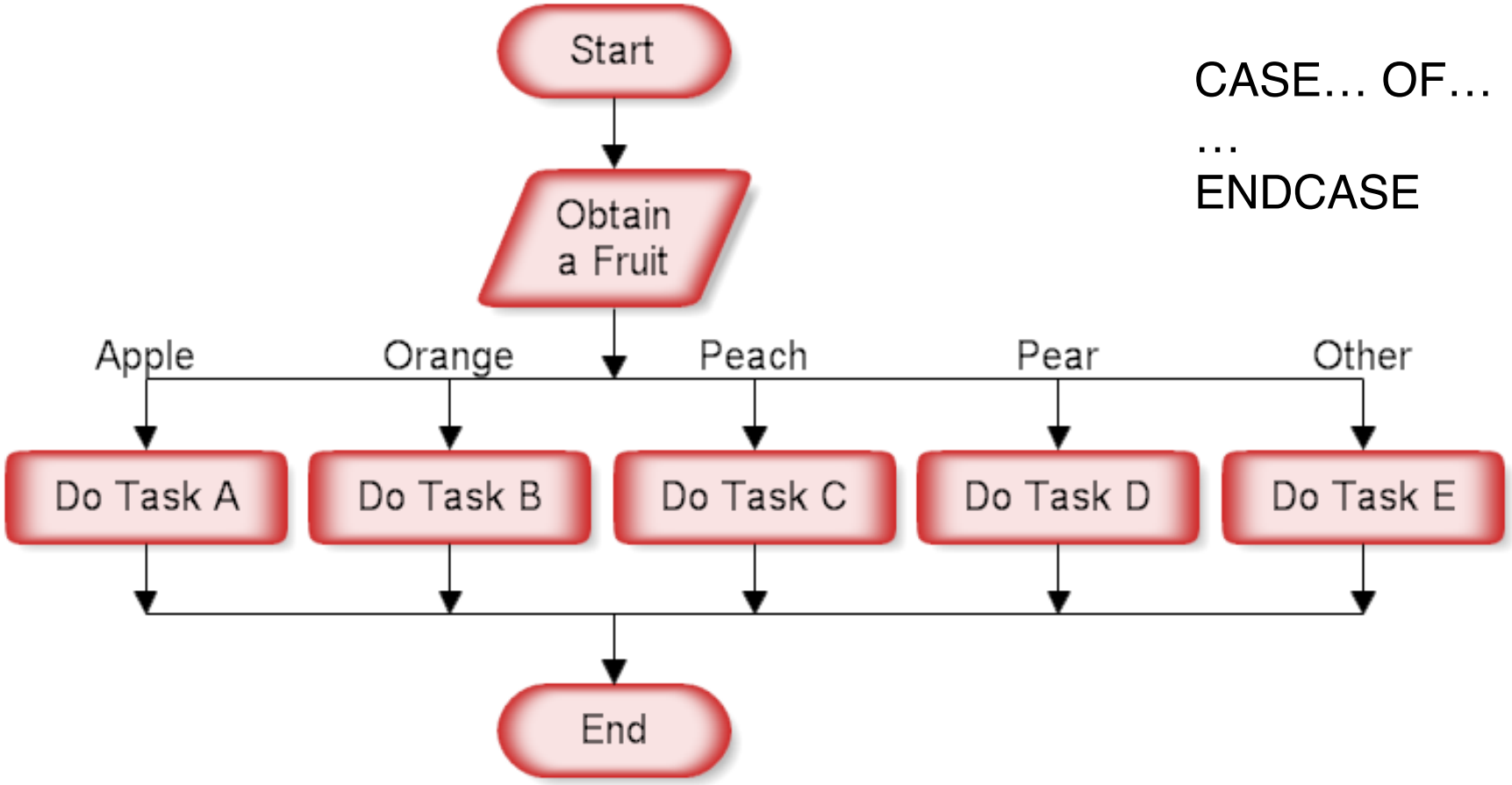


IF… THEN…
ou
IF… THEN… ELSE…

Instructions de structure

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

Tests multivariés

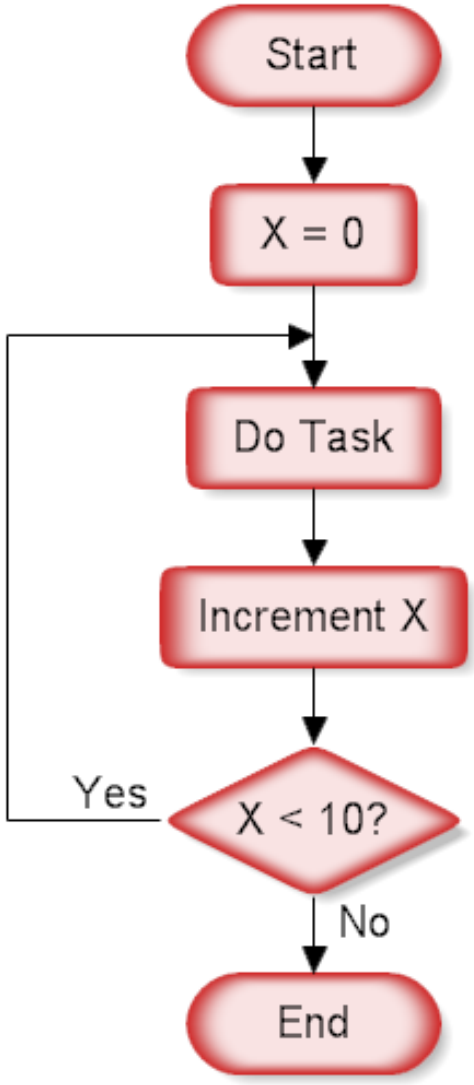


CASE… OF…
…
ENDCASE

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

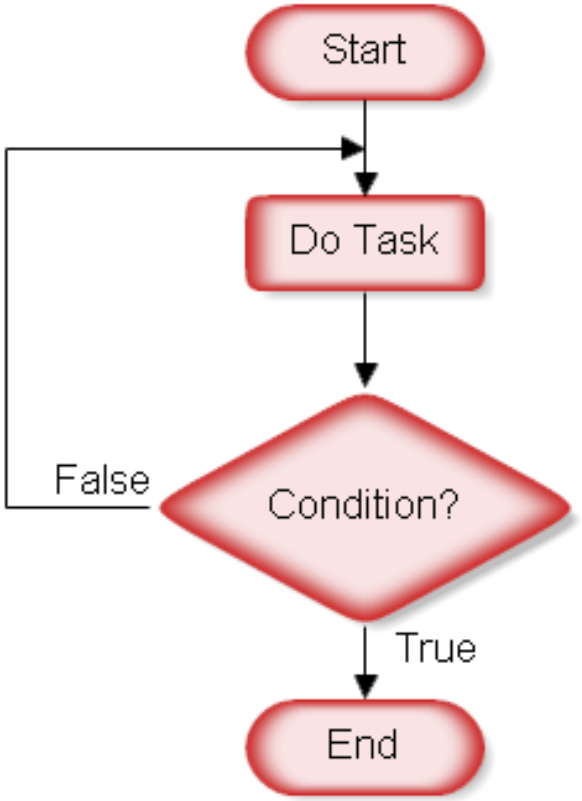
Instructions de structure

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

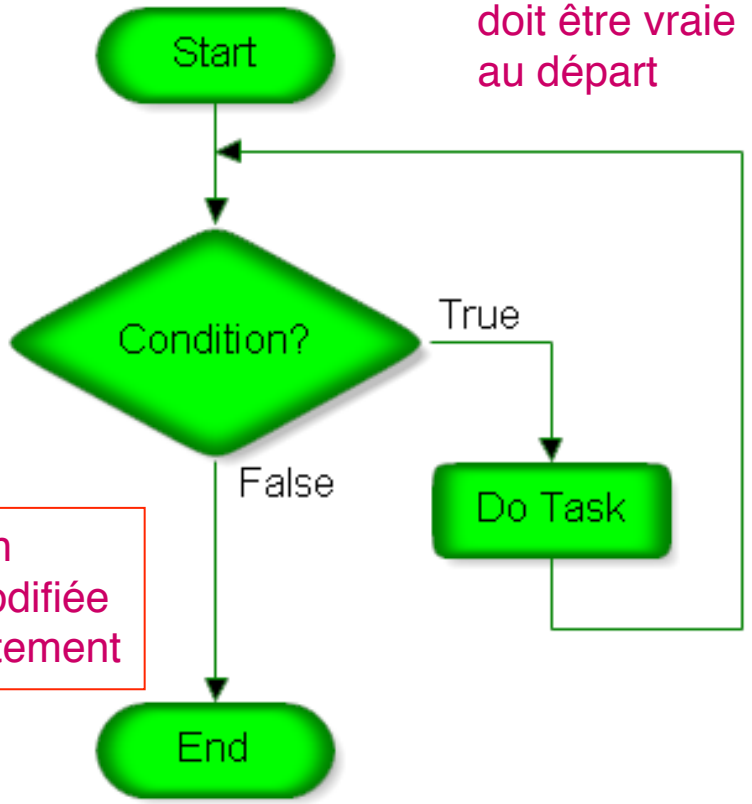
Exécutée au moins une fois



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

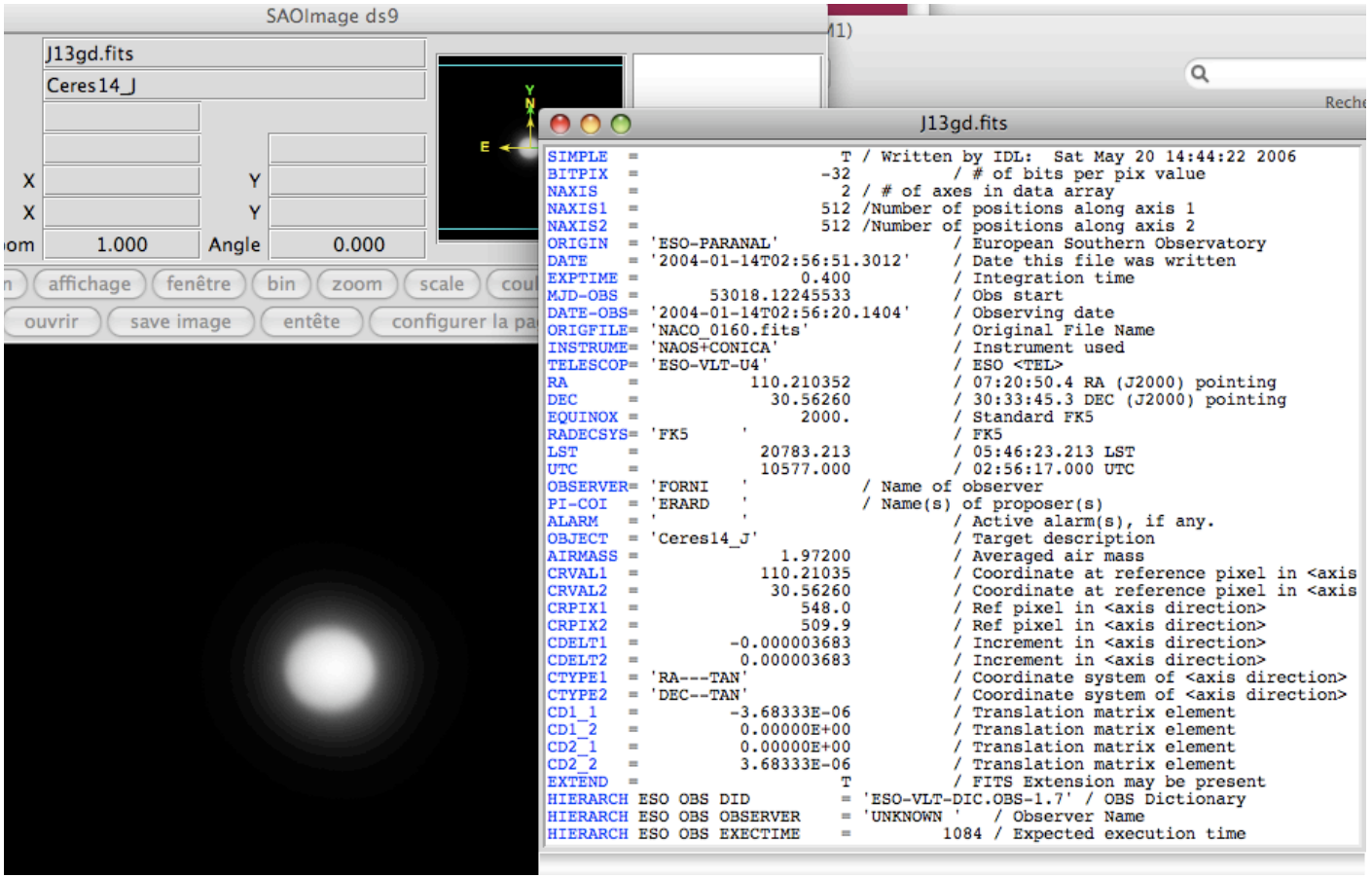
```
restore, 'fichier.ext'
save, file = 'fichier.ext', <liste de variables>
```

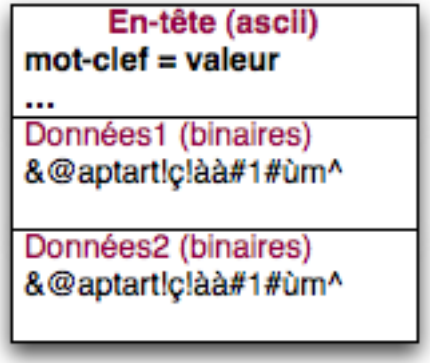
Lecture/écriture de fichiers

Format FITS (images astro)

Dans la bibliothèque ASTRON

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





Lecture/écriture de fichiers

Format FITS (images astro)

Lecture de l'en-tête avec les données ima = readfits('fichier.fits', header)
=> tableau de chaînes, on peut chercher dedans

Extraction de la valeur correspondant à un mot-clef tp_integ=sxpar(header, 'EXPTIME')

Ajout d'un mot-clef et d'une valeur associée sxaddpar, header,'COMMENT2', com2
dans une variable

Ecriture d'un fichier et de son en-tête writefits, 'fichier2.fits', data, header
(les premiers mots-clefs sont obligatoires dans le
standard FITS, mais la routine ne vérifie pas l'intégrité de l'en-tête)

Lecture/écriture de fichiers

Ouvrir un fichier

Associe un fichier à une unité logique openr, unit, 'fichier.ext', /get_lun

openr / openw / openu: lecture / écriture / les deux (update)
/get_lun: gère automatiquement l'unité logique
/comp : compresse/décompresse (gzip)

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 redimensionables

```
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

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
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
Tourne sous Windows	(maintenant supporté par GDL, expérimental)