Guide de programmation IDL

Stéphane Erard, IAS juin 1999

(Notes pour les travaux pratiques de traitement d'images du DEA Astrophysique et techniques spatiales, Paris-7 Paris-11)

Publié dans Micro-Bulletin **69**, 22-46, 1997.

Mises à jour : http://www.ias.fr/cdp/infos/idl/idl.html

Cette documentation a été écrite à l'origine pour les TP de traitement d'images du DEA d'Astrophysique et Techniques Spatiales de Meudon (Paris 7-Paris 11). Technique, mais destinée à des non spécialistes, elle aborde un certain nombre de points délicats dont l'expérience montre qu'ils se posent avec insistance aux utilisateurs. Ces notes ne sont pas destinées à se substituer aux manuels IDL, mais plutôt à les compléter en situation réelle en indiquant comment réaliser telle opération ou pourquoi une fonction ne marche pas comme on s'y attend. Elles supposent une certaine familiarité avec le langage lui-même : en particulier la syntaxe des commandes de structure (FOR, IF…) n'est pas discutée ici, mais laissée à la consultation des manuels de base.

Ce document est explicitement consacré au traitement d'images et ne détaille que les points importants dans ce contexte : optimisation de la programmation, entrées/sorties, calcul vectoriel simple, traitement de données, graphiques et images, impression. Les méthodes de calcul numérique et le graphisme en trois dimensions ne sont pas abordées ici. Les fonctions dont il est question sont celles des versions 4.0 et 3.6 d'IDL, et des principales bibliothèques de routines en libre accès ; les comportements spécifiques de certains systèmes d'exploitation (Unix, VMS et Macintosh) sont discutés quand ils sont source de dysfonctionnement.

Trucs essentiels

Les grands principes

• Sur une machine multitâche, le nombre d'utilisateurs simultanés d'IDL est limité par le nombre de licences. Il faut donc impérativement sortir d'une session active qui ne sert plus et ne pas conserver une session ouverte alors qu'on part faire autre chose… Sur les machines Unix, ctrl-Z suspend une tâche active sans la terminer ; il faut taper `job` pour voir les tâches suspendues, `fg <n° de la tâche>` pour en reprendre une, ou `kill %<n° de la tâche>` pour la terminer (`kill -9 <n° de la tâche>` dans les cas désespérés). Il vaut mieux vérifier avant de partir qu'aucune session IDL n'est active : on a vu des tâches suspendues rester actives après la fin d'une session.

• Il est important d'économiser la mémoire et le temps de traitement, qui sont partagés entre les utilisateurs. Un programme lent pénalise aussi les petits camarades… En pratique ça veut dire qu'il faut chercher à profiter au maximum des possibilités d'IDL

(calcul vectoriel, fonctions précompilées…) et ne pas programmer comme si c'était du Fortran à l'ancienne mode.

• Il faut toujours chercher à optimiser les entrées/sorties, c'est ce qui ralentit le plus les programmes. Plutôt que d'initialiser une routine d'usage fréquent avec de longs fichiers ascii, on commencera par sauver les données réellement utilisées en binaire, qu'on relira beaucoup plus vite pendant les exécutions suivantes (commandes SAVE et RESTORE).

• De façon générale il faut éviter les boucles, surtout quand l'indice varie beaucoup : c'est presque toujours possible en utilisant les fonctions sur les tableaux, et toujours (très) payant pour le temps d'exécution.

• Si l'on fait malgré tout des boucles sur les indices d'un tableau, il faut absolument faire tourner les premiers indices dans les boucles les plus internes, surtout sur les machines qui ont peu de mémoire (ceci minimise les accès disque).

• Et ne pas multiplier les tableaux inutilement : une image 512 x 512 en entiers représente 500 Ko de mémoire, sa transformée de Fourier occupe 2 Mo… Les stations de travail ont typiquement entre 16 et 128 Mo de mémoire à répartir entre les (nombreux) utilisateurs et le système (à titre de comparaison, un roman de format courant codé en ascii représente 1 Mo d'information — et certains sont très compressibles).

• Par ailleurs, la concaténation de tableaux, y compris la définition explicite avec des constantes, est peu efficace. Il faut donc éviter de multiplier ces instructions, en particulier éviter d'initialiser un tableau dans une fonction appelée fréquemment.

Trucs d'optimisation

1- Faire attention à l'ordre dans lequel on écrit une expression. Jouer notamment avec la préséance des opérateurs pour que les fonctions de recherche dans les tableaux ne soient pas appelées inutilement.

ex : `255 / max(IMAGE) * IMAGE`

est *beaucoup* plus rapide que `IMAGE * 255 / max(IMAGE)`

2- Ne jamais écrire un test sur un élément de tableau à l'intérieur d'une boucle, mais utiliser les opérations logiques sur les tableaux : c'est toujours possible.

ex : `For i = 0, (N-1) do if B(i) gt 0 then A(i) = A(i) + B(i)`

peut être dix fois plus lent que : `A = A + (B > 0)`

3- La plupart des opérations sur les tableaux sont d'ailleurs réalisables sans faire de boucle (quasiment les seules exceptions sont certaines situations où la valeur d'un élément est fonction de son indice).

ex : pour inverser une image 512x512 tête en bas, la procédure naïve à base de boucle est environ 30 fois plus lente que l'instruction compacte :

```
IMAGE2 = IMAGE(*,511 - indgen(512))
```

4- Certaines de ces opérations sont réalisables grâce à des fonctions IDL optimisées pour utiliser au mieux la mémoire et le temps de calcul. On y gagne aussi en soucis d'écriture pour peu qu'on connaisse l'existence de ces fonctions pré-programmées.

ex : On gagne encore un facteur dix pour résoudre le problème précédent en faisant :

```
IMAGE2 = rotate(IMAGE,7)
```

5- Plus classique : dans une expression, il faut utiliser des constantes de même type que les variables pour éviter les conversions de types ; c'est également important pour fixer le type du résultat, notamment quand on travaille sur des images 8 bits : ajouter une constante réelle donne un résultat réel, quatre fois plus gros et qu'il faudra reconvertir au moment de l'affichage.

```
ex :  B = A + 5      ; si A est entier,
      B = A + 5.     ; si A est réel,
      B = A + 5B     ; si A est une image sur 8 bits.
```

C'est également important pour les calculs :

```
print, 240  * 380
```

donne un résultat pour le moins curieux, car les arguments et donc le résultat sont des entiers codés sur 16 bits. Dans le même ordre d'idée, les entiers (courts) sont toujours signés jusqu'à la version 5.2 : il faut s'en souvenir quand on fait de l'arithmétique avec des images codées sur 16 bits (on a souvent intérêt à convertir en entiers longs pour utiliser toute la dynamique).

6- Ne pas laisser traîner d'instructions inutiles dans une boucle (initialisations d'expressions constantes…), mais les effectuer avant la boucle. Par ailleurs les tableaux sont initialisés à zéro par défaut, ce qui n'est pas forcément utile.

```
A=fltarr(512,512,/nozero)
readu, 1, A
```

Évite de mettre A à zéro alors qu'on va le modifier tout de suite.

7- Minimiser les déplacements à longue portée dans les tableaux qui multiplient les accès disque, beaucoup plus lents que les accès mémoire (les stations travaillent en mémoire virtuelle, c'est à dire que seule une partie des variables est présente en mémoire à un moment donné).

Dans les boucles imbriquées, il faut *impérativement* faire tourner à l'intérieur les premiers indices de tableaux pour accéder aux éléments selon l'ordre dans lequel ils résident en mémoire.

```
ex :    for i = 0, 511 do for j= 0,511 do A(j,i)=0.
```

(qui par ailleurs est idiot…) est au moins 50 fois plus rapide que de le faire en ordre inverse : dans ce cas, il faut recharger en mémoire physique la fin ou le début du tableau à chaque affectation.

8- On récupère facilement la place utilisée par les tableaux qui ne servent plus en leur affectant une valeur scalaire. C'est la bonne réponse à un message d'erreur "`UNABLE TO ALLOCATE MEMORY`".

```
ex :    A = fltarr(512,512)
        …
        (fini pour A)
        A =0            On récupère 1 Mo de mémoire.
```

Quand on travaille avec de gros fichiers, il arrive qu'on n'ait pas même la place de lire une image et de la transformer dans un autre tableau. Penser à la fonction ASSOC, qui évite de lire l'ensemble du fichier avant de travailler dessus.

9- Toujours pour épargner la mémoire, il faut tuer les gros tableaux avant de les ré-initialiser, et utiliser la fonction TEMPORARY.

```
ex :    A = IMAGE
        …
        A = 0                   ; convertit A en scalaire
        A = fct(IMAGE)
```

peut sembler étrange, mais évite d'occuper deux fois la place de IMAGE durant la dernière affectation.

```
        A = temporary(A) + 1
```

N'utilise que la place nécessaire pour stocker A, au lieu du double sinon (A = A + 1).

10- On peut utilement penser à épargner les disques aussi. Les fichiers de données secondaires de plusieurs Mo n'ont pas vocation à rester longtemps sur un disque en ligne. S'ils sont importants ou difficiles à reconstruire, il peut être plus judicieux de les

conserver sur un support externe, ou au moins de les comprimer. Avant d'empiler des dizaines d'images de 200 Ko dans un répertoire, il est bon de se demander ce qu'on va en faire, et si elles serviront encore une fois imprimées. Les gros stockages se font sur les disques poubelles (quand il y en a) plutôt que sur les disques communs, ou sur CD. Les CD sont délicats d'utilisation sous Unix (requièrent un accès système), mais il existe une procédure de sauvegarde sur disque optique pour certaines machines du labo.

11- Dans le même ordre d'idées, il est souvent judicieux de stocker des données lues et relues en binaire plutôt qu'en ascii. Les fichiers ascii sont plus volumineux et surtout beaucoup plus longs à lire. Si une routine lisant un fichier ascii doit être exécutée de nombreuses fois (pour la mettre au point ou la modifier) on a donc intérêt à stocker les données en binaire dès la première exécution, surtout si la lecture est suivie d'un long calcul. L'instruction SAVE fait ça très facilement :

```
save, <liste de variables>, filename = "pipo.dat"
```

écrit le fichier PIPO.DAT contenant les variables indiquées. Il se relit avec :

```
restore, "pipo.dat"
```

Cette instruction redéfinit les variables sauvegardées et leur réaffecte les valeurs correspondantes (en écrasant éventuellement des variables homonymes). Ne pas préciser de liste de variables sauvegarde tout le contexte.

Ce format est de plus indépendant de la machine et permet d'éviter les problèmes de stockage interne de façon transparente. Sous VMS, il faut ajouter l'option /XDR pour sauvegarder dans ce format d'échange commun. Il semble bien que le format de sauvegarde ait changé avec la version 5.1, les archives plus anciennes sont souvent difficiles à relire.

12- La même procédure SAVE peut être utilisée pour éviter bien des drames : elle permet de sauver des résultats intermédiaires dans les programmes qui tournent toute une nuit, et de limiter la portée d'éventuels problèmes système.

Erreurs fréquentes

- Attention à la division entière : 4 / 3 = 0 ; on l'oublie facilement en manipulant des tableaux, et on obtient des images noires ou seuillées. Faire en particulier attention aux types des variables système.

- Autre piège des variables entières : les indices de boucles supérieurs à 32767

(compteurs incrémentés manuellement dans les boucles while en particulier).

- Les paramètres de sortie des routines ne peuvent pas être des éléments de tableaux ou des champs de structures : ceux-ci sont transmis par valeur et ne sont donc pas modifiables. Les arguments doivent être des scalaires ou des tableaux complets, c'est une chose à vérifier en priorité lors d'un fonctionnement anormal : IDL ne détecte pas d'erreur dans ce cas.

- En particulier READF, TAB(5) ne permet pas de modifier tab(5)…

- Il y a parfois des problèmes avec le signe - utilisé comme négation (c = -c) ; il est prudent de le mettre en parenthèses.

- Une cause d'affolement du compilateur est l'homonymie entre variables et fonctions. Éviter les noms trop simples qui pourraient être utilisés par des fonctions obscures des bibliothèques utilisateur… En fait IDL ne se donne même pas la peine de vérifier ses mots-clefs réservés à la compilation, et accepte des noms de variables tels que float ou min sans broncher (une très mauvaise idée est par exemple d'appeler long un tableau contenant des longitudes…).

- Si IDL ne trouve pas une routine, vérifier qu'elle est dans le chemin d'accès défini dans la variable !PATH d'IDL. Vérifier aussi qu'on est dans le répertoire où l'on croit être, et que le chemin ne pointe pas sur un répertoire contenant une routine homonyme. Une cause d'erreur classique est que la routine ne porte pas le même nom que le fichier qui la contient. Si c'est une fonction appelée par une autre routine, vérifier qu'il n'y a pas de conflit de nom avec un tableau et essayer de la compiler séparément, avant de l'appeler. Sous Unix, il faut fournir le nom exact du fichier pour le compiler (en tenant compte des majuscules).

- Quant aux fichiers de données, on a intérêt à ne pas les pointer avec des chemins absolus, mais à l'aide de variables systèmes définies dans le fichier idl_start.pro (on se garde ainsi des réorganisation de disques) :

```
DEFSYSV, '!images_dir', '~/lune/10jul98/'      ; dans le idl_start
openr, /get=lun, !images_dir+'LU73b04.FIT'     ; dans la routine
```

- À la suite d'une erreur d'exécution, IDL s'arrête et reste dans la procédure active (à moins qu'on ait indiqué de faire autrement). On a donc accès aux variables de la procédure qui pose problème, mais pas à celles du programme appelant. On remonte d'un cran dans le hiérarchie d'appel en faisant RETURN (ou RETURN, 0 à partir d'une fonction), et au niveau principal avec RETALL. HELP décrit seulement les variables du niveau courant.

- Après avoir modifié une routine dans un éditeur de texte, il faut sauver le fichier et recompiler sous IDL. Sinon, on relance la version précédente — et on oublie souvent de sauver…

- Toutes les routines qui exploitent un pointeur (souris) supposent qu'il possède trois boutons. Dans la version Mac le seul bouton est considéré comme étant celui de gauche, le bouton du milieu et celui de droite sont simulés par les flèches gauche et droite respectivement (versions < 5), ou par Option-clic et Comm-clic (version 5).

- Dans une instruction composée on ne peut pas chaîner plusieurs instructions avec & ; il faut obligatoirement utiliser un bloc BEGIN… ENDxxx.

```
if machin then plot, a & print, b      ; affiche b à tous les coups
```

Eléments de base

Instructions de contrôle et caractères spéciaux

Les instructions de contrôle sont fournies au clavier depuis la fenêtre de commande IDL.

@	Exécute une séquence d'instructions.
.r	Compile une routine ou lance un programme principal.
.c	Reprend l'exécution d'une routine arrêtée (on peut modifier les variables entre temps).
.go	Reprend l'exécution au début de la routine courante.
.s	Reprend l'exécution pas à pas.
.r -t <nom>	Recompile en affichant un listing de la procédure (avec les numéros de lignes utilisés par la routine d'erreur).
$	En début de ligne, introduit une commande transmise au système d'exploitation.
Spawn	Ouvre un sous-process dans la session IDL, utilisable depuis une procédure (permet de lancer une application externe par exemple).

Journal, 'fichier' Stocke les commandes en ligne dans un fichier (idlsave.pro par défaut). Utile pour garder une trace de ce qu'on bricole en mode interactif ; les résultats d'affichage sont stockés sous forme de commentaires. Une fois le fichier ouvert, cette même commande permet d'introduire des chaînes de commentaires dans le journal.

Les vingt commandes les plus récentes peuvent être rappelées avec la flèche vers le haut. On peut augmenter ce nombre en redéfinissant par exemple !EDIT_INPUT=50 dans le fichier idl_start.pro (pas au cours d'une session). Sous Unix, il arrive que le rappel de commande deviennent inopérant quand le système a des indigestions ; la raison en reste mystérieuse, mais il arrive que tout fonctionne normalement en démarrant IDL depuis une fenêtre Xterm plutôt que DTterm (qui est le défaut sous CDE ; taper xterm dans la fenêtre, et démarrer IDL depuis la nouvelle fenêtre).

IDL ne distingue pas les majuscules des minuscules, sauf dans les commandes envoyées au système d'exploitation, notamment dans les arguments des instructions de contrôle (en particulier .r sous Unix). Sous Unix, les procédures dont les noms contiennent des majuscules doivent toujours être compilées à la main. Les caractères spéciaux sont :

& séparateur d'instructions sur une même ligne.

$ caractère de continuation en fin de ligne, permet d'écrire une même instruction sur plusieurs lignes (ne doit pas apparaître dans une chaîne).

Les routines et programmes principaux sont compilés, mais en mode interactif les commandes sont interprétées au fur et à mesure : les instructions doivent être écrites sur une seule ligne (les blocs d'instructions ne sont pas compris, mais on peut chaîner plusieurs lignes avec des caractères de continuation).

Les bibliothèques de procédures

IDL consiste en un noyau de commandes de base, une bibliothèque de routines dites « utilisateur » installée en standard, et des bibliothèques « contributives » qui ne font pas partie du langage lui-même, bien que certaines soient fournies avec IDL.

Les commandes de base incluent les commandes de structure, les fonctions mathématiques, certaines procédures graphiques élémentaires comme PLOT ou TV, qui ensemble forment le langage de programmation. La bibliothèque utilisateur est composée

de routines écrites avec ces commandes, dont les sources doivent être compilées à l'exécution ; la plupart sont écrites et maintenues par les programmeurs d'IDL. En fait, elles évoluent lentement d'une version à la suivante, et il arrive qu'on ait des surprises en changeant de version — quoi qu'il en soit, elles font partie du langage et ne doivent pas être modifiées. Les routines qui disparaissent lors d'une mise à jour sont généralement présentes dans un répertoire « obsolete ». Selon la version d'IDL, la bibliothèque utilisateur est d'un seul tenant ou séparée en modules (Statlib, Mathlib…). Enfin, les bibliothèques contributives sont développées par des programmateurs indépendants et constituent des ensembles à la cohérence parfois douteuse : les noms de routines sont parfois en conflit, on ne peut donc pas les inclure simultanément dans le chemin d'accès. On peut les réutiliser et les modifier, à conditions bien sûr qu'elles ne soient pas partagées par d'innocents utilisateurs qui en pâtiraient — bref, on fait ça dans son propre répertoire.

Parmi les plus intéressantes la bibliothèque MERON, qui était livré avec la version 3.6 d'IDL, contient de nombreuses routines mathématiques et graphiques. La bibliothèque ASTRON développée par la Nasa est extrêmement riche de fonctions utiles pour le traitement d'images télescopiques et la gestion des fichiers FITS et HST ; d'autres bibliothèques sont disponibles sur le réseau, en général dédiées à l'exploitation de jeux de données particuliers. Le site du Centre de données planétaires de l'IAS contient notamment une page d'information sur les bibliothèques publiques IDL (www.ias.fr/cdp/infos/idl/biblio_idl.html). Le présent document évoque des routines qui ne font pas partie de la distribution standard d'IDL mais sont particulièrement adaptées à certains traitements ; leur origine est toujours signalée explicitement.

Documentation

HELP	Liste et type des variables déclarées dans la procédure courante. Avec les mots-clefs appropriés on a accès aux caractéristiques du système graphique (HELP, /DEV), aux fichiers ouverts (HELP, /FILES), aux structures (HELP, /STRUCTURE)…
?	Aide hypertexte à partir de la version 4.0. Comprend le manuel utilisateur complet et la description de toutes les routines internes.
Doc_library	Aide en ligne simple. Donne accès à la documentation de toutes les routines des répertoires pointés par la variable !IDL_PATH.
Man	Liste toutes les bibliothèques pointées par !IDL_PATH. Simule la

fonction d'aide de la version 3.6 (une routine contributive distribuée avec ASTRON).

Doc Similaire, mais groupe toutes les routines (bibliothèque SERTS).

Online_help Documentation avec interface graphique (version 5), remplace XDL des versions 3.x.

Mk_html_help, ['dir1', 'dir2'], 'page.html' Crée une documentation en format html à partir des en-têtes des routines des répertoires dir1 et dir2 (version 5).

Variables, constantes et opérateurs

Les variables peuvent être de trois natures (scalaires, tableaux ou structures), et de huit types : octets, entiers, entiers longs, réels, réels double précision, complexes et chaînes. Les noms de variables commencent obligatoirement par une lettre, mais peuvent contenir des chiffres ou le caractère '_'. Faire attention à ne pas utiliser un nom attribué à une fonction (il n'y a pas de vérification systématique à la compilation).

Contrairement à d'autres langages, en particulier le Fortran, les entiers sont codés sur 2 octets et les entiers longs sur 4 octets. Les entiers non-signés n'apparaissent que dons la version 5.2. Avec les précédentes les entiers courts ont des valeurs comprises entre -32768 et 32767, et il faut donc travailler en entiers longs pour manipuler des images codées sur 16 bits (on risque des surprises avec l'arithmétique 16-bits sinon). Les variables sont déclarées à chaque initialisation avec le type explicitement fourni. Leur type peut donc être modifié très souvent, et l'initialisation des variables n'a pas en général pas grand sens. On force le type des constantes de la manière suivante :

3B	est Byte (sur un octet, donc)
3	est entier (sur 2 octets)
3L	est entier long (sur 4 octets)
3.	est réel (sur 4 octets)
3.D	est réel double précision (sur 8 octets)
complex(3,0)	est complexe (2 fois 4 octets)
dcomplex(3,0)	est complexe double précision (2 fois 8 octets, existe depuis la version 4 seulement).

Les types entiers peuvent être entrés en numération décimale, octale ou

hexadécimale (voir la doc papier pour la syntaxe).

Les variables sont stockées en mémoire de façons différentes selon les machines et les types (les deux octets d'un entier court sont stockés dans un ordre ou dans l'autre, par exemple). Deux routines permettent de convertir les variables entre les différents systèmes de représentation interne : BYTEORDER convertit seulement les entiers (scalaires et tableaux), SWAP_ENDIAN convertit tous les types (il existe plusieurs options pour les réels) et gère les structures. Ces fonctions sont parfois indispensables pour travailler sur des données binaires provenant d'une autre machine, y compris entre deux machines Unix (les fichiers save IDL sont à l'abri ces problèmes).

Les opérateurs numériques sont semblables à ceux du Fortran, à part l'exponentiation qui est notée ^ (et accepte des arguments complexes), et le modulo qui s'utilise comme un opérateur (`RESTE=A MOD B`). Il existe deux opérateurs supplémentaires < et >, binaires tous les deux, qui renvoient respectivement le plus petit ou le plus grand des opérandes (à ne pas confondre avec les opérateurs de comparaison GT et LT), et deux types de multiplication matricielle (voir plus loin). Depuis la version 5, il existe également un opérateur ? semblable à celui du C (équivalent à une instruction IF). Les opérateurs booléens et de comparaison retournent des valeurs 1 pour vrai et 0 pour faux (c'est différent du Fortran et du C).

Chaînes de caractères

Les chaînes sont considérées comme des variables, pas comme des tableaux (comme c'est le cas en C). Un tableau de chaînes peut contenir des chaînes de longueurs différentes sans qu'on s'en préoccupe.

Les fonctions sur les chaînes de caractères sont détaillées dans le manuel utilisateur (chapitre 8 pour la version 4). La seule particularité un peu obscure concerne les transformations de chaînes en valeurs numériques ; il y a équivalence entre caractères ascii et valeurs de type Byte seulement :

`string([67,105,97,111])`	renvoie une chaîne composée des chiffres tabulés
`string([67B,105B,97B,111B])`	renvoie les 4 caractères ascii correspondants
`byte(chaîne)`	renvoie les codes ascii des caractères
`fix(chaîne)`	renvoie la valeur numérique de la chaîne (si elle est valide)

Structures

Les structures regroupent des variables de types quelconques apparaissant

toujours dans le même ordre. Elles sont typiquement utilisées pour regrouper les différentes entrées d'une base de données, et apparaissent généralement sous forme de tableaux de structures :

```
st = {ligne, nom:' ', A:1., tab:fltarr(64)}  ; définit la structure 'ligne'
base = replicate(st, 30)                     ; déclare un tableau de 30 structures 'ligne'
…
print, base.a                          ; affiche les 30 valeurs du champ A
print, base.tab(0)                     ; affiche les 30 valeurs de tab(0)
print, base(0).tab                     ; affiche tab en entier pour la première ligne
```

La fonction HELP ne décrit pas par défaut les structures déclarées. Pour obtenir une information sur une structure existante, il faut faire HELP, /STRUCTURE, <variable>. La bibliothèque ASTRON contient de nombreuses procédures qui facilitent la gestion des structures.

Différents types de programmes IDL

En dehors du mode interactif, il existe trois méthodes pour envoyer des commandes à IDL : séquences d'instructions, programmes principaux, et routines (procédures et fonctions). Celles-ci sont décrites plus loin.

Séquences d'instructions ("batch")

On peut enregistrer une séquence de commandes IDL fréquemment exécutée, qui sera lancée en faisant :

```
@Test
```

où Test est un fichier d'instructions IDL qui sont interprétées comme si elles étaient entrées au clavier, pas comme une routine (les blocs d'instructions sur plusieurs lignes ne sont pas compris). IDL recherche d'abord un fichier Test.pro, puis un fichier Test. « Test » est un nom de fichier, il doit donc respecter les règles de syntaxe du système utilisé (sensible aux majuscules sous Unix).

Pour lancer une routine de cette façon, il faut écrire un fichier d'instructions qui appelle la routine et lui fournit des paramètres comme si on l'appelait depuis la fenêtre de commande :

```
a=[1, 2, 3]
```

```
plot, a
…
```

Une telle séquence doit être complètement paramétrée comme dans l'exemple ci-dessus ; si les routines qu'elle contient demandent des valeurs, celles-ci doivent être entrées au clavier, on ne peut pas les fournir dans le fichier (ce n'est donc pas à proprement parler un batch, contrairement à ce que dit la doc IDL).

Programmes principaux

IDL utilise d'autre part des « programmes principaux » compilés avant exécution, et supportant donc les instructions sur plusieurs lignes. Ils se terminent obligatoirement par une instruction END, mais ne comportent pas de ligne de déclaration comme les routines (PRO…). Un seul programme peut être présent en mémoire. Il doit être explicitement compilé à la première exécution (lancement par .r <nom> depuis la fenêtre de commande) ; une fois en mémoire on peut le relancer par .go (avec les versions anciennes d'IDL, il faut recompiler à chaque exécution).

Les programmes principaux sont équivalents à des procédures mais sont exécutés au niveau principal : en clair, ils travaillent sur les variables déclarées au moment de l'appel et on a accès à toutes les variables après exécution sans devoir transmettre de paramètre — par contre on ne peut pas appeler ces programmes depuis une procédure ou un autre programme. Leur principal intérêt est l'accès direct aux variables, qui pose tout de même un problème sérieux d'encombrement mémoire pour de gros programmes (les routines constituent une méthode de programmation beaucoup plus efficace). Ils servent essentiellement à écrire les commandes principales de grosses applications (appels de routines en séquence).

Appel d'une routine en batch sous Unix

Sous Unix, il existe au moins deux méthodes pour faire tourner une routine en batch (session détachée des consoles). Dans les deux cas les affichages à l'écran produisent des erreurs, puisque la sortie n'est pas définie ; si les accès écran sont indispensables, utiliser le pilote d'écran virtuel Z au lieu de X (les images sont perdues à la fin de l'exécution, mais on peut les enregistrer en cours de route). Attention tout de même, ce pilote ne gère pas le fenêtrage et l'instruction WINDOW dans une procédure provoque une erreur ; si la routine gère explicitement la sortie PS, il vaut mieux l'appeler avec cette option.

La première méthode consiste à se déconnecter en laissant tourner IDL en tâche de fond. Il faut pour cela avoir invoqué IDL de façon adéquate :

```
unix> nohup IDL  /* Permettra de continuer en tâche de fond après déconnexion */

IDL> set_plot, 'Z'              ; utilise le pilote virtuel pour éviter les accès écran
IDL> routine, parametres        ; Appel du prg avec ses paramètres
IDL> <CTRL> Z                   ; suspend IDL

unix> bg        /* relance IDL en tâche de fond (fournir éventuellement le n° de job) */
unix> logout    /* sort de la session Unix */
```

L'autre méthode ne marche qu'à condition d'être autorisé à lancer des batch sur la machine (nom d'utilisateur déclaré dans le fichier /var/adm/cron/at.allow du système), ce qui comme toujours sous Unix ne va pas de soi (voir l'administrateur système). C'est cependant la méthode la plus civique, puisqu'elle permet de lancer l'exécution aux heures creuses et ne gèle pas une licence IDL.

```
unix> at -finst.com 23:00   /* lancera le fichier inst.com à 23h (attention au f !) */
```

Le fichier inst.com est donc un fichier de commandes Unix contenant :

idl batch.pro <return>

Et le fichier batch.pro est une routine IDL contenant :

```
set_plot, 'Z'
routine, parametres
exit
```

Si tout va bien, les textes affichés sur le terminal (mais pas les images) sont envoyés par mail à la fin de l'exécution, avec diverses informations sur le déroulement du process.

Procédures et fonctions

Écriture

Les procédures ont la forme suivante :

```
PRO machin, <liste de variables>
…
END
```

Les fonctions sont similaires, mais retournent un argument explicite :

```
FUNCTION machine, <liste de paramètres>
…
RETURN, <valeur>
END
```

Il est préférable d'écrire procédures et fonctions dans des fichiers séparés qui portent le même nom qu'elles, avec l'extension .PRO. Chaque routine doit avoir un nom unique, et différent des noms de variables. En cas de problème, vérifier qu'IDL ne prend pas une fonction pour un tableau (faire HELP) ; vérifier aussi qu'il n'existe pas plusieurs fichiers portant le même nom dans les différents répertoires accessibles. Si le compilateur ne trouve pas une routine sous Unix, c'est probablement parce que le nom de fichier contient des majuscules (la compiler à la main).

Quand une routine appelle des sous-routines qui lui sont spécifiques, qu'elle seule utilise, on peut écrire l'ensemble dans un même fichier. Dans ce cas, la routine principale doit apparaître en dernière position du fichier, sinon IDL ne verra pas les sous-routines (le compilateur s'arrête après la routine mère, et ne retrouve pas ensuite de fichier correspondant).

Appel, compilation

• Les procédures sont appelées comme des commandes :

```
Machin, <liste de paramètres>
```

Une procédure en état de marche peut être simplement appelée par son nom quand on en a besoin. Si cette procédure n'est pas déjà compilée, IDL cherche dans les répertoires prédéfinis un fichier texte appelé machin.pro, le compile et l'exécute (sous Unix, le compilateur convertit d'abord en minuscules, et ne trouve pas les fichiers contenant des majuscules…).

• Les fonctions sont appelées en y faisant référence dans une expression, et les paramètres sont fournis entre parenthèses (même quand on ne fournit pas d'arguments, ou quand ceux-ci sont initialisés par mots-clefs…) :

```
A=machine(<liste de paramètres>, /<mots-clef binaires>)
```

Cette syntaxe ne permet pas de distinguer si MACHINE est une fonction ou un tableau ; IDL vérifie d'abord s'il existe un tableau de ce nom, puis cherche un fichier machine.pro dans les répertoires prédéfinis, et enfin s'arrête en disant que la variable n'est pas définie. En cas de problème, il faut donc vérifier si MACHINE est considérée comme fonction ou

non (faire HELP) et éventuellement compiler la fonction avant de l'appeler. À partir de la version 5, l'indexation des tableaux avec la notation […] supprime ce problème, mais est incompatible avec les versions précédentes. La commande FORWARD_FUNCTION permet de réserver un nom à une fonction.

• Quand on met au point une routine, il faut explicitement la recompiler après chaque modification en faisant :

```
.r <nom de fichier>
```

sinon, IDL voit la routine compilée en mémoire et ne va pas chercher le fichier correspondant. Il faut évidemment avoir sauvegardé la nouvelle version pour qu'IDL la prenne en compte (sortir carrément de l'éditeur en cas de doutes).

• Dans la version 5.1, la routine RESOLVE_ALL recompile toutes les procédures appelées dans les procédures déjà compilées — ça évite les bêtises quand on modifie un ensemble de procédures. Attention, l'appel de cette routine empêche de faire tourner des programmes 5.1 avec les versions plus anciennes.

• Toutes les routines sont récursives.

• On peut appeler indirectement les routines à l'aide des commandes CALL_FUNCTION et CALL_PROCEDURE (avec le nom de routine dans une variable). CALL_EXTERNAL permet d'exécuter une routine écrite en C ou en Fortran (la transmission d'arguments est parfois incertaine, toujours laborieuse).

Listes de paramètres

Les arguments de routines peuvent être passés directement ou par mot-clef. Une routine peut être appelée sans définir tous ses arguments. D'autres paramètres peuvent être passés par COMMON.

• Les paramètres simples sont interprétés selon leur ordre d'apparition dans la définition de la routine et l'instruction d'appel. Certains peuvent être optionnels.

ex : dans Plot, X, Y le premier vecteur est tracé en abscisse. S'il est absent, il est remplacé par la liste des indices de Y.

• Les mots-clefs permettent de s'abstraire de l'ordre d'apparition des paramètres ; ils servent surtout à choisir des options.

ex : `PRO truc, A, B, bidule=chose`

peut-être appelée par TRUC, bidule=23
ou TRUC, /bidule (équivalent à TRUC, bidule=1)

Dans la routine elle-même, c'est chose qui intervient comme variable (bidule ne sert qu'à la correspondance entre l'appel et la routine). Les mots-clefs sont par définition des paramètres optionnels, il faut donc tester leur présence à l'appel et prévoir des valeurs par défaut dans les routines.

• Les blocs COMMON contiennent des données accessibles à toute routine qui veut les utiliser. Ils sont déclarés dans une routine par :

```
COMMON <nom du bloc>, <liste de variables locales>        ; sans virgule après COMMON
```

Ils permettent notamment de passer des valeurs entre plusieurs appels successifs d'une même routine sans encombrer le programme principal. Certains blocs sont maintenus par IDL soi-même, notamment pour gérer les tables de couleurs. À noter qu'on ne peut pas redéfinir un bloc COMMON dans une session IDL (il faut sortir, et relancer IDL).

• Une même variable ne peut être déclarée qu'une seule fois dans une routine donnée. Si elle appartient à plusieurs blocs COMMON, un seul peut être déclaré dans une routine. De même, une variable déclarée à l'intérieur d'une routine dans un bloc COMMON ne peut pas être passée comme argument à cette routine.

• Il n'y a pas de différence formelle entre paramètres d'entrée et de sortie ; cependant, seules les variables utilisateur non indicées peuvent être modifiées dans une routine. Si un paramètre est initialisé à l'appel par une constante, expression, *variable système*, *élément de tableau* (y compris un sous-tableau) ou champs de structure, on ne récupère pas sa nouvelle valeur à la sortie.

Ex : `Truc, tableau(5)`

ne permet pas de modifier tableau(5). Il faut écrire

```
temp=tableau(5)
truc, temp
tableau(5)= temp
```

L'exemple le plus fréquemment rencontré est celui de l'instruction READF, qui ne permet pas de lire un élément de tableau, mais ne le signale pas.

• On peut passer des mots-clefs à une routine à travers une autre routine dans laquelle ils ne sont pas définis. Ceci permet d'alléger considérablement l'écriture et la maintenance de routines intermédiaires, où on ne déclare que les mots-clefs qu'on intercepte.

```
PRO tartufo, a, b, _extra=pipo, color=c, cause=cause
```

```
                                        ; Stocke les mots-clefs non déclarés dans pipo
…
if keyword_set(cause) then print, 'blah blah blah'
plot, a, b, _extra=pipo, color=c+10      ;…et les passe tel quel à Plot
                                        ; Color est intercepté puis transmis explicitement,
                                        ; Cause est réservé à la routine
end
```

Et on appelle par exemple : `Tartufo, a, b, thick=5, color=3, /cause`

Attention, IDL 4 plante méchamment quand on envoie ainsi des mots-clefs à une routine qui ne sait pas quoi en faire. Sous IDL 5, les routines appelées n'utilisent que les mots-clefs qu'elles savent traiter, ce qui permet de les transmettre à plusieurs routines sans avoir à les intercepter.

Tests sur les paramètres d'appel

A l'intérieur d'une routine, on peut utiliser les fonctions suivantes pour vérifier les conditions dans lesquelles la routine a été appelée :

N_params()	retourne le nombre d'arguments à l'appel (ne compte pas ceux qui sont transmis par mot-clef).
Size(A)	Retourne les dimensions d'un paramètre et son type.
Keyword_set(motclef)	Vaut 1 si motclef a été initialisé à une valeur non nulle (généralement utilisé pour tester les options binaires).
N_elements(A)	Retourne le nombre d'éléments de A, 0 s'il n'est pas défini. C'est la méthode normale pour tester l'absence d'un mot-clef à l'appel de la routine et fixer une valeur par défaut (N_elements(motclef) eq 0).
Arg_Present(A)	Vaut 1 si la variable est retournée au programme appelant (*ie*. l'argument est présent et correspond à une variable transmise par référence). Utile surtout pour gérer les pointeurs (depuis la version 5.0).

```
Pro JoliePassante, x, y, z, OToiQueJEusseAime=a, ToiQuiLeSavais=s
 if N_params() ne 3 then begin
  print, 'Je veux des arguments'
   return
 endif
 If (size(x))(0) ne 2 then begin
  print, 'Et en 2d'
   return
 endif
```

```
 If not(keyword_set(s)) then begin
  print, 'Que tu dis'
   return
 endif
 If N_elements(a) le 0 then a = 0
 print, (['Pas du tout','Un peu', 'Beaucoup','Enormément','A la folie'])(a<4)
 If a ne 0 then print, 'Et maintenant on fait quoi ?'
End
```

Entrées-Sorties

Ouverture et fermeture de fichiers

Les fichiers s'ouvrent de manière différente selon ce qu'on veut en faire :

Openr, LUN	Ouvre en lecture seulement.
Openw, LUN	Ouvre un nouveau fichier en lecture-écriture.
Openu, LUN	Ouvre un fichier existant en lecture-écriture.

LUN est le numéro d'unité logique qui désignera le fichier par la suite :

0, -1, et -2 sont réservés.
1 à 99 sont attribuables directement.
100 à 128 sont attribués par le mot-clef GET_LUN (ne pas les affecter manuellement).

Mots-clefs de OPEN :

/Append	place le pointeur à la fin du fichier.
open*, unit, /get_lun	choisit une LUN (entre 100 et 128) et la stocke dans UNIT.
/XDR	Utilise une représentation binaire externe, portable (lecture ou écriture, pas les deux à la fois)

Sous VMS seulement :

/Fixed	Format à longueur d'enregistrement fixe.
/Fortran	Utilise les mêmes CR que le Fortran pour un nouveau fichier.
/Segmented	Permet de relire des fichiers Fortran segmentés.

Open accepte un troisième argument qui est la taille des enregistrements en longueur fixe. Ce mot-clef n'est pas pris en compte sur d'autres systèmes, et n'est utile qu'en écriture.

Sous Unix seulement :

/F77_unformatted	Utilise les mêmes codes de structure que le Fortran Unix.

On ferme les fichiers explicitement avec :

Close, unit	ou /ALL pour fermer tous les fichiers ouverts
Free_lun, unit	Libère le numéro d'unité et ferme le fichier

Les fichiers sont fermés proprement quand on sort d'IDL, mais pas à la sortie ni à l'interruption d'une routine (les fichiers ouverts restent accessibles à tous les niveaux de procédures). Il faut donc explicitement fermer les fichiers dans les procédures qui les utilisent, dès qu'on n'en a plus besoin — et d'autre part prendre l'habitude de les ouvrir avec GET_LUN dans les routines pour éviter de possibles conflits de numéros d'unités.

La fonction EOF(unit) retourne 1 si on est arrivé à la fin du fichier. Elle permet de lire en boucle un fichier de longueur inconnue.

Accès aux fichiers

IDL cherche les fichiers uniquement dans le répertoire courant (par défaut, celui où l'on était en lançant IDL). On change le répertoire courant avec :

```
cd, '<chemin d'accès>'
```

Quand on change de répertoire dans une procédure, on ne revient pas à la situation initiale en en sortant. On peut gérer une pile de répertoires avec PUSHD et POPD, ou stocker le nom du répertoire qu'on quitte avec :

```
cd, '<chemin d'accès>', current=vieuxrep
```

PRINTD affiche le répertoire courant et la pile de répertoires. Pour accéder à d'autres répertoires, il faut préciser le chemin dans l'argument de OPEN (en utilisant la syntaxe du système d'exploitation de la machine). Les fonctions de recherche de fichiers utiles sont :

Findfile('<Fichier>',count=Nb) Retourne dans un tableau la liste des fichiers dont les noms correspondent, et leur nombre dans Nb. Les noms retournés sont les chemins d'accès relatifs aux fichiers, dans la syntaxe du système (ça peut être un moyen de rendre une routine indépendante du système). Accepte les abréviations du système dans les noms de fichiers ; cherche par défaut dans le répertoire courant.

Dialog_Pickfile() Ouvre une fenêtre de sélection de fichiers, puis ouvre le fichier choisi (en lecture ou écriture avec les mots-clefs READ et WRITE). La valeur retournée est le chemin complet du fichier sélectionné, selon la syntaxe du système. Interface paramétrable très sophistiquée, utilisable dans des routines interactives (remplace la routine PICKFILE depuis la version 5.1).

E/S Formatées

Sur fichiers :	Readf et Printf	
Sur clavier/écran :	Read et Print	
Sur chaîne :	Ch = 'coucou'+string(I)	Ecrit dans une chaîne
	StrPut, ch, ch1, 0	Insère ch1 au début de ch
	Reads, Ch, a	Lit depuis une chaîne

Les formats sont fournis comme chaînes entre guillemets (ce sont des chaînes) *et* parenthèses :

```
Printf, 1, format='(<format>)', truc
```

Ils sont similaires à ceux du Fortran. Parmi les choses supplémentaires :

/ Va à la ligne en sortie, passe à l'enregistrement suivant en entrée.
$ En sortie, supprime le saut de ligne final.
: En sortie, supprime une chaîne à la fin d'un format répétitif.

Les chaînes n'ont pas de longueur donnée, mais sont redimensionnées automatiquement. Quand on lit une chaîne depuis un fichier, toute la ligne est transférée dans la chaîne (le marqueur de fin de ligne n'est pas inclus).

E/S non formatées

Sur fichiers :	Readu, Writeu et Assoc
Sur clavier :	Get_kbrd (un seul caractère ; attend le suivant si /WAIT).

- Pour lire des chaînes dans un fichier non formaté normal, il faut les avoir déclarées avec la bonne longueur. Pour les fichiers XDR, ce n'est pas la peine.

- Dans les fichiers XDR, les tableaux d'octets (images) sont stockés et relus en une seule fois, ils doivent être déclarés avec la taille correcte avant lecture.

- Fichiers Fortran non formatés créés sous Unix : ouvrir avec /F77_UNFORMATTED, et lire le même nombre d'éléments à la fois que dans le programme d'écriture (à cause des CR et marqueurs de longueur cachés dans le fichier).

- Fichiers Fortran non formatés créés sous VMS : écrire de préférence des fichiers à longueur d'enregistrement fixe (ouverts avec open(…, recordtype=fixed) par un programme Fortran). Sinon, essayer en ouvrant le fichier avec le mot-clef /SEGMENTED. Relire le même nombre d'éléments à la fois qu'en écriture.

Point_lun, unit, n Pointe sur le n-ième octet du fichier (compté à partir de 0). Avec les fichiers VMS à taille d'enregistrement fixe, il faut pointer au début

d'un enregistrement.

Variables et fichiers associés

Cette fonction est utile pour relire des plans indépendants dans un même fichier (suites d'images notamment). À utiliser chaque fois que c'est possible, c'est la forme d'entrées/sorties la plus rapide — rapide surtout si la taille des tableaux est un multiple de 512 octets (taille du bloc machine).

```
openu, unit, /Get_lun, <nom de fichier>
A= assoc(unit, fltarr(n,m, /nozero), offset)
```

où offset est la taille en octets d'une éventuelle en-tête à sauter. Le fichier est considéré comme une suite de tableaux de dimensions n × m, lus et écrits indépendamment.

B=A(1)	Charge le 2e tableau du fichier dans B. Lit le fichier : si plusieurs accès à ce tableau sont nécessaires, le conserver dans une variable intermédiaire.
B=A	Associe B au même fichier que A.
z=A(i,j,tab)	Écrit (i,j) du tableau n° tab+1 dans z (scalaire). À n'utiliser que si l'on n'accède qu'à un seul élément : ça lit le tableau en entier de toutes façons.
A(3)= z	Écrit z dans le 4e tableau du fichier (doit avoir les mêmes dimensions).

Cette méthode permet de relire et de mettre à jour les enregistrements d'un fichier préexistant, mais pas de créer un nouveau fichier ni d'ajouter des enregistrements. On écrit un tel fichier de la manière habituelle ; sous VMS il faut spécifier une longueur d'enregistrement fixe (le fichier est illisible de cette façon sinon) :

```
openw, unit, <nom de fichier>, 512      ; Sous VMS, force l'écriture de blocs
for i=0,N do begin                      ; de 512 octets.
…
 writeu, unit, tab
endfor
```

Exemples de lecture de tableau

L'indice de colonne apparaît en premier sous IDL, comme pour indicer une image (comme en Fortran ; c'est l'opposé du C, du Pascal et de l'écriture matricielle) :

```
tableau(colonne, ligne)
```

Cette écriture correspond à l'ordre de stockage en mémoire (tous les éléments de la première ligne, puis ceux de la seconde…), il faut en tenir compte pour optimiser les accès disque.

L'ordre dans lequel le tableau est écrit rend la lecture plus ou moins facile :

1) A(1) A(2) A(3)… A(N)
B(1) B(2) B(3)…
…

```
a=(b=(c=fltarr(n,  /nozero)))
readf, unit, a, b, c    ; Pas de problème, mais il faut connaître les dimensions.
```

2) A(1) B(1) C(1)
A(2) B(2) C(2)
…

Attention, l'équivalent du Fortran :

```
for i=0 to N-1 do readf, unit, A(i), B(i), C(i)
```

ne marche pas, et renvoie une erreur "Attempt to store into an expression".
(Les paramètres de sortie d'une procédure, ici READF, doivent être des scalaires ou des tableaux, pas des éléments de tableau — ces paramètres sont passés par valeur et considérés comme expressions dans READF). Il faut faire :

```
tab=fltarr(3,N,  /nozero)
Readf, unit, tab
```

La variable i est dans tab(i,1:N). Pour réaffecter les variables à des tableaux séparés, faire :

```
tab=transpose(tab)
a=tab(*,1)  &  b=tab(*,2)  &  c=tab(*,3)
tab=0          ; récupère la place de tab
```

Après transposition, les valeurs d'un même tableau sont consécutives en mémoire, et le transfert est beaucoup plus rapide.

3) L'exemple précédent marche seulement si les variables sont de même type (fichier non formaté), où si elles sont toutes numériques (fichier ascii). Dans le cas contraire, le plus efficace est de créer un tableau de structures :

```
st = {ligne, As:1., Bs:1, Cs:' '}                    ; définit la structure d'une ligne
base = replicate(st, beaucoup)                       ; et la duplique
i=0
while (not eof(unit)) do begin              ; on peut aussi lire 'base' en une seule fois
```

```
        readf, unit, base(i)            ; si on connaît le nombre de lignes
        i=i+1                           ; du fichier
endwhile
As = base(0:i-1).As         ; on réaffecte les champs à des tableaux simples, ou bien…
base=base(0:i-1)…           ; on ne garde que la partie initialisée de la structure.
```

4) Si on connaît la structure des lignes mais pas leur nombre, on peut retrouver celui-ci avec la fonction FSTAT :

```
struc=FSTAT(unit)               ; renvoie les infos dans une structure
taille=struc.size               ;Taille en octets du fichier
longeur=struc.rec_len           ;Longueur des enregistrements pour des fichiers
                                ;       VMS à taille d'enregistrement fixe.
```

Attention avec les fichiers binaires : en Fortran les réels *et* les entiers sont par défaut sur 4 octets.

Dans un fichier ascii, IDL reconnaît les valeurs si elles sont séparées par des espaces. Si ce n'est pas toujours le cas, il faut donner un format explicite. Les problèmes de relecture sont fréquemment dus à l'absence de séparateur de champs quelque part dans le fichier.

Lecture et écriture d'images

Les images sont souvent stockées sous des formats particuliers, plus ou moins standard. IDL utilise habituellement des procédures du style READ_* et WRITE_* dont le premier argument est un nom fichier, le deuxième un nom de variable et les suivants des paramètres qui dépendent du type d'image (avant la version 5.0, les routines TIFF_READ et TIFF_WRITE faisaient exception à cette nomenclature ; par ailleurs, certaines routines de lecture sont des fonctions, pas des procédures). Par exemple :

```
Read_pict, 'Immagine.pict', ima, R,G,B
Write_pict, 'Immagine.pict', ima, R,G,B
```

lisent et écrivent en format PICT une image et sa table de couleurs. Des routines similaires existent en GIF, JPEG, SR… Les formats 8 bits (le Gif notamment) posent des problèmes à l'écriture quand on ne prend pas de précaution : le haut de la table de couleurs est fréquemment saturé quand on relit le fichier sur un autre système (voir le document sur les graphiques IDL). Les formats dits « scientifiques » (HDF, CDF et netCDF) possèdent leurs propres instructions d'interface. Les format de mises en page (PostScript, HP-GL, PCL, CGM) sont gérés par des pilote spécifiques.

Des routines contributives donnent accès à d'autres formats spécialisés, et leur syntaxe est souvent différente. Les plus importantes sont celles de la bibliothèque

ASTRON, qui contient de nombreuses fonctions de base pour la gestion du format FITS et de ses dérivés (c'est le format habituel des images télescopiques) :

```
ima=Readfits('PicDuMidi.fits',EnTete)     ; lit l'en-tête et l'image
writefits,'PicDuMidi.fits', image, EnTete ; les écrit
date=sxpar(EnTete, 'DATE-OBS')            ; renvoit la date d'acquisition
```

L'en-tête FITS décrit la structure des données et les conditions d'acquisition à l'aide de mots-clefs standard dans l'en-tête du fichier. Ces routines permettent toutes les manipulations courantes des images FITS. La fonction SXPAR permet notamment de récupérer la valeur associée à un mot-clef dans l'en-tête, sans avoir à se soucier de la structure exacte de celui-ci. ASTRON contient par ailleurs une autre procédure de lecture/écriture FITS plus puissante (MRDFITS) qui gère l'ensemble du standard FITS (les fichiers peuvent contenir également des tables, des cubes spectraux… en plus des images).

Une routine similaire existe pour la lecture de fichiers au format PDS (standard pour les images planétaires des missions spatiales) ; la structure des données est également définie dans un en-tête précédent le champ de données, mais les règles d'écriture de cet en-tête sont différentes de celles du FITS. Cette routine READPDS est disponible sur le site Web du nœud Petits Corps du PDS, et peut poser quelques problèmes (voir la page du Centre de données planétaires de l'IAS, http://www.ias.fr/cdp/infos/idl/idl.html).

Enfin, ASTRON contient des routines de lecture d'images MIDAS (le logiciel de traitement d'images de l'ESO), AIPS (logiciel de traitement d'images radio), HST…

Vecteurs et matrices

Définition, initialisation

- IDL gère les tableaux comme des variables, toutes les opérations peuvent donc être effectuées sans faire de boucle sur les indices. On y gagne en simplicité d'écriture et surtout en temps d'exécution.

- Pour la petite histoire, le mode d'indexation des tableaux a changé avec la version 5.0. Les indices étaient jusque alors donnés entre parenthèses, on peut maintenant les fournir entre crochets. Cette nouvelle méthode est fortement recommandée par

l'éditeur, elle permet d'éviter certaines confusions entre tableaux et fonctions. Mais les routines écrites ainsi ne tournent absolument pas sur les versions plus anciennes d'IDL.

- Le dimensionnement est automatique dans les expressions, et est donc superflu :

```
A = B + C
```

où B et C sont des matrices de mêmes dimensions impose les dimensions de A. Par contre le dimensionnement explicite est obligatoire pour les affectations élément par élément (sinon la variable est déclarée scalaire à la première utilisation, et les affectations suivantes entrent en conflit avec cette définition).

- Les déclarations sont du type Fltarr(dim), Lonarr(dim)… et précisent le type du tableau. Les tableaux sont mis à zéro par défaut, ce qui est long et parfois inutile :

```
A=Bytarr(40,  /Nozero)
```

définit un tableau de 40 octets et inhibe l'initialisation à zéro (les valeurs initiales sont quelconques).

```
sz=Size(A)              ;retourne un tableau décrivant les caractéristiques de A
dim=sz(0)               ; Nb de dimensions de A
EltDim=sz(1:dim)        ; le nombre d'éléments dans chaque dimension
type=sz(dim+1)          ; code du type de variable
Nelt=sz(dim+2)          ; nb total d'éléments

Nelt=N_ELEMENTS(A)      ;retourne le nombre total d'éléments de A
```

- On peut adresser directement le résultat vectoriel d'une fonction :

```
dim  =  (size(A))(0) ; équivalent à sz=Size(A) & dim=sz(0)
```

- Les éléments de vecteurs sont toujours numérotés à partir de l'indice 0. Ils sont par contre toujours définis en donnant le nombre d'éléments (indice max + 1).

```
A=fltarr(10)        ; définit A(0) à A(9), c'est à dire A(0:9)
```

- Les indices de tableaux sont inversés par rapport à la notation matricielle classique. Il faut en tenir compte pour écrire des opérations sur des matrices :

```
A(colonne, ligne)
```

Les indices sont donnés comme A(2:7), A(2:*), A(*:7), A(*,5)…
A(B) est l'ensemble des éléments de A dont les indices sont dans B…
Si A est un tableau N × M, B(i,j) = A est équivalent à B(i:i+N-1, j:j+M-1) = A. Si on

précise les limites en indices, il faut que les dimensions coïncident.
Un tableau peut être indicé avec un seul nombre, quelle que soit sa dimension (y compris si c'est un scalaire) : ses éléments sont alors comptés ligne après ligne A(0:N,0) puis A(0:N,1)…

- Les tableaux constants sont entrés avec leurs valeurs entre crochets :

[1,3,7] (première ligne)
[[11,12,13],[21,22,23]] (première puis deuxième ligne)

En particulier :
[1,2] est un vecteur-ligne
[[1],[2]] est un vecteur-colonne
La fonction TRANSPOSE transforme d'un type à l'autre.

- Les concaténations se font sur le même mode. Si A et B sont des vecteurs :

[B,A] colle A après B
[[B],[A]] B en première ligne, A en deuxième ligne (doit avoir la même dimension).
transpose([[B],[A]]) met A et B côte à côte.

Si A et B sont des matrices, les dimensions doivent correspondre :
[B,A] met A après B (allonge les lignes, qui doivent être en nombre égal)
[[B],[A]] met A au-dessous de B (allonge les colonnes).

- Les opérations sont effectuées élément par élément : A + B donne $(A_{ij} + B_{ij})$

- Il existe depuis la version 4.0 deux multiplications matricielles. La multiplication proprement dite est notée ## et s'utilise comme dans l'écriture mathématique usuelle. Dans les versions antérieures, seule existe la multiplication de tableaux-images, notée #. A cause de l'inversion de notation, les expressions sont transposées :

A # B donne ce qu'on écrit maintenant B ## A
(avec A(N,M) et B(M,P), on obtient C(N,P))

Les routines de calcul matriciel sont écrites pour gérer la notation classique, et attendent des arguments qui sont des matrices au sens habituel (depuis la version 4.0 seulement).

- XvarEdit est un outil de d'édition de tableaux interactif, utile pour modifier quelques valeurs dans une image.

- Il existe des moyens rapides d'initialiser des vecteurs :

Replicate(2., 100, 100) est une matrice réelle 100 × 100 dont tous les éléments valent 2.

Findgen(10) est un vecteur réel de dimension 10 dont les éléments valent leur indice (de 0 à 9). Des fonctions similaires existent pour tous les types de variables.

Indgen(10,20) est une matrice entière 10 × 20 dont les éléments valent leur indice mono-dimensionnel (de 0 à 10 × 20 - 1).

Dist(10,20) est une matrice dont les éléments valent leur distance euclidienne au centre du tableau (en fait, les quadrants du tableau sont inversés). Si un seul argument est fourni, renvoie une matrice symétrique.

Make_array(Nx,Ny) Définit un tableau de dimensions données par des variables.

Reform(tab,d1,d2…) Redimensionne un tableau sans modifier son contenu (en particulier, tue les dimensions dégénérées).

Opérations sur les tableaux et les matrices

Certaines routines mathématiques d'IDL reprennent des algorithmes classiques des bibliothèques de calcul (celles de Numerical Recipes). Avant la version 4 d'IDL, ces routines exigeaient des arguments tableaux suivant la notation matricielle (indice de ligne en premier, comme en C). Il fallait donc transposer les tableaux avant d'appeler ces routines (toutes celles dont le nom est en NR_* en version 3.6 ; cette transposition est inutile pour des matrices symétriques réelles). Avec la version 4 les routines ont été réécrites pour surmonter cette contrainte, et elles ont changé de noms.

On peut indicer directement le résultat d'une fonction qui retourne un tableau :

```
print, (size(Tab))(1)
```

On effectue facilement certaines opérations vectorielles ou matricielles :

```
Total(A*A)      est le module carré
Total(A*B)      est le produit scalaire
Crossp(A,B)     calcule le produit vectoriel

invert(A, res)  inverse une matrice carrée ; res contient le statut du résultat
determ(A)       calcule le déterminant d'une matrice carrée
transpose(A)    fournit la transposée de A (vecteur ou matrice). On peut
```

également retourner les tableaux et vecteurs à l'aide REVERSE et ROTATE (voir le document Graphiques IDL).

```
A=fltarr(N,N)                 Définit une matrice identité N × N
A(indgen(N),indgen(N))=1.     (équivalent à IDENTITY(N) en version 5)

A # replicate(1.,M)           est la somme en lignes (deuxième indice…) de A(N,M)
replicate(1.,N) # A           est la somme en colonnes (premier indice…)

A # replicate(1.,N)  est une matrice dont les N lignes sont égales au vecteur A(P)
replicate(1.,N) # A  est une matrice dont les N colonnes sont égales au vecteur A(P)
```

Pour dupliquer une matrice 2D dans un tableau 3D, il faut utiliser REBIN (ou la fonction REPLICAS de la bibliothèque FKK) :

```
sz=size(a)
b=rebin(a,sz(1),sz(2),3)          ;Les 3 plans du tableau b contiennent a
```

Exemple : calcul d'une matrice Nx × Ny dont les éléments donnent la distance au centre.

```
i0=(Nx-1.)/2.                ; Coordonnées du centre
j0=(Ny-1.)/2.
d0=findgen(Nx)               ; Distance en x, recopiée sur chaque ligne
d0=abs(d0-i0)
d1=d0#replicate(1,Ny)
d2=d1^2
e0=findgen(Ny)               ; Même chose en y
e0=abs(e0-j0)
e1=replicate(1,Nx)#e0
e2=e1^2
distE=sqrt(e2+d2)            ; Distance euclidienne
```

La distance est calculée depuis le centre du tableau. Le même résultat est obtenu pour des matrices de dimension impaire en faisant :

```
distE = dist(Nx,Ny)
distE=  shift(distE,(Nx-1)/2.,(Ny-1)/2.)
```

La fonction SHIFT décale les lignes sans interpoler. Si le nombre en est pair, l'origine des distances n'est pas au centre mais sur une des lignes qui l'entourent.

Lorsqu'on manipule de grands tableaux on a intérêt à utiliser des fonctions optimisées pour les affectations, qui sont beaucoup ainsi plus rapides et moins exigeantes en mémoire. En particulier :

Blas_axpy Ajoute le multiple d'un tableau à un autre tableau.

Replicate_InPlace Remplace certains éléments d'un tableau par une constante.

Statistiques

Les fonctions statistiques courantes sont décrites dans le document Graphiques IDL. Parmi les autres fonctions utiles :

Min et Max — Renvoient les valeurs extrêmes d'un tableau (voir le document Graphiques IDL pour les options). Pour comparer plusieurs variables, faire :

```
mini = min([tab, a, c1], /max=maxi)
                    ; si leurs dimensions sont différentes
mini = a < b < s    ; si elles sont identiques
```

Correlate(A,B) — Calcule le coefficient de corrélation ou la covariance entre les vecteurs A et B, ou entre les colonnes d'un seul argument matriciel.

Standardize(A) — Normalise un tableau de variables (retire la moyenne et divise par l'écart-type pour chaque colonne).

Sort(A) — Tri ascendant des éléments d'un tableau (faire reverse(sort(a)) pour le tri descendant). A(sort(A)) est le tableau trié.

Par ailleurs, la version 5 d'IDL contient des fonctions calculant les distributions statistiques courantes (BINOMIAL, GAUSSINT…) ou comparant deux distributions (FV_TEST, TM_TEST, RS_TEST…).

Interpolation et lissage

Smooth(X,N) — Moyenne mobile du tableau X dans une fenêtre de largeur N (impair). Pour lisser un vecteur seulement à l'affichage, utiliser le mot-clef NSUM de PLOT.

Interpol(V,N) — Retourne un vecteur à N points linéairement interpolé à intervalles réguliers. Suppose que les valeurs de V sont régulièrement espacées.

Interpol(V,X,U) — Interpolation linéaire irrégulière d'un vecteur, ou d'un vecteur irrégulier. X contient les abscisses des points de V, U celles où on interpole.

Interpolate(Tab,X,Y,Z) — Retourne les valeurs linéairement interpolées de Tab en X,

(X,Y) ou (X,Y,Z) selon la dimension de Tab. Avec le mot-clef /GRID retourne une interpolation régulière sur une grille.

Spline(X,Y,U,s) Interpole la fonction Y(X) aux points U avec un spline cubique. La valeur de s (de 0.01 à 10) détermine l'erreur tolérée aux points d'observation X. X et U doivent être croissants. Quand on fait des interpolations successives des mêmes données, il vaut mieux utiliser SPL_INIT une fois au départ, puis SPL_INTERP à chaque calcul.

Spline_P,X,Y,u,v Interpole la courbe (X,Y) par une courbe (u,v) — le spline est calculé dans les deux dimensions. La routine décide du nombre de points, à moins qu'on ne le fixe avec INTERVAL. C'est essentiellement une routine de lissage de courbes à 2D, à finalité graphique.

Les interpolations de tableaux (grandissement et maillage) sont décrites dans le document Graphiques IDL à propos des transformations géométriques d'images.

Ajustements

Les versions 4 et 5 d'IDL contiennent de nombreuses fonctions d'ajustement, et d'estimation de leur qualité ; parmi celles-ci les plus utiles sont les fonctions de distribution du Chi2, CHISQR_PDF et CHISQR_CVF (retournent la probabilité d'avoir une valeur supérieure à une limite, et la limite correspondant à une certaine probabilité). Des fonctions similaires existent pour les distributions gaussienne, F et T.

Linfit(X,Y) Ajustement linéaire de Y(X). On peut fournir une erreur de mesure avec le mot-clef SDEV, et récupérer le Chi2 et les erreurs sur les paramètres dans les variables introduites par CHISQ et SIGMA. La fonction LADFIT est similaire, mais moins sensible aux points aberrants.

Comfit(X,Y,A) Ajuste Y(X) à l'aide de six formes analytiques courantes (exponentielle, Log carrée…). A contient une première estimation des paramètres. Les gaussiennes avec continuum s'ajustent avec GAUSSFIT.

Poly_fit(X,Y,N,Yf) Ajuste Y(X) avec un polynôme de degré N, et renvoie un vecteur de N+1 coefficients. On peut récupérer la fonction ajustée dans Yf,

ainsi que les erreurs dans des arguments supplémentaires.

PolyfitW(X,Y,W,N,Yf) Identique à la précédente, mais permet de pondérer les observations à ajuster avec le vecteur W. Initialiser W=1./Y pour une pondération statistique (toutes les observations comptent autant, quelle que soit la valeur observée), ou W=1./sigma(X) pour une pondération selon l'incertitude de mesure (les mesures les moins bonnes comptent moins).

Fitexy, X,Y,a,b Ajustement linéaire de variables bruitées. Renvoie les paramètres de la droite a et b (dans ASTRON).

Sfit(Tab,N,Kx=Var) Ajuste une surface Tab(x,y) par un polynôme en (x,y) de degré N. Retourne le tableau ajusté, et les coefficients dans la variable VAR.

Regress(X,Y,W,Yf,A0) Ajustement linéaire sur plusieurs variables de Y(X) — Y est un vecteur, X est un tableau de variables indépendantes.

Curvefit(X,Y,W,a,s,Function_name="fonct") Ajustement général non linéaire. Ajuste X(Y) avec la fonction spécifiée, contenue dans le fichier FONCT.PRO — qui doit être écrite suivant des règles précises. Les fonctions SVDFIT et LMFIT permettent également ce type d'ajustement.

Mpfit Est une alternative à curvefit, plus souple et plus solide paraît-il (bibliothèque de l'Université du Wisconsin).

Équations non linéaires / racine de fonctions

La recherche des zéros d'une fonction est équivalente à la résolution d'un système d'équations. IDL inclut plusieurs routines adaptées à différents cas. De façon générale, les fonctions non-linéaires peuvent admettre différentes solutions, qui sont donc très sensibles à une estimation initiale.

FZ_roots(Coef) Renvoie les racines d'un polynôme à coefficients réels ou complexes (nommée NR_ZROOTS dans la version 3.6).

FX_root(X,"fct") Renvoie les racines d'une fonction univariée réelle ou complexe. fct.pro contient la définition de la fonction dont on cherche les zéros, X est un vecteur à trois éléments contenant trois estimations de la solution. Cette fonction ne peut pas être utilisée pour chercher

simultanément plusieurs valeurs d'une même fonction (f(x)=y), il faut faire l'appel dans une boucle. Broyden marche plutôt mieux et n'est pas plus long.

Broyden(X,"fct") et Newton(X,"fct") Deux algorithmes légèrement différents pour le cas multivarié (méthodes quasi-newtoniennes). On peut ajuster les critères de convergence et la précision du calcul. Les deux routines diffèrent par leur réaction à l'imprécision de la première approximation donnée dans X. Elles se nomment respectivement NR_BROYDN et NR_NEWT dans la version 3.6.

Minimisation de fonction

La recherche des minima d'une fonction passe par la recherche des zéros du gradient de la fonction. IDL inclut deux routines de minimisation de fonctions multivariées (à ne pas utiliser sur le Chi2 pour ajuster une fonction, utiliser plutôt curvefit) :

DFPMin Requière la forme analytique de la fonction et de son gradient, ce qui peut être difficile à écrire.

Powell Apparemment moins précise, elle ne requière que la forme analytique de la fonction et une direction de recherche initiale.

Amoeba Moins rapide que Powell, mais plus précise dans certains cas.

Analyse de Fourier

fct2=FFT(fct,dir) Calcule la transformée de Fourier d'une fonction ou d'une image, directe si dir =–1, inverse si dir=1 (ces transformations ne sont pas symétriques). Les premiers éléments de chaque dimension représentent les basses fréquences ; à deux dimensions il faut donc rétablir l'ordre de la matrice avec SHIFT comme pour la fonction DIST ci-dessus. Attention : cette routine fonctionne quelles que soient les dimensions de l'image de départ, mais n'utilise un algorithme rapide que si ce sont des puissances de 2 (obligatoire en pratique) ; le résultat est complexe, et génère facilement des tableaux énormes quand on travaille sur des images.

Wtn(arr,N)	Calcule une décomposition en ondelettes en utilisant les fonctions de Debauchy avec N composantes (4, 12 ou 20). Les dimensions de ARR doivent être des puissances de 2 (vecteur ou matrice). Cette fonction s'appelle NR_WTN dans la version 3.6.
Convol(fct,noy,ech)	Convolue une fonction par un noyau. La convolution mathématique s'obtient en imposant CENTER=0 (c'est une fonction de traitement d'image sinon) ; le résultat est divisé par le facteur ech (faire ech=TOTAL(noy) pour moyenner). Le résultat a le type de l'image : il faut d'abord la transformer en complexe si le noyau est complexe.
Blk_Con(Filtre, Signal)	Convolue Signal par une réponse impulsionnelle.
Digital_filter	Calcule le noyau de convolution correspondant à un filtre défini par ses fréquences de coupure (à une dimension seulement).
Hanning	Fabrique un cosinus de la taille voulue à une ou deux dimensions. Utile pour définir des fenêtres de convolution.

La bibliothèque WWB (Wavelets WorkBench) contient de nombreuses routines de traitement du signal par ondelettes à une ou deux dimensions, disponibles à travers une interface graphique.