IDL 講習会 (初級編)

2018年7月26日(木), 7月27日(金)
於 国立天文台 三鷹キャンパス
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)

目次

- Journal 機能
- OSのコマンド実行(SPAWN)
- Dollar Sign ($)

8. グラフィックス ～ データプロット(plot) etc.
   - Direct Graphics vs. Object Graphics
   - ウィンドウの操作
     - ウィンドウを開く
     - 開いたウィンドウを操作する
   - データを表示する
   - PLOT (procedure)
     - 基本的な使い方
     - よく使うオプション
     - 外見を整えるオプション
   - 複数データを重ねてプロット(OPLOT)
   - 点や線のオーバーレイ(PLOTS)
   - 文字列のオーバーレイ(XYOUTS)
   - IDLグラフィックスの座標
   - 軸の作成(AXIS)
   - 画面を分割する(!p.multi)
   - Position キーワード
   - Postscript ファイルに出力する
   - エラーバーのプロット
   - CONTOUR (procedure)
   - SURFACE (procedure)
     - SHADE_SURF
     - XSURFACE
   - CGPLOT
   - Coyote Graphics によるファイル出力
   - ヒストグラム作成
     - CGHISTOPLOT
   - 関数型のグラフィックスルーチン
9. 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
   - 型変換と生成の関数
   - 手動変換(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
10. 配列
    - IDLの配列 (Array)
    - 多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

11. 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

12. 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

13. IDL のプログラミング
- スクリプト
- プロシージャとファンクション
- プロシージャ (Procedure)
- 関数 (Function)
- プログラムソースファイルの作成
  - ファイル名の付け方(ルール)
  - 複数のルーチンをまとめる(サブルーチン)
- 変数のスコープ(有効範囲)
- COMMON ブロック
- コンパイル
  - 自動コンパイル
  - 手動コンパイル
- エラー
  - プログラムが存在しない場合のエラー
  - 実行時エラー対処の際の注意点
- 引数
  - 位置パラメータ
  - キーワードパラメータ
  - 引数のチェック
  - _EXTRA キーワード
  - 引数の引き渡し(値渡しと参照渡し)

14. 簡単なプログラム
- プログラムの基本構造とUsage
- Usage (一般的な書き方と読み方)
- 制御文
  - IF 文
  - FOR 文
  - WHILE 文
  - CASE 文

1. IDL について

IDL とは？

- Interactive Data Language
- 「データの解析と可視化に特化した配列指向型のプログラミング言語」
- データの処理、その科学的解析から視覚化まで IDL だけで行うことが可能
- 対話的に使用する事が出来て、手軽に扱える。すぐに結果が見られる

etc.

- 豊富なグラフィックスルーチンが用意されている、画像処理に長けたプログラミングソフトウェア
- ポストスクリプト(PS)出力の他、JPEG や PNG などの形式も出力可能
- ライブラリを使用することで FITS ファイルの読み書きも出来る
- 天文・宇宙の分野ではスタンダードに使用されている
  - ほかにも、高層大気や気象分野、医療画像処理などで広く使用されている
- 画像処理用のほか数学的、統計的な処理機能などデータ解析ルーチンも豊富
- むしろどんなルーチンがすでに準備されているのか把握するのが難しい

言語の特徴

- 配列計算が得意。ベクトル化された処理により、大量の要素、次元を持った配列も見通しよく扱える
- 直感的に書けて、理解がしやすい。比較的少ない行数で読みやすい
- 変数の事前宣言が不要。いつでも新しい変数を作成できる。変数のデータ型は(基本的には)IDL が自動的に判断して動的に決定できる

→ 習得が容易

- 構文は歴史的には FORTRAN 風の傾向が強かったが、最近は C 言語的な部分も多い。
  オブジェクト指向の機能等も追加されている
- IDL 自体は C 言語で書かれている
- コンパイラ型とインタプリタ型の中間的存在。
  通常、自動コンパイルで使用される。インタープリタ的な使い勝手だが、コンパイラ型に見劣りしない速度も出る
- Linux のほか、Windows や Mac OS、Sun Solaris でも動作する(クロスプラットフォーム)
  - どの OS で開発した IDL プログラムでも、同じように使える
- ライセンスは高価

IDL の配列について

- 最大の特徴である IDL の配列操作(ベクトル化された処理)は C 言語などで書かれた処理と比べても十分に(同等レベルに)高速
- しかし、配列の個々の要素に対してループ処理を行うなどすると、途端に遅くなる
- 配列操作は IDL 的に取り扱うべし
  - 配列同士の演算に不用意なループは用いない
  - IDL に用意されているプログラムも多くは配列をそのまま入力できる
- ただし、ループ処理は IDL でも多用される
  (配列処理に不必要に使わないことが重要)

開発元の歴史

- 大本は大学の研究所(コロラド大学ボルダー校の大気宇宙物理学研究所 LASP)による開発。
- その後、LASP から独立して設立された会社、Research Systems Inc. (RSI) が長年の開発元に。
- 2004年に ITT の子会社となり、2006年に RSI は ITT Visual Information Solutions に改称。
- 2011年には、分社化して Exelis VIS (Visual Information Solutions) となった。
- 2015年2月、米国 Harris 社が米国 Exelis 社を買収。
- 2018年1月より社名変更「Exelis VIS」から「Harris Geospatial」に
- (余談)インストールディレクトリ名にデフォルトでは社名が入る(rsi, itt, exelis, harris)ため、インストールパスが頻繁に変わり、(システムに複数バージョンをインストールしたい場合などに)一時混乱の元だった(今も?)。

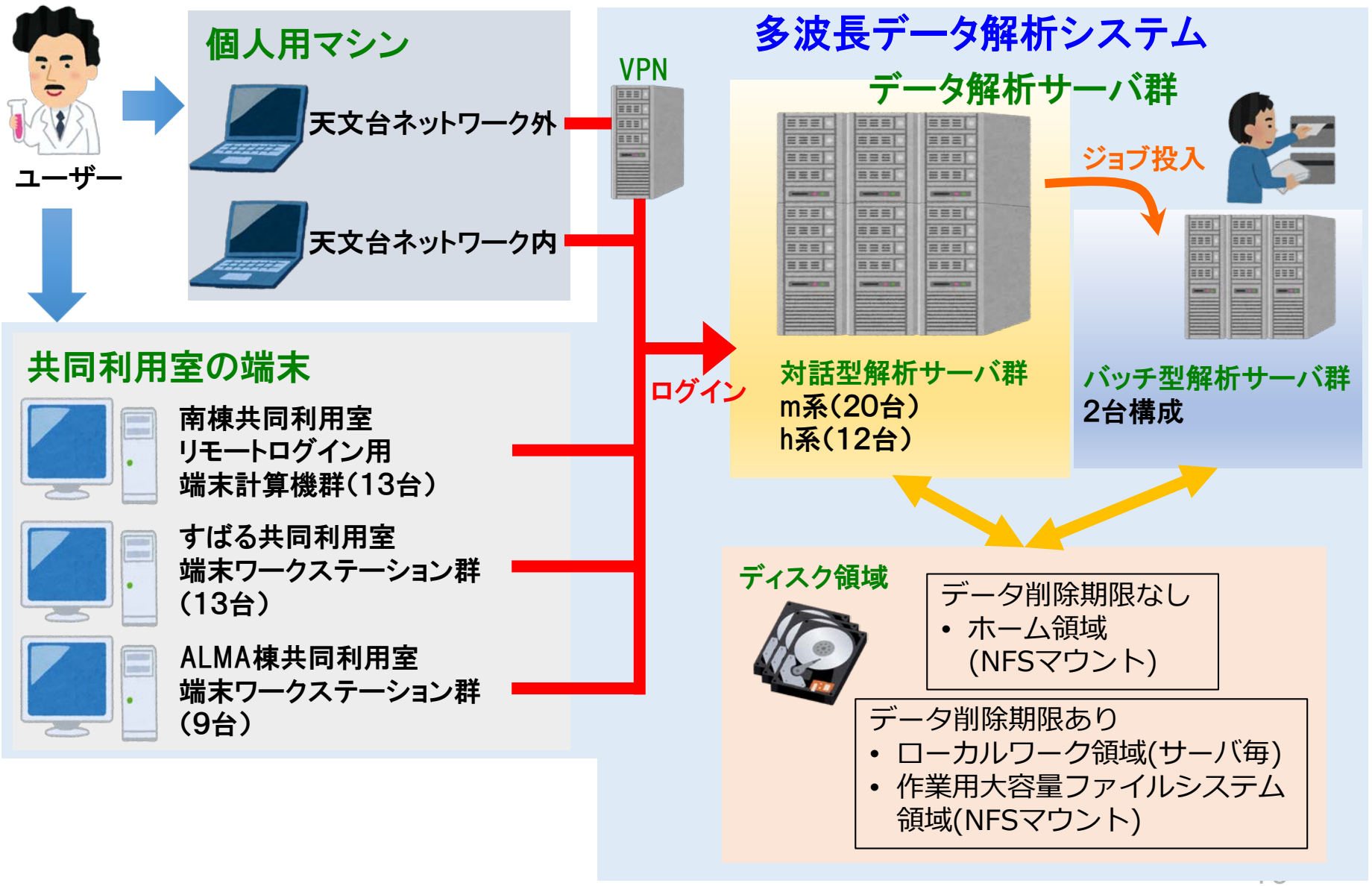
IDL の入手

- IDL のインストールファイルはメーカーサイトでユーザ登録して申請することでダウンロード可能
- VM (Virtual Machine)が含まれる
- VM は無料で使用できる
  - IDL プログラムを VM アプリケーションとして配布することが可能。(ただし、動作上のいくつかの制限もある)
- ヘルプファイルも含まれる
- ライセンスされていない IDL 本体はデモモードで起動する(試用可能)

2. 使用準備

天文台データ解析システムにおける IDL 環境

- 国立天文台 天文データセンターが運用している多波長データ解析システムには、様々な天文データ解析に使用されるソフトウェア類がインストールされており、IDL も(ほぼ)最新版が使用できる
- 多波長データ解析システムを使用できる人
  - 国立天文台の職員および国内外の天文学研究者(大学院生を含む)
    - 学部生は個別の許可が必要
- 使用できる計算機
  - 解析サーバの OS は Linux (Red Hat Enterprise Linux)
  - 解析サーバには天文台内の自分のマシンからリモートログインするか、共同利用室に設置された端末からログインする
  - 天文台外からの使用には VPN を通した接続が必要



by courtesy of Ozawa, T. & Kamegai, K.

IDL の利用とライセンス

- IDL の使用にはライセンス認証が必要
  - ライセンス認証がされない場合は7分間のデモモードで起動する
- 多波長データ解析システム全体では、通常時最大80人が同時使用できるライセンスを備えている
- 同じホストにログインして IDL を複数起動しても使用されるライセンスはひとつ分
  - 別ホストで使用すると、ライセンスを余計に消費する
- idlstat コマンドで現在のライセンス使用状況を確認することが出来る
  - idlstat は多波長解析システムで用意された独自ツール (IDL のベンダー提供コマンド の wrapper プログラム)

(参考情報) ライセンス管理の仕組みが ver. 8.6 以降で変わった

操作方法とエディタ

- IDL を対話型で使用する際は、シェルターミナルから起動してそのままコマンドを入力していく
- IDL のプログラムを作成する場合は、任意のテキストエディタを起動してソースコードを記述する
- テキストエディタは、IDL 用に用意された統合開発環境 IDLDE (後述)を用いても良いし、その他のエディタを使用しても構わない
- 多波長解析システムでは Emacs, Vim, gedit (GNOMEの標準テキストエディタ) などが使用できる。使い慣れたもの、使いやすいものを使えば良い

環境設定(1) パス(IDL_PATH)

- IDL パス (IDL_PATH) の役割
  - IDL のプログラムは、手動コンパイルして実行することももちろん可能だが、自動コンパイルして実行する使い方が普通
  - IDL はプログラム(ファイル名が “*.pro” のプロシージャや関数)を IDL パスの中から自動的に検索して、コンパイル、実行できる
  - IDL のパスは通常、環境変数 IDL_PATH に設定する
  - ベンダーが提供するプログラム(デフォルトライブラリ)のほか、ユーザが追加したライブラリルーチンや自作プログラムが置かれたディレクトリのパスをあらかじめ設定しておけば、毎回同じ環境で使用できる

- プログラムは、パスに書かれたディレクトリから指定した順番で検索されて、最初に見つかったプログラムがコンパイルされる
- このため、同じ名前のプログラムが異なるディレクトリに存在すると、混乱の元になるので注意。
  たとえば、
  - 意識せずに同じ名前で作成されたプログラム
  - 同じプログラムだがバージョンが異なる

  の存在などがトラブルの元になりやすい。

自作プログラムの名前は、既存プログラムと重複しないように意識することが必要

環境変数 IDL_PATH の設定方法

bash の場合の例

```
$ IDL_PATH=~/work:+~/idl/mylib:"<IDL_DEFAULT>":${IDL_PATH}
$ export IDL_PATH
```

- ✓ : (コロン)でディレクトリのパスをつなぐ
- ✓ 先頭に + を付けると、そのサブディレクトリもすべて追加される
- ✓ "<IDL_DEFAULT>" は開発元が用意したライブラリの全パスを意味する
- ✓ この設定前にすでに IDL_PATH 環境変数が存在しており、それを残したまま追加する場合は、${IDL_PATH} を含める

csh/tcsh の場合の例

```
$ setenv IDL_PATH ~/work:+~/idl/mylib:"<IDL_DEFAULT>":${IDL_PATH}
```

- ◘ 毎回使う設定は、シェルの設定ファイル .bashrc や .cshrc に書いておく
- ◘ 多波長解析システムではデフォルトで標準的な IDL_PATH が共通設定されている

環境変数 IDL_PATH の確認方法

- 設定した(設定されている) IDL_PATH 環境変数を確認するには、

```
$ printenv IDL_PATH
```

✓ printenv は環境変数を表示する Linux のコマンド

あるいは、

```
$ echo $IDL_PATH
```

環境設定(2) startup file (起動ファイル)

- startup ファイルは IDL の起動時に自動実行される IDL で書かれたバッチ(スクリプト)ファイル
- IDL コマンドを使った環境設定や、作業の事前操作など、毎回決まった手順を実行したい場合に、設定しておく
  - たとえば、作業やプロジェクトごとに独自に必要なパスを設定する、コンパイル時やエラー発生時の挙動を設定する、保存してある毎回使う変数データを展開する、など
- ファイル名は何でも構わないが、*.pro としておくのが無難。(IDL のファイルとして識別するため。ただし、中身はプロシージャではなくスクリプトである)
- 環境変数 IDL_STARTUP にファイル名を指定する

環境設定の注意点

- IDL_PATH や startup file の設定や確認にはワークベンチ(IDLDE)の機能を使用すると分かりやすい
  (詳細は次章)
  - 「ウィンドウ」->「設定」-> 「IDL」-> ....

ただし、注意が必要

- 環境変数による設定とワークベンチによる設定の両方が存在する場合は、環境変数が優先される
- 多波長データ解析システムでは、デフォルトのユーザ環境として環境変数 IDL_PATH が設定済み。
  (この状態でワークベンチ側の設定を行っても設定は保持されない)

3. 起動と終了

コマンドライン環境の起動と終了

■ 起動

- Linux では端末(ターミナル)でコマンド名 "idl" を入力
  - (参考) Windows の場合は、スタートメニューなどから「IDL 8.x Command Line」を実行。(パスが通っていればコマンドプロンプトから "idl" でも可。)

```
$ idl
```

■ 終了

- IDL のプロンプトに対して "exit" を入力

```
IDL> exit
```

! 起動時にライセンス認証が実行される。ライセンスが無い(あるいは認証が出来ない)場合は、7分間だけ使用できるデモモードになる。(7分が経過すると自動終了する)

起動時に表示されるメッセージ (ver.8.6.1の例)

```
$ idl
IDL 8.6.1 (linux x86_64 m64).
(c) 2017, Exelis Visual Information Solutions, Inc., a subsidiary of Harris Corporation.

Licensed for use by: Fujitsu Limited - Aoyama via 133.40.130.160:7070
License: MNT-5509912:****-****-****-E890
A new version is available: IDL 8.7
https://harrisgeospatial.flexnetoperations.com

IDL>
```

← 新バージョンがある場合のお知らせ表示

ライセンスエラーの場合に表示されるメッセージ (ver.8.5.1の例)
7分間だけ使用可能なデモモードで起動する

```
$ idl
IDL Version 8.5.1 (linux x86_64 m64). (c) 2015, Exelis Visual Information Solutions, Inc., a subsidiary of Harris
Corporation.
LICENSE MANAGER: Cannot find license file.
 The license files (or license server system network addresses) attempted are
listed below.  Use LM_LICENSE_FILE to use a different license file,
 or contact your software provider for a license file.
Feature:       idl
(略)
Entering timed demo mode. Each session is limited to 7 minutes of operation. Printing and file saving are disabled.

To learn more about our license options for this product, please contact your account manager or Exelis Visual
Information Solutions, Inc., a subsidiary of Harris Corporation. at info@exelisvis.com.

IDL>
```

IDL ワークベンチ(IDLDE)の起動と終了

■ 起動

- Linux では端末(ターミナル)でコマンド名 "idlde" を入力

```
$ idlde &
```

  - ✓ コマンドの後ろに & (バックグラウンドで実行する)を付けないと、起動した端末を占拠してしまうので注意
  - ✓ (参考) Windows の場合は、スタートメニューなどから「IDL 8.x」を実行

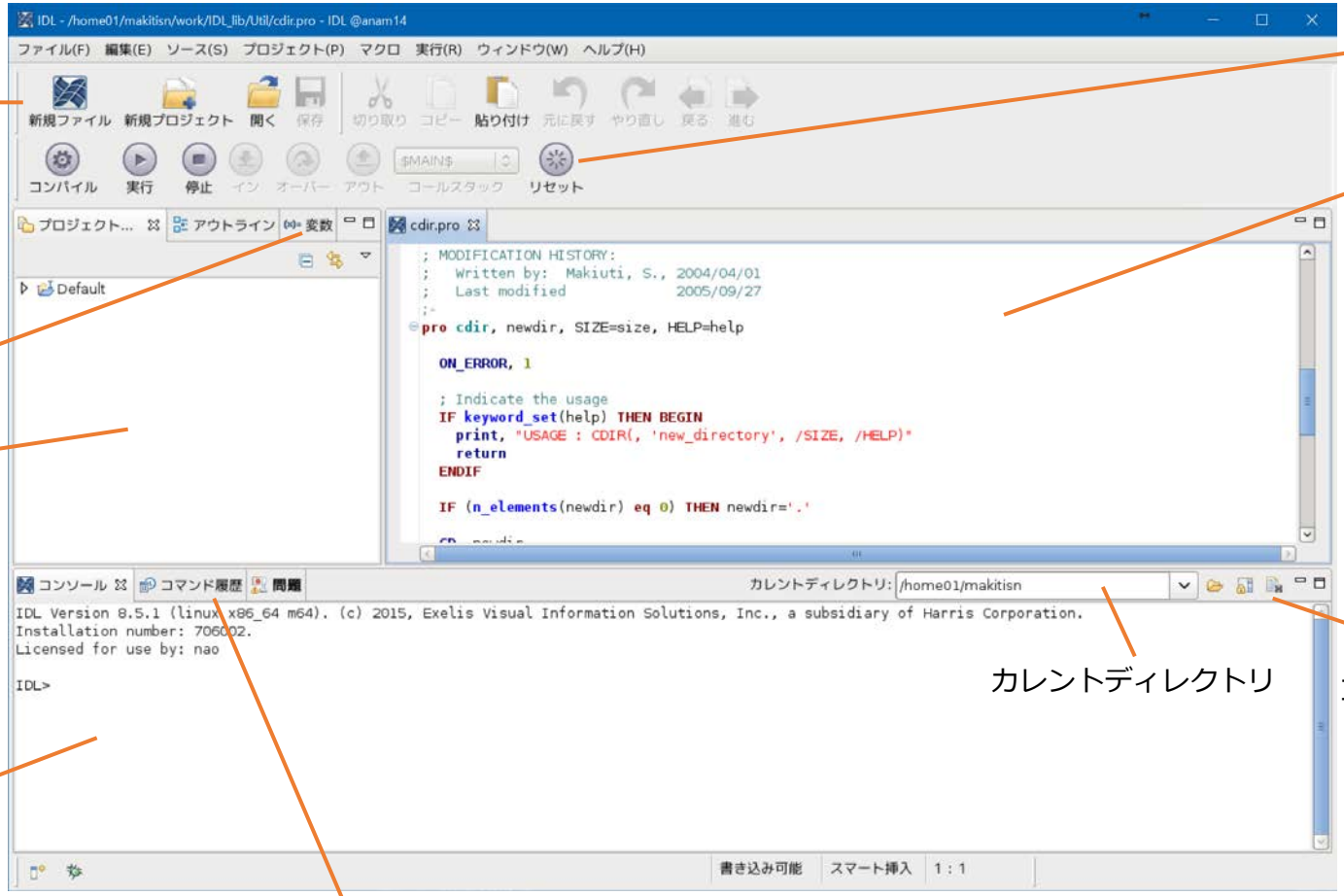
■ 終了

- メニューバーから「ファイル」→「終了」を選択
- 「コンソール」領域の IDL プロンプトに対して "exit" を入力
- IDLDE ウィンドウのクローズボタンで閉じる (この場合は通常、確認ダイアログが表示される)

IDL Workbench (プログラム統合開発環境) - IDLDE -

- オープンソースの Eclipse フレームワークをベースにして用意された、IDL 用のグラフィカルな開発環境
- コマンドラインの操作も可能。(コンソール領域がある)
  - ただし、純粋なコマンドライン環境の方が「より軽い」「コンソールの表示領域が大きく取れる」などの利点がある
- 複数のソースを連携したプロジェクトの管理や、デバッグツールなどを使いこなせれば、多機能な IDLDE は便利に使える
- IDLDE のレイアウトや各項目の表示・非表示などは、かなり自由にカスタマイズが可能
  - 「ウィンドウ」→「ビューのリセット」でデフォルトに戻せる
- IDL_PATH など各種設定を GUI で確認、操作できる
- エディタ画面では IDL の各構文が分かりやすく色分けして表示される

IDLDE (IDL Development Envionment)



IDLDE を使った環境設定

■ よく使う項目

「ウィンドウ」→「設定」

→「IDL」"起動ファイル", "作業ディレクトリ"

→「パス」(IDLパス)

- IDL パスは、環境変数 IDL_PATH が別に設定されている場合、そちらが優先される。(IDLDE で行ったパス設定は、起動時に IDL_PATH に置き換えられる)
  IDL_PATH が設定されていない場合は、IDLDE の環境設定で設定した IDL パスはコマンドライン環境にも反映される
- IDL パス以外の設定項目も基本的に同様。IDL をコマンドライン環境で使用する場合も、設定やその確認は IDLDE から行うと GUI で分かりやすい

作業ディレクトリ(カレントディレクトリ)

- IDL のプロセスは「作業ディレクトリ」"current working directory" 内で実行される
- コマンドラインモードでは IDL を起動したディレクトリ
- ワークベンチ(IDLDE)には「デフォルトの作業ディレクトリ」の設定がある。(環境変数 IDL_START_DIR でも設定可能)
- プログラム(*.pro) の最初の検索パスやファイル入出力の対象ディレクトリは、作業ディレクトリ
  - もちろん、ファイルを扱う際に、ファイル名に上位パスを付けたフルパスで指定することは可能
- この認識が曖昧だと、書き出したファイルがどこに保存されたのかなど、分からなくなることもあるので注意
- IDL 内部で作業ディレクトリを変更するには、下記コマンド群が使える
  - CD, PUSHD, POPD

実行中のプログラムの中断方法

- 実行中のプログラムを手動で止めたい場合、キーボードから Ctrl+c (Cntl キーを押しながら c キー) をタイプすることで可能
  - ワークベンチ使用時は、Ctrl+c でプログラムを止めるとき、コマンドラインエリアにフォーカスしておく。エディタにフォーカスされている場合は Ctrl+c はコピー操作になる
- 間違って無限ループを作ってしまったときや、長時間かかるプログラムを途中で止めたくなったときなどに使用する
- デバッグなどのために、プログラムの中にあらかじめ中断を仕込む用途には STOP プロシージャを使う。中断されたプログラムは .CONTINUE コマンドで再開できる

IDL の使い方と実行モード

- IDL の使い方には、対話式に一行ずつコマンドを実行していくインタラクティブモードと、プログラムを書いて実行するプログラムモードがある
- 大量のデータを処理する場合や、同じ処理を繰り返す必要がある場合にはプログラムを書く必要があるが、基本的なデータ処理と可視化などのデータ解析の場面など多くの場合はインタラクティブモードの使用で事足りる
- インタラクティブモードでの一連の処理はファイルに保存しておいて、バッチ処理(スクリプトの実行)という形で実行することもできる
- また、手順を記録(メモ)したテキストファイルから、まとめてコピー＆ペーストでコマンドラインに流し込むような使い方も可能で、便利で効率的に使える

4. 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

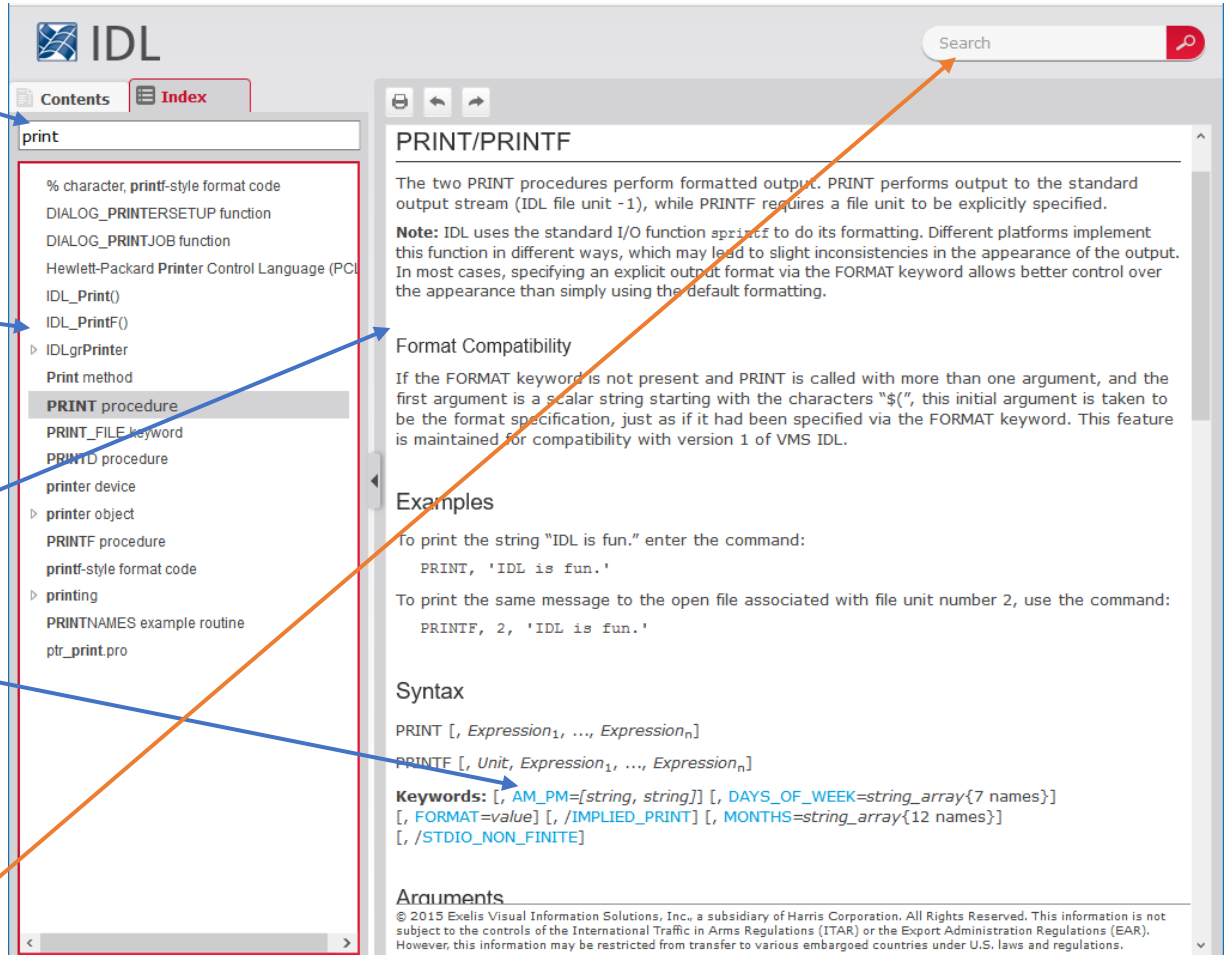
「Index」タブを選択して下の欄に検索ワードを入力

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

マニュアル全文からの検索窓



「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には、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

5. 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 と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

6. 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
- 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> ; 特殊文字の例
IDL> ; このようにセミコロンの後ろはコメントになる
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(,)区切りの構文になっている
  - 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

7. 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残される
- 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のに使うなどする

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

RETURN & RETALL

- return コマンドは、ひとつ上のプログラムレベル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

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)で実行することが可能。(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のには 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 (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
exit
IDL>
```

8. グラフィックス

データプロット(PLOT) etc.

Direct Graphics vs. Object Graphics

- Object Graphics はオブジェクト指向プログラミングによるグラフィックシステムとして IDL version 5.0 から実装された
- Object Graphics では、オブジェクトの生成や初期化等、描画するまでにいくつかの手順が必要なため、手軽さには劣る
- Direct Graphics の方が軽い
- インタラクティブな使用には Direct Graphics の方が適している
- Object Graphics では、描画情報を保持して後から変更することが可能

※ 本講習会では Direct Graphics を中心に解説する。

ウィンドウの操作

- 通常は必要なグラフィックスウィンドウは自動で開かれる
  たとえば、plot コマンドを使うと、自動的に開いたウィンドウにプロットされる
- ウィンドウの大きさや(初期の)位置を指定したり、複数のウィンドウを開いて使い分けるなどするために、ウィンドウ操作コマンドを使って自分で操作することも出来る

WINDOW	WDELETE	WSET	ERASE
開く	閉じる	出力先 (current window) を切り替える	中身を消す

ウィンドウを開く

- [使い方]

WINDOW[, window_index, /free] [, {x|y}size=value, {x|y}pos=value, title=string]

- ✓ window_index : ウィンドウ番号(0-31, 指定が無ければ0から)
- ✓ /free : 32番以上の空いている番号を使う
- ✓ {x|y}size : ウィンドウの縦/横幅をピクセル単位で指定する
- ✓ {x|y}pos : ウィンドウを開く位置を指定する
- ✓ title : デフォルトでは 'IDL *n*' (*n* はウィンドウ番号)と表示されるウィンドウタイトルを指定する

- すでに開いている番号に開くと、前のウィンドウは消される
- 新しいウィンドウを開くと、それが current window になる

```
IDL> window, 1, xsize=800, ysize=600
```

開いたウィンドウを操作する

WDELETE[, window_index, ...]

- ✓指定した番号(window_index)のウィンドウを閉じる
- ✓指定が無ければ current window を閉じる
- ✓指定する番号は複数並べることが可能

WSET[, window_index]

- ✓ current window を指定した番号に切り替える

ERASE

- ✓開いている current window の中身を消す
  (背景色で塗りつぶす)

データを表示する

■ データ可視化のための主なコマンド (procedures)

- plot　　: 散布図、ライングラフ
- oplot　: 既存のグラフ上に別のグラフを重ね描き
- plots　: 既存のグラフの上に点または線を描く
- axis　　: 軸を定義して表示する
- xyouts　: グラフィック上に文字列を表示する
- contour: 3次元データをコントア表示する
- surface : 3次元データをメッシュで表現する

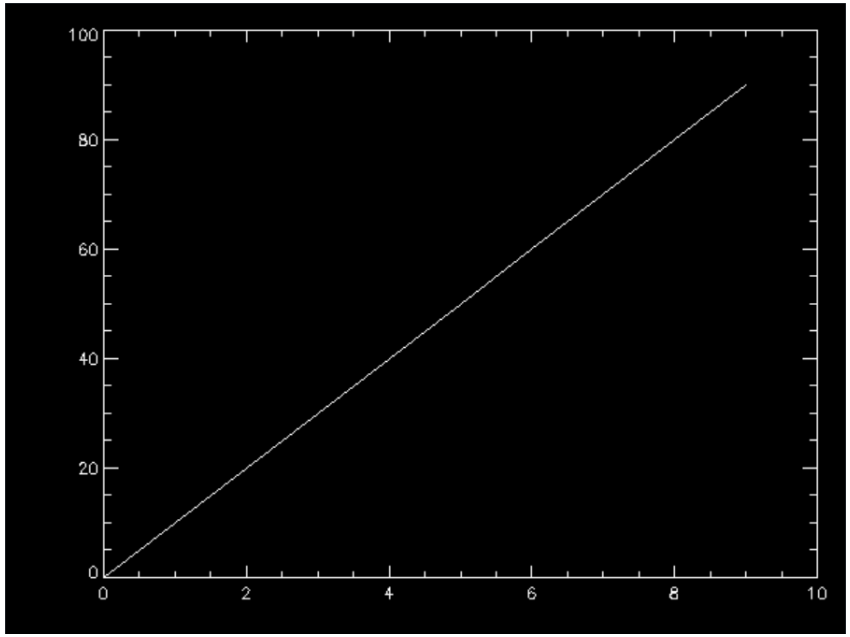
PLOT (procedure)

- (おそらく)もっとも頻繁に使う procedure
  ただし、より高機能な拡張されたライブラリツールもある(後述)
- もっとも基本的な使い方は、

plot, [x,]y　　(x, y はプロットするデータアレイ)

例)
IDL> plot, indgen(10)*10

(※ indgen() はインデックス配列の生成コマンド)



- ✓ 引数がひとつ(y)のみの場合、x は自動的にインデックス値となる
- ✓ 描画範囲やレイアウトなどは IDL によって適当に調整される
  (手早くデータを確認したいときに便利)
- ✓ オプションを使うことで、様々な箇所をカスタマイズできる

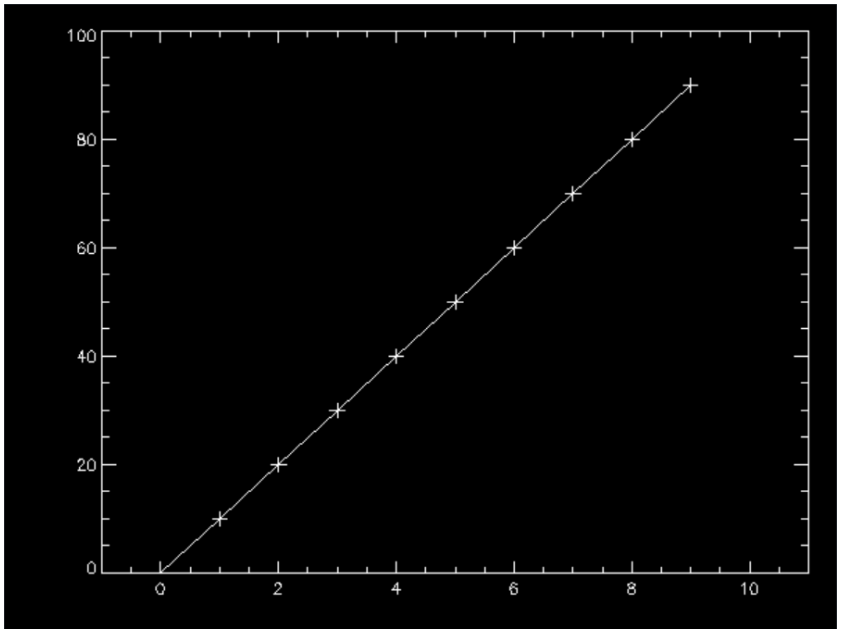
よく使うオプション (1)	
psym=number	データ点のシンボル 1: Plus sign (+) 2: Asterisk (*) 3: Period (・) 4: Diamond (◇) 5: Trianle (△) 6: Square (□) 7: X 8: User-defined 数字にマイナス符号(-)を付けると、シンボルと線の両方を表示する
symsize=value	シンボルのサイズ(default: 1.0)
{x|y}range=[min, max]	(x,y)軸の範囲(レンジ)の指定 指定しない場合、データ全体を含む様に自動調整される
/ynozero	yデータがすべて正の場合に、Y軸がゼロから描かれるのを禁止する

よく使うオプション (2)	
{x|y}style =value	(X,Y)軸の表示方法 下記数字の合計値で指定する 0: (default) IDLが判断した適切なレンジ 1: 指定値あるいはデータの最大・最小値の正確な範囲 2: 適当なマージンを設ける 4: 軸を描画しない 8: ボックススタイルの軸を表示しない(左側、下側の軸線のみ) 16: /ynozero と同じ(Y軸のみで有効)
/{x|y}log	対数(log)スケールでプロットする
/isotropic	X軸とY軸のスケールを合わせる
/nodata	軸のみを描いて、データはプロットしない
/noerase	既存のグラフィックスを消さずに、上に重ねて描く

オプションを使用した例

```
IDL> plot, indgen(10)*10, psym=-1, symsize=1.5, xrange=[-1,11], xstyle=1
```

- シンボルマーク“+”で、実線付き
- シンボルのサイズは標準の1.5倍
- X軸のレンジは -1 から 11 までの正確な範囲でプロット



さらに見栄えを整える

- オプションを使ってプロットの外見を整える
- これらのオプションの多くは、PLOT procedure だけではなく、CONTOUR procedure など他の多くのグラフィックスコマンドに共通

プロットの外見を整えるオプション類 (1)	
position=[x0, y0, x1, y1]	ウィンドウ内のプロット位置を指定。左下と右上の相対座標(0.0 - 1.0)で指定する。
{x|y}margin=[v0, v1]	プロット領域外側の余白サイズを指定する。 単位は charsize(文字サイズ)。 デフォルトはx:[10,3], y:[4,2] ただし、position が優先される。
title=string	プロットのタイトル
subtitle=string	プロットのサブタイトル
{x|y}title=string	(x|y)軸のタイトル
charsize=value	文字サイズ(default 1.0)
{x|y}charsize=value	(x|y)軸ごとの文字サイズ(default 1.0)
charthick=value	文字の太さ (default 1)
color=value	色の指定(default: highest color table index) 詳しくは後述。

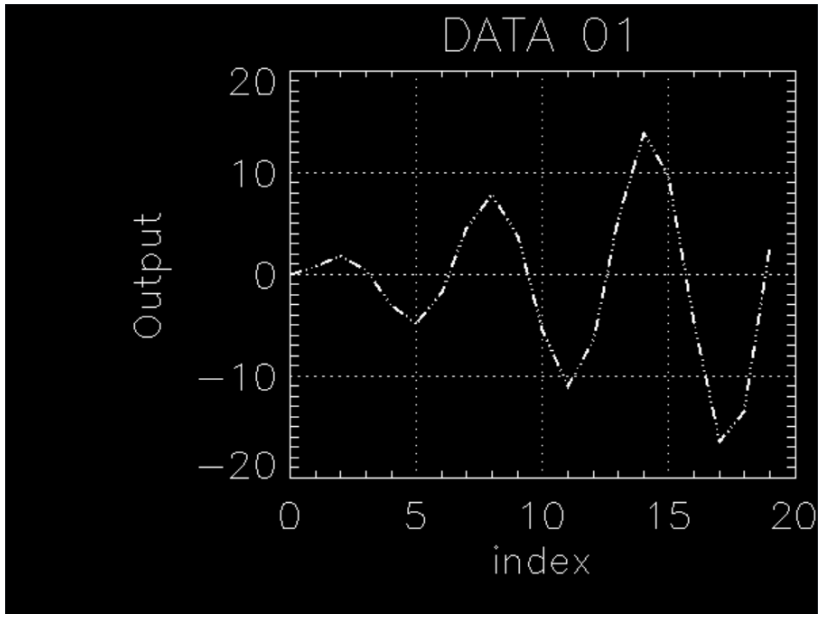
プロットの外見を整えるオプション類 (2)	
linestyle={0 to 5}	線の種類 0 Solid (default) 1 Dotted 2 Dashed 3 Dash Dot 4 Dash Dot Dot 5 Long Dashes
thick=value	線の太さ (default 1.0)
{x|y}thick=value	軸と目盛線の太さ (default 1.0)
{x|y}ticks=value	目盛の数
{x|y}minor=value	目盛間の補助目盛の数
ticklen=value	目盛り線の長さ (default 0.02; プロット範囲全体の長さに対する比)
{x|y}ticklen=value	(x,y)軸ごとの目盛り線の長さ (default 0.02)
{x|y}gridstyle={0 to 5}	グリッド線の種類(種類は linestyle と同じ) グリッド線を描くには ticklen=1.0 を設定する。

オプションを使用した例 (2)

```
IDL> plot, indgen(20)*sin(indgen(20)), xmargin=[12,1], $
     title='DATA 01', xtitle='index', ytitle='Output', charsize=2.5, $
     linestyle=4, thick=2.0, ticklen=1.0, xgridstyle=1, ygridstyle=1
```

行末の '$' 記号はコマンド途中の改行。上の例で '$' を付けずに、一行ですべてを書いても良い。

各オプションで何を設定してあるか確認してみて下さい。



複数のデータを重ねてプロットする; OPLOT

- plot で最初のデータをプロットした後、oplot で2番目以降のデータを重ねてプロットする
- plot と同様に、thick, linestyle, psym, symsize 等のオプションが使える
- 表示される範囲(レンジ)は最初の plot で決まる
- 区別しやすいように色を変えるなどすると良い

color オプションによる色の指定 (decomposed colors)

Black	: '000000'XL
Red	: '0000FF'XL
Green	: '00FF00'XL
Blue	: 'FF0000'XL
Yellow	: '00FFFF'XL
Gray	: '7F7F7F'XL

(Blue, Green, Red の順にそれぞれ 00 から FF の間の 256 階調で指定する)

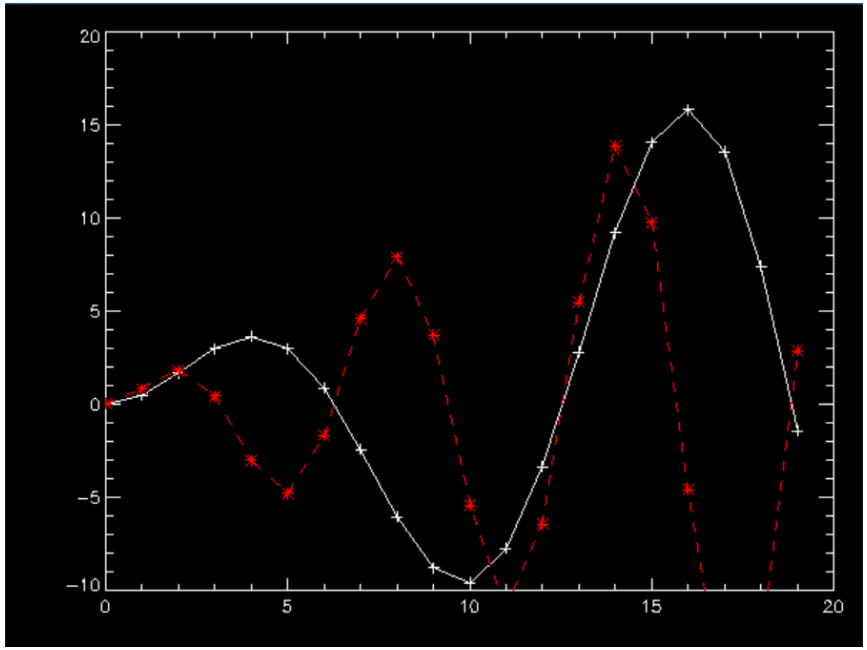
(注) 色の指定がうまく働かない場合は、IDL で使用しているカラーモデルが"decomposed color model" (通常はこれが起動時のデフォルト)になっていない。その場合は下記コマンドを実行する。

```
IDL>device, decomposed=1
```

OPLOT を使った例

最初のグラフの上に、別のグラフを赤色の破線で重ねる

```
IDL> plot, indgen(20)*sin(indgen(20)/2.0), psym=-1
IDL> oplot, indgen(20)*sin(indgen(20)), psym=-2, $
IDL>              linestyle=2, color='0000FF'XL
```



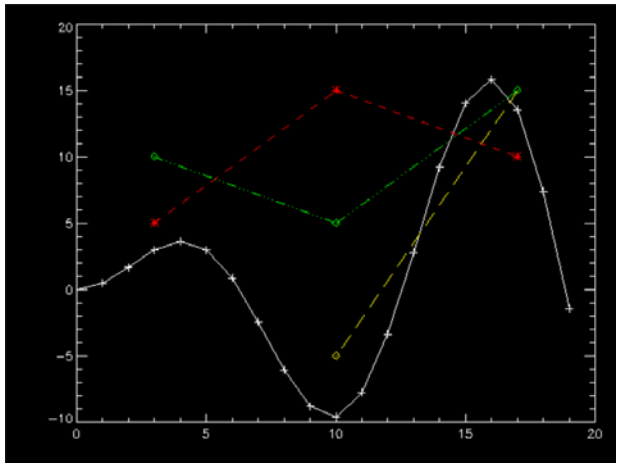
既存グラフィック(plotなど)の上に点や線を描く; PLOTS

- [使い方]

```
plots, x[, y], [, /continue]
```

✓ x, y はスカラーまたはベクトル(1次元アレイ)
✓ /continue オプションは最後のプロット点から続けて線を引く

例)
```
IDL> plot, indgen(20)*sin(indgen(20)/2.0), psym=-1
IDL> plots, [3,10,17], [5,15,10], psym=-2, linestyle=2, color='0000FF'XL
IDL> plots, [3,10,17], [10,5,15], psym=-4, linestyle=4, color='00FF00'XL
IDL> plots, 10, -5, psym=-4, linestyle=5, color='00FFFF'XL, /continue
```



既存のグラフィックの上の座標 x, y に文字列(string)を表示する; XYOUTS

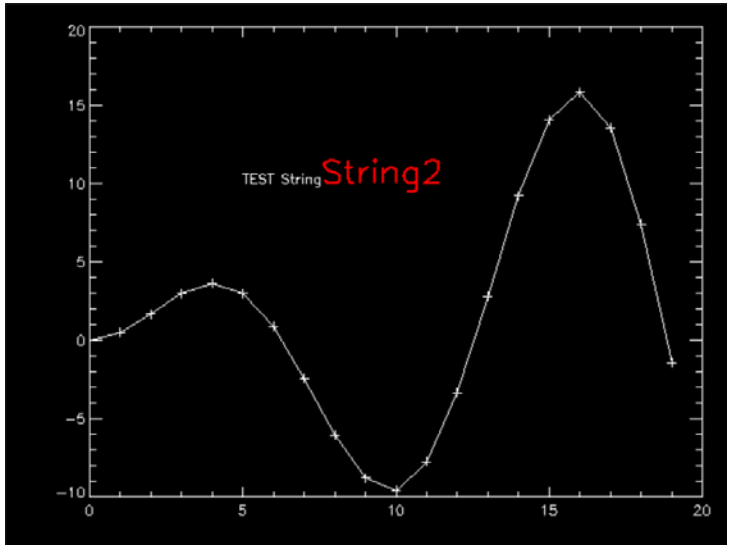
- [使い方]

```
xyouts[, x, y], string
```

✓ 座標 x, y を省略した場合は直前の最終位置が起点になる
✓ オプション charsize, charthick などが使える

例)

```
IDL>plot, indgen(20)*sin(indgen(20)/2.0), psym=-1
IDL>xyouts, 5, 10, 'TEST String'
IDL>xyouts, 'String2', color='0000FF'XL, $
IDL>            charsize=1.8, charthick=2
```



✓ デフォルトでは指定座標(x,y)が文字の左下
✓ ALIGNMENT オプションを使って右揃えや中央揃えにも出来る
✓ 座標の与え方については次ページ

IDL グラフィックスの座標

- プロットグラフィックス上で指定できる座標系は次の3種類が存在する
  - DATA　　; plot 等を実行した後から有効になる座標系。プロットしたデータの値で定義される
  - NORMAL; グラフィックウィンドウの左下を[0.0,0.0]、右上を [1.0,1.0] と定義した座標系
  - DEVICE　; グラフィックウィンドウの左下を原点[0, 0]として、ピクセル単位で定義される座標系

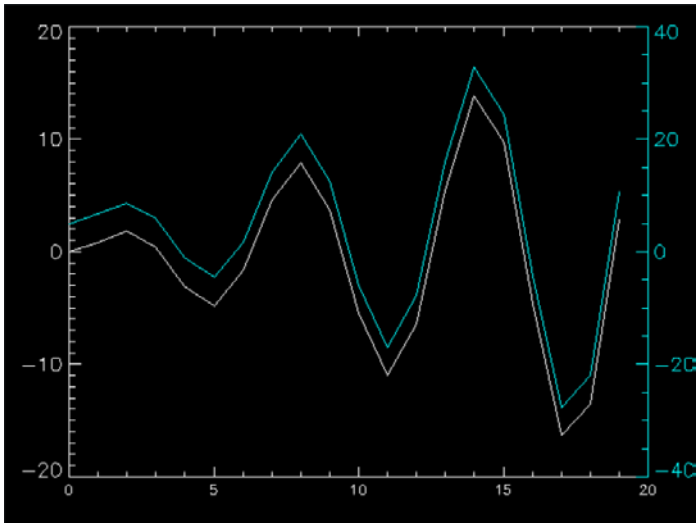
例) グラフィックウィンドウ中央を起点として文字を書く場合

```
IDL> xyouts, 0.5, 0.5, 'TEST String', /Normal
```

軸の作成; AXIS

- たとえば、異なるスケールを持つデータに対して、それぞれ独立のスケールのY軸を左右に作成する(第1軸と第2軸を描く)

```
  ; データ作成(2種類)
IDL> y1 = indgen(20)*sin(indgen(20))
IDL> y2 = indgen(20)*sin(indgen(20))*2+5
  ; データはプロットしないでX軸だけ描く
IDL> plot, y1, ystyle=4, xmargin=[8,6], /nodata
  ; 左側のY軸を描く
IDL> axis, yaxis=0, charsize=1.5
  ; 最初のデータをプロット
IDL> oplot, y1
  ; 右側のY軸を描く。
  ; /save オプションで新しいY軸のスケールを保存
IDL> axis, yaxis=1, yrange=[min(y2), max(y2)], $
IDL>          color='FFFF00'XL, charsize=1.5, /save
  ; 次のデータをプロット
IDL> oplot, y2, color='FFFF00'XL
```



画面を分割する; !p.multi

- グラフィックウィンドウを分割して、複数のプロットを描く
- !p はプロット関係の設定値を保存しているシステム変数構造体。(システム変数は先頭に！が付いている)

(参考) システム変数の構造体 !p の中身を表示する

```
IDL>help, !p
```

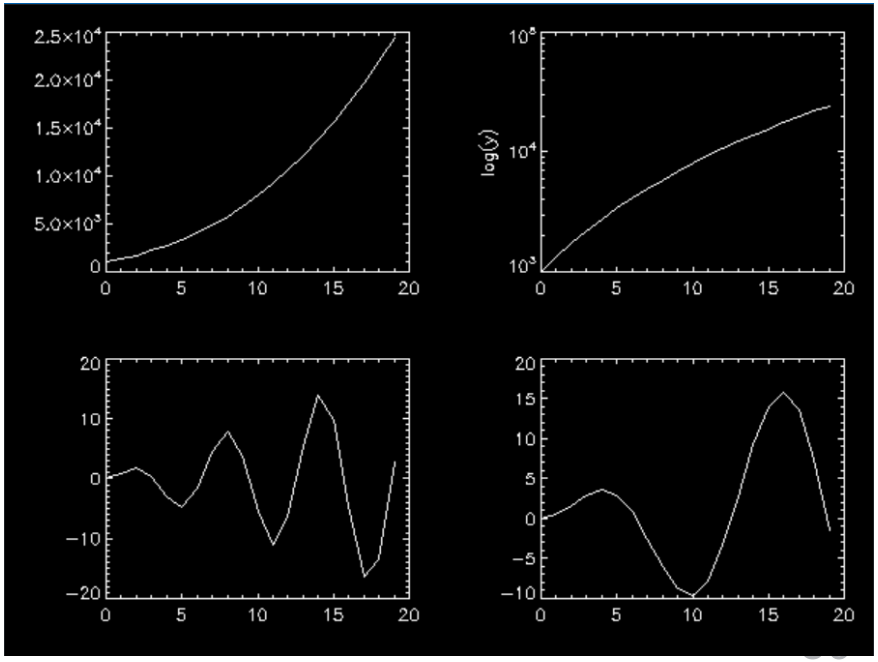
- !p で設定できる値のほとんどは plot や contour などのコマンドのオプションとして個別に指定できる
- しかし、画面分割を指定する !p.multi はオプションでは指定できない
- !p.multi は5つの要素を持つ配列。2番目の要素(!p.multi[1])と3番目の要素(!p.multi[2])がそれぞれ、画面を横と縦にいくつに分割するかを指定する値

(参考) デフォルト(起動時)の !p.multi

```
IDL> print, !p.multi
           0           0           0           0           0
```

例) 画面を4つ (2x2) に分割してプロットする場合

```
IDL> !p.multi=[0,2,2]   ; 配列の3番目まで書けばOK
  ;; これ以降、4つの領域に順番にプロットされる
IDL> plot, (findgen(20)+10)^3
IDL> plot, (findgen(20)+10)^3, /ylog, ytitle='log(y)' ; Y軸logスケール
IDL> plot, findgen(20)*sin(findgen(20))
IDL> plot, findgen(20)*sin(findgen(20)/2.0)
IDL> !p.multi=0          ; 分割設定を元に戻す
```



❗分割した画面の数を超えてプロットすると、また最初の位置から順番にプロットされる(前のプロットは消える)

position キーワード

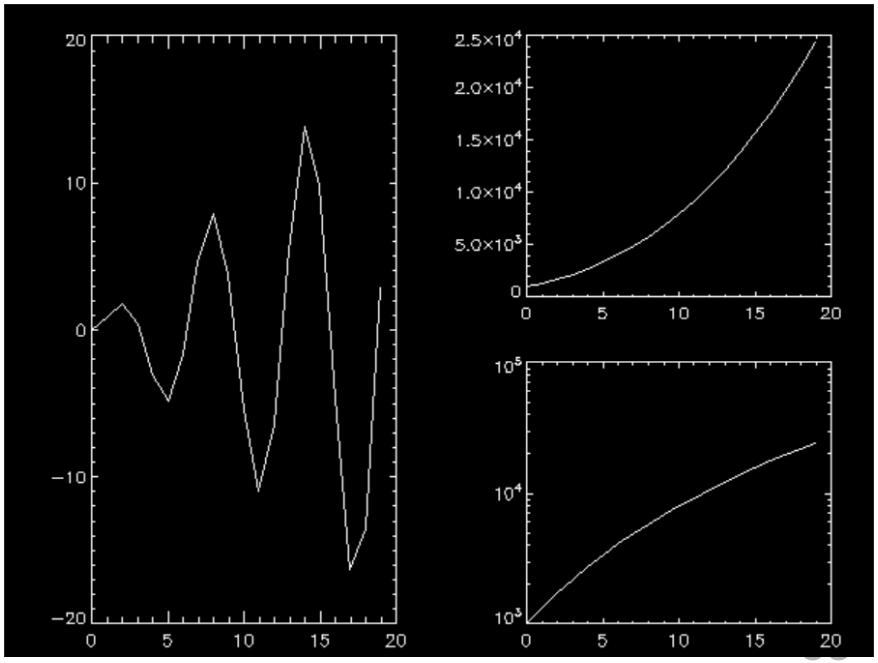
- !p.multi による画面分割は手軽だが、出力される位置や順番は固定されており、細かい設定は出来ない
- 代わりに、ひとつずつ位置 (position) を指定していくことで同じような複数プロットが可能
- [キーワードの書式]

```
position=[x_min, y_min, x_max, y_max]
```

例) 3つのプロットをレイアウトする

```
  ;; データを準備
IDL> dt1=(findgen(20)+10)^3
IDL> dt2=findgen(20)*sin(findgen(20))
  ;; 位置を指定しながらプロット
IDL> plot, dt1, position=[0.60, 0.55, 0.95, 0.95]
IDL> plot, dt1, position=[0.60, 0.05, 0.95, 0.45], /noerase, /ylog
IDL> plot, dt2, position=[0.10, 0.05, 0.45, 0.95], /noerase
```

(注) 2つめ以降は前のプロットを消さないように /NoErase オプションを付ける



Postscript ファイルに出力する

- グラフィックス出力先のデバイスを PS に切り替えてプロットする
  - 最後にファイルを閉じる処理を忘れずに (忘れやすい!)
  - 切り替えた出力デバイスも元に戻す

PS デバイス用のオプション (DEVICE procedure)	
filename	出力ファイル名 (default: idl.ps)
/closefile	ファイルを閉じる
/portrait	縦置き (default)
/landscape	横置き
/color	カラー出力 (default: grayscale)
{x|y}size=value	描画範囲のサイズ
{x|y}offset=value	描画範囲の原点位置
/inches	size と offset の単位を inch にする (default: cm)
font_size=value	フォントサイズ(単位:ポイント, default: 12 points)

例) ポストスクリプト出力手順例

```
;; 前準備
IDL> init_dev = !d.name                ; 現在の出力デバイスを保存しておく
                                       ; !d はデバイスに関するシステム変数の構造体
IDL> set_plot, 'PS'                    ; 出力先として PS デバイスを指定する
IDL> device, filename='output.ps'      ; 出力するファイル名(省略すると 'idl.ps')

;; プロットする
IDL> plot, indgen(20)*sin(indgen(20))      ; 画面には何も出力されない

;; 後処理
IDL> device, /close_file           ; 出力ファイルを閉じる
                                   ; ## これを忘れると白紙のままになったりする ##
IDL> set_plot, init_dev            ; 出力先のデバイスを元に戻す
```

IDL から Linux シェルに抜けて、出力した Postscript ファイルを確認

```
IDL>$
$ gv output.ps
```

- ✓ 通常時の出力デバイスは 'X' (Linux の X Window System の場合)
- ✓ Microsoft Windows の画面の場合 'WIN', プリンタ出力の場合 'PRINTER'
- ✓ PS にカラー出力する場合は decomposed=0 をセット(Indexed color を使用)

エラーバーのプロット

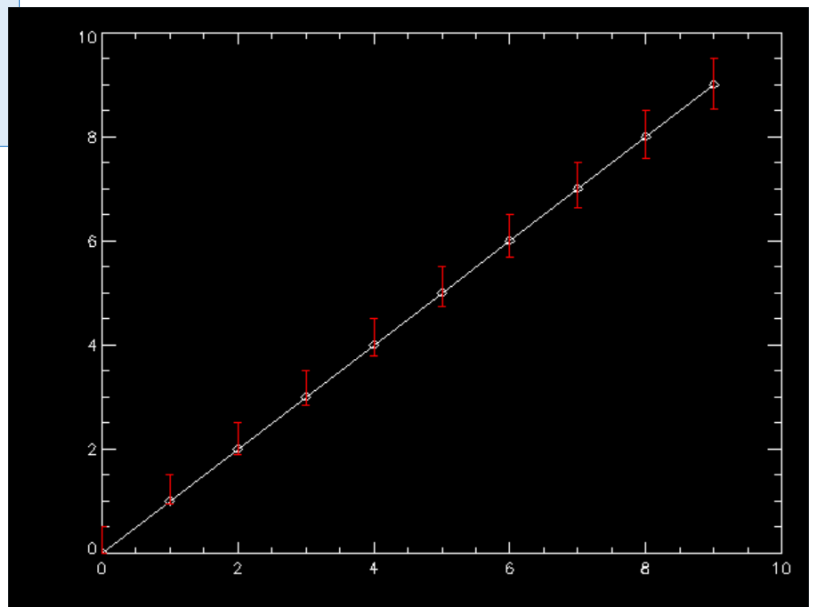
- ライブラリツールを含めていろいろある
- 使用方法や使えるオプションに差があるなどするので、用途と必要に応じて使い分けると良い

エラーバー表示	
errplot	既存プロットの上にエラーバー(正負個別指定)
ploterr	エラーバー付きプロット
oploterr	既存プロットの上にエラーバー(正負同じ)
ploterror	(astrolib) エラーバー付きプロット(x, y 方向) 正負個別プロット可
oploterror	(astrolib) 既存プロットの上にエラーバー(x, y 方向)
errorplot()	function (IDL 8.0 以降)

ERRPLOT を使った例

```
  ;; データ
IDL> dt = indgen(10)
  ;; 下側エラー
IDL> err1 = indgen(10)*0.05
  ;; 上側エラー
IDL> err2 = make_array(10, /float, value=0.5)

  ;; プロット
IDL> plot, dt, psym=-4
IDL> errplot, dt, dt-err1, dt+err2, color='0000ff'xl
```



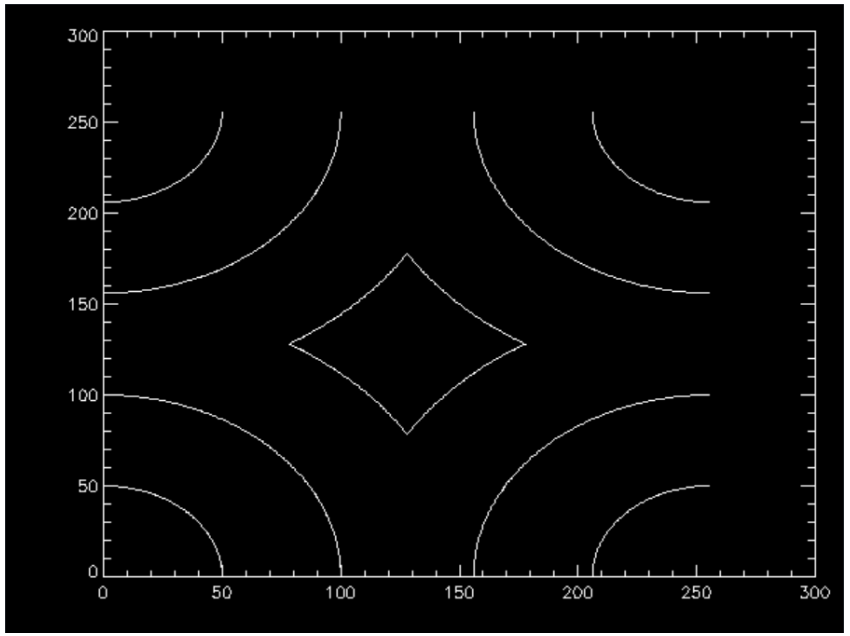
CONTOUR (procedure)

- 2次元イメージ(画像)をコントアで表示する
- [使い方]

contour, z[, x, y]　(z は 1 or 2次元データアレイ)

例)
IDL>contour, dist(256)

✓x, y 座標値を与えない場合は
ピクセル番号でマップされる

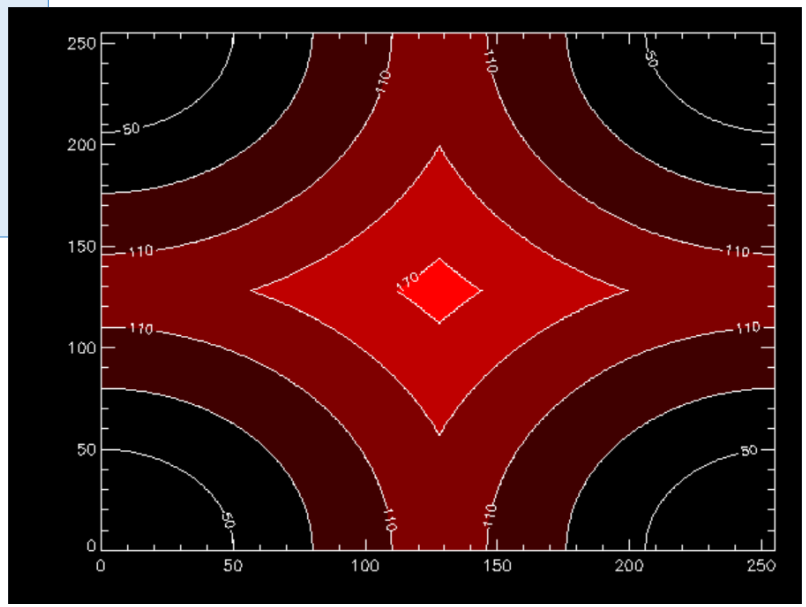


CONTOUR でよく使うオプション	
levels	コントアレベル(任意値)
nlevels	等高線の数(整数 1-60)
/follow	等高線の値を表示(自動調整)
c_labels	ラベル(等高線の値)の任意表示切替 (0/1, default 0)
c_charsize	コントアラベルの文字サイズ
c_color	コントア線の色
c_linestye	コントア線の種類 (0-5, default 0)
/overplot	既存グラフィックスの上にプロット
/fill	レベルごとに色を付ける
/cell_fill	レベルごとに色を付ける
max_value	プロットされる最大値
min_value	プロットされる最小値
/irregular	不規則に配置されたデータでプロット (x, y が必須)

コントアプロット例

- 等高線5本で、ラベルはひとつおきに表示、色を付けてコントア表示する

```
  ;; 等高線の設定値
IDL> lev=[50,80,110,140,170]
IDL> c_lab=[1,0,1,0,1]
  ;; 色を付けてプロット
IDL> contour, dist(256), levels=lev, /fill, /xsty, /ysty
  ;; ラベルを付ける(重ねる)
IDL> contour, dist(256), levels=lev, c_labels=c_lab, /over
```



✓ 色づけとラベル表示は同時に行えないため、2度に分けて描画している

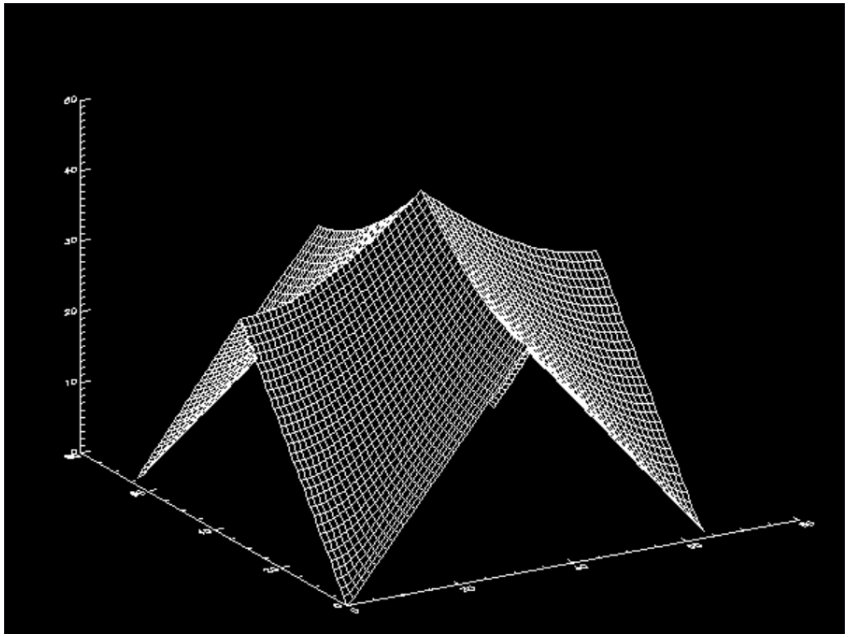
SURFACE (procedure)

- 2次元イメージ(画像)をメッシュ表示する
- [使い方]

surface, z[, x, y]　(z は 1 or 2次元データアレイ)

例)
IDL>surface, dist(64)

✓x, y 座標値を与えない場合は
ピクセル番号でマップされる

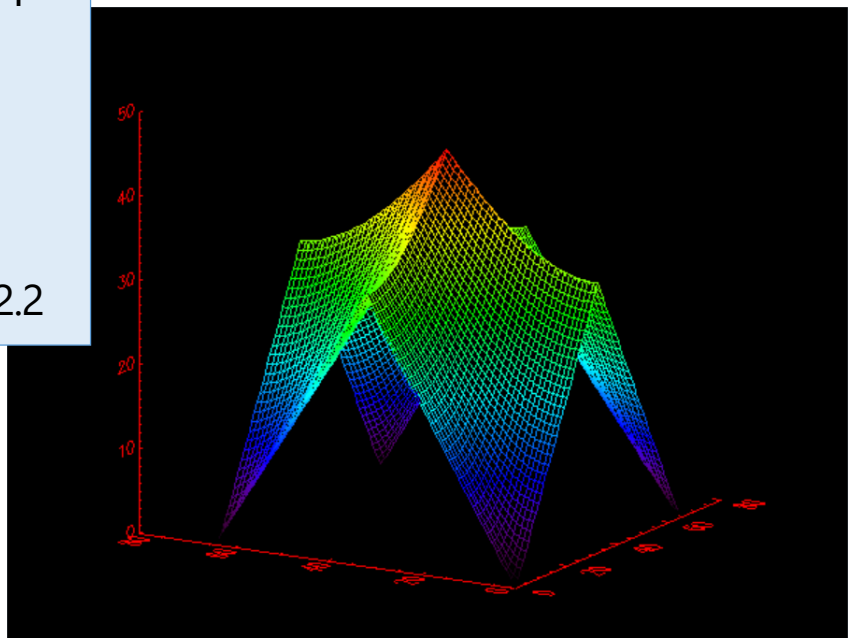


SURFACE でよく使うオプション	
az	Z 軸周りの回転角(度) (右回り、default 30度)
ax	X 軸周りの回転角(度) (右回り、default 30度)
shades	z レベルに対するカラーインデックス (default テーブル最大値)
skirt	裾を表示する場合の z レベル (指定が無ければ裾無し)
/lower_only	下面のみを表示
/upper_only	上面のみを表示
min_value	表示する最小値
max_value	表示する最大値
/{x|y|z}log	(x,y,z)軸を対数スケールにする

surface プロット例

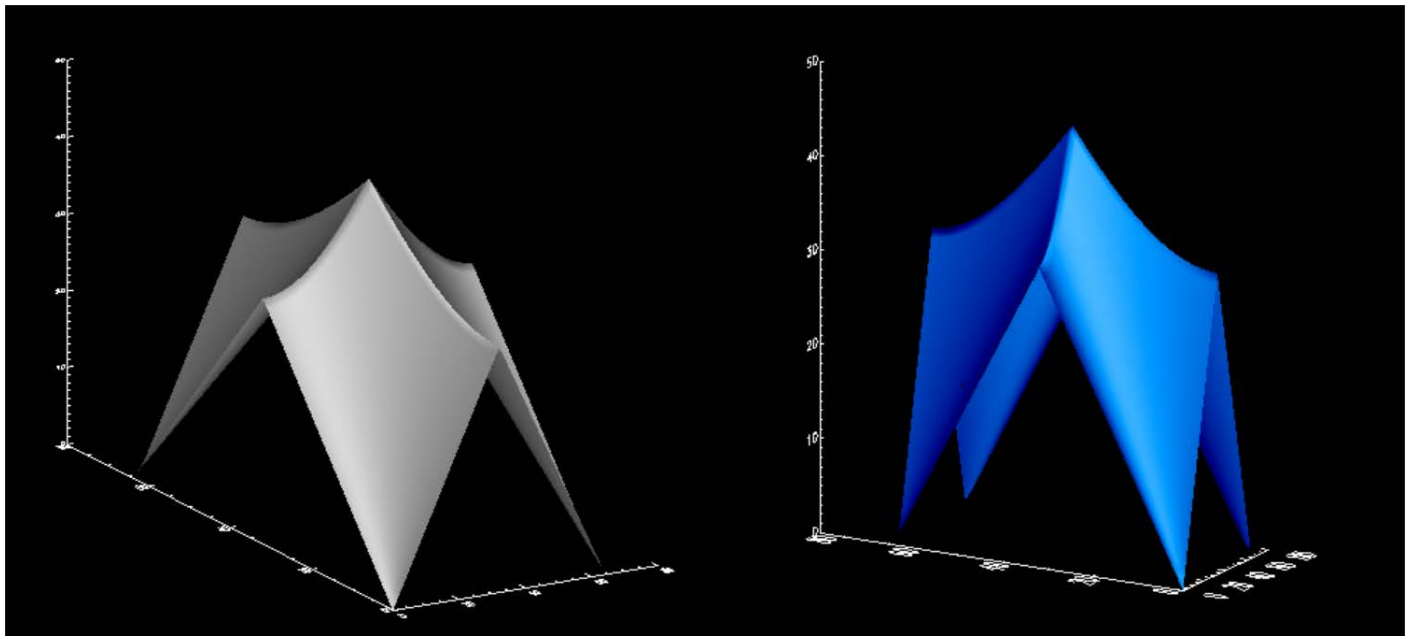
- Indexed color モードにして、
  レインボーカラーテーブルを使って表示する

```
 ;; カラーモードをセットしてカラーテーブルを使用
IDL> device, decomposed=0
IDL> loadct, 13 ; レインボーカラーテーブルのロード
 ;; z データ作成
IDL> z=dist(64)
 ;; カラーインデックスをスケーリング
IDL> shades=bytscl(z)
 ;; 描画
IDL> surface, z, az=60, ax=20, shades=shades, chars=2.2
```



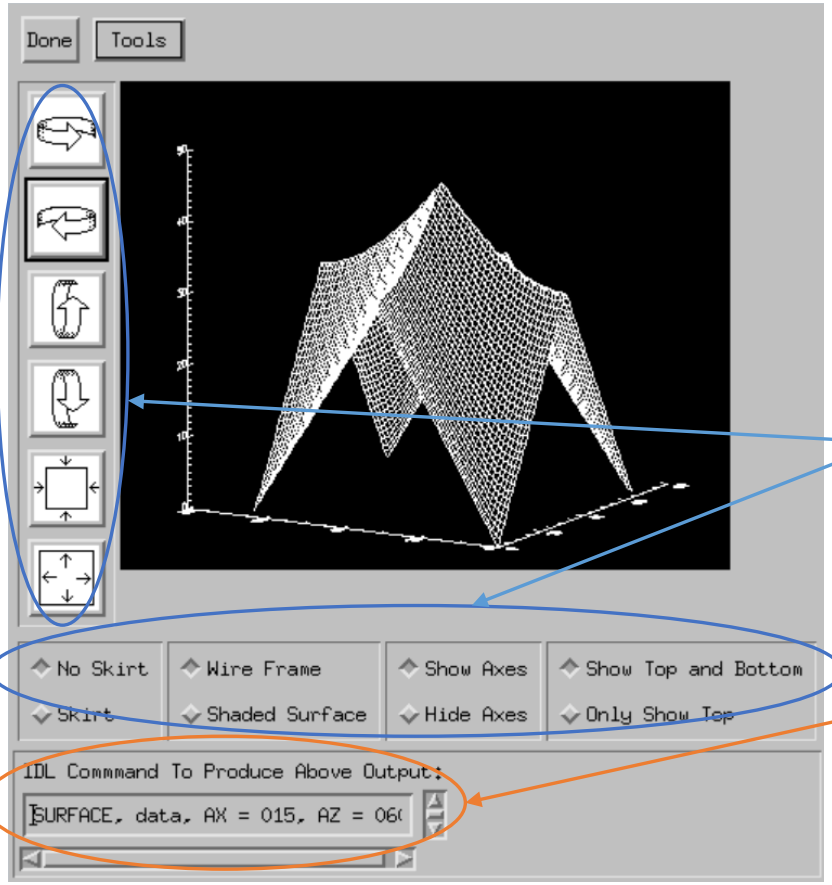
SHADE_SURF (procedure)

- 2次元イメージ(画像)を連続的な影付きの面として表示する



```
; カラーテーブル1番を使用
IDL> device, decomposed=0
IDL> loadct, 1
```

XSURFACE



- surface, shade_surf の結果をGUIで手軽に確認できるユーティリティーツール
- 表示角度や、いくつかのオプションが簡単に試せる
- それぞれの結果を得るためにコマンドラインに入力するべきコマンドも表示される

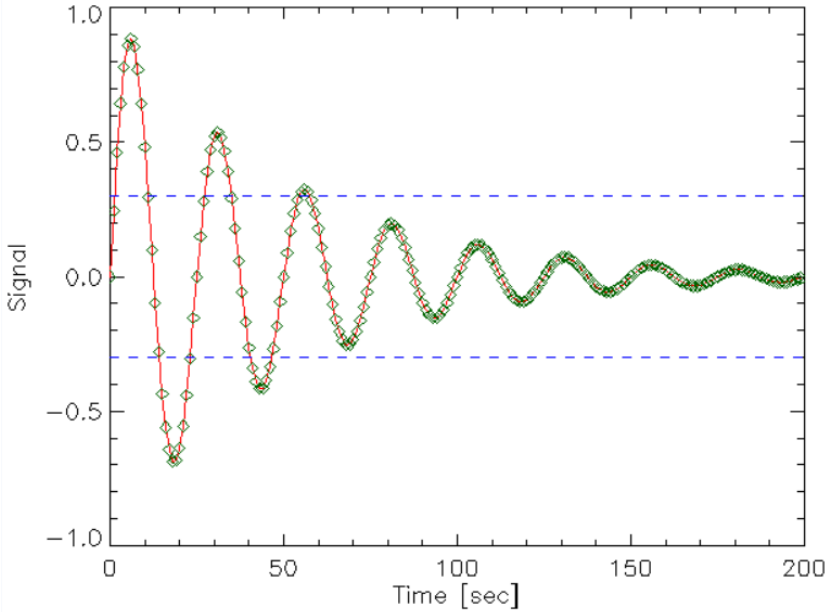
CGPLOT

- Coyote IDL Program Libraries に含まれるプロット作成プロシージャ
- Astronomy User's Library にも導入されている(一緒に配布されている)
- IDL の本家 PLOT プロシージャよりも多機能で、見た目的にも優れたプロットが容易に作成できる
- 基本的に本家 PLOT の拡張になっているので、オプションなどほぼ同様の使い方が可能で、PLOT の代わりとして(置き換えて)使える

❗ Plot のほかにも、Contour, Surface, TV などの代替となる各種ルーチンが Coyote Graphics System (CGS) として公開されている

```
IDL> data = sin(2.0*findgen(200)*!PI/25.0)*exp(-0.02*findgen(200))
IDL> cgPlot, data, psym=-4, color='red', $
IDL>              symcolor='darkgreen', symsize=1.2, $
IDL>              xtitle='Time [sec]', ytitle='Signal'
IDL> cgPlot, [0,200], [0.3,0.3], linestyle=2, color='blue', /Overplot
IDL> cgPlot, [0,200], [-0.3,-0.3], linestyle=2, color='blue', /Overplot
```

- ✓ デフォルトの背景色は白色
- ✓ 色の指定に名前 (red, blue, cyan, etc.) が使える
- ✓ cgPlot は Plot とともに OPlot の拡張 (wrapper) プログラムになっている。/Overplot オプションを使用すると、前の描画の上に重ねて表示される



Coyote Graphics によるグラフィックスのファイル出力

- cgPlot 等は Output オプションを使用してグラフィックスを直接ファイル出力する機能を持つ
  - ただし、ImageMagick および Ghostscript がインストールされている環境が必要

```
cgPlot, x[, y][, Output={'PS'|'EPS'|'PDF'|'BMP'|'GIF'|'JPEG'|'PNG'|'TIFF'} ]
```

あるいは、ファイル名を直接指定する。形式は拡張子で判断される
(例 Output='out.png')

- ポストスクリプト出力デバイス切り替えルーチン
  cgPS_Open, cgPS_Close (各種オプション使用可)

(例)

```
IDL> cgPS_Open, file='test.ps'
IDL> Plot, indgen(10), psym=-4
IDL> cgPS_Close
```

ヒストグラム作成

- IDL には入力データの密度分布を求めるための HISTOGRAM 関数が用意されている
- HISTOGRAM() で得た密度分布をプロットすることでヒストグラムプロットが得られる
- しかし、この IDL 標準の手順はいささか面倒なので、ここでは一気にヒストグラムプロットを行う Coyote Libraries の CgHistoPlot プロシージャを紹介する
- Astronomy Library には含まれて(一緒に配布されて)いない

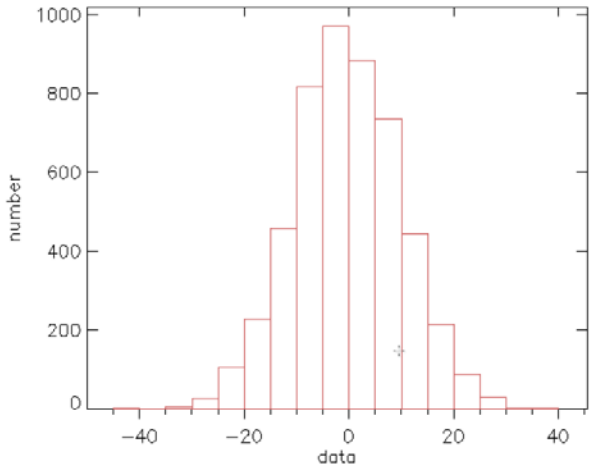
CGHISTOPLOT

cgHistoPlot, data[, binsize=value, nbins=value, datacolorname=string, /fillpolygon, polycolor=string , histdata=variable, locations=varibable, ...]

- ✓ 区切り(bin) は binsize または nbins で調整
- ✓ ヒストグラムの色を指定したり、塗りつぶしたりもできる
- ✓ histdata, locations は出力変数。ヒストグラムデータを再利用したい場合にこの変数に保存する
- ✓ 他にも多数のオプションが存在する (cghistoplot.pro のヘッダの説明参照)

```
IDL> data = RANDOMN(seed, 5000)*10
IDL> cghistoplot, data, bins=5, histd=hd, loc=loc, $
IDL>                    xtitle='data', ytitle='number'
```

(※ randomn() はガウス分布の乱数生成コマンド)



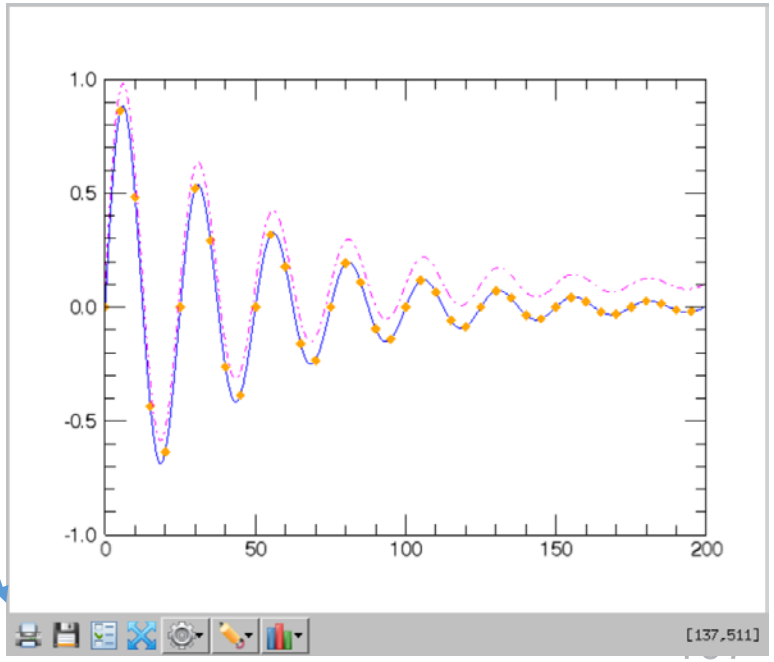
関数型のグラフィックスルーチン

- IDL 8.0 以降、**オブジェクトグラフィックス**による新しいグラフィックスルーチンが導入された
- PLOT ほか基本的なグラフィックス機能が一通り用意されている
- 従来のダイレクトグラフィックスによるルーチンがプロシージャ型だったのに対して、関数型になっている
- IDL6.0 から iTools というオブジェクトグラフィックスルーチンが導入されていたが、動作が重いなどあまり評判は良くなかった(と思う)
- 新しい関数型ルーチンは iTools からは大幅に改良されいてる
- とは言え、ダイレクトグラフィックスに比べると軽快さには劣る
- オブジェクトの属性を後から変更できる、保存や印刷機能が備わっている、などの利点もあるので、試してみて、用途に応じて使い分けると良いだろう

```
IDL> y = sin(2.0*findgen(200)*!PI/25.0)*exp(-0.02*findgen(200))
IDL> ; Plot
IDL> p1 = plot(y, SYMBOL=4, COLOR='blue')
IDL> p2 = plot(y+0.1, COLOR='magenta', linestyle=3, /overplot)
IDL> ; オブジェクトのプロパティを変更
IDL> p1.SYM_INCREMENT = 5
IDL> p1.SYM_COLOR = "orange"
IDL> p1.SYM_FILLED = 1
```

ツールバー:
プロパティの変更、グラフィックスファイル出力、印刷、線や文字などの追加などが GUI からも可能

軸などを選択した状態でマウスホイールを使った拡大縮小も可能



9. 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
IDL> help, 2 + 3L
<Expression>    LONG      =        5
IDL> help, 2 + 3.0
<Expression>    FLOAT     =      5.00000
IDL> help, 3 / 2
<Expression>    INT       =        1
IDL> help, 3 / 2.
<Expression>    FLOAT     =      1.50000
```

← INT + INT

← INT + LONG

← INT + FLOAT

← INT / INT

← 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
```

■ 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
```

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
```

❗ 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

■ ファイル名として使うための、
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_2018Jul24_143514.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()
IDL> print, stime
Wed Jul 28 15:35:59 2018
IDL> month = strmid(stime, 4, 3)
IDL> date = strmid(stime, 8, 2)
IDL> hh = strmid(stime, 11,2)
IDL> mm = strmid(stime, 14,2)
IDL> ss = strmid(stime, 17,2)
IDL> ypos = strpos(stime, '2018')
IDL> year = strmid(stime, ypos, 4)
IDL> fname = 'log_'+year+month+date+'_'+hh+mm+ss+'.txt'
IDL> print, fname
log_2018Jul28_153559.txt
```

10. 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早くなる。
  (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、そちらの方が早い)

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作成関数を使う

```
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.

- 一般的な配列作成関数を使う

```
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]
       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、REFORRM(), 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> 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
```

✓ 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。

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各要素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+1)*2
       4       6       8
```

→ 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

11. 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にしても構わない

- 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

12. 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

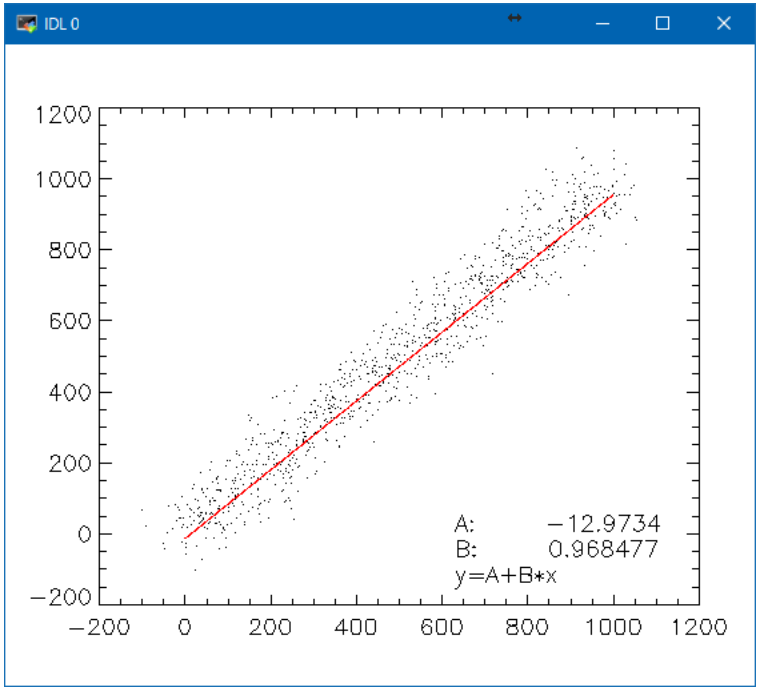
LINFIT 関数

- (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

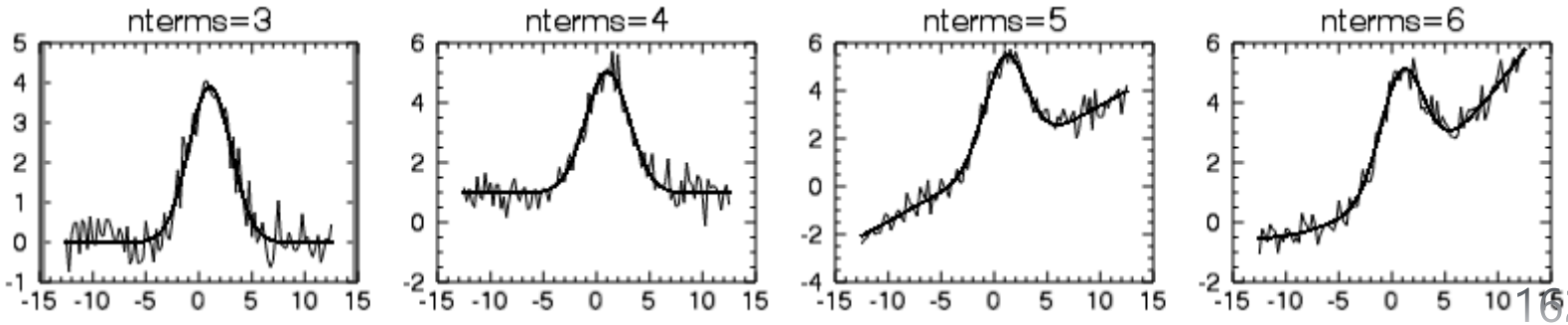
- (x,y)データに対して、ガウシアンカーブと2次関数の組み合わせでフィッティングを行う

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)

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



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)

13. IDL のプログラミング

スクリプト

- IDL のバッチ処理を行う
- すなわち、スクリプトファイルに記述された IDL のコマンドを、1行ずつ順番に、連続的に自動実行する
- コマンドラインから手動で1行ずつ入力する作業を省き、一気に実行できる
- プログラム化するまでもない一連の処理を行うのに便利
- 複数行に渡るループ文(For文など)などを使うためには、プロシージャや関数としてプログラムを書く必要がある
- 実行方法 @[スクリプトファイル名]

例) スクリプトファイル testcode.pro を実行する場合
IDL> @testcode

- スクリプトファイル名は何でも構わないが、*.pro としておくのが無難
  - ユーザが IDL 関連のファイルであると識別しやすくなる。
  - IDL も *.pro は IDL のファイルであると認識できる。(たとえば testcode.pro の実行の際には "testcode" のみの指定でOKで、".pro"まで書く必要が無い。)
  - (pro は本来 procedure の頭 3 文字だが、必ずしもプロシージャではない)

プロシージャとファンクション

- IDL のプログラムにはプロシージャ(procedure)と関数(function)が存在する
- 関数には必ず返値がある
  返値 = function(引数1, 引数2, 引数3, ...)
- プロシージャに返値は無いが引数に処理の結果を戻す事はできる
  procedure, 引数1, 引数2, 引数3, ...
- 引数は入力と出力の両方に使用できる。見た目の区別は無いので、その区別はプログラマと使用者の責任になる
- 処理結果の変数を複数得るためには、引数を使う
  (関数の返値はひとつのみ)
- プロシージャと関数に決定的な違いは無い
  形式の違いのみなので、分かりやすくなるように、用途に合った方を選んで使用すれば良い
  - たとえば、計算結果の数値をひとつ得るためであれば、関数の方が分かりやすい
  - 処理結果として返値を必要としない場合、逆に複数の処理結果を得たい等の場合はプロシージャが分かりやすい

プロシージャ(Procedure)

- IDL のプログラムの基本形

[書き方のルール]

  - 'PRO プロシージャ名' で始めて、'END' で終わる

例) 引数無しの場合

```
-----
PRO PROC_NAME
  ; プログラムコード
  ...
END
-----

[実行]
IDL> proc_name
```

例) 引数(位置パラメータ, キーワード)を持つ場合
　　* 引数については後述

```
-----
PRO PROC_NAME, arg1, arg2, key1=key1, key2=key2
  ; プログラムコード
  ...
END
-----

[実行]
IDL> proc_name, arg1, arg2, key1=key1, key2=key2
```

関数 (Function)

- (パラメータを与えて)実行すると「定められた処理を行った結果として或る値(返値)を返す」という動作を行うプログラム

[書き方のルール]

- 'FUNCTION 関数名' で始めて、RETURN コマンドで値を返し、'END' で終わる

```
例)
-----
FUNCTION FNC_NAME, arg1, arg2, key1=key1
  ; プログラムコード
  ...
  return, value
END
-----
[実行]
IDL> ret= fnc_name(arg1, arg2, key1=key1, key2=key2)
```

- プロシージャと異なり、関数は必ず返値を持つ。従って、基本的に返値の出力先が存在しなくてはならない

例1) 10個の要素を持ったインデックス配列を作成し、変数 'ret' に代入する

```
IDL> ret = indgen(10)
```

例2) 10個の要素を持ったインデックス配列を作成し、内容を画面(標準出力)に表示する

```
IDL> print, indgen(10)
```

❗ ただし、IDL 8.3 からは、コマンドラインで実行された関数は自動的に print に渡されるようになった

❗ プロシージャや関数を自作する際は、 すでに存在するプロシージャ名、関数名と名前が重ならないように注意が必要

プログラムソースファイルの作成

■ ファイル名の付け方(ルール)

- IDL プログラム(プロシージャ・関数)のソースファイル名は、原則としてプログラム先頭で定義した名前(先頭の PROCEDURE や FUNCTION に続けて書いたプログラム名)と一致させる。こうしておくことにより、プログラム実行時に IDL が自動的にソースをコンパイルする(詳しくは後述)
- 拡張子はプロシージャ、関数ともに *.pro
- ファイル名はすべて小文字に揃えることを推奨

複数のルーチンをまとめる

- ひとつのソースファイルの中に複数のルーチン(プロシージャ、関数)をまとめて書くことも出来る
- ファイル名には最初に呼び出されるメインルーチンの名前を付ける
  - 自動コンパイルはプログラム名と同じ名前のファイルを探してコンパイルするため
- ファイル内部ではメインルーチン(親プログラム)を最後に記述し、サブルーチン(子プログラム)を先に記述する
  - 自動コンパイルの場合、プログラムの先頭からコンパイルされる。ファイルの中から呼び出した名前のプログラムをコンパイルすると直ちに実行に移り、以降のプログラムはコンパイルされないため。
  - 手動コンパイル(.compile コマンド)した場合は、ファイル全体がコンパイルされる。

サブルーチンの書き方の例

- 例えば、プログラムの親子関係(呼び出しの順序)が次のようになっている場合のファイル内部の順番は下図

proc0.pro

```
PRO PROC1
  …
END
PRO PROC3
  …
END
PRO PROC2
  …
END
PRO PROC0
  …
END
```

変数のスコープ(有効範囲)

- IDL で通常作成する変数はローカル変数
- ローカル変数は変数を作成したプログラム内でのみ有効で、プログラム実行終了時には破棄され、メモリが解放される
- どのプログラムレベルでも(メインレベルでも、呼び出されたどのプログラムの実行中でも)有効な(使用できる)変数はグローバル変数と呼ぶ。
  IDL では、通常は ! (exclamation mark) で始まるシステム変数だけがグローバル変数。

グローバル変数の例

[read-only] !PI (単精度の円周率)

[writable] !p.multi (画面分割)

COMMONブロック

- 同じ変数データを複数のプログラムから使用したい場合は、たとえば引数としてプログラム間で受け渡しを行うなどする
- しかし、煩雑になったり、そのような実装が実質的に困難だったりする場合もある。
  このため、グローバルなスコープを持ち、
  異なるプログラムの間で変数を共有する仕組み
  としてCOMMONブロックが用意されている

- COMMON ブロックの定義書式

```
COMMON ブロック名, 変数1, 変数2, 変数3, ....
```

- ✓ この宣言を行うことにより、以降の IDL セッション中は、宣言した変数の集合がメモリ内に保持される
- ✓ 他のプログラムから使用する時は、ブロック名のみ宣言すれば良い(変数名まで記述する必要は無い)
- ✓ COMMON ブロック内の変数の型や、含まれる変数の数をあとから変更することは出来ない。変更したい場合は、一度、現在の IDL (セッション)を終了するか、.reset_session コマンドでメモリをクリアする必要がある

```
PRO proc1
  COMMON SHARE1, var1, var2
  ....
END
```

```
PRO proc2
  COMMON SHARE1
  ....
END
```

COMMONブロックの注意点

- COMMONブロックは複数のプログラムで変数データを共有できる便利な仕組みだが、安易に使うべきではない。他の方法が困難な場合のみ使用するようにする
- なぜならば、下記のようなトラブルの元になりやすい
  - 一度宣言したCOMMONブロックはセッション中の変更ができないなど、融通が利かない
  - 多くのプログラムで使用していると、どこでどんな変数が使われているか、見通しが悪くなる。変数名の衝突が起こったり、プログラム群のメンテナンスを困難にしたりする
  - よく使うCOMMONブロック名で宣言しようとして、すでに存在するCOMMONブロックと重複させてエラーになる

コンパイル

- IDL のプログラム(プロシージャ、関数)は、コンパイルされ、メモリ上に保存されてから実行される
- メモリ上に保存されたプログラムは２度目の使用からはコンパイル処理はされず、メモリから直接呼び出される
  - 従って、ソースファイルを修正した場合は再コンパイルが必要

自動コンパイル

- まだコンパイルされていない(メモリ上に存在しない)プログラムが呼ばれると、IDL は自動的に、
  1. 最初は、カレントディレクトリから
  2. 次に、設定された検索パス(IDL_PATH)から順番に

  プログラム名と同じ名前のファイル(*.pro)を検索して、最初に見つかったファイルをコンパイルする

  ❗別のディレクトリに同名ファイルが存在した場合、検索順が後ろのファイルはいつまでもコンパイルされない

- プログラムの中で別のプログラムを使用している場合、それらも芋づる式に検索してコンパイルする

- 自動コンパイルの動作例

```
IDL> cgplot, indgen(10)
% Compiled module: CGPLOT.
% Compiled module: CGSETCOLORSTATE.
% Compiled module: CGGETCOLORSTATE.
% Compiled module: SETDEFAULTVALUE.
% Compiled module: CGCHECKFORSYMBOLS.
% Compiled module: CGDEFAULTCOLOR.
% Compiled module: COLORSAREIDENTICAL.
% Compiled module: CGDEFCHARSIZE.
% Compiled module: CGDISPLAY.
% Compiled module: CGQUERY.
% Compiled module: CGERASE.
% Compiled module: CGCOLOR.
% Compiled module: CGCOLOR24.
```

呼び出したプログラム名が付いたファイル cgplot.pro が検索され、メインプログラムの CGPLOT がコンパイルされ、さらに CGPLOT から呼び出されている別のプログラムも、自動的にコンパイルされている

手動コンパイル

- 通常は自動コンパイルでプログラムを実行するが、IDL のドットコマンド .compile を使用して、手動で明示的にコンパイルすることが出来る
  - セッション中に行ったプログラム修正を反映したい場合
  - パスが通っていないディレクトリにあるプログラムをコンパイルしたい場合
  - 既存プログラムと同名の別プログラムをコンパイルしたい場合

例)
```
IDL> .compile sample.pro   ; .compile の後ろにカンマ(,)は必要無し
IDL> .compile sample       ; 拡張子 .pro は省略可能
IDL> .com sample.pro       ; .compile コマンド名は短縮可能
IDL> .compile sample1 sample2 sample3  ; 複数ファイルを一度にコンパイル
```

プログラム(ファイル)が存在しない場合のエラー

✓ 存在しないプロシージャを実行しようとした場合

```
IDL> noexist, a, b
% Attempt to call undefined procedure: 'NOEXIST'.
% Execution halted at: $MAIN$
```

✓ 存在しない関数を実行しようとした場合

```
IDL> r = noexist(a, b)
% Variable is undefined: NOEXIST.
% Execution halted at: $MAIN$
```

↑ 関数 noexist() が存在しないので、'noexist' は配列であると解釈したがこれも存在しないので変数の未定義(undefined)エラーが表示される

プログラム実行時エラー対処の際の注意点

- プログラムを実行したがエラーで止まった場合、修正して再コンパイル・再実行を試みてもうまくいかない場合がある。原因として IDL がエラーで止まったプログラムレベルに留まったままであるためであることが考えられる
- この場合は retall コマンドを実行してメインレベル(最上位のプログラムレベル)まで戻ることで解決する
- 他の対処法として .reset_session コマンドを実行して、セッションをすべてリセットする。ただしこの場合、メモリ上に保存されていたすべての変数、コンパイル済みのプログラムがクリアされることに注意

引数

- 引数とは、プロシージャや関数を呼び出すときに渡す値や変数
- IDL のプロシージャ、関数に渡すことが出来る引数には、位置パラメータとキーワードパラメータがある
- あらかじめ定義してプログラムに渡す入力引数と、プログラムの処理結果を保存するための出力引数がある(しかしこれらは外見からは区別できない)
- もちろん、引数が無い(必要としない)プログラムもある

位置パラメータ

- プロシージャや関数の動作に必要性が高い(それらを呼び出すときに与えられるケースが多い)引数に用いられる (⇔キーワードパラメータ)
- とはいえ、必須とは限らない。位置パラメータを与えなくても動作するようにプログラムが作られていれば問題ない
- どの引数がどの役割を持つかは、プログラムに与えられる順番(位置)で定義される
- プログラムで定義されているすべての位置パラメータが必要とは限らない。引数が少ない場合、与えられた個数分だけで動くように作られているプログラムでは問題ない
- ただし、途中を飛ばして与える(1番目と3番目の引数を与えて2番目を飛ばす、など) ことは出来ない

例)

plot[, x], y

- PLOT プロシージャの場合、二つの引数(位置パラメータ) x, y を取る
- 1番目の引数が X データ、2番目が Y データとして扱われる
- 1番目の引数 x は省略できる。引数がひとつの場合は、それを Yデータとして扱うように作られている

キーワードパラメータ

- 基本的にオプショナルな引数に使われる
- 位置パラメータと異なり、名前で区別される
- 実行時の与え方(書式)は

```
Keyword_Name = keyword_Value
```

  - 左辺がプログラムで定義されたキーワード名。右辺は与える値や変数。プログラム内部で使う右辺の変数名は何でも構わないが、左辺のキーワード名と同じか似た名前にしておくと分かりやすい。
- 実行時に指定が無い場合は、デフォルト値で動作する(ようにプログラムを書く)
- 位置パラメータと一緒に使う場合、どの位置にあっても問題ないが、位置パラメータの後ろにまとめるのが分かりやすい
- 実行時のキーワード名は省略が可能 (ambiguity が無い限り)

キーワードパラメータの注意点

- 通常の場合は “左辺=右辺” のように書かれていると、左辺に右辺を代入する (例 a = 1.0, b=c)
- これに対して、キーワードパラメータでは、プログラム内に "キーワード名=変数" と書かれているが、引数として与えられた値などが代入されるのは右辺の変数
- 実行時に、
  - "キーワード名=値" として与えた場合、右辺の値が変数に代入される
  - "キーワード名=変数"として与えた場合、右辺の変数がプログラム内の変数に渡される(参照渡し)

    ※ 参照渡しについては後述

```
; X, Y 軸タイトルを設定
IDL> plot, x, y, xtitle='time [sec]', ytitle='Flux [Jy]'

; キーワード名は省略できる
IDL> plot, x, y, xtit='time [sec]', ytit='Flux [Jy]'

; xti だけでは xtitle の他、xticks や xtickv などと区別できないため、
; 次の例はエラーになる
IDL> plot, x, y, xti='time [sec]', yti='Flux [Jy]'
```

■ キーワードの特別な与え方: /KEYWORD

- キーワードは 0 (off, false) か 1 (on, true) のフラグ指定に使われることが多い。このため、特別な指定方法が用意されている
- /KEYWORD は KEYWORD=1 と同じ意味を持つ

```
IDL> plot, y, /ylog
と
IDL> plot, y, ylog=1
は同じ。
```

❗デフォルト値が 1 のキーワードを 0 にセットする場合は通常通りの書式で KEYWORD=0 と書く。

引数のチェック

- 引数の与え方によって動作を変える場合など、実際に与えられた引数をプログラム内でチェックする必要がある
- また、プログラムの使用者が、プログラマが想定した通りに引数を与えるとは限らない。意図と異なる使われ方をした場合の挙動を定義しておくことも重要

引数のチェックでよく使われる関数	
n_params()	渡された引数(位置パラメータのみ)の数を返す
n_elements()	変数(配列)の要素数を返す。変数未定義の場合は 0
keyword_set()	キーワードがセットされているかどうか判定する
size()	変数の型やサイズ、次元などの情報を調べる

■ プログラムの先頭で、引数チェックを行う例

```
PRO SMPL1, arg1
  ; check arguments
  nel = n_elements(arg1)       ; arg1 の要素数を変数 nel に代入
  IF (nel EQ 0) THEN return    ; arg1 が未定義ならばここで終了
  .....
END
```

```
PRO SMPL2, arg1, arg2
  ; check arguments
  np = n_params()     ; 与えられた入力パラメータの数を np に代入
   CASE np of         ; np の値によって処理を分岐
      1: .......
      2: .......
      ELSE:
    ENDCASE
END
```

```
PRO SMPL3, arg1, arg2, key1=key1
  ; check keywords
  IF keyword_set(key1) THEN BEGIN ; key1 がセットされている場合
    ........
  ENDIF ELSE BEGIN                ; key1 がセットされていない場合
    ........
  ENDELSE
  .......
END
```

```
PRO SMPL4, arg1, arg2, key1=key1
  ; check keywords
  IF ~keyword_set(key1) THEN BEGIN ; key1 がセットされていない場合
    ........
  ENDIF
  .......
END
```

✓ 例えば、処理に必須のパラメータをキーワードで与えるが、指定されなかった場合は内部でデフォルト値にセットする、など。

_EXTRA キーワード

- あるプログラムが内部で他のプログラムを使用する場合、呼び出されるプログラムが使うことの出来るすべてのキーワードを書き下すのは困難
- 例えば plot プロシージャの wrapper プログラムとして、常に対数表示を行うプログラムを作成しようとするとき、/xlog, /ylog の他に、plot プロシージャに許されているすべてのオプションを明示的に書くのは大変
- そんな場合は、明示的に定義されていないキーワードパラメータをサブルーチンに渡すために _extra キーワードが使用できる
- そのプログラムに定義されていないキーワードを付加した場合、_extra キーワードがあればそこに保存され、サブルーチンに渡される

例えば、

```
PRO LOGPLOT, x, y, BACKGROUND=bgcolor, CHARSIZE=chsize, $
      COLOR=dtcolor, LINESTYLE=ls, NODATA=nod, SYMSIZE=syms, ……
  plot, x, y, /xlog, /ylog, BACKGROUND=bgcolor, CHARSIZE=chsize, $
      COLOR=dtcolor, LINESTYLE=ls, NODATA=nod, SYMSIZE=syms, ……
     …….
END
```

のように、使いそうなキーワードをすべて書くかわりに、

```
PRO LOGPLOT, x, y , _extra=ex
  plot, x, y, /xlog, /ylog, _extra=ex
     …….
END
```

と書くことで、logplot にセットされた未定義のすべてのキーワードが、そのまま plot のキーワードとして引き継がれる

✓ _extra キーワードは値渡し
✓ 参照渡しにする場合は _ref_extra キーワードを使用する

※ 値渡し、参照渡しの説明は次ページ

引数の引き渡し(値渡しと参照渡し)

- 引数が「式」「定数」「配列の要素」「構造体の要素」「システム変数」の場合は値渡しになる
- 引数が「スカラー変数」「配列変数」「構造体」の場合は参照渡しになる

- 値渡しの場合、その値が呼び出したプログラムの内部変数にコピーされて使用される(渡された変数の値が別のアドレスにコピーされる)。そのため、元の引数の値は変化しない
- 参照渡し(アドレス渡し)の場合、呼び出されたプログラムの内部で変数の値が変更されると、(同じアドレスが指す、同じ変数なので)元の変数の値も変わる

twice.pro

```
PRO twice, arg
  arg = arg * 2
  print, arg
END
```

```
IDL> arr = [2,4,6,8,10]
IDL> print, arr
       2       4       6       8      10
IDL> twice, arr
       4       8      12      16      20
IDL> print, arr
       4       8      12      16      20
```

配列の全体が引数の場合

← twice プロシージャ実行

← arr の中身が変更された

```
IDL> arr = [2,4,6,8,10]
IDL> print, arr
       2       4       6       8      10
IDL> twice, arr[0:4]
       4       8      12      16      20
IDL> print, arr
       2       4       6       8      10
```

配列の要素が引数の場合

← twice プロシージャ実行

← arr の中身は変更無し

- 参照渡しを行った結果、内部でその変数に手を加える処理があると、意図せずに変数の値を変えてしまうというケースが発生するので注意が必要
- プログラムの作成者の注意として、引数として受け取った変数の中身を変えたくない場合は、内部ではまず別の変数にコピーして、その変数に対して処理を行うようなコードを書くようにする

14. 簡単なプログラム

IDLプログラムの基本構造とUsage

```
;+
; NAME:
;       MYPROC
;
; PURPOSE:
;       .....
;
; CALLING SEQUENCE:
;       MYPROC, X, Y, A[, KEY1=key1, /KEY2]
;
; INPUTS:
;       X:    .....
;       Y:    .....
;
; OUTPUTS:
;       A:    .....
;
; KEYWORDS:
;       KEY1:    .....
;
; EXAMPLE:
;       myproc, x, y, /key2
;
; MODIFICATION HISTORY:
;    Written by ******
;    Last Modified 2017/??/??
;-
PRO MYPROC, ARG1, ARG2, ARG3, KEY1=key1, KEY2=key2
       .....
       (code)
       .....
END
```

Usage

- IDL のプログラムの先頭には、通常 ;+ 行と ;- 行に囲まれた documentation header が書かれる
  - すべての行はコメントとして記述される (先頭に ;)
- 内容は、プログラムの目的、使用方法、引数の説明、変更履歴、など
- 記述は必須ではない
- このヘッダが書かれていれば、使いたいプログラムのソースコードをエディタで開いて先頭を読むことで、その利用方法などを確認できる
- DOC_LIBRARY プロシージャを使って表示や印刷することもできる

制御文

- 条件によって処理の流れを変えたり、処理を繰り返したりするための、フロー制御を行う仕組みが、他の言語と同様 IDL にも用意されている
- IDL の特性上、配列計算には繰り返し処理を行うループではなく配列処理を行うのが良いが、それ以外の場合で、ループや分岐が必要になる場面は多い
- IF, FOR, FOREACH, WHILE, CASE, SWITCH, GOTO, etc.

IF 文

- 基本形

- 条件式が真(値が1)の場合は実行文1を実行する
- 偽(値が0)の場合に、ELSE 文があれば実行文2を実行する

```
IF (条件式) THEN (実行文1){ ELSE (実行文2)}
```

✓ すべて一行に書かれる($で改行することは可能)
✓ 実行文は1コマンド

```
IF (条件式) THEN BEGIN
  (実行文1)
ENDIF{ ELSE BEGIN
  (実行文2)
ENDELSE}
```

✓ 実行文は複数行(複数コマンド)が可能。
✓ BEGIN で実行文のブロックを開始する。
BEGIN ブロックの最後は ENDIF か ENDELSE。

- 条件 IF は入れ子にして、分岐を増やすことが出来る

```
IF (条件式1) THEN BEGIN
  (実行文1)
ENDIF ELSE IF (条件式2) THEN BEGIN
  (実行文2)
ENDIF ELSE BEGIN
  (実行文3)
ENDELSE
```

✓ 上の場合、「条件1が成立」か「条件2が成立」か「それ以外」かによって、実行する処理が分かれる

❗ ELSEIF は無い (ELSE IF である)

条件式の書き方

- 条件式(評価式)では次の演算子がよく使われる
  EQ, NE, GE, GT, LE, LT, AND, OR, &&, ||, ~
- 条件式は複数の条件を組み合わせる事も出来る

(例1)

```
IF (a EQ 5) THEN .....
```

(例2)

```
IF ((a EQ 5) AND (b NE 0) OR (c GT 100)) THEN .....
```

- 条件式の括弧は必ずしも必要無い。ただし、読みやすさ、複数条件を組み合わせた場合の分かりやすさのために、適宜、括弧でくくるのがおすすめ

FOR 文

- 基本形

- 処理を所定の回数繰り返す
- ループ変数(カウンタ)を増減させて、定められた条件が満たされるまで繰り返す

```
FOR i = n1, n2{, inc} DO (実行文)
```

✓ i の値を n1 から始めて、実行文を処理するたびに inc 分増加させ(指定が無ければ +1)、n2 まで達したら終了する

```
FOR i = n1, n2{,inc} DO BEGIN
  (実行文)
ENDFOR
```

✓ BEGIN ブロック(BEGIN で開始して ENDFOR で終わる)の中には複数行の実行文が書ける

ループ変数について

- ループ変数(i, j がよく用いられる)は通常、整数を用いることが多いが、実数でも構わない
- 以前は INT (16bit整数)を用いた場合、上限値(32767)を超えるとエラーが発生した。このため、LONG (32bit整数)の使用が推奨された

例) FOR i=0L, 40000 DO ....

→IDL 8.0 以降では、オーバーフローする場合、自動的に型変換が行われるようになった

```
IDL> FOR i=0,32000 DO j = i
IDL> help, i
I               INT       =    32001
IDL> FOR i=0,33000 DO j = i
IDL> help, i
I               LONG      =       33001
IDL> FOR i=0,33000.0 DO j = i
IDL> help, i
I               FLOAT     =       33001.0
```

ただし、符号無し整数を指定した場合 (ex. i=0B) は、自動的な型変換はされない

WHILE 文

- 基本形
- 条件式が真である間は実行文を繰り返す

```
WHILE (条件式) DO (実行文)
```

```
WHILE (条件式) DO BEGIN
  (実行文)
ENDWHILE
```

✓ BEGIN ブロック(BEGIN で開始して ENDWHILE で終わる)の中には複数行の実行文が書ける

❗ いつまで経っても条件式が真のままで処理が止まらない
無限ループを作らないように気をつけること

CASE 文

- 基本形

- 条件よってケースを分けて処理を分岐する
- IF 文を入れ子にして複数の条件判断を組み合わせるような場合は、代わりに CASE 文を使うと見通しが良くなる場合がある

```
CASE value OF
  expression: (実行文)
  ....
  expression: (実行文)
  ELSE: (実行文)
ENDCASE
```

✓ 実行文が複数行になる場合は BEGIN ブロック(BEGIN で開始して END で終わる)を使用する

❗ 似た用途では SWITCH文もある

IF 文と CASE 文の比較

- IF 文を使った例

```
IF (x EQ 1) THEN BEGIN
  print, 'CASE 1'
ENDIF ELSE IF (x EQ 2) THEN BEGIN
  print, 'CASE 2'
ENDIF ELSE BEGIN
  print, 'CASE 3'
ENDELSE
```

- CASE 文を使った例

```
CASE x OF
  1: print, 'CASE 1'
  2: print, 'CASE 2'
  ELSE : print, 'CASE 3'
ENDCASE
```

❗ CASE 文では分岐条件がすべての場合に対応できるように注意する。指定された条件のどれにも当てはまらない場合、エラーになってしまう。ELSE を有効に使う。

❗ この例では分岐判定に使う x は数値になっているが、文字列を使う事も出来る。

三項演算子 ?:

- IF-THEN-ELSE の代わりに "?:" を使うとすっきりする場合がある
- X ? A : B の形式で使用して、条件 X が真なら A を、偽なら B を返す

(IF 文を使って書いた場合)
IF (x GT y) THEN z = x ELSE z = y
↓
(三項演算子を使って書いた場合)
z = (x GT y) ? x : y

BREAK & CONTINUE コマンド

- BREAK コマンドは、FOR 文や WHILE 文のループの中から、あるいは CASE 文や SWITCH 文の分岐から、処理を終わって抜け出す
- CONTINUE コマンドは、FOR 文や WHILE 文のループ処理の中で、以降の処理を飛ばして、次のループの処理に移る

15. データの入出力

コンソール上の入出力

標準入力・標準出力を使った入出力

- キーボードからの入力 READ
- 画面への出力 PRINT

□ FORMAT キーワードオプションが使用可能

```
IDL> read, a, b
: 8
: 12
IDL> print, a, b
      8.00000      12.0000
```

❗ READ により値を格納する変数が未定義の場合、FLOAT 型になる

✓ 整数を与えても FLOAT 型になる。文字列を与えるとエラーになる
✓ あらかじめ定義された変数の場合は、その型が維持される

- read で FLOAT 型以外にしたい場合は、まず変数を希望の型で作成してから read で読み込む

```
IDL> a=0d
IDL> read, a
: 1.2
IDL> help, a
A               DOUBLE    =        1.2000000
```

- FORMAT オプションの利用例 : 16進数で入力する

```
IDL> read, a, format='(Z)'
: ff
IDL> help, a
A               FLOAT     =        255.000
```

テキストファイル入出力

基本手順

1. ファイルを開く OPENR, OPENW, OPENU
2. 読み書きを行う READF, PRINTF
3. ファイルを閉じる
    1. ファイルを閉じる CLOSE
    2. 論理ユニット番号を解放してファイルを閉じる FREE_LUN

ファイルを開く

OPENR	既存ファイルを読み込み専用モードで開く
OPENW	新規ファイルを読み書きモードで開く
OPENU	既存ファイルを読み書きモードで開く

```
OPENR[W,U], lun, File[, /GET_LUN]
```

- 開いたファイルには論理ユニット番号 (Logical Unit Number: LUN) が割り当てられる
- LUN は自分で指定するほか、空いている番号を自動で割り当てるオプション /GET_LUN を使うことも出来る

```
IDL> OPENW, 1, 'test1.txt'
IDL> OPENR, lun, 'test2.txt', /get_lun
```

読み書きを行う

- テキストファイルの読み込み READF
- テキストファイルへの書き出し PRINTF

```
READF, lun, var1, var2, var3, .....[, FORMAT=value]
PRINTF, lun, var1, var2, var3, .....[, FORMAT=value]
```

✓ 通常は IDL が自動的に変数の内容を判断して書式が処理されるが、意図通りになるとは限らない
✓ その場合、必要に応じて明示的に書式指定(format=)を行う

```
IDL> printf, lun, 'TEST: ', 1.3, format='(A, D4.2)'
```

➢ readf を使って formatted file からデータを読み込むためには、あらかじめ、その内容(表記の形式や、何行何列のデータか、など)を知っておく必要がある

ファイルを閉じる

- CLOSE

```
CLOSE[, lun]
```

❗ ファイルを閉じる処理を行わないと、printf などでファイルに出力した内容がきちんと反映されない

- FREE_LUN

```
FREE_LUN[, lun]
```

(ファイルが開いていたら、そのファイルを閉じてから) LUN を解放する。通常、/GET_LUN オプションを使ってフィルを開いていた場合に使用して、割り当てられていた LUN (100-128) を他のファイルで再使用できるようにする。

READCOL プロシージャを使ってテキストファイルを読み込む

- Astronomy User's Library に含まれるプログラム
- (デフォルトでは)カンマかスペースで区切られたデータ列が書かれたテキストファイルから、内容を簡単に読み込むことが出来る
- ユーザは、自分でファイルを開いたり閉じたりの処理はせずに、直接ファイルを指定して実行できる
- 読み込めるデータ列数は、最大 50
  (2017年現在。過去、どんどん増えてきた)
- 汎用的に使えるように作られている反面、巨大ファイルの読み込みにはスピードの面で不向き。スピード重視なら専用の読み込みルーチンを自作するべき

```
READCOL, filename, v1, [ v2, ... v50 , DELIMITER= ,FORMAT = , SKIPLINE = ]
```

✓ 変数 v1, v2, ... (変数名は自由に付けて良い)にデータを読み込む
✓ DELIMITER: 区切り文字の指定, FORMAT: フォーマット指定,
SKIPLINE: ファイル先頭のコメント行などを無視したい場合に指定

[演習] READCOL 使用例

- 天体カタログファイルを読み込んでみる
- サンプルファイル AKARI_BSC_sub.txt

```
       RA          DEC    FLUX90[Jy] FLUX140[Jy]      FERR90      FERR140    FQ90  FQ140
353.33686    59.27372      0.84210      0.97231     0.08600     0.22323       2       1
354.92008    61.21272      1.38426      8.39954     0.22443     1.47207       1       3
348.95788    64.87155      0.78687      3.30825     0.07724     0.52001       3       3
353.64587    61.35542      1.21891      4.98273     0.11845     1.09378       3       1
354.61934     4.00820      0.78750          NaN     0.38694         NaN       3       1
239.08909   -47.03910      0.77609      3.35555     0.11283     1.68992       3       3
306.10227    43.82911      2.17332      2.72243     0.35359     0.89352       3       1
348.58965    62.69154      1.57482      5.44634     0.10212     0.86262       3       3
268.32883   -24.19715      3.61970     14.13531     0.31894     3.13412       3       1
...
...
```

[データ各列の内容] "赤経RA", "赤緯DEC", "90μm Flux", "140μm Flux", "90μm Fluxエラー", "140μm Fluxエラー", "90μm質指標", "140μm質指標"

[手順]

- ファイル内容をエディタなどで確認
- データ列数と内容に合わせて、受け取る変数名を適当に指定

[演習回答例]

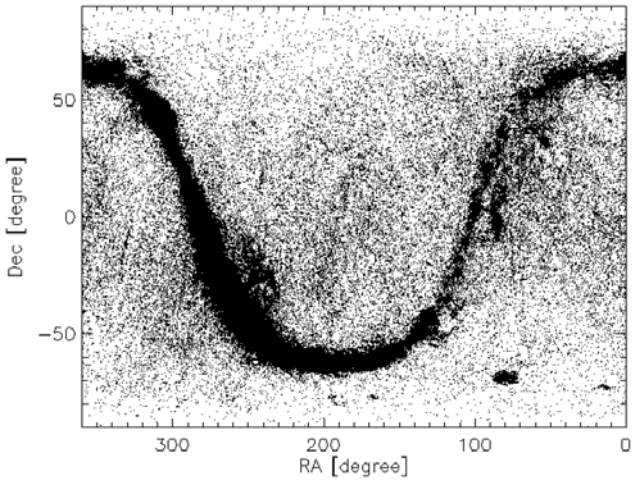
```
IDL> readcol, 'AKARI_BSC_sub.txt', ra, dec, $
IDL>      f90, f140, ferr90, ferr140,  fqual90, fqual140, $
IDL>      format='(F,F,F,F,F,F,I,I)', skip=1
```

正しく読み込めているかどうか
確認してみる。

例えば、

- 座標位置(RA, DEC)をプロットして、
  天体の分布がもっともらしいかどうか
  確かめる
- 天体のフラックスの分布を見てみる
- 異なるバンド(波長)間のフラックスの
  相関を見てみる

など



天体位置分布の散布図

```
IDL> cgplot, ra, dec, psym=3, xra=[360,0], yra=[-90,90],
IDL> xtitle='RA [degree]', ytitle='Dec [degree]'
```

90μmバンドの天体フラックス分布(ヒストグラム, 対数スケール)

```
IDL> cghistoplot, alog10(f90), xtitle='log(Flux90)', ytitle='source number'
```

90μmバンドと140μmバンドの天体フラックス相関(対数スケール)

```
IDL> cgplot, f90, f140, /xlog, /ylog, psym=3, $
IDL>     xtitle='Flux90 [Jy]', ytitle='Flux140 [Jy]', $
IDL>     xrange=[0.1,10^4], yrange=[0.1,10^4]
```

FITS ファイルの取り扱い

- FITS (Flexible Image Transport System) フォーマットは天文学で使用される標準化されたデータフォーマット。画像のほかスペクトルデータや、ASCIIまたはバイナリの表(テーブル)形式のデータも格納できる
- IDL の組み込みルーチンには FITS は扱うものは無いが、Astronomy User's Library (AstroLib) のルーチンによりサポートされている
- AstroLib には IDL 入出力・編集を行うルーチンセットが複数存在している
  - それぞれに長所・短所があるので、目的に応じて、あるいは好みで選択

FITS ファイルの読み込み
MRDFITS() の使用

- MRDFITS() は標準的な FITS ファイルから、画像データをアレイに、ASCIIまたはバイナリテーブルデータを構造体に読み込むことが出来る
- 圧縮された *.gz (gzip compressed) ファイルもそのまま読み込める
- 対応する FITS 書き込み用の関数は MWRFITS()

```
Result = MRDFITS(Filename/FileUnit,[Exten_no/Exten_name, Header, ...])
```

✓ FITS データは、1つまたは複数の Header and Data Units (HDUs) のシーケンスで構成されている
✓ 拡張 HDU を読み込むには Exten_no を指定する (デフォルトは 0 で、プライマリ HDU を読み込む)
✓ Exten_no の後ろに出力用変数 Header を指定すると、ヘッダ情報が文字列アレイに返される

イメージ FITS データの読み込み例

```
IDL> file = 'M31_100um.fits'
IDL> img = mrdfits(file, 0, hd)
% Compiled module: MRDFITS.
% Compiled module: FXPOSIT.
% Compiled module: MRD_HREAD.
% Compiled module: FXPAR.
% Compiled module: GETTOK.
% Compiled module: VALID_NUM.
MRDFITS: Image array (300,300)  Type=Real*4
% Compiled module: MRD_SKIP.
IDL> help, img, hd
IMG             FLOAT     = Array[300, 300]
HD              STRING    = Array[135]
```

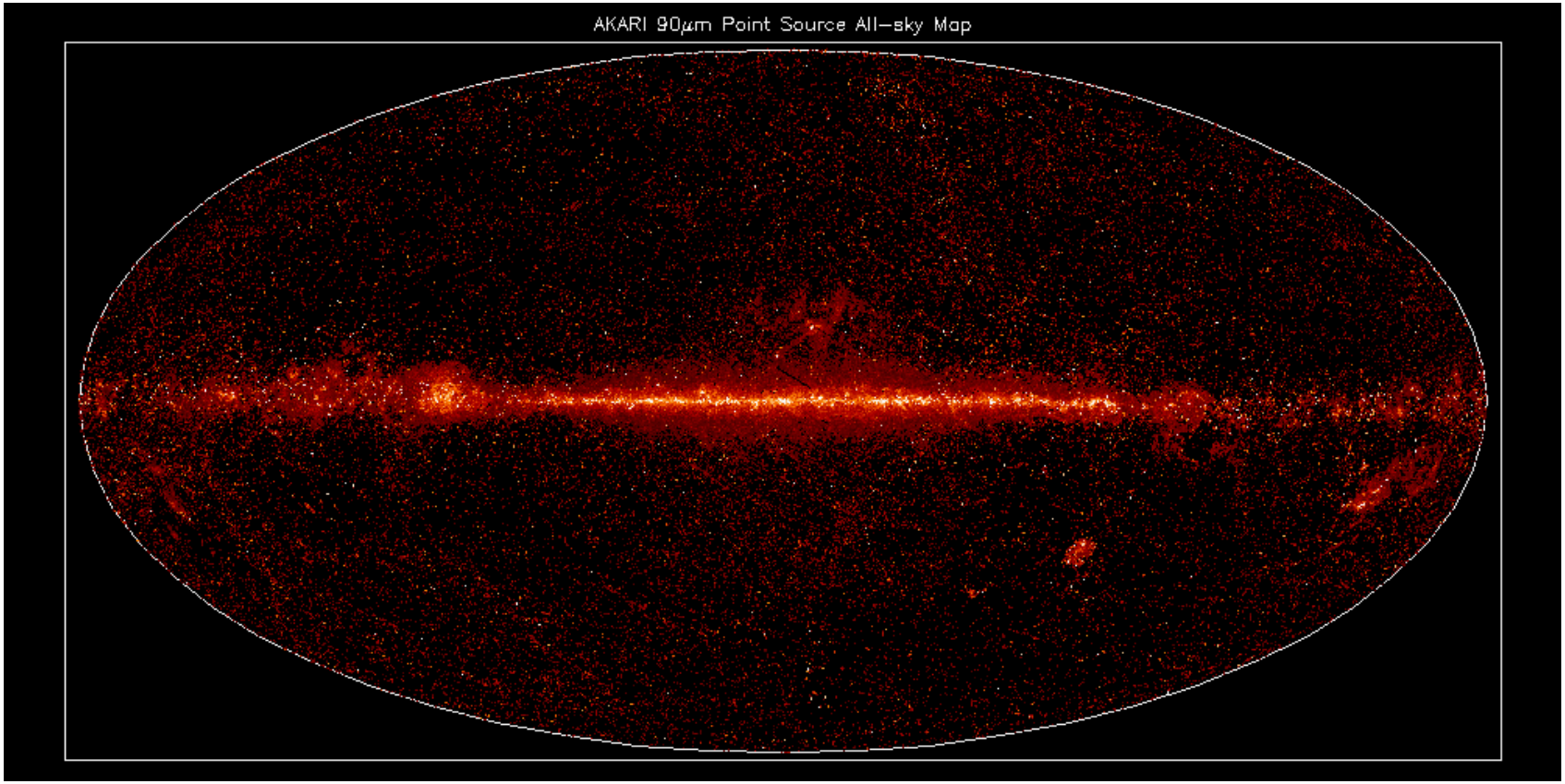
→ 演習問題２へ

バイナリテーブル FITS データの読み込み

```
IDL> file='AKARI-FIS_BSC_V1.fits.gz' ; AKARI/FIS BSC ver.1
IDL> cat = mrdfits(file, 1, hd)
% Compiled module: MRDFITS.
% Compiled module: FXPOSIT.
% Compiled module: FXMOVE.
% Compiled module: MRD_HREAD.
....
MRDFITS: Binary table.  36 columns by  427071 rows.
IDL> help, cat, hd
CAT             STRUCT    = -> <Anonymous> Array[427071]
HD              STRING    = Array[99]
  ;; 別にプライマリヘッダも読み込む
IDL> dmy = mrdfits(file, 0, phd)
MRDFITS: Null image, NAXIS=0
IDL> help, phd
PHD             STRING    = Array[21]
```

→ 演習問題３へ

AKARI/FIS All-Sky Survey Bright Source Catalogue (BSC)



(演習用データ)

演習問題

[演習1] 検出器信号の確認と評価

サンプルデータ

赤外線天文衛星「あかり」の遠赤外線検出器 FIS の時系列信号 (1時間分の観測データ; IDL save フィル形式)
ファイル名 : FIS_SW_20061102140000_gb.sav

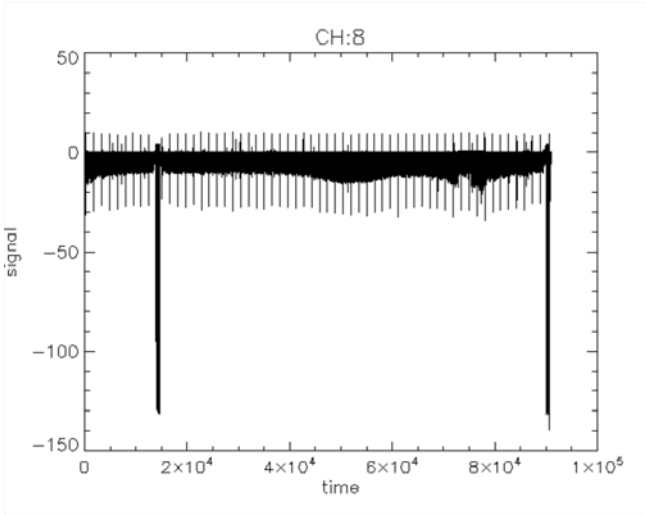
データファイルの内容

1. flux
   100チャンネル(ピクセル)を持つFIS検出器のサーベイ観測中の時系列信号
2. bad
   バッドデータ(様々な理由から適正な検出器信号と認められないデータ)の位置を示す bad フラグ(1/0)。1 が bad

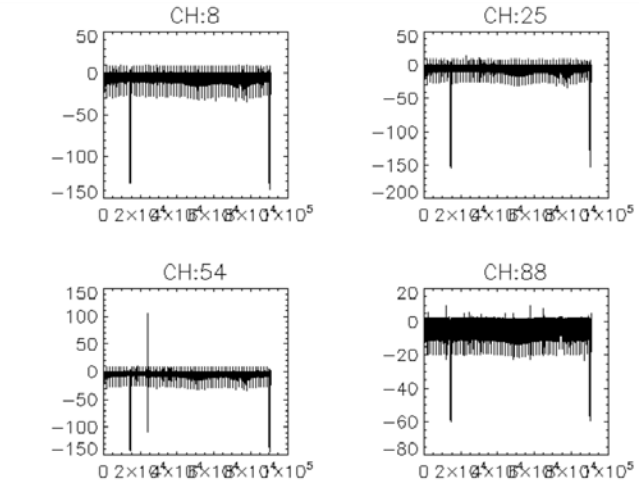
1. 適当なチャンネル(0-99)を選んで信号をプロットせよ。
   ヒント: cgplot (plot などでもOK)
2. 複数のチャンネル(例えば任意の4つほど)の信号をマルチプロットしてみよ。
   ヒント: !p.multi
3. 適当なチャンネルの信号を bad フラグ(1 が bad data)を参照して、good データのみでプロットせよ。
   ヒント: where()
4. good データの適当な一部を取り出して拡大プロットせよ。そのデータにスムージングをかけよ。結果を重ねてプロットせよ。
   ヒント: smooth(), cgplot with /overplot オプション

(参考)

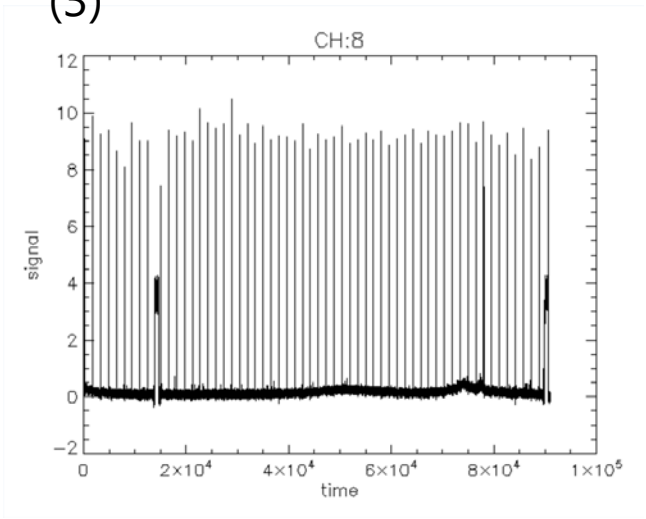
(1)



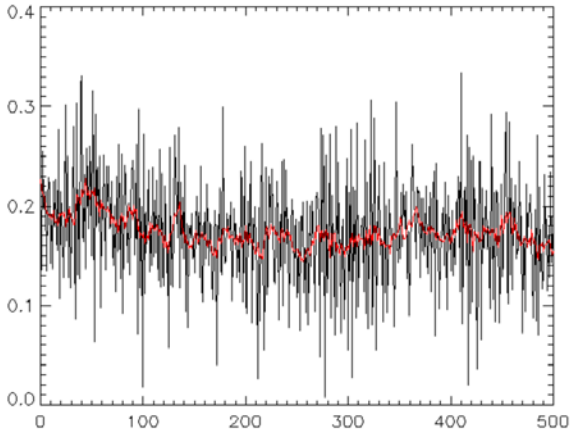
(2)



(3)



(4)



5. good データから較正信号(約1分ごとの周期的に入っている信号)などが入っていない、適当な平穏な範囲のデータを取り出して、ノイズレベルを評価せよ(信号の標準偏差を見積もれ) 。

   (※)すべて bad フラグが立っている bad channel も存在するので注意

   ヒント: stddev()

6. 全100チャンネルのノイズレベルを評価せよ。評価した100チャンネル分のノイズレベルをプロットせよ。

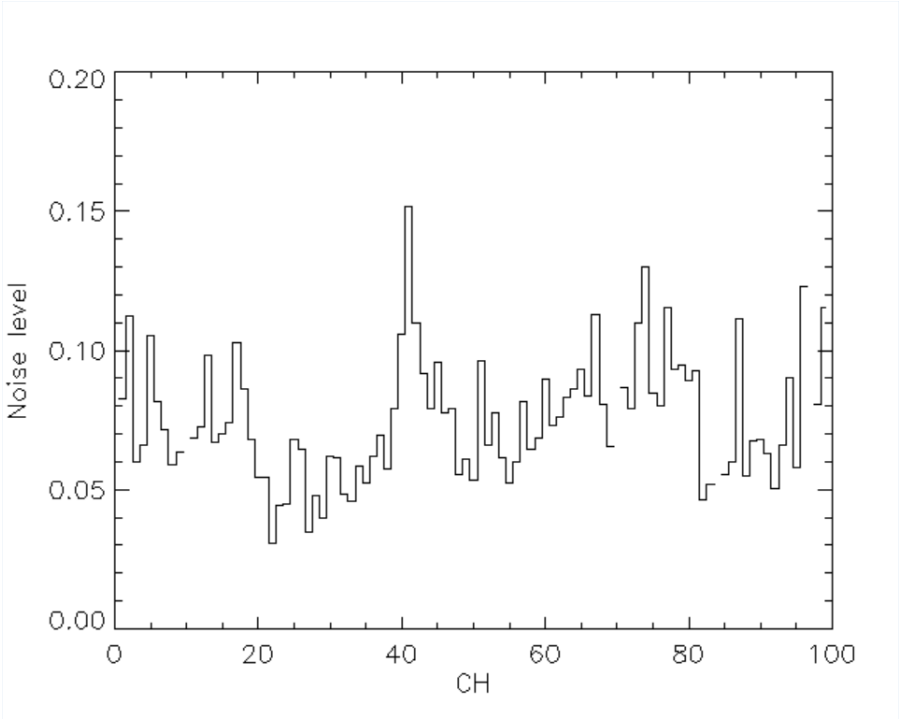
   ヒント: where(), !values.d_nan (bad data に上書きしてマスク), FOR文, stddev(/nan)

7. 適当なチャンネルの信号成分のヒストグラムを作成して、ガウシアンフィッティングせよ。

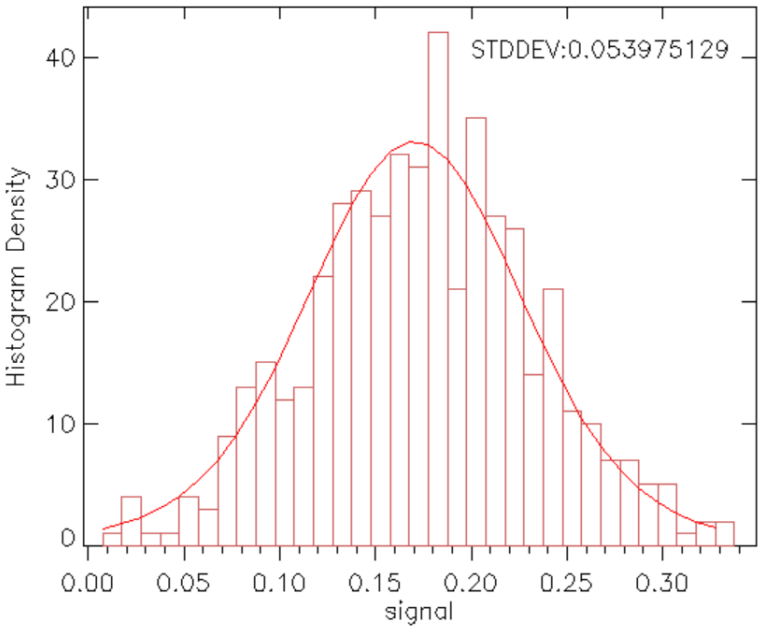
ヒント: cghistoplot, gaussfit(), cgtext (xyouts と同じ使い方)

(参考)

(6)


0.20
0.15
0.10
0.05
0.00
Noise level
0
20
40
60
80
100
CH

(7)


STDDEV:0.053975129
40
30
20
10
0
Histogram Density
0.00
0.05
0.10
0.15
0.20
0.25
0.30
signal

[演習2]
FITS イメージを読み込んで操作

サンプルデータ

M31 の IRAS 衛星による 3バンドの画像 FITS (3x3度)

- 波長25μm　M31_25um.fits
- 波長60μm　M31_60um.fits
- 波長100μm M31_100um.fits

1. 各ファイルを読み込み、読み込んだヘッダを確認せよ。また、イメージを表示せよ。

   ヒント: mrdfits(), tvscl, image()

   ✓ tvscl プロシージャは続けて使用したとき(plotなどとは異なり)前の描画を消去しない。IDL> tvscl, image, position(0,1,2,...) と position を順番に指定すると、画像をウィンドウ内部でタイル状に配置して表示できる。

2. 任意のバンドで任意の位置の、経度方向や緯度方向の放射強度プロファイルを表示せよ。

3. 各バンドの放射強度分布をヒストグラム表示せよ。

   ヒント: cghistoplot

4. 任意のバンド間で放射強度の相関をプロットせよ。

   • 余力があれば直線フィッティングしてみよ。

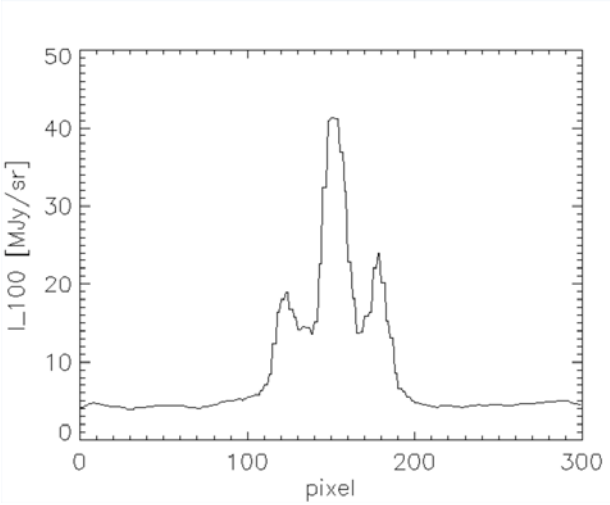
   ヒント: cgplot, linfit()

(参考)

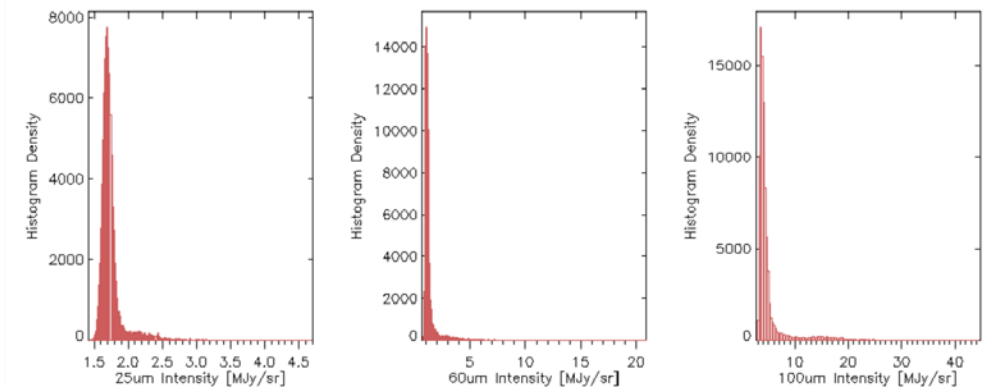
(1) tvscl 使用

25μm　　60μm　　100μm

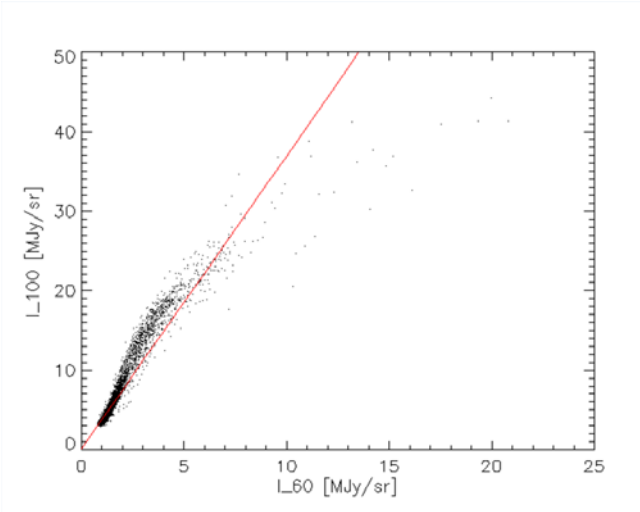
(2)



(3)



(4)



5. 任意のバンド間でイメージの差分を取って表示せよ。
   - インデックスカラーモードにして、カラー表示してみよ。

   ヒント:

   device, decomposed=0
   Rainbow カラーテーブルのロード loadct, 13
   tvscl

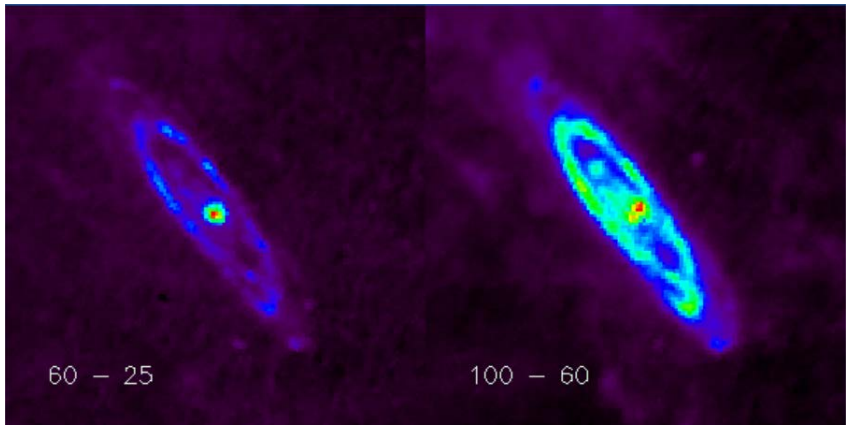
6. 3バンドのイメージを合成して、疑似カラー表示せよ。

   ヒント:

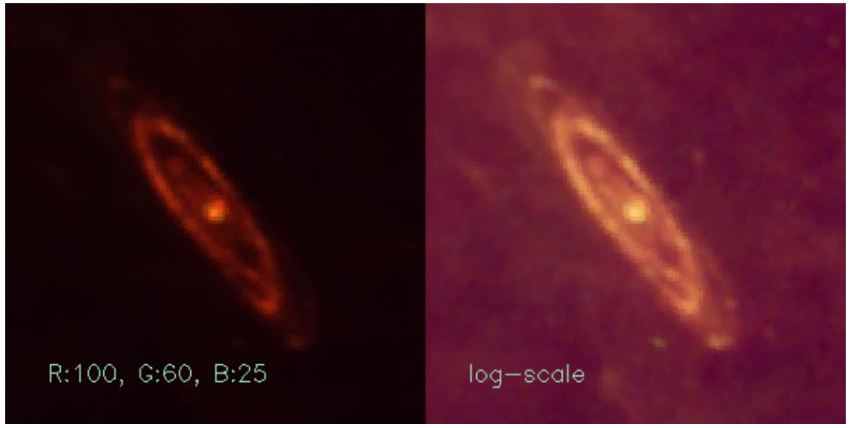
   - ✓ [m, n] の2次元アレイ a,b,c を3枚重ねて [m, n, 3] にする
     → d = [[[a]], [[b]], [[c]]]
   - ✓ image() は [m, n, 3] のアレイを自動的にカラー表示する
   - ✓ tvscl で [m, n, 3] のアレイをTrueColor 表示するには
     true=3 のオプションを付ける

(参考)

(5)



(6) tvscl 使用



(6) image() 使用

[演習3] FITS バイナリテーブルを読み込んでデータ解析

サンプルデータ

赤外線天文衛星「あかり」の遠赤外線点源天体カタログ(Bright Source Catalogue ver.1)
ファイル名 : AKARI-FIS_BSC_V1.fits.gz

データファイルの内容

全天サーベイ観測によって取得した遠赤外線の4バンド(λ = 65, 90, 140, 160 μm)の点源天体の位置(赤道座標)とフラックス密度(Jy)

1. カタログデータ(FITSバイナリテーブル)を読み込み、データ(構造体になっている)とヘッダの内容を確認せよ。
   - ✓ FITS のテーブルは拡張領域 (HDU 1) に格納されている
   - ✓ プライマリヘッダも確認する場合は、HDU 0 を別に読み込む

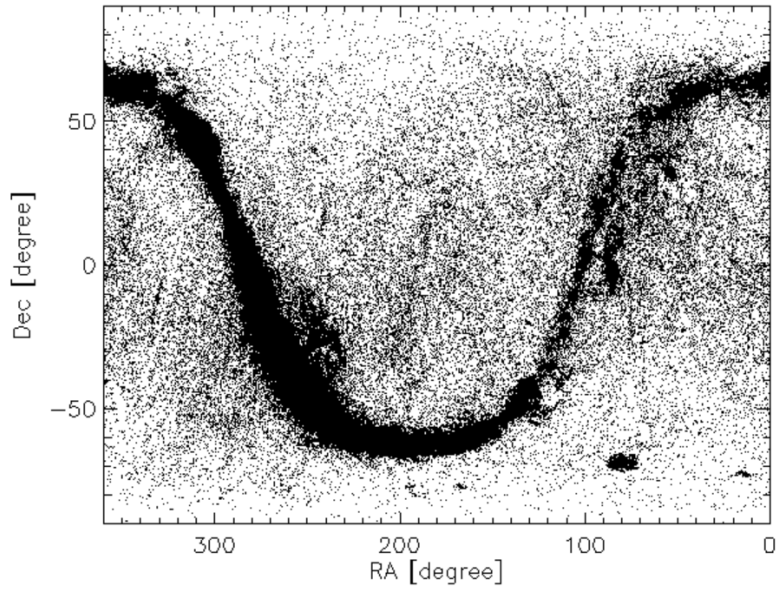
   ヒント: mrdfits(), help, /structures オプション
2. カタログに含まれる全天体の座標位置をプロットせよ。

   ヒント: 構造体名を cat とした場合、
   座標データは cat.ra & cat.dec 。単位は degree (0 - 360度)
3. 赤道座標から銀河座標に座標変換してプロットせよ。
   - 余力があれば黄道座標でもプロットしてみよ。

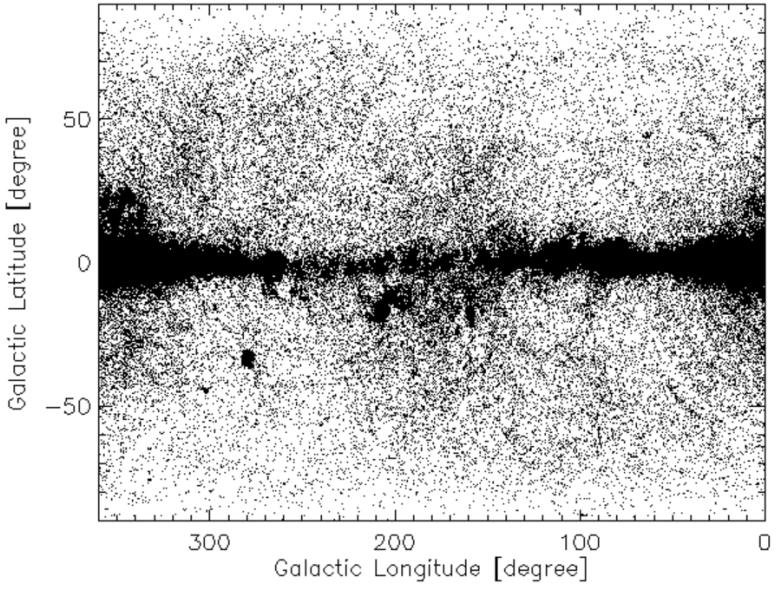
   ヒント: euler

(参考)

(2)


50
0
−50
Dec [degree]
300
200
100
0
RA [degree]

(3)


50
0
−50
Galactic Latitude [degree]
300
200
100
0
Galactic Longitude [degree]

4. 90um バンドと 140um バンドで、それぞれクオリティ指標が良い(FQual90=3, FQual140=3)データだけを取り出せ。それぞれの Flux 値から logN-logS プロット(ヒストグラム)を作成せよ。
   - ✓ カタログデータには4バンドそれぞれにクオリティ指標 (3,2,1,0) が付いている
   - ✓ フラックスデータは cat.flux90, cat.flux140 。単位は Jy 。
   - ✓ logN-logS は明るさS (flux)ごとの天体個数密度 N の分布を両対数のグラフにしたもの

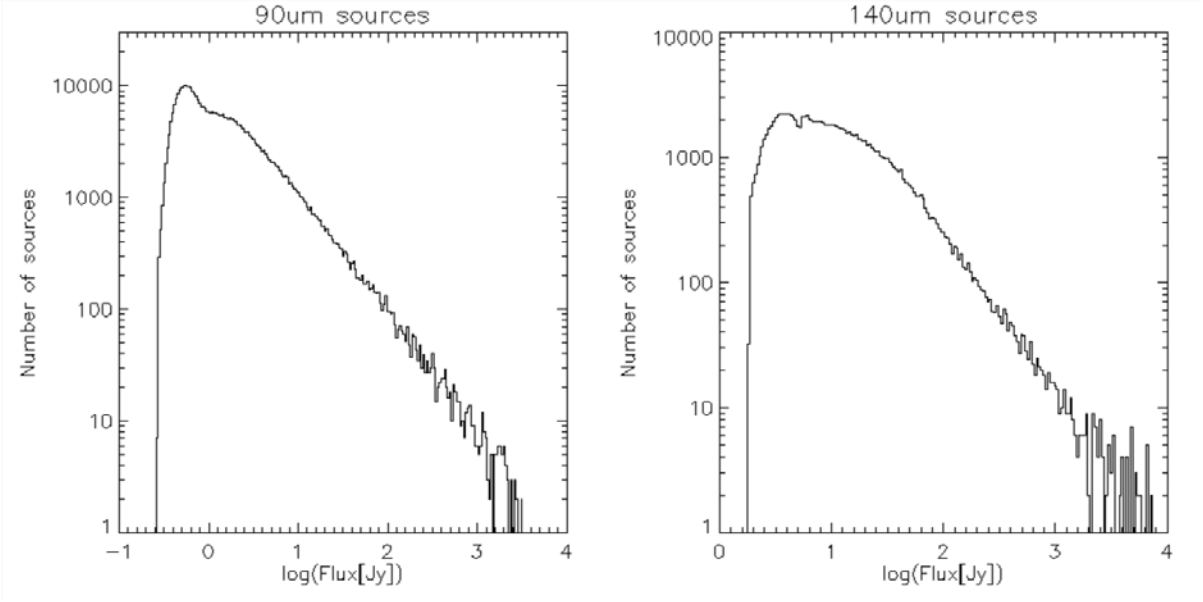
   ヒント: where(), cghistoplot, alog10()
5. 90um バンドと 140um バンドの両方でクオリティ指標が良い(FQual90=3 & FQual140=3)データだけを取り出せ。両者の Flux の相関をプロットせよ。

   ヒント: 構造体名が cat とすると、Flux は cat.flux90, cat.flux140

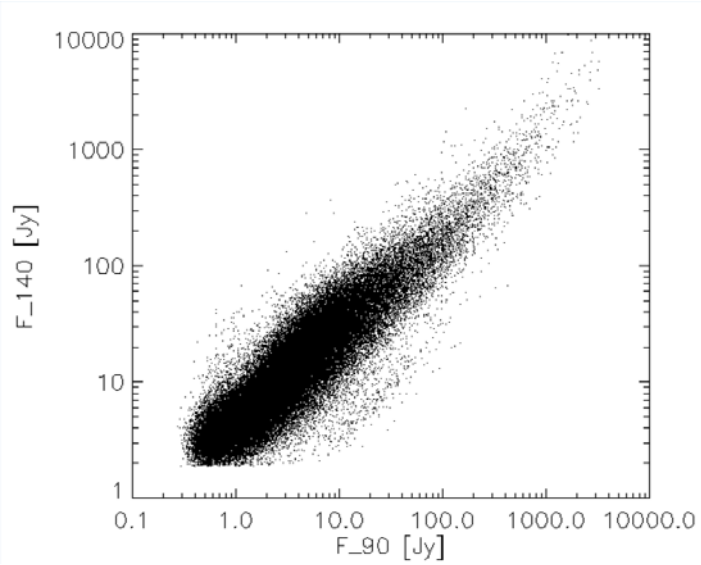
   ヒント: where()

(参考)

(4)


90um sources
140um sources
Number of sources
log(Flux[Jy])
10000
1000
100
10
1
−1
0
1
2
3
4

(5)


F_140 [Jy]
F_90 [Jy]
10000
1000
100
10
1
0.1
1.0
10.0
100.0
1000.0
10000.0

補遺

数学演算子

演算子	説明	例
+	加算 (文字列の連結にも使われる)	IDL> a = 4 + 7
++	インクリメント	IDL> a=1 IDL> a++ IDL> print, a 2
-	減算 正負反転	IDL> a = 8-3 IDL> a = -a
--	デクリメント	IDL> a = 5 IDL> a-- IDL> print, a 4
*	乗算	IDL> a = 2*5
/	除算	IDL> a = 10.0/2.4
^	指数演算	IDL> print, 2^3 8
MOD	剰余	IDL> print, 8 mod 3 2

数学関数

関数名	説明	関数名	説明
ABS	絶対値	ACOS	$\cos^{-1}(x)$
ALOG	自然対数	ATAN	$\tan^{-1}(x)$
ALOG10	常用対数	SINH	sinh(x)
SIN	sin(x)	COSH	cosh(x)
COS	cos(x)	TANH	tanh(x)
TAN	tan(x)	EXP	自然指数関数
ASIN	$\sin^{-1}(x)$	SQRT	平方根

- ✓ 三角関数の引数の角度の単位はラジアン
- ✓ degree との変換には、円周率πのシステム変数 !PI (単精度)や!DPI (倍精度)、あるいは、変換係数のシステム変数 !RADEG (180/ π @ 57.2958) や !DTOR (π /180 @ 0.01745)を使用する

統計関数

関数名	説明
MIN	最小値
MAX	最大値
MEAN	平均
VARIANCE	分散
STDDEV	標準偏差
SKEWNESS	歪度(左右の歪み)
KURTOSIS	尖度(とがり)
MOMENT	mean, variance, skewness, and kurtosis
TOTAL	総計
MEDIAN	中央値

```
Result = MOMENT( X [, SDEV=variable][, /NAN] )
```

- ✓ moment 関数の返値はmean, variance, skewness, and kurtosi の4要素配列。SDEV オプションには標準偏差を返す

よく使いそうな機能

画像表示 TV, TVSCL

TV(SCL), Image [, Position] [, TRUE={1 | 2 | 3}]
or
TV(SCL), Image [, X, Y [, TRUE={1 | 2 | 3}]]

- ✓ Direct Graphics ウィンドウに画像を表示する。TV はピクセル値をそのまま使用するのに対して、TVSCL では最大値と最小値の間を256階調にスケーリング(ストレッチング)してから表示する
- ✓ 続けて使用したときには (plotなどとは異なり) 前の描画を消去しない
- ✓ Position(0,1,2,...) を指定すると、ウィンドウ内部でタイル状に配置して表示する
- ✓ X, Y オプションで描画位置の座標を指定することも出来る
- ✓ RGB情報を持つ3次元配列を TrueColor 表示するには TRUE キーワードに、(3,m,n) 次元配列の場合は 1, (m,3,n) 次元配列の場合は 2, (m,n,3) 次元配列の場合は 3 を指定する

```
IDL> file= '/usr/local/exelis/idl85/examples/data/glowing_gas.jpg'
IDL> read_jpeg, file, image
IDL> tv, image, 50,20, true=1
```

1次元配列の補間 INTERPOL 関数

```
Result = INTERPOL( Y, X, XOUT [, /LSQUADRATIC] [, /QUADRATIC] $
                                [, /SPLINE] )
```

✓ (X,Y) データに対して X=XOUT の位置の補間データを作成する。オプションで補間アルゴリズムを選択できる

```
; サイン波
x = findgen(21)/10*3 - 3
y = sin(x)
; 補間位置
xintp = [-2.5, -0.4, 1.4, 2.5]
; 補間
r = INTERPOL(y, x, xintp)

cgplot, x, y, psym=-4
cgplot, /over, xintp, r, psym=7, color='red'
```

1～3次元配列のリサイズ CONGRID 関数

Result = CONGRID(Array, X[, Y][, Z] [, CUBIC=value{-1 to 0}] [, /INTERP])

- ✓ 配列Array のサイズを X*Y*Z (最大3次元配列まで) に拡大縮小する
- ✓ デフォルトアルゴリズムは Nearest-neighbor sampling

```
IDL> im = dist(300)                    ; 300 x 300 の 2次元アレイ
IDL> im2 = congrid(im, 450, 450)  ; 拡大
IDL> im3 = congrid(im, 150, 150)  ; 縮小

  ;; tvscl プロシージャで表示
IDL> window, xsize=900, ysize=450
IDL> tvscl, im
IDL> tvscl, im2, 300, 0
IDL> tvscl, im3, 750, 0
```

スムージング SMOOTH 関数

Result = SMOOTH(Array, Width [, /EDGE_MIRROR] [, /EDGE_TRUNCATE] $
[, /EDGE_WRAP] [, /NAN])

✓ 指定幅(width)の 移動平均(boxcar average)フィルターで平滑化を行う

```
  ;; 1次元データのスムージング
IDL> dt = randomu(seed, 100)
IDL> dt2 = smooth(dt, 10)
IDL> plot, dt, psym=-1
IDL> oplot, dt2, psym=-7, color='0000ff'xl

  ;; 2次元データのスムージング
IDL> im = sin(dist(300)/3)
IDL> im2 = smooth(im, 10)
IDL> window, xsize=600, ysize=300
IDL> tvscl, im
IDL> tvscl, im2, 300, 0
```

AstroLib のよく使いそうな機能

■ 赤道座標の表示

RADEC

- RA, Dec の角度(degree)をHours, Min, Sec, Deg, Min, Sec の数値に変換する

```
radec, ra, dec, ihr, imin, xsec, ideg, imn, xsc
```

```
IDL> radec, 125.5 , -20.5, ihr, imin, xsec, ideg, imn, xsc
IDL> print, ihr, imin, xsec, ideg, imn, xsc
      8      22      0.00000     -20      30      0.00000
```

ADSTRING()

- RA, Decの角度を60進法形式の文字列に変換する

```
result = ADSTRING( ra,dec,[ precision, /TRUNCATE ] )
```

```
IDL> help, adstring(125.5 , -20.5)
<Expression>    STRING    = ' 08 22 00.0  -20 30 00'
```

■ 分点変換

JPRECESS

- B1950 から J2000 への変換

```
jprecess, ra, dec, ra_2000, dec_200
```

■ 角度の制限

CIRRANGE

- 角度の値をを 0-360°の範囲にする

```
CIRRANGE, ang, [/RADIANS]
```

```
IDL> crd = 420.5
IDL> cirrange, crd
IDL> print, crd
      60.500000
```

■ 座標変換

EULER

- 赤道座標・銀河座標・黄道座標の相互変換

EULER, LONIN, LATIN, LONOUT, LATOUT, [SELECT]

SELECT	From	To
1	RA-Dec (2000)	Galactic
2	Galactic	RA-Dec (2000)
3	RA-Dec (2000)	Ecliptic
4	Ecliptic	RA-Dec (2000)
5	Ecliptic	Galactic
6	Galactic	Ecliptic

```
IDL> euler, 30.5, 42.3, glon, glat, 1
IDL> print, glon, glat
      136.61886      -18.688709
```

■ 離角計算

GCIRC

- 天球面座標上の2点間の角距離の計算

GCIRC, U, RA1, DC1, RA2, DC2, DISTANCE

(U) Units of inputs and output	
0	everything radians
1	RAx in decimal hours, DCx in decimal degrees, DIS in arc seconds
2	RAx and DCx in degrees, DIS in arc seconds

```
IDL> gcirc, 2, 30.123, 54.038, 30.275, 53.994, dist
IDL> print, dist
      358.41299
```

ダイレクトグラフィックスのファイル出力

画像出力 (PNG, JPEG, etc.)

- IDL には画像ファイルを読み書きするためのルーチンが用意されている
- たとえば、PNG の入出力なら READ_PNG と WRITE_PNG
- ほか、BMP, GIF, JPEG, TIFF などにも対応
- Direct Graphics の画面に表示されたプロットを画像ファイルとして保存するには、TVRD() 関数で取り込んで、それをファイルに出力する

```
  ;; 画像ウィンドウにプロットした状態で
IDL> write_png, 'output.png', tvrd(/true)
```

IDL のカラーモデル

カラーモデル

- 環境(device)と目的に応じて、Decomposed Color (分解型カラー) と Indexed Color (インデックス型カラー) の2種類のカラーモデルが設定できる

現在使用しているカラーモデルの確認
IDL> device, get_decomposed=d
IDL> print, d
1
;➔ 1: Decomposed Color, 0: Indexed Color

- 通常のフルカラーディスプレイ使用時、初期設定は Decomposed Color になっている

カラーモデルの変更方法
IDL> device, decomposed=0 ; Indexed Color に設定

Decomposed Color

- 色を R, G, B (赤,緑,青) の3色で指定する
- 各色の指定に 8 ビット (256階調) を使い、
  合計 24ビットで色指定 → 最大1677万7216色を表現できる (TrueColor)
- 16進数で指定する場合、2文字ずつ B, G, R の順番で
  00～FF の文字で指定する

```
IDL> blue = 'FF0000'XL  ; blue という名前の変数に青色の色指定値を保存
IDL> white = 'FFFFFF'XL ; 同じく、白色の色指定値を保存
  ; 線を青色で、背景を白色でプロットする
IDL> plot, indgen(10), color=blue, background=white
```

✓ XL は Hexadecimal (16進数)の Long 型であることを示す

- 10進数で表現しても構わない
  例) オレンジ色は16進数で '0080FF'xl, 10進数では33023

主な色の RGB 色成分

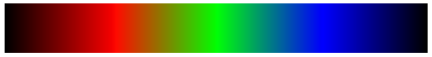
Color	Blue(B)	Green(G)	Red(R)
black	00	00	00
white	FF	FF	FF
gray	80	80	80
red	00	00	FF
green	00	FF	00
blue	FF	00	00
cyan	FF	FF	00
magenta	FF	00	FF
yellow	00	FF	FF
orange	00	80	FF
purple	80	00	80

Indexed Color

- 256色のカラーテーブルからインデックス値(0-255)で色を指定する
- IDL には初めから 74 種類のカラーテーブルが準備されている。自分で作成することも可能
- カラーテーブルを選択するには loadct コマンドを使用して、テーブル番号でセットする
- GUI なポップアップウィンドウから選択する xloadct で対話的に選択することも出来る。ガンマ値などを変更することも可能
- 'PS' (Postscript) device に PLOT コマンドによるラングラフなどを出力する場合は Indexed Color (8-bit color)を使う

カラーテーブル見本

online-help → Loading a Default Color Table

	Name	Sample			Name	Sample
0	Black-White Linear			41	CB-Accent	
1	Blue-White Linear			42	CB-Dark2	
2	Green-Red-Blue-White			43	CB-Paired	
3	Red Temperature			44	CB-Pastel1	
4	Blue-Green-Red-Yellow			45	CB-Pastel2	
5	Standard Gamma-II			46	CB-Set1	
6	Prism			47	CB-Set2	
7	Red-Purple			48	CB-Set3	
8	Green-White Linear			49	CB-Blues	
9	Green-White Exponential			50	CB-BuGn	
10	Green-Pink			51	CB-BuPu	

IDL の代表的なエラー

エラーサンプル

文法エラー(タイプミスなど)

```
IDL> print 'Test'

print 'Test'
      ^
% Syntax error.
```

存在しないプロシージャ

```
IDL> windw, 1
% Attempt to call undefined procedure: 'WINDW'.
% Execution halted at: $MAIN$
```

存在しない関数

```
IDL> print, median(a)
      2.00000
IDL> print, medan(a)
% Variable is undefined: MEDAN.
% Execution halted at: $MAIN$
```

不適切な引数

```
IDL> window, [1,2]
% WINDOW: Expression must be a scalar or 1 element array in this context: <INT       Array[2]>.
% Execution halted at: $MAIN$
```

配列の添え字が配列サイズの範囲外

```
IDL> a=indgen(5)
IDL> print, a[5]
% Attempt to subscript A with <INT      (       5)> is out of range.
% Execution halted at: $MAIN$
IDL> print, a[0:5]
% Illegal subscript range: A.
% Execution halted at: $MAIN$
```

浮動小数点のアンダーフローエラー

```
% Program caused arithmetic error: Floating underflow
```

- ✓ 計算処理を含むプログラム実行中(後)に出ることが多いエラー
- ✓ 計算結果が浮動小数点数で表現できないほど非常に小さくほぼゼロになるときに表示される
- ✓ arithmetic error は(このほかも含め)通常は無視して構わない事が多い