IDL 講習会 (初級編)

2021年9月29日(水), **30日(木)**, 10月6日(水), 7日(木)
Zoom オンライン開催
主催 天文データセンター
講師 巻内 慎一郎 (国立天文台 天文データセンター)

二日目

- ヘルプシステム
  - マニュアル
  - リファレンスの使い方
  - IDL セッションからのヘルプ使用
  - 参考になるサイト
- ライブラリルーチン
  - IDL のライブラリ
  - The IDL Astronomy User's Library
  - Coyote IDL Program Libraries
  - Markwardt IDL Library
- IDL の基本文法
  - 大文字と小文字
  - IDLで使用される特殊文字(;, $, &)
  - 予約語
  - その他の特徴
- IDLの基本的なコマンド
  - .reset_session
  - .compile
  - return & retall
  - save & restore
  - print
  - Help
  - Journal 機能
  - OSのコマンド実行(SPAWN)
  - Dollar Sign ($)
- 変数・定数・データ型
  - 変数とデータ型
  - 文字列定数
  - 変数の作成と注意
  - 型変換と型別配列生成の関数
  - 型変換の例
    - 明示的に変換
    - 自動変換)
  - 浮動小数点数から整数への変換関数
  - データ型の判定 SIZE 関数
  - 特別な値 Null
  - 特別な浮動小数点数 NaN, Inf
  - Math errors を取り除く (finite関数)
  - 文字列操作

- 配列
  - IDLの配列 (Array)
  - 他言語の配列との比較
  - 配列の作成
    - 要素を直接指定する
    - 配列生成関数を使う
  - 配列の結合
  - 配列の指定方法(部分配列)
  - 配列の変形
    - Reform 関数
    - Transpose 関数
  - 配列操作でよく使う機能
    - Shift 関数
    - Where関数
      - 条件の書き方(比較演算子・論理演算子)
    - SORT関数
  - 配列の演算
  - 行列演算
- 構造体
  - IDLの構造体 (Structures)
  - 無名構造体 (Anonymous structures)
  - 記名構造体 (Named structures)
  - 構造体の操作
  - 構造体についての注意点
- カーブフィッティング
  - フィッティングルーチン
  - LINFIT関数(線形)
  - GAUSSFIT関数(ガウシアン)
  - LMFIT関数(任意関数)
    - LMFIT() を使ったテスト
  - フィッティング処理の注意点

■ ヘルプシステム

マニュアル

- IDL のマニュアルはオンラインヘルプの形で IDL 本体と共に配布されている
- 以前は分厚い本が何冊も付いていたが(おそらく)廃止されている
- オンラインヘルプはブラウザアプリケーション(Firefox など)で開かれる

```
$ idlhelp                ← 端末から IDL ヘルプを呼び出すコマンド
```

- チュートリアル的な文書(Getting Started)から、言語の詳細な解説、コマンドリファレンスまで揃っている。(全部を読むのは困難)
- 検索機能が付いているので、調べたいコマンド名やキーワードを入力して検索する使い方が便利
- 関連項目へのリンクも張られている

リファレンスの使い方

「Index」タブを選択して下の欄に検索ワードを入力

検索ワードに一致する項目がリストされる(インクリメンタルサーチ)

リファレンスページが表示される

主な項目はリンクをたどってジャンプできる

マニュアル全文に対する検索窓

IDL

Search

Contents | Index

print

- % character, printf-style format code
- DIALOG_PRINTERSETUP function
- DIALOG_PRINTJOB function
- Hewlett-Packard Printer Control Language (PCL
- IDL_Print()
- IDL_PrintF()
- IDLgrPrinter
- Print method
- PRINT procedure
- PRINT_FILE keyword
- PRINTD procedure
- printer device
- printer object
- PRINTF procedure
- printf-style format code
- printing
- PRINTNAMES example routine
- ptr_print.pro

PRINT/PRINTF

The two PRINT procedures perform formatted output. PRINT performs output to the standard output stream (IDL file unit -1), while PRINTF requires a file unit to be explicitly specified.

Note: IDL uses the standard I/O function sprintf to do its formatting. Different platforms implement this function in different ways, which may lead to slight inconsistencies in the appearance of the output. In most cases, specifying an explicit output format via the FORMAT keyword allows better control over the appearance than simply using the default formatting.

Format Compatibility

If the FORMAT keyword is not present and PRINT is called with more than one argument, and the first argument is a scalar string starting with the characters "$(", this initial argument is taken to be the format specification, just as if it had been specified via the FORMAT keyword. This feature is maintained for compatibility with version 1 of VMS IDL.

Examples

To print the string "IDL is fun." enter the command:

```
PRINT, 'IDL is fun.'
```

To print the same message to the open file associated with file unit number 2, use the command:

```
PRINTF, 2, 'IDL is fun.'
```

Syntax

PRINT [, *Expression*$_1$, ..., *Expression*$_n$]

PRINTF [, *Unit*, *Expression*$_1$, ..., *Expression*$_n$]

Keywords: [, AM_PM=[*string*, *string*]] [, DAYS_OF_WEEK=*string_array*{7 names}] [, FORMAT=*value*] [, /IMPLIED_PRINT] [, MONTHS=*string_array*{12 names}] [, /STDIO_NON_FINITE]

Arguments

© 2015 Exelis Visual Information Solutions, Inc., a subsidiary of Harris Corporation. All Rights Reserved. This information is not subject to the controls of the International Traffic in Arms Regulations (ITAR) or the Export Administration Regulations (EAR). However, this information may be restricted from transfer to various embargoed countries under U.S. laws and regulations.

「Contents」タブからはマニュアル全体を項目からたどって読むことが出来る

IDL セッションからのヘルプ使用

- IDL を使用中に、簡単にオンラインヘルプを呼び出す事が可能。コマンドラインから ? (クエスチョンマーク)に続けて調べたいコマンド名などのキーワードを入力する

使い方: ? キーワード

(例)

```
IDL> ? print
```

! キーワードを付けずに「?」一文字の場合は、ヘルプシステムのトップページが開く

参考になるサイト

IDL の開発・販売元の会社のページ

https://www.harrisgeospatial.com/docs/using_idl_home.html

- 最新のマニュアル類が公開されている
- サポートのためのフォーラムページや、第三者作成のライブラリ情報などもある

Coyote's Guide to IDL Programming

http://www.idlcoyote.com/

- 個人による、たぶん一番有名な IDL のページ
- IDL のパワーユーザ・エキスパートである管理者による、IDL プログラミング解説、プログラム例、FAQ、Tips など、膨大な量の役立つ資料が公開されている
- かつては新しい情報も頻繁に追加されていたが、ご本人は2014年に引退を宣言されて、現在更新はストップしている
- しかし今でも非常に有用な情報の宝庫

■ ライブラリルーチン

IDL のライブラリ

- 開発元から提供されている IDL コマンドの多くは IDL 言語で書かれたライブラリルーチンとなっている
- IDL で書かれたプログラム(バイナリではなく、テキスト形式のソースコード)なので、内容を確認したり、それを参考にしたプログラムを自作したり出来る
- 開発元以外の、IDL の個人ユーザ、あるいはユーザコミュニティによって作成されて、一般に公開されているライブラリルーチンも数多く存在する

The IDL Astronomy User's Library (AstroLib)

https://idlastro.gsfc.nasa.gov/

- 天文学関係のデータ処理や計算に用いられる機能を中心とした IDL のプログラム集
- たとえば、天球面座標の計算や、FITS形式のデータファイルの読み書き、作成などを行うツールが揃っている
- 個々の望遠鏡や観測機器に固有なソフトウェアは基本的に含まない
- 天文分野に限らない、一般的に便利なツールも含まれる
- すべてをダウンロードしても良いし、必要な個々のプログラムだけを持ってきて使うことも出来る
- 天文台の多波長データ解析システムではデフォルトで使用できるようになっている(インストール済みでパスが通っている)
- 頻繁に更新が続けられている
- アップデートによるバージョン間の互換性の問題が発生することもある

Coyote IDL Program Libraries

http://www.idlcoyote.com/documents/programs.php

- Coyote's Guide の管理者が作成した、たいへん有用なライブラリツールの数々
- グラフィックス関係を始めとして、データ入出力関係、カラーハンドリングやポストスクリプト作成、その他、様々なユーティリティーツールなど
- IDL ネイティブでは使いにくかったり、分かりにくかったりするコマンドや操作を、より簡単に、より高機能に使えるようにするプログラムが多数提供されている
- Coyote ライブラリの一部は、サブセットライブラリとして前述の AstroLib に含まれており、一緒に配布されている

Markwardt IDL Library

http://cow.physics.wisc.edu/~craigm/idl/idl.html

- Curve Fitting 関係のプログラムを始めとして、数学関係や、データの読み書きユーティリティー、プロットツールなどが公開されている
- IDL による、もっとも堅牢性 (robustness) と信頼性 (reliability) が高いフィッティング関数として広く利用されている
  - ✓ **MPFIT** - Robust non-linear least squares curve fitting など

■ IDL の基本文法

大文字と小文字

- IDL では大文字と小文字を区別しない
- コマンドやオプション、キーワードの指定、変数名など、大文字と小文字は好みで(見やすさ、分かりやすさを考えて)使い分ければ良い

```
; たとえば下記はどれも同じ
IDL> print, a, format='(f)'
IDL> PRINT, A, FORMAT='(F)'
IDL> Print, a, Format='(F)'
```

↑ printコマンドによる表示。a (A) は表示したい変数名。formatオプションで表示形式を指定。

- 一方、Linux 環境では大文字と小文字は区別される。
  そのため、IDL からファイル名を扱うときには注意が必要

IDLで使用される特殊文字

- セミコロン (;)
  - コメント開始文字。同じ行の中で ; 以降はすべてコメントとして扱われる
- ドル記号 ($)
  - 継続文字。行末に $ を書くことによって、コマンドを次の行に続けて書くことが出来る。通常は1コマンドは一行に書く
  - 行頭に用いる場合は、OS のコマンド実行(後述)
- アンパサンド (&)
  - 複数のコマンドを & でつなげて書くと、複数コマンドを一行で書くことが出来る。通常は1行1コマンド

```
IDL> ; 特殊文字の使用例
IDL> ; このようにセミコロンの後ろはコメントになる
IDL> plot, x, y; 行の途中からコメントを書いた例
IDL>
IDL> ; 下の例は x と y への代入処理を1行にまとめた
IDL> x=indgen(100)*0.1 & y=sin(x)*x
IDL>
IDL> ; 下の例は、本来1行のコマンドを2行に分けて書いた
IDL> plot, x, y, psym=-4, yrange=[-8,8], $
IDL> ystyle=1, title='Test Plot'
```

予約語

- 一般に、変数名やユーザ作成のプログラム名として、既存のプログラム名などを使うのは避けるべき
- とくに、次の語(予約語)の使用は文法エラーとなり、禁止されている

AND	BEGIN	BREAK	CASE	COMMON	COMPILE_OPT
CONTINUE	DO	ELSE	END	ENDCASE	ENDELSE
ENDFOR	ENDFOREACH	ENDIF	ENDREP	ENDSWITCH	ENDWHILE
EQ	FOR	FOREACH	FORWARD_FUNCTION	FUNCTION	GE
GOTO	GT	IF	INHERITS	LE	LT
MOD	NE	NOT	OF	ON_IOERROR	OR
PRO	REPEAT	SWITCH	THEN	UNTIL	WHILE
XOR					

その他の特徴

- 基本書式はカンマ(,)区切りの構文
  - IDL> command, arg1, arg2, …
- 変数の型宣言を最初に行う必要が無い
- 多くのコマンド(プログラム)は配列入力に対応
  - y=sin(x) と書いたとき、x はスカラーでも配列でも構わない
- 実行コマンド(.compile など)や、プログラムのキーワードオプションの名前は省略(短縮)して使用可能 (例えば .comp だけでOK)

- IDL の基本的なコマンド

- プログラムや個々のコマンドの実行など、IDL を操作している間に使う機会が多いと思われる基本的なコマンドを説明する。

.RESET_SESSION

- ドット(.)コマンド(IDLの制御など特別なコマンド)のひとつ
- 現在の IDL セッションの状態(のほとんど)を起動直後の状態に戻す
- メモリ上に展開されていた変数やコンパイルされたプログラムのバイナリ情報がクリアされる
- 一度 exit してから IDL を再スタートさせることなく、環境をリセットできる
- 変数が増えすぎた、メモリが圧迫されている、複数のファイルを修正したがひとつずつコンパイルし直すのが面倒、などの場合に実行すると良い
- startup ファイルも実行される
- 実行レベルはメインレベル($MAIN$)まで戻る
- ドット(.)コマンドは短縮入力が可能。たとえば .reset だけでもOK

.COMPILE

- プログラム(プロシージャ, 関数)をコンパイルする。コンパイルしたプログラムはメモリ上に保持される
- 同じ IDL セッションの中で、同じプログラムを2回目以降に使用する際は、コンパイル処理は行われず、メモリから直接実行される
- そのため、自作プログラムを修正した後そのまま実行すると、メモリ上に残っていた修正前のプログラムが実行されることになるで、.compile コマンドで明示的にコンパイルし直すことが必要になる

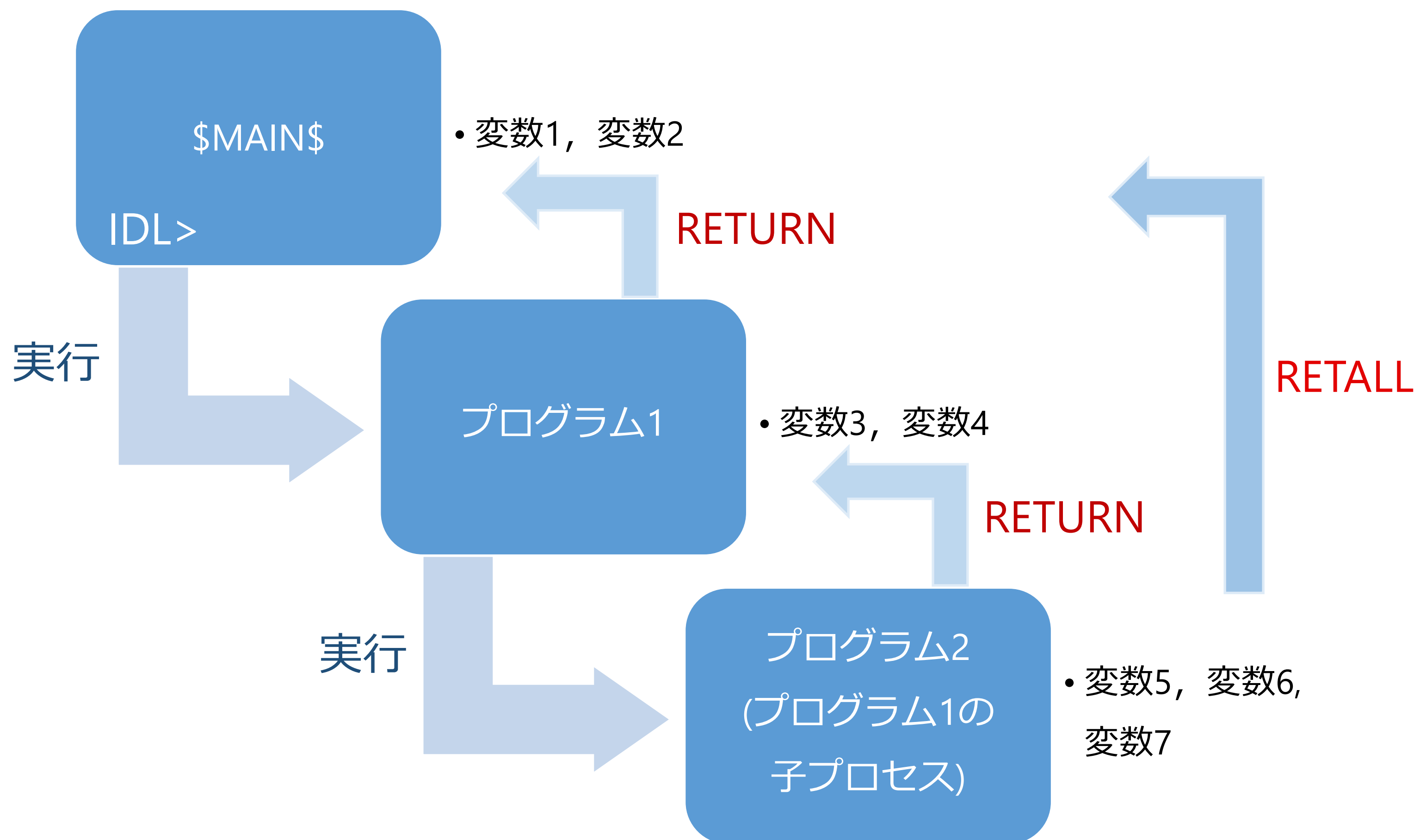
```
IDL> .compile File1[, File2, File3, ...]
```

(注) .compile とコンパイルするファイル名の間にはカンマ(,)は必要ない

RETURN & RETALL

- return コマンドは、ひとつ上のプログラムレベル(親プロセス)にコントロールを戻す
- 関数 (function) の内部で、引数を付けて使用する場合は、その値を呼び出したプログラムへ返す(返値)
- retall コマンドは、プログラムレベルを一番上位のメインレベル($MAIN$)まで戻す。return をメインレベルまで繰り返すのと同じ
- エラーによりプログラムが途中で止まった場合、そのまま次の処理を行おうとしても、存在するはずの変数が見えなかったりする。これはプログラムレベルが、中断した位置に止まっているため。この場合、return や retall で元のレベルまで戻る必要がある
- 逆に、デバッグのため変数の値などを調べるには、レベルを戻す前に行う必要がある

(補足) プログラム実行レベルの階層構造例



SAVE & RESTORE

- IDL では、セッションの間に作成された変数や、コンパイルされたプログラムは、メモリ上に展開されているが、セッションを終了すると消えてしまう
- それらをバイナリイメージとして保存できる
- たとえば、繰り返し使用したい、一時的に保存しておきたい、他者と共有したい、といったデータ(変数)を SAVE ファイルとして保存しておける
- 保存したバイナリプログラムはライセンス不要のIDL バーチャルマシン(VM)で実行することも可能。(ただし制限あり)

- IDL の SAVE ファイルは save コマンドで保存して、restore コマンドでメモリ上にリストアする
- SAVE ファイルの名前は何でも構わないが、通常は *.sav としておくのが良い

```
; メモリ上の変数をすべて保存する
IDL> save, filename='hozon.sav'
```

```
; 変数 var1, var2 を保存する
IDL> save, var1, var2, filename='hozon.sav'
```

```
; メモリ上に復元する
IDL> restore, 'hozon.sav'
```

❢ 上記のようにファイル名のみで指定した場合は、カレント作業ディレクトリ上で読み書きされる。他のディレクトリを使う場合はフルパスで指定することも可能

PRINT

- IDL でもっともよく使うコマンド(プロシージャ)
- 標準出力(ディスプレイ)に変数等を書き出す
- FORMAT オプションを付けることで書式設定を行い、出力結果を整形できる
- ファイルに書き出す場合は PRINTF プロシージャ

➢通常の使用のほか、たとえば、現在の変数の内容(値)を知りたい場合などに頻繁に使う。対話型セッション中だけでなく、実行プログラムの中で、変数に意図した値が入っているか確かめたい場合などにあらかじめ仕込んでおくなど、バグ取り用途でも多用する

❢ IDL 8.3 以降、コマンドラインでは Print 自動実行が可能になった (Implied Print)

PRINT の書式設定(FORMAT オプション)

- 基本的には FORTRAN 的。一方 C言語スタイルの書き方もサポートされている

基本形 [n]*FC*[+][-][*width*]
n: 繰り返しの数、FC: Format Code, +: 正記号付加、-: 左寄せ、width: 表示桁数

Format Code	内容
A	文字列に変換
F, D, E, and G	浮動小数点数に変換
B, I, O, and Z	2, 10, 8, 16 進数に変換

```
IDL> print, 1.23, format='(F)'
      1.2300000
IDL> print, 1.23, format='(F8.3)'
   1.230
IDL> print, 1.23, format='(E10.3)'
 1.230E+00
IDL> print, 1.23, format='(A)'
      1.23000
```

HELP

- IDL セッション実行中の各種情報を表示するコマンド(プロシージャ)
- 引数無しで実行すると、現在メモリ上にある、作成された変数やコンパイル済みのプログラムの情報が表示される

```
IDL> help
% At $MAIN$
A               FLOAT     =       12.3400
B               STRING    = 'IDL lesson'
Compiled Procedures:
    $MAIN$  EULER     GCIRC

Compiled Functions:
```

- 変数名を引数にすると、変数データの値と型が表示される

➢ このコマンドもバグ取り作業で頻繁に使われる

JOURNAL 機能

- 対話型で進める IDL セッション(IDLプロンプトへの入力と一部の出力結果)をログファイルに記録する
- セッションのメモ(記録)や、スクリプトの簡易的な作成方法として使用できる
- Journal 機能を使って記録したセッションをスクリプトとして実行すれば、セッションを再実行(リプレイ)できる

使い方: Journal[, filename]

✓ journal コマンドの引数にジャーナルファイル名の指定がない場合は、'idlsave.pro' が作成される
✓ 引数無しで再度 jounal コマンドを実行することで保存を停止する

```
IDL> journal, 'mylog.pro' ; logging start
IDL> (IDLコマンド)
IDL> journal ; logging stop
```

OS のコマンド実行 (SPAWN)

- 子プロセスとして、OS のコマンドを実行する

使い方: SPAWN [, *Command* [, *Result*] [, *ErrResult*]]

✓ OS のコマンドを第一引数として文字列で与える
✓ 第二引数に変数名を指定すると、出力結果がその変数に保存される

```
IDL> spawn, 'ls -l'
合計 12
-rw-r--r-- 1 makitisn adcusers 7131 11月  4 16:02 2016 output.ps
-rw-r--r-- 1 makitisn adcusers   61  1月 13 16:50 2017 test.pro
-rw-r--r-- 1 makitisn adcusers  109  1月 20 14:31 2017 test2.pro
```

- SPAWN コマンドは、自作の IDL プログラムの中で使用することもできる
- たとえば、プログラム中に

```
PRO mypro
  ....
  SPAWN, 'mkdir new_directory'
  ....
END
```

と書いて、ディレクトリを作成する、など

- ただし、IDL 自身はクロスプラットフォームな言語であるが、このようなプログラムは OS に依存してしまう(実行環境依存性を持つ)ことになるので注意
- ディレクトリ作成用のコマンド FILE_MKDIR など、ファイル操作用の IDL コマンドも用意されている。極力 IDL の機能を使用するのが望ましい

Dollar Sign ($)

1. コマンド文の行末の $ 記号は、本来1行に書かれるべきコマンド文を次の行につなげる、継続文字
2. IDL のプロンプト(IDL>)の直後の $ 記号は、その後ろの入力文字をOSのコマンドとして実行する
3. IDLコマンドライン環境を使用中に、$ 記号単体で入力すると、OSの子プロセス(シェル環境)が始まる。
   exit コマンドでシェルを抜けると元の IDL 環境に戻る

```
IDL> $pwd                            ; Linux の pwd コマンドを実行
/home01/makitisn/work
IDL> $                               ; シェルに抜ける
makitisn@kaim14:~/work[1]$ exit      # シェルから IDL に戻る
exit
IDL>
```

■ 変数・定数・データ型

変数とデータ型

- IDL では、変数は必要になった場所で自由に作成できる (プログラム先頭で使用する変数を宣言する必要は無い)
- 値を代入するだけで変数を作成できる
- データ型はプログラムの途中でも自由に変更できる。あるいは自動的に変更される
- 変数の名前は文字で始める(アルファベット、およびアンダーバー記号 _)。2文字目以降には数字も使える
- 予約語(AND, IF, END, etc.)は使用できない
- 変数名にビルトインコマンド名も避けるべき (変数の作成は可能だが、トラブルの元になる)
- 大文字小文字の区別は無い
- 変数名は最大1000文字(1001文字目以降は無視される)

主なデータ型

Type	説明	Bits	最小値	最大値	Suffix	例
Byte	符号無し整数	8	0	255	B	1B
Integer	整数	16	-32768	32767	(S)	1
Unsigned Integer	符号無し整数	16	0	65535	U	1U
Long	倍長整数	32	-2^{31}	$2^{31} - 1$	L	1L
Unsigned Long	符号無し倍長整数	32	0	$2^{32} - 1$	UL	1UL
64-bit Long	4倍長整数	64	-2^{63}	$2^{63} - 1$	LL	1LL
64-bit Unsigned Long	符号無し4倍長整数	64	0	$2^{64} - 1$	ULL	1UL
Float	浮動小数点	32	-10^{38}	10^{38}	. E	1.0 1.0E+1
Double	倍精度浮動小数点	64	-10^{308}	10^{308}	D	1.0D+1
String	文字列	8*文字数			' or "	'1.0'

他に、数値型には複素数(Comprex, Double-Complex)や、非数値型には構造体やポインタなどがある

8進数、16進数による整数表現

	Type	例
16進数 (Hexadecimal)	Byte	'FF'XB or 0xFFUB
	Integer	'FF'X or 0xFF
	Unsigned Integer	'FF'XU or 0xFFU
	Long	'FF'XL or 0xFFL
	Unsigined Long	'FF'XUL or 0xFFUL
8進数 (Octal)	Byte	"12B
	Integer	"12 or '12'O
	Unsigned Integer	"12U or '12'OU
	Long	"12L or '12'OL
	Unsigined Long	"12UL or '12'OUL

＊上記の他、64ビットの型もある

文字列定数

- 文字列定数はシングルクオート(')またはダブルクオート(")の対で囲む
- 異なる種類のクオートで囲むことで、文字列の中にクオートを含める事が出来る
  - クオートを二つ続ける事でも可能だが見にくい

```
IDL> print, 'Test'
Test
IDL> print, "I'm Taro."
I'm Taro.
IDL> print, 'I''m Hanako.'
I'm Hanako.
```

❗ ~~ダブルクオートは8進数定数を表す文字としても使われるため、文字列用として数字の前には使えない。print, "12TEST" はエラー~~
➔ ver. 8.6.1 から文字列として認識するようになった

変数の作成と注意

- x=1 としたときの整数 x は Integer (INT)
  - 使用中に INT の最大値(32767)を超えてしまってエラーになることがあるので注意
    - ~~たとえば FOR ループのカウンタ変数。~~(*)
- x=1.0 としたときの浮動小数点数 x は Float
  - 多くの IDL 計算ライブラリのデフォルトも Float
  - 計算過程で精度が不十分な場合があるので注意
- 安全のためには、整数は Long を、浮動小数点数は Double を意識的に使うようにすると良い
  - もちろんメモリ節約などを優先したい場合はその限りでは無い

(*) IDLの最近のバージョンでは自動的に Long に変化するようになった

```
IDL> x=1
IDL> help, x
X               INT       =        1
IDL> x=1L
IDL> help, x
X               LONG      =              1
IDL> x=1.
IDL> help, x
X               FLOAT     =       1.00000
IDL> x=1d
IDL> help, x
X               DOUBLE    =        1.0000000
IDL> x='1'
IDL> help, x
X               STRING    = '1'
```

型変換と型別配列生成の関数

Type	Type Name	型変換関数	配列生成	インデックス配列生成
byte	**BYTE**	byte()	bytarr()	bindgen()
Integer	**INT**	fix()	intarr()	indgen()
Unsigned Integer	**UINT**	uint()	uintarr()	uindgen()
Long	**LONG**	long()	lonarr()	lindgen()
Unsigned Long	**ULONG**	ulong()	ulonarr()	ulindgen()
64-bit Long	**LONG64**	long64()	lon64arr()	l64indgen()
64-bit Unsigned Long	**ULONG64**	ulong64()	ulon64arr()	ul64indgen()
Float	**FLOAT**	float()	fltarr()	findgen()
Double	**DOUBLE**	double()	dblarr()	dindgen()
String	**STRING**	string()	strarr()	sindgen()

型変換の例 (明示的に変換)

```
IDL> a=20.3
IDL> help, a
A               FLOAT     =      20.3000
IDL> help, fix(a)
<Expression>    INT       =      20
IDL> help, double(a)
<Expression>    DOUBLE    =      20.299999
IDL> help, string(a)
<Expression>    STRING    = '    20.3000'
IDL> help, float('18.5')
<Expression>    FLOAT     =      18.5000
```

← 整数に変換

← 倍精度に変換
(誤差が発生している)

← 文字列に変換

← 文字列から数値に変換

型変換の例 (自動変換)

```
IDL> help, 2 + 3
<Expression>    INT       =        5              ← INT + INT
IDL> help, 2 + 3L
<Expression>    LONG      =              5        ← INT + LONG
IDL> help, 2 + 3.0
<Expression>    FLOAT     =       5.00000         ← INT + FLOAT
IDL> help, 3 / 2
<Expression>    INT       =        1              ← INT / INT
IDL> help, 3 / 2.
<Expression>    FLOAT     =       1.50000         ← INT / FLOAT
```

✓ 異なるデータ型の変数の演算結果は、式の中の精度が高い変数の型に揃う

❗ 整数同士の割り算の結果は整数なので小数点以下が失われる

浮動小数点数から整数への変換

- 下記の関数を使っても浮動小数点数を整数に変換できる
  - round()　丸める
  - floor()　小数点以下を切り捨てる
  - ceil()　繰り上げる

[使い方] Result = round(x[, /L64])

✓ x が浮動小数点数なら 32-bit (long) 整数に変換する。
ただし、x が byte や int なら、返値も同型

✓ /L64 オプションを使うと返値は 64-bit 整数となる

```
IDL> r = round([3.1, 3.8])
IDL> help, r
R               LONG      = Array[2]
IDL> print, r
           3           4
```

データ型の判定 SIZE() 関数

変数のデータ型を調べる方法

- コマンドラインからは help コマンドを使っても良い
- プログラムの中では SIZE 関数が使える。変数の型の他に、配列サイズや次元などを調べることが出来る

```
IDL> print, size(5)
       0       2       1
IDL> print, size(findgen(3,7))
       2       3       7       4       21
```

← 0次元, INT(2), 1要素

← 2次元(3x7), FLOAT(4), 21要素

- データ型のみを調べるのには /TYPE や /TNAME のオプションが使える

```
IDL> print, size(5, /type)
       2
IDL> print, size(5, /tname)
INT
```

特別な値 Null

- !NULL はデータ型未定義を示すシステム変数。IDL8.0から導入された
- !NULL 値は通常の操作では無視される
  - [1, !NULL, 2, !NULL, 3] は [1,2,3] と同じになる

```
;; 変数が定義済みか調べる
IDL> help, a
A               UNDEFINED = <Undefined>
IDL> help, a EQ !NULL
<Expression>    BYTE      =    1

;; 定義済みの変数に!NULLを代入することでメモリを解放できる
IDL> var=!NULL
```

他の使い方は調べてみて下さい。

特別な浮動小数点数 NaN, Inf

- !VALUES という read-only のシステム変数(構造体)の中に定義されている。
  Single- and double-precision それぞれの (IEEE で定義された) 浮動小数点数。
- 浮動小数点演算のエラー結果として現れる未定義性を表現する特殊な値
- !NULL とは異なり、操作上、無視されない
  (エラー値として伝播する)
- プログラム実行中に現れた場合、通常 Math errors の warning メッセージを表示するが、処理は止まらない

■ NaN (not-a-number)

- !Values.F_NAN (単精度), !Values.D_NAN (倍精度)

```
IDL> print, sqrt(-1)
        -NaN
% Program caused arithmetic error: Floating illegal operand
```

←負の数の平方根

■ Inf

- !Values.F_INFINITY (単精度), !Values.D_INFINITY (倍精度)

```
IDL> print, 1./0
         Inf
% Program caused arithmetic error: Floating divide by 0
```

← ゼロ除算の結果

✓ 多くのライブラリルーチンでは適切に処理される。例えば以下の例

```
IDL> a = findgen(10)
IDL> a[[3,7]] = !VALUES.F_NAN
IDL> plot, a, psym=-4
```

← 配列 a の要素3と7にNaN を代入
← NaN を含むデータのプロット

Math errors を取り除く

- Math errors を取り除き、伝播するのを防ぐ
- 明示的に NaN や Inf のチェックを行う FINITE 関数を使用する

```
IDL> a = findgen(10)
IDL> a[[3,7]] = !VALUES.F_NAN
IDL> print, mean(a)
          NaN
IDL> print, finite(a)
   1   1   1   0   1   1   1   0   1   1
IDL> print, mean(a[where(finite(a) EQ 1)])
      4.37500
```

mean 関数で平均を求める

← math errors の位置は偽(0)

- 関数の中で math errors をチェックして取り除くオプション (/NAN) を持つ関数も多い。上の mean() を使った例では、下記も可

```
IDL> print, mean(a, /nan)
      4.37500
```

文字列操作

- IDL でデータ処理や解析を行う際には、
  文字列を扱う処理も使う機会は多い
- たとえば、
  - 処理結果の数値などを見やすく整形して
    ディスプレイ上に表示する
  - グラフとともに、値をプロット枠の内部に示す
  - 外部にデータとして出力する

  など

主な文字列操作ルーチン

文字列操作でよく使うライブラリルーチンなど	
連結	+
空白除去	strcompress()
	strtrim()
文字列長さ	strlen()
文字列位置検索	strpos()
抜き出し	strmid()
置き換え	strput
比較	strcmp()
一致検索	strmatch()
大文字・小文字変換	strupcase(), strlowcase()

STRCOMPRESS関数

```
Result = STRCOMPRESS( String, /REMOVE_ALL )
```

- ✓ 連続する空白(スペースあるいはタブ)を一つに縮める
- ✓ /REMOVE_ALL オプションを付けると、空白をすべて取り除く

STRTRIM関数

```
Result = STRTRIM( String [, Flag] )
```

- ✓ Flag: 0(default): 後方の空白を除去, 1: 前方の空白を除去, 2: 両方を除去

```
IDL> a=1.24
IDL> str='Value='
IDL> print, str + string(a)
Value=      1.24000
IDL> print, str + strcompress(string(a))
Value= 1.24000
IDL> print, str + strcompress(string(a),  /rem)
Value=1.24000
IDL> print, str + strtrim(string(a),1)
Value=1.24000
```

❗ これらの関数は、実は文字列以外を引数にしても自動的に文字列に変換する

STRPOS関数

Result = STRPOS(文字列, 検索文字列)

- ✓ '文字列'の中から'検索文字列'が何文字目に現れるか、その位置を返す
- ✓ マッチする文字列が見つからない場合の返値は -1

STRMID関数

Result = STRMID(文字列, 開始位置[, 長さ])

- ✓ '文字列'の中から'開始位置'から'長さ'分の部分文字列を切り出す
- ✓ '長さ'の指定がない場合は最後まで

STRPUT プロシージャ

STRPUT, 文字列, 置き換え文字列 [, 開始位置]

- ✓ '文字列'の'開始位置'から'置き換え文字列'で置き換える
- ✓ '開始位置'の指定がない場合は0文字目から
- ✓ 元の文字列の長さは変化しない。
置き換え文字列がはみ出した部分は切り捨てられる

STRCMP関数

Result = STRCMP(文字列1, 文字列2 [, N], /FOLD_CASE)

- ✓ ‘文字列1’と’文字列2’を比較して一致すれば 1、不一致なら 0 を返す
- ✓ ‘EQ’ 演算子を用いた比較と同様だが、はじめのN文字のみで比較や、大文字小文字の違いの無視 (/FOLD_CASE) ができる

STRMATCH関数

Result = STRMATCH(文字列, 検索文字列 , /FOLD_CASE)

- ✓ ‘文字列’と’検索文字列’が一致すれば 1、不一致なら 0 を返す
- ✓ ‘検索文字列’にはワイルドカード(*, ?)が使用できる

```
IDL> str = ['foot', 'Feet', 'fate', 'FAST', 'ferret', 'fort']
IDL> PRINT, str[WHERE(STRMATCH(str, 'f*t', /FOLD_CASE) EQ 1)]
foot Feet FAST ferret fort
```

↑ 大文字小文字は区別せず、fで始まりtで終わる文字列だけを取り出す例

❗ このほか、正規表現を使用した文字列マッチングを行う STREGEX 関数もある

[演習] 文字列操作

■ ログファイル名として使用するために、現在時刻を含んだ文字列を作ってみる

- (ファイル名の例)
  log_2021Sep15_115657.txt

- 現在時刻を取得する関数は systime()
  - 引数なしで実行すると、ローカルタイムの日付時刻を文字列で返す

❗ systime() 関数以外にも、bin_date() や timestamp() 関数などが現在時刻を取り出す機能を持つ

演習の回答例

```
IDL> stime = systime()     # システム時刻(現在時刻)の取得
IDL> print, stime       # 取得したシステム時刻の文字列確認
Wed Sep 15 11:56:57 2021
IDL> month = strmid(stime, 4, 3)
            #「月名」を切り出し。4文字目から3文字分
IDL> date = string(strmid(stime, 8, 2), format='(i02)')
    # 「日付」は切り出した後、必ず2桁になるようにフォーマットを整えている。
    # 文字列への型変換関数 string() は format オプションが使える。
IDL> hh = strmid(stime, 11, 2)      # 以降、時刻等を同様に切り出し
IDL> mm = strmid(stime, 14, 2)
IDL> ss = strmid(stime, 17, 2)
IDL> ypos = strpos(stime, '2021')   #「年」の位置(何文字目か)を取得
IDL> year = strmid(stime, ypos, 4) # 取得位置から4文字を切り出し
IDL> fname = 'log_'+year+month+date+'_'+hh+mm+ss+'.txt'
                 # 得られた文字列を結合
IDL> print, fname # 結果を確認
log_2021Sep15_115657.txt
```

■ 配列

IDL の配列 (Array)

IDL 言語の最大の特徴
IDL は配列指向に設計された言語であり、
効率的で分かりやすい配列処理を行うことができる

呼び方
スカラー(単一の数値・変数)、ベクトル(1次元アレイ)、アレイ(次元を持った構造)

配列の表現方法
(例)
要素数 n の 1 次元アレイの場合
a[i] (i=0〜n-1)
要素数 m×n の 2 次元アレイの場合
a[i,j] (i=0〜m-1, j=0〜n-1)

他言語の配列との比較

- 複数次元の場合、一組の [ ] の間に要素数を書く
  (ex. a[5,8]) ⇌ C言語では [ ] を並べる (ex. a[5][8])
- [ ] ではなく、FORTRAN のように ( ) も使える
  (が、推奨されない)
- 要素を指定する添え字の i や j は 0 から始まる
  (C言語と同じ。FORTRAN は 1 から)
- アレイ要素のメモリ内での並びは [ ] 内の左側から
  (FORTRAN と同じ。C言語では右から)

  → FOR文のループ処理では、左側の添え字を内側のループに入れる方が処理が速くなる。
  (ただし、ループ処理以外の方法で配列を扱えるのであれば、そちらの方が速い)

配列の作成 (1)

- 要素を直接指定する

```
IDL> arr = [2,4,8,16]
```

- 多次元の場合

```
IDL> arr = [[0,1,2],[3,4,5],[6,7,8]]
IDL> help, arr
ARR             INT       = Array[3, 3]
IDL> print, arr
       0       1       2
       3       4       5
       6       7       8
```

```
IDL> a=[1,2,3] & b=[3,4,5] & c=[6,7,8]
IDL> arr = [[a],[b],[c]]                    ; ➔ 配列の結合(後述)
```

配列の作成 (2)

- 型別に用意された配列生成関数を使う

```
IDL> arr=fltarr(256, 256)
IDL> help, arr
ARR             FLOAT     = Array[256, 256]
```

✓ 上の例では 256x256 の float 型の配列を作成してメモリ上に領域を確保
✓ 各要素の初期値はゼロ
✓ データ型によって、他にも intarr(), lonarr(), dblarr(), etc.

- 一般的な配列生成関数を使う

```
IDL> arr=make_array(256, 256, /FLOAT)
```

✓ 上の例と同じ結果を得る
✓ データ型によって、オプションは他に /INTEGER, /LONG, /DOUBLE, etc.

```
IDL> arr=make_array(256, 256, /FLOAT, Value=3.0)
```

✓ 初期値を VALUE キーワードオプションで指定できる

配列の結合

```
IDL> a=[1,2,3]   ; 1次元アレイ
IDL> b=[4,5,6]
IDL> c=[a,b]     ; 1次元アレイを直列に結合
IDL> d=[[a],[b]] ; 1次元アレイを結合して2次元アレイに
IDL> help, c,d
C               INT       = Array[6]
D               INT       = Array[3, 2]
IDL> e=[[[d]],[[d]]]      ; 2次元アレイを結合して3次元アレイに
IDL> help, e
E               INT       = Array[3, 2, 2]
```

❢ 角括弧[ ]で括ることによって結合する次元が変わる

配列の指定方法(部分配列)

- 要素を指定しない場合は配列全体

```
IDL> a=[2,4,6,8,10]
IDL> print, a
       2       4       6       8      10
```

- すべての要素を指定する * (アスタリスク)

```
IDL> print, a[*]
       2       4       6       8      10
```

- 要素の位置を指定する

```
IDL> print, a[2]
       6
```

- 要素の連続した位置を指定する : (コロン)

```
IDL> print, a[1:3]
       4       6       8
```

- 多次元配列の場合、それぞれの次元を指定する例

```
IDL> a=[[2,4,6,8,10],[3,6,9,12,15]]
IDL> help, a
A               INT       = Array[5, 2]
IDL> print, a[2:4,*]
       6       8      10
       9      12      15
```

- 多次元配列を1次元配列として扱って指定

❢ 配列はすべて1次元配列(ベクトル)として表現できる

```
IDL> print, a[8]
      12
```

- 要素の指定に別の配列を使う

```
IDL> a=indgen(10)*5
IDL> print, a
       0       5      10      15      20      25      30      35      40      45
IDL> ind=[1,3,5]
IDL> print, a[ind]                    ← a[[1,3,5]] と同じ意味になる
       5      15      25
```

配列の変形 よく使われる関数

REFORM

- 次元(配列の形)を変更する

TRANSPOSE

- 次元を(2次元アレイの場合、行と列を)入れ替える

REVERSE

- 指定した次元の要素の順番(向き)を反転する

ROTATE

- 2次元アレイをX軸、Y軸方向に任意の組み合わせで反転する

※ 次ページ以降に、REFORM(), TRANSPOSE() を使った例を示す

REFORM 関数

- 全体の要素数は同じまま、次元を変更する

```
IDL> a=indgen(4,3)
IDL> help, a
A               INT       = Array[4, 3]
IDL> b=reform(a,3,4)
IDL> help, b
B               INT       = Array[3, 4]
```

- この関数は、次元数を下げる目的で使われることが多い(次ページ)

- reform() を次元を下げるのに利用する
  - 多次元配列から部分配列を取り出した際に、ある次元方向の要素数が1になって残ることがある。これを取り除く。

```
IDL> a=indgen(4,4,4)
IDL> help, a
A               INT       = Array[4, 4, 4]
IDL> b=a[2,*,*]
IDL> help, b
B               INT       = Array[1, 4, 4]
IDL> c=reform(b)
IDL> help, c
C               INT       = Array[4, 4]
```

❢ 一番右側の次元を1要素だけ残した場合は自動的に次元が下がる

```
IDL> d=a[*,*,2]
IDL> help, d
D               INT       = Array[4, 4]
```

TRANSPOSE 関数

- 2次元アレイの行と列を入れ替える

```
IDL> arr = [[1,2,3,4,5],[6,7,8,9,10]]
IDL> help, arr
ARR             INT       = Array[5, 2]
IDL> print, arr
       1       2       3       4       5
       6       7       8       9      10
IDL> help, transpose(arr)
<Expression>    INT       = Array[2, 5]
IDL> print, transpose(arr)
       1       6
       2       7
       3       8
       4       9
       5      10
```

✓ 3次元以上の配列の次元を任意の順番に並べ替えることもできる

配列操作でよく使う機能

SHIFT 関数

- 配列要素を指定の方向へ指定の数だけずらす(循環する)

```
IDL> a=indgen(5)
IDL> print, a, shift(a,1), shift(a,-1)
       0       1       2       3       4
       4       0       1       2       3
       1       2       3       4       0
```

- [使用例] 微分(差分)データを作る

```
; 2次関数の微分
IDL> a=indgen(10)^2
IDL> print, a-shift(a,1)
     -81       1       3       5       7       9      11      13      15      17
```

✓ 一つ隣の要素との差分を取っているが、端の要素のみ異なることに注意

❗ (参考) ISHFT 関数は整数(byte, int, long)のビットをずらす処理を行う

WHERE 関数

- 配列の中から、指定した条件に合う要素の位置(添え字の値)を取り出す

使い方: vector1 = WHERE(条件 [, Count] [, COMPLEMENT=vector2])

- オプションにより、条件に合った要素の数(Count)や、条件に合わなかった要素の位置(vector2) も得られる

```
IDL> a=[2,5,-3,9,-7,1]
IDL> r=where(a GE 0, cnt, complement=r2)  ; GE=Greater than or equal to (以上)
IDL> print, r, r2, cnt
       0       1       3       5
       2       4
       4
IDL> print, a[r]
     2     5     9     1
```

❗条件に合う要素がない場合の返値は -1 となる(Countは0)
(IDL8.0 以降、-1 の代わりに !NULL を返すオプション /NULL ができた)

Where() の条件の書き方

- Where 関数の「指定条件」は、条件に合う位置の要素の値が '0ではない' (通常は1) ベクトル
- 次の演算子がよく使われる(詳細は次ページ以降)
  - 比較演算子 EQ, NE, GE, GT, LE, LT
  - 論理演算子 AND, OR, NOT

```
IDL> a=indgen(7)-3
IDL> print, a
      -3      -2      -1       0       1       2       3
IDL> print, a GT 0    ; 比較演算子を使った結果
   0   0   0   0   1   1   1
IDL> print, where(a GT 0)  ; それを条件とした where() の結果
           4           5           6
IDL> print, a[where(a GT 0)]  ; where() で条件に合う要素だけ取り出す
       1       2       3
```

比較演算子(関係演算子)

演算子	説明
EQ	Equal to (等しい)
NE	Not equal to (等しくない)
GE	Greater than or equal to (以上)
GT	Greater than (より大きい)
LE	Less than or equal to (以下)
LT	Less than (より小さい)

- 比較演算の結果が真 (True) ならば 1 (Byte) が、偽 (False) ならば 0 が返される

論理演算子

論理演算の結果 → 真(True, 1)、 偽(False, 0)

演算子	例
AND	複数の条件がすべて成立するかどうか。 「かつ」
OR	複数の条件のどれかひとつが成立するかどうか。 「または」
~	条件が成立しないかどうか。

AND, OR(, NOT, XOR) は本来 Bitwise Operators (ビット演算子)。
0, 1 以外の数に使用するときは注意！

✓ IDL 6.0 から論理演算子(Logical Operators)として &&, ||, ~ が導入された。
ただし、**&&, || は配列を受け付けない。**

➡ where() の条件式では通常 AND や OR を使用

例) 配列 a の -2 より大きく 2 以下の値の取り出し
a[where((a GT -2) AND (a LE 2))]

```
IDL> print, 1 AND 1
       1
IDL> print, 1 AND 0
       0
IDL> print, 1 OR 0
       1
IDL> print, ~0
     1
IDL> print, ~1
     0
```

SORT 関数

- 要素の値の小さい順に並び替えた「添え字の配列」を作る

```
IDL> a=[5,3,8,2,6]
IDL> r=sort(a)
IDL> print, r
           3           1           0           4           2
IDL> print, a[r]
       2       3       5       6       8
```

大きい順に並べ替えるには REVERSE 関数を組み合わせる

```
IDL> print, a[reverse(r)]
       8       6       5       3       2
```

❗ 並べ替えた後、同じ値を取り除くのにはさらに UNIQ() を使う

配列の演算

- IDL の配列演算ではループ処理を行う必要はない (ループを使うと遅くなる)
- 配列とスカラー値の加減乗除
  - 配列の各要素とスカラー値の計算結果が得られる

```
IDL> a=[1,2,3]
IDL> print, (a+1)*2
      4       6       8
```

- 配列同士の加減乗除
  - 配列の同じ位置にある要素同士の計算結果が得られる

```
IDL> a=[1,2,3] & b=[2,4,6]
IDL> print, a * b
      2       8      18
```

配列演算の注意

- 要素数や次元が異なる配列同士の演算結果は…

```
IDL> a=[2,4,6,8,10] & b=[1,2,3]
IDL> print, a+b
       3       6       9
```

→ 対応する要素がある位置でのみ計算した結果が返される

```
IDL> a=[[1,2,3],[4,5,6],[7,8,9]] & b=[[2,2],[3,3]]
IDL> print, a*b
       2       4
       9      12
```

→ 二次元アレイでも、それぞれを1次元アレイと見なしたときに先頭から対応する要素がある位置まで計算が行われる

❢ エラーにはならない

行列演算

- これまで見た通常の IDL の配列処理は、数学的な演算とは異なっているが、IDL には線形代数的な行列演算を行うための演算子 #, ##も用意されている

```
IDL> a=[[0,1,2],[3,4,5]] & b=[[0,1],[2,3],[4,5]]
IDL> help, a,b
A               INT       = Array[3, 2]
B               INT       = Array[2, 3]
IDL> print, A#B
           3           4           5
           9          14          19
          15          24          33
IDL> print, A##B
          10          13
          28          40
```

※ 詳細はヘルプファイルを参照のこと

■ 構造体

IDL の構造体(Structures)

- 配列などの通常の変数データは、すべての要素で同じデータ型を持つ
- 構造体ではデータ型の異なる変数を一つにまとめた集合体として取り扱うことができる
- IDL の構造体には、定義名を持たない無名構造体 (Anonymous structures)と、定義名を持つ記名構造体 (Named structure)がある
- 構造体の要素(メンバ)に構造体を用いることも出来る (Hierachical structures, Nested structures)

無名構造体 (Anonymous structures)

- 構造体の作成
  - 波括弧 { } の中に構造体を構成する要素を、タグ名とデータ型を決める初期値のペアにして : (コロン) を挟んで並べる

```
IDL> strct = {name:' ', ra:0.0, dec:0.0, flux:0.0d}
```

✓ 初期値は定義済みの変数や式を使ってもＯＫ
✓ 上記の例では要素はすべてスカラーだが、配列でも構わない

- 構造体の配列
  - REPLICATE 関数を使ってアレイが作成できる

```
IDL> strctarr = replicate(strct, 100)
```

記名構造体 (Named structures)

- 構造体の作成
  - 最初に構造体の定義名を書く。
    あとは無名構造体と同じ。

```
IDL> strct = {star, name:' ', ra:0.0, dec:0.0, flux:0.0d}
```

  - 定義名を使って、同じ構造を持った別の構造体変数を作る事ができる (初期値は0 or "null string" になる)

```
IDL> strct2 = {star}
```

- 構造体の配列
  - 構造体の定義名を{} で囲んで REPLICATE 関数に渡すことでアレイを作ることができる

```
IDL> strctarr = replicate({star}, 100)
```

✓ 無名構造体と同じように既存の構造体変数から複製しても構わない

構造体の操作

- 構造体の中身にアクセスする書式は、

```
構造体変数名.タグ名
```

(例)

```
IDL> strct.name='Vega' & strct.flux=1.23
IDL> print, strct.name, strct.flux
Vega       1.2300000
```

- 構造体が配列の場合、構造体変数名に添え字指定をする

```
IDL> print, strctarr[15].name
```

- 定義と異なる型が代入された場合は、可能ならば定義された型に変換される (不可能ならばエラーとなる)

```
IDL> strct.name=3.54
FISDR> help, strct.name
<Expression>    STRING    = '      3.54000'
```

構造体の配列操作の注意

- 構造体の配列の一部を指定する場合、配列要素指定の添え字を付ける位置に注意！

- 構造体が配列の場合、構造体変数名に添え字指定をする

例) strctarr**[15]**.tag1

- 構造体を構成する要素が配列の場合、配列になっている要素のタグ名に添え字指定をする

例) strct.tag2**[0]**

- どちらも配列の場合、それぞれに添え字指定をする

例) strctarr**[15]**.tag2**[0]**

構造体についての注意点

- 通常の変数と異なり、動的にデータ型やサイズを変更することはできない

- 記名構造体は定義名を使って、同じ構造の構造体変数を後から増やすことができる
- 記名構造体の定義名はグローバルメモリに保存され、変更(再定義)できない。
  変更したい場合は .reset コマンドでセッションをリセットする必要がある
- 無名構造体はいつでも再定義(再作成)できる

■ カーブフィッティング

フィッティングルーチン

- IDL には、
  - 様々な目的 (・フィッティングの対象は1次元データか？2次元データか？・ガウス関数か？多項式関数か？ など)に応じて、
  - また、各種アルゴリズムによって実装された、

  複数のカーブフィッティングプログラムが標準で用意されている。

- さらに、より堅牢で高機能と評価されているフィッティングルーチンが、公開ライブラリツールとしてユーザによって公開・提供されている。

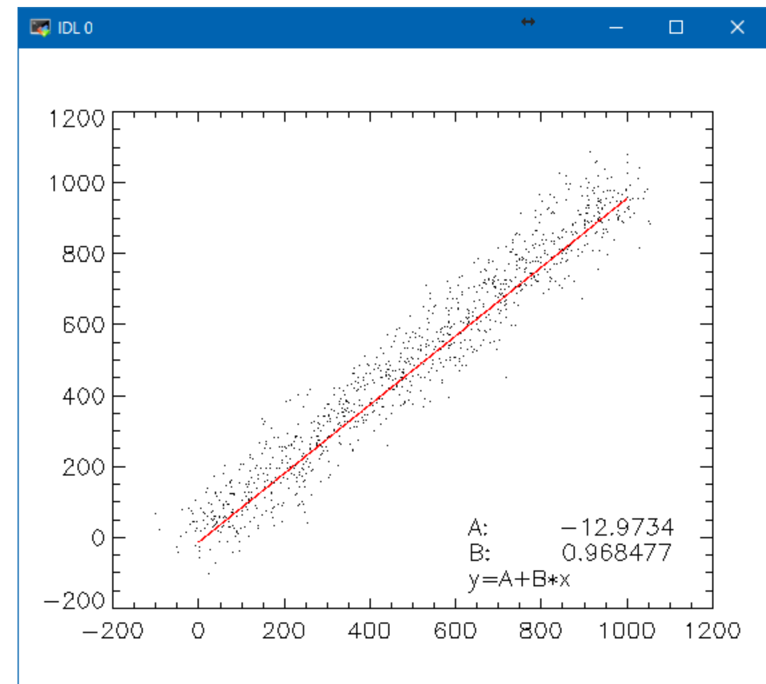
以降ではそのうちのいくつかを紹介する

LINFIT 関数

- (x,y)データを直線 y = A + Bx でフィットする

```
Result = LINFIT( x, y [, /DOUBLE, MEASURE_ERRORS=vector, etc.])
```

```
(使用例)
 ;; 疑似データ作成
IDL> x=findgen(1000)+randomn(seed, 1000)*50
IDL> y=findgen(1000)+randomn(seed, 1000)*50
IDL> merr = SQRT(ABS(y))
 ;; 直線フィッティング
IDL> r = LINFIT(x, y, MEASURE_ERRORS=merr)
IDL> print, r
     -12.9734     0.968477
```



➢ LINFIT() の返値は直線関数の係数

GAUSSFIT 関数

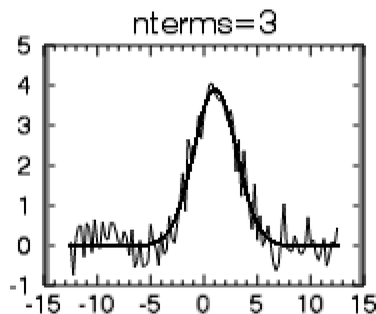
- (x,y)データに対して、ガウシアンカーブと2次関数の組み合わせでフィッティングを行う

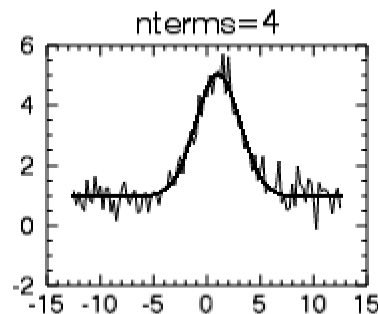
```
Result = GAUSSFIT( x, y [, a] [, MEASURE_ERRORS=vector, NTERMS={3 to 6}, etc.])
```

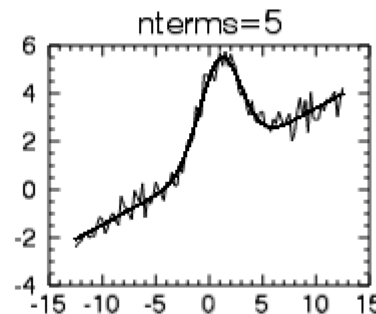
➢ GAUSSFIT() の返値は当てはめ曲線のf(x)データ配列。パラメータ数を nterms で調整。関数の係数はオプションの引数Aに返される

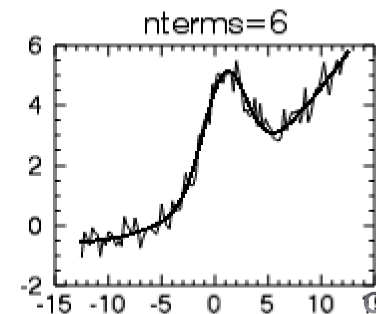
$$f(x) = A_0 e^{\frac{-z^2}{2}} + A_3 + A_4 x + A_5 x^2$$ (nterms=6 の場合)

$$z = \frac{x - A_1}{A_2}$$







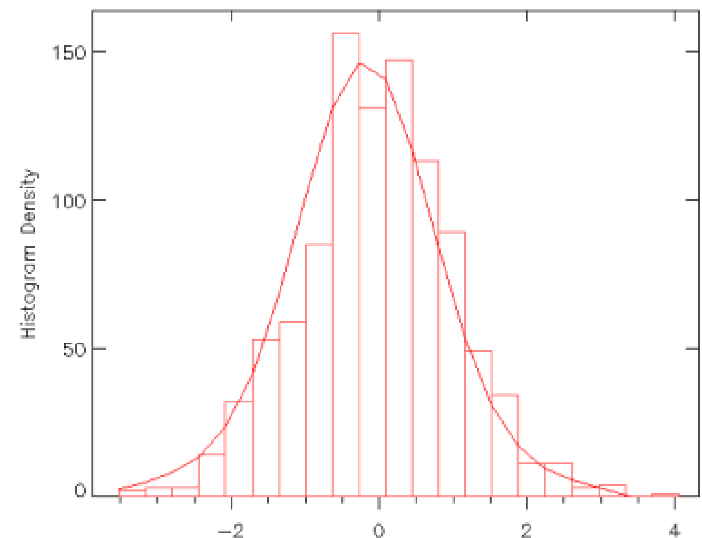


```
;;正規曲線(ガウシアンカーブ)フィッティング例

; 疑似データ作成(正規分布に従う乱数を1000個)
dt = randomn(seed, 1000)
; ヒストグラム表示; hdt, loc にヒストグラムデータを格納
cghistoplot, dt, histdata=hdt, loc=loc

x = loc
y = hdt
; さらに2次関数を付加してみる
;y = hdt+10+5*x+2*x^2
;cgplot, x, y, psym=10

; ガウシアンフィッティング
yfit = GAUSSFIT(x, y, coeff, NTERMS=6)
; フィッティングした曲線を重ねて表示
cgplot, x, yfit, /over, color='red'
```



LMFIT 関数

- (x,y)データに対して、レーベンバーグ・マーカート法(Levenberg-Marquardt Method)による非線形最小二乗フィットで任意の数のパラメータを持つユーザ定義関数に当てはめる

```
Result = LMFIT(x, y, a[, FUNCTION_NAME=string, FITA=vector, etc.])
```

- LMFIT() の返値は当てはめ曲線の f(x) データ配列
- a には関数のパラメータの初期推定値を与えておく。処理後には、フィッティング結果のパラメータに変化する
- FUNCTION_NAME にはフィットする関数を定義した *.pro (IDL の関数としてあらかじめ作成)の名前を指定する
- FITA オプションで固定パラメータとフリーパラメータを指定できる
- ほか、収束したかどうかを確かめるためのキーワード、フィット結果の精度を調べるための chi-square 値や実施した iterations の数を得るためのオプションもある

LMFIT() を使ったテスト

Online help の例を参考に

フィットする関数 f(x)=a[0] * exp(a[1]*x) + a[2] + a[3] * sin(x)

myfunct.pro

```
FUNCTION myfunct, X, A
   bx = a[0]*EXP(a[1]*x)
   return, [ [bx+a[2]+a[3]*SIN(x)], [EXP(a[1]*x)], [bx*x], $
              [1.0], [SIN(x)] ]
END
```

まず上記の IDL 関数を作成、保存しておく

- ✓ フィットする関数は x と a を受け取り、a のパラメータ数+1の要素数を持った配列を返値とする
- ✓ 返値は、関数値f(x)と、残りは各パラメータに対する偏微分になっている

```
;; (x, y) データ
x = findgen(40)/20.0
y = 8.8 * EXP(-9.9 * x) + 11.11 + 4.9 * SIN(x)
merr = 0.05 * y

;; パラメータ a の初期値の与え方によって結果がどう変わるか？試してみる
a = [10.0, -0.1, 2.0, 4.0] ; online-help で例示されている値
; うまくいかない場合、初期値を動かして結果の変化を見てみる

print, a   ; パラメータ初期値を確認
;; フィッティング実行
yfit = LMFIT(x, y, a, MEASURE_ERRORS=merr, FUNCT = 'myfunct' $
                    , iter=iter, conv=conv)
;; フィッティング結果を確認
PLOTERR, x, y, merr
OPLOT, x, yfit, color='0000ff'xl
print, a, iter, conv   ; iter はiterations の数, conv は収束結果(1なら収束)
```

フィッティング処理の注意点(経験的に)

- フィッティングはわりと試行錯誤が必要になることが多い
- 思いもよらないフィッティング結果が出てくることも多いので、結果を注意深く確認することが必要
- パラメータ初期推定値やパラメータ範囲の設定、反復計算の最大回数や収束条件の指定などの微妙な差が結果に大きな差をもたらすこともある
- うまくいかない場合は他のプログラム(ライブラリルーチン)を使ってみるのも手。(指定できるオプションや使い方なども異なる)