IDL 講習会 (初級編)

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

講師 巻内 慎一郎 (国立天文台 天文データセンター)

三、四日目

- 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)
- 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
    - IF 文と CASE 文の比較
  - 三項演算子 ?:
  - Break & Continue コマンド
- データの入出力
  - コンソール上の入出力
  - テキストファイル入出力
    - ファイルを開く
    - 読み書きを行う
    - ファイルを閉じる
  - READCOLによるテキストファイル読み込み
  - FITS ファイルの取り扱い
    - イメージ FITS
    - バイナリテーブル FITS

- 演習問題
  - ✓ 演習1
    - 検出器信号の確認と評価
      (1次元時系列データの操作)
  - ✓ 演習2
    - FITS イメージデータの読み込みと画像操作
      (2次元画像データの操作)
  - ✓ 演習3
    - FITS バイナリテーブルを読み込んでデータ解析
      (天体カタログデータの取り扱い)
- 補遺
  - 数学演算子
  - 数学関数
  - 統計関数
  - 画像表示 TV, TVSCL
  - 1次元配列の補間 INTERPOL
  - 配列のリサイズ CONGRID
  - スムージング SMOOTH
  - AstroLib
    - 赤道座標表示 RADEC, ADSTRING()
    - 分点変換 JPRECESS, 角度の制限 CIRRANGE
    - 座標変換 EULER
    - 離角計算 GCIRC
  - グラフィックスのファイル出力
  - IDL のカラーモデル
    - Decomposed color vs. Indexed color
    - カラーテーブル
  - IDL の代表的なエラー

■ 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作成すれば良い
  - 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, key2=key2
  ; 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)
```

(注) IDL 8.3 以降では、コマンドラインで実行された関数の渡す先が明示されていない場合は自動的に print に渡されるようになった(Implied Print)ため、上記の例の “print, ” は省略できる
➡
IDL> indgen(10)
で OK

❢ 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(順番)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 ─┬─ proc1
       └─ proc2 ── proc3

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。

<u>グローバル変数の例</u>

[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、通常は、引数を使用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(,)は不要
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 のプロシージャや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

- 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
- 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(短縮)が可能 (ambiguity が無い限り)

```
; 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。

自作プログラムを作成する際の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、左辺の変数に右辺を代入する (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” と書くが、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、右辺の値がプログラム内の変数に代入される
  - "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

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=keyv1
  ; check keywords
  IF keyword_set(keyv1) 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=keyv1
  ; check keywords
  IF ~keyword_set(keyv1) 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呼び出して使用する場合、呼び出されるプログラムが使うことの出来るキーワードをすべて書き下すのは困難
- 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メモリ上の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
```

← twice プロシージャ実行
引数が配列の全体の場合

← 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
```

← twice プロシージャ実行
引数が配列の要素の場合

← 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

■ 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
(すべてコメント)

プログラム本体

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)]
```

一行で書く場合

✓ 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]
```

複数行で書く場合

✓ 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)
```

一行で書く場合

✓ 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
```

複数行で書く場合

✓ 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)
```

一行で書く場合

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

複数行で書く場合

✓ 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 "?:" を使うとすっきりと書ける場合がある
- 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

■ 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 (AstroLib) 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]

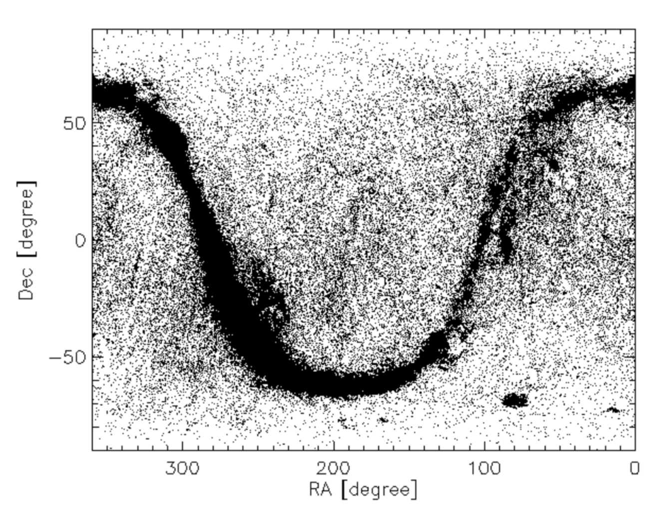
```
IDL> readcol, 'AKARI_BSC_sub.txt', ra, dec, $
IDL>      f90, f140, ferr90, ferr140,  fqual90, fqual140, $
IDL>      format='(F,F,F,F,F,F,I,I)', skip=1
```

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 FITS 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]
```

読み込んだデータを確認のため表示する例

```
; 読み込んだ画像データを表示して確認
; tvscl (「補遺」p.93 参照) で表示
tvscl, img

; congrid() 関数 (「補遺」 p.95参照) を使用して拡大して表示
tvscl, congrid(img, 600, 600)

; Coyote ライブラリの cgimage を使用して表示
; # ウィンドウサイズに合わせて表示のサイズが調整される
cgimage, bytscl(img) ; cgimage は AstroLib には含まれていない

; コントアで表示
contour, sqrt(img)
```

→ 演習問題２へ

FITS イメージデータの読み込みと画像操作

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

FITS バイナリテーブルを読み込んで
データ解析

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

AKARI 90μm Point Source All-sky Map

(演習用カタログデータ; 427071天体)

演習問題

[演習1] 検出器信号の確認と評価

～ 1次元時系列データの操作

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

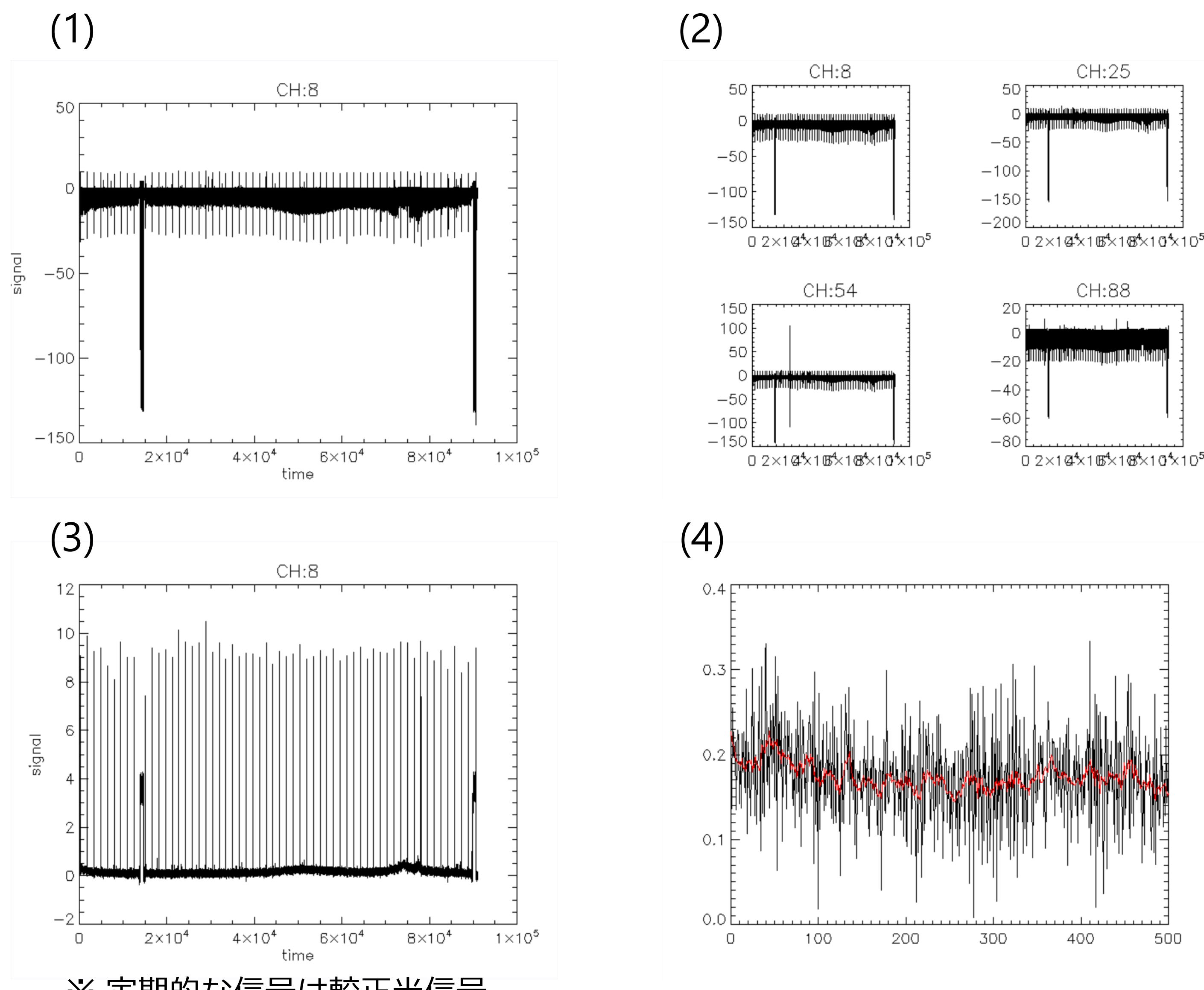
   100チャンネル(ピクセル)を持つFIS検出器がサーベイ(掃天)観測で取得した時系列信号データ。サンプリングレートは約25Hz。

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(flux データに対応)を示す bad フラグ(1/0)。1 が bad

0. データ(IDL save フィル形式)をメモリ上に展開せよ
   ヒント: restore
1. 適当なチャンネル(0-99のどれか)を選んで時系列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)の信号を並べてプロット(マルチプロット表示)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 データのみを取り出しプロットせよ。
   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(平滑化)処理をかけよ。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)



※ 定期的な信号は較正光信号

5. good データから較正光信号(約1分ごとの周期的に入っている)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()

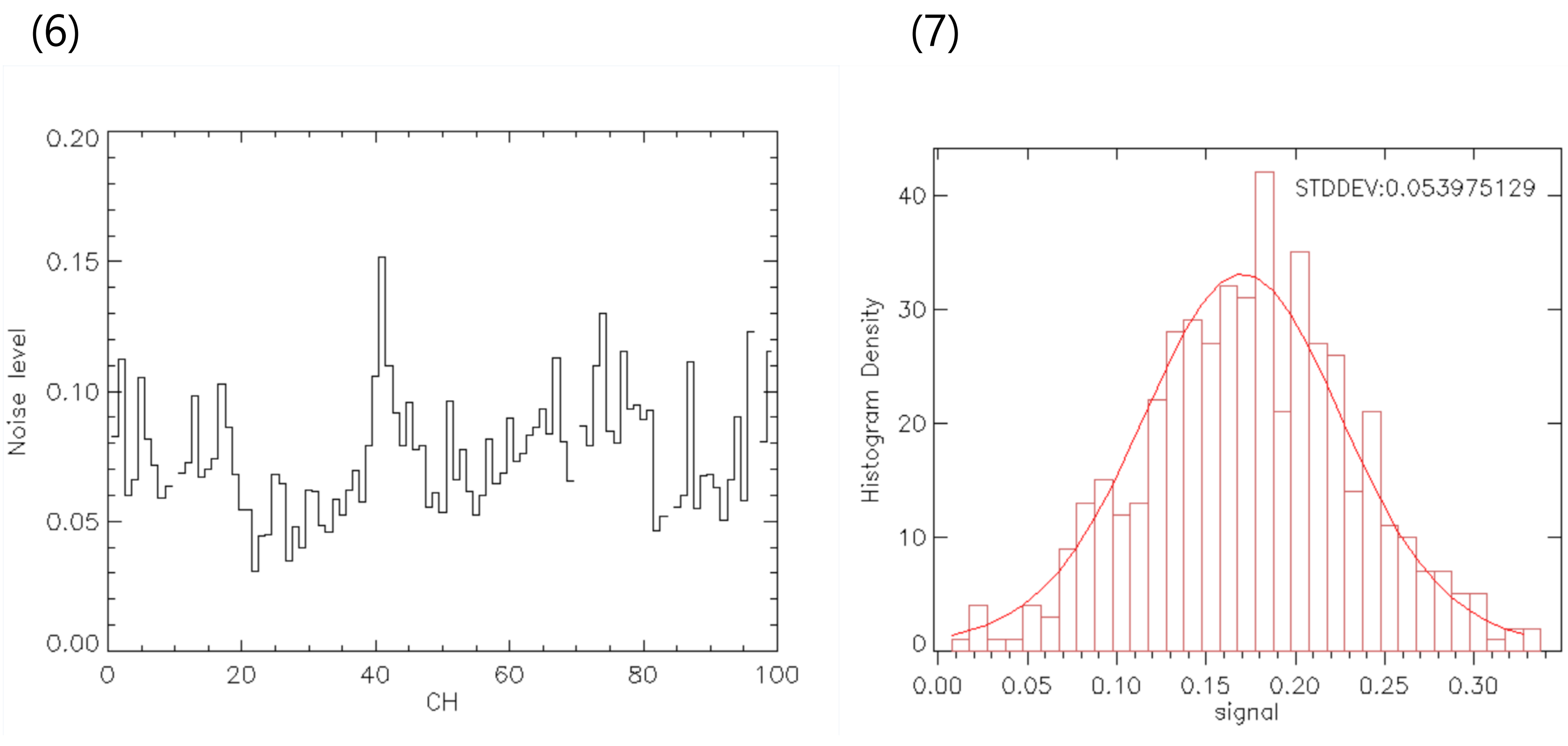
6. 全100チャンネルのノイズレベルを評価せよ。評価した100チャンネル分のノイズレベルをグラフに示せ(プロットせよ)。

   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適当な範囲の信号値を取り出して、値の分布のヒストグラムを作成せよ。ガウシアンフィッティングして信号値の分布の幅(標準偏差)を見積もってみよ。

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

(参考)



[演習2]
FITS イメージデータの読み込みと画像操作

～ 2次元画像データの操作

サンプルデータ

M31 の IRAS 衛星による 3バンドの画像 (3x3度) の FITS データ

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

1. 各バンドの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 あるいは image()

   ✓ tvscl プロシージャは連続して使用したとき、(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 番号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。

   ヒント: 2次元アレイの取り扱い

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

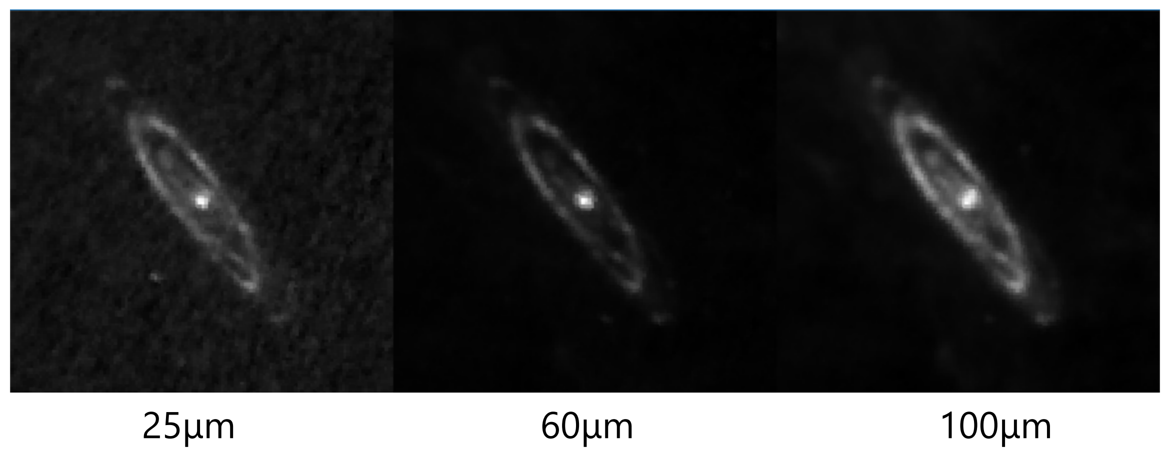
4. 任意のバンド間で放射強度の相関をプロットせよ。

   - 余力があれば直線フィッティングしてみよ。

   ヒント: cgplot, linfit()

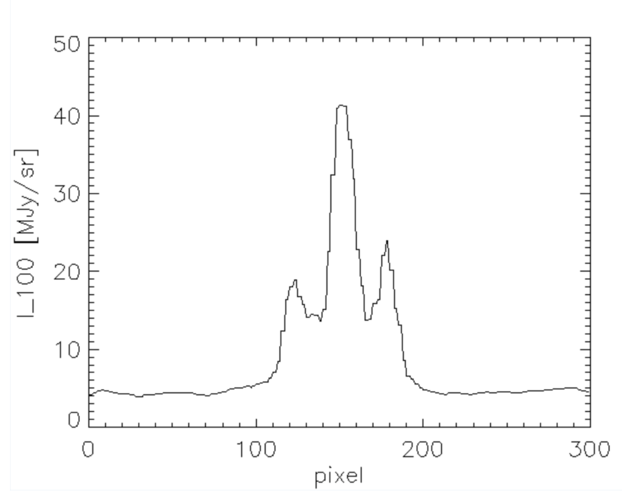
(参考)

(1) tvscl 使用

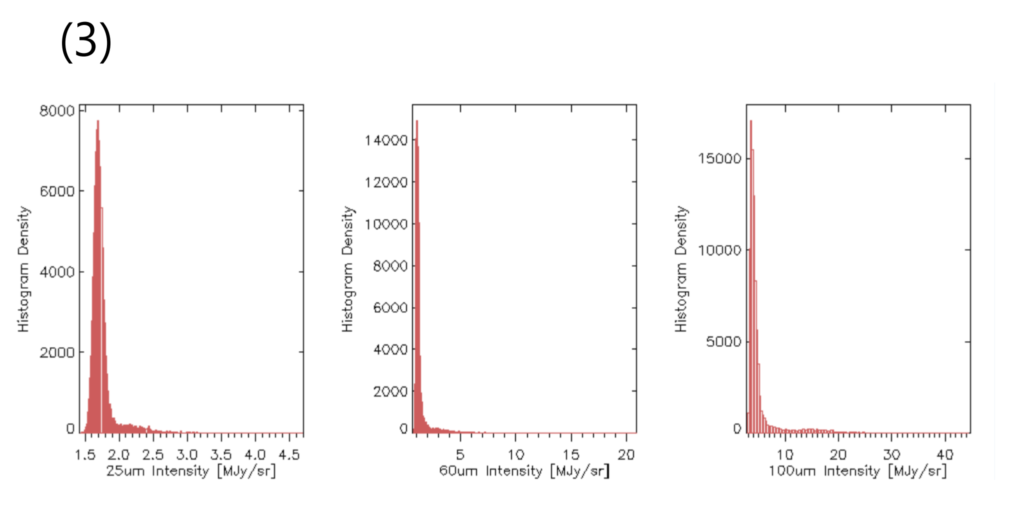


25μm　　60μm　　100μm

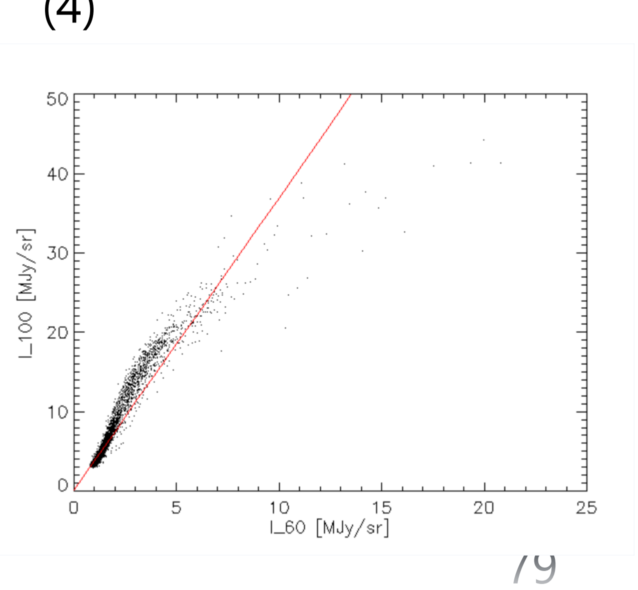
(2) 100um データ, 中心付近縦方向プロファイル



(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:

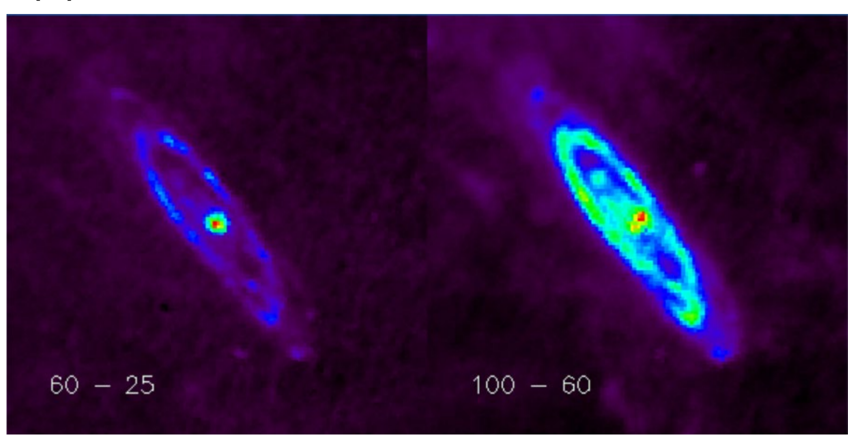
   ✓ [m, n] の2次元アレイ a,b,c を3枚重ねて [m, n, 3] にする
   → d = [[[a]], [[b]], [[c]]]

   ✓ image() は [m, n, 3] のアレイを自動的に3色合成してカラー表示する

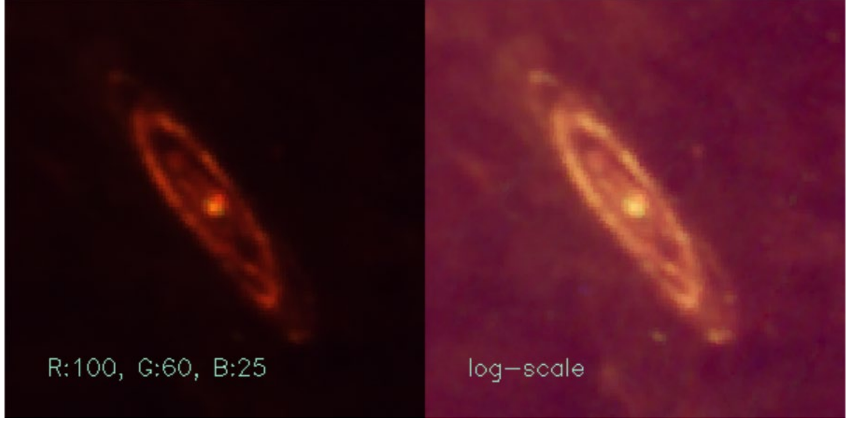
   ✓ tvscl で [m, n, 3] のアレイを TrueColor 表示するには true=3 のオプションを付ける

(参考)

(5)

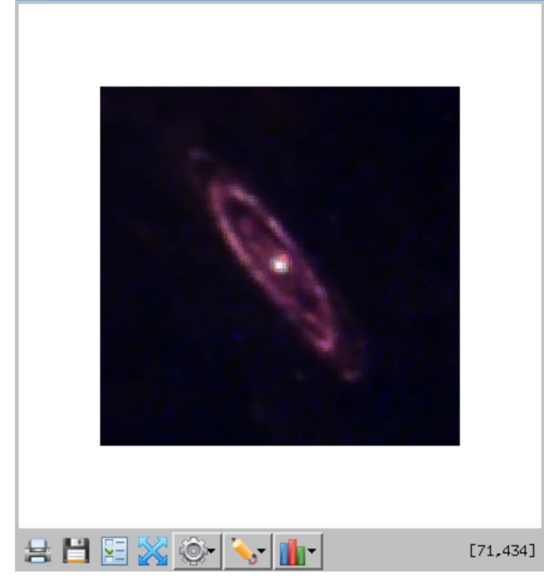


(6) tvscl 使用



※ 右図は対数スケールで表示

(6) image() 使用



■ 補足

(6′) 画像合成および表示の際のスケール最適化の効果

左: 単純な合成
中: bytscl でスケールして合成
右: 対数を取ってから bytscl でスケールして合成
表示はすべて平方根スケール

おまけ: もし時間があれば
WISE の M81 データ(3.4μm, 12μm, 22μm)でも同様の作業で3色合成画像作成 ➡

データ: /lfs12/idl2021/data/M81_FITS/

[演習3] FITS バイナリテーブルを読み込んでデータ解析

～ 天体カタログデータの取り扱い

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。
   - ✓ データ本体(FITS のテーブル形式)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、プライマリ領域 HDU 0 を別に読み込む

   ヒント: mrdfits(), help, /structures オプション

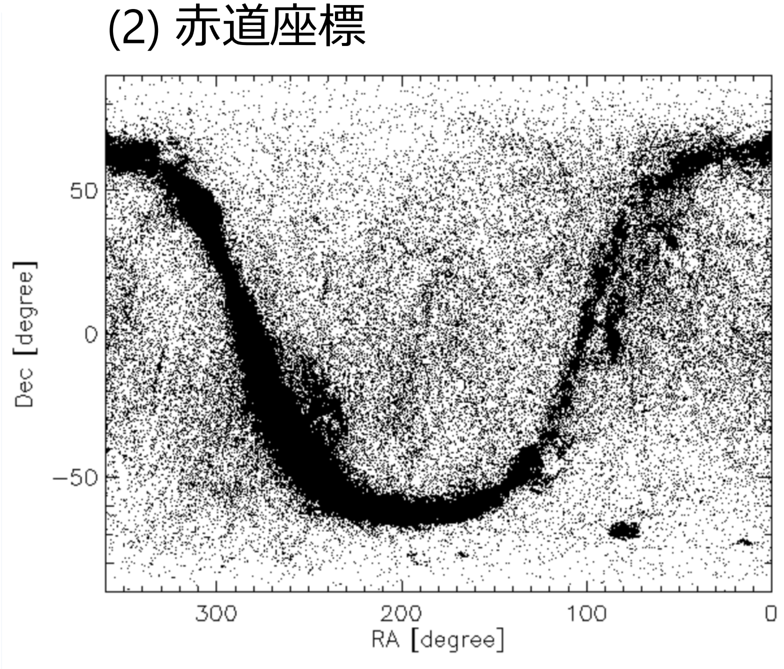
2. カタログに含まれる全天体の座標位置をプロットせよ。(全天マップの表示)

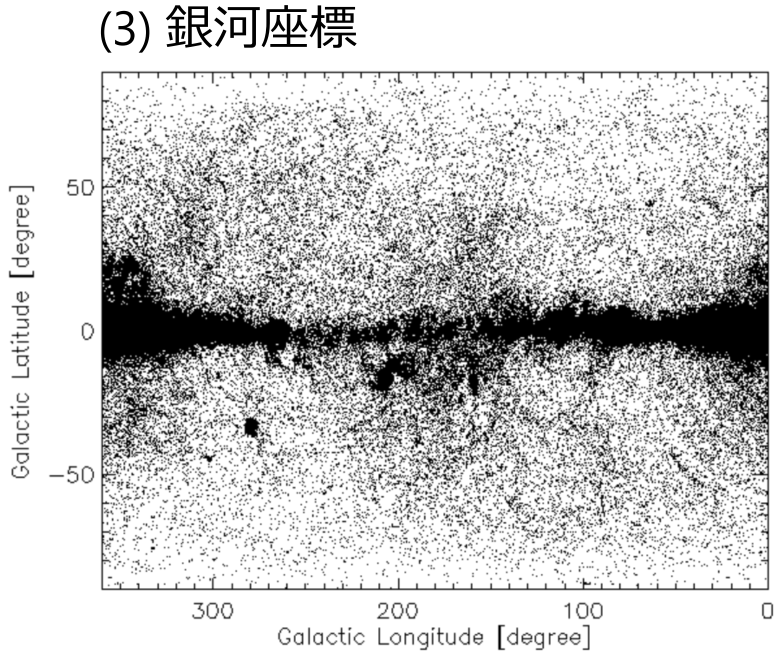
   ヒント: 構造体名を cat とした場合、
   座標データは cat.ra & cat.dec 。単位は degree (0 - 360度)

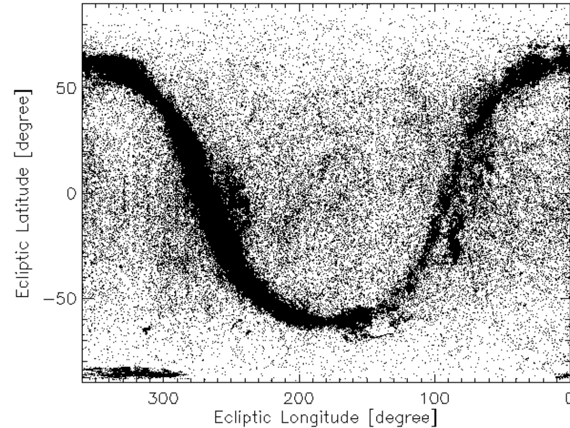
3. 赤道座標から銀河座標に座標変換してプロットせよ。
   - 余力があれば黄道座標でもプロットしてみよ。

   ヒント: euler

(参考)







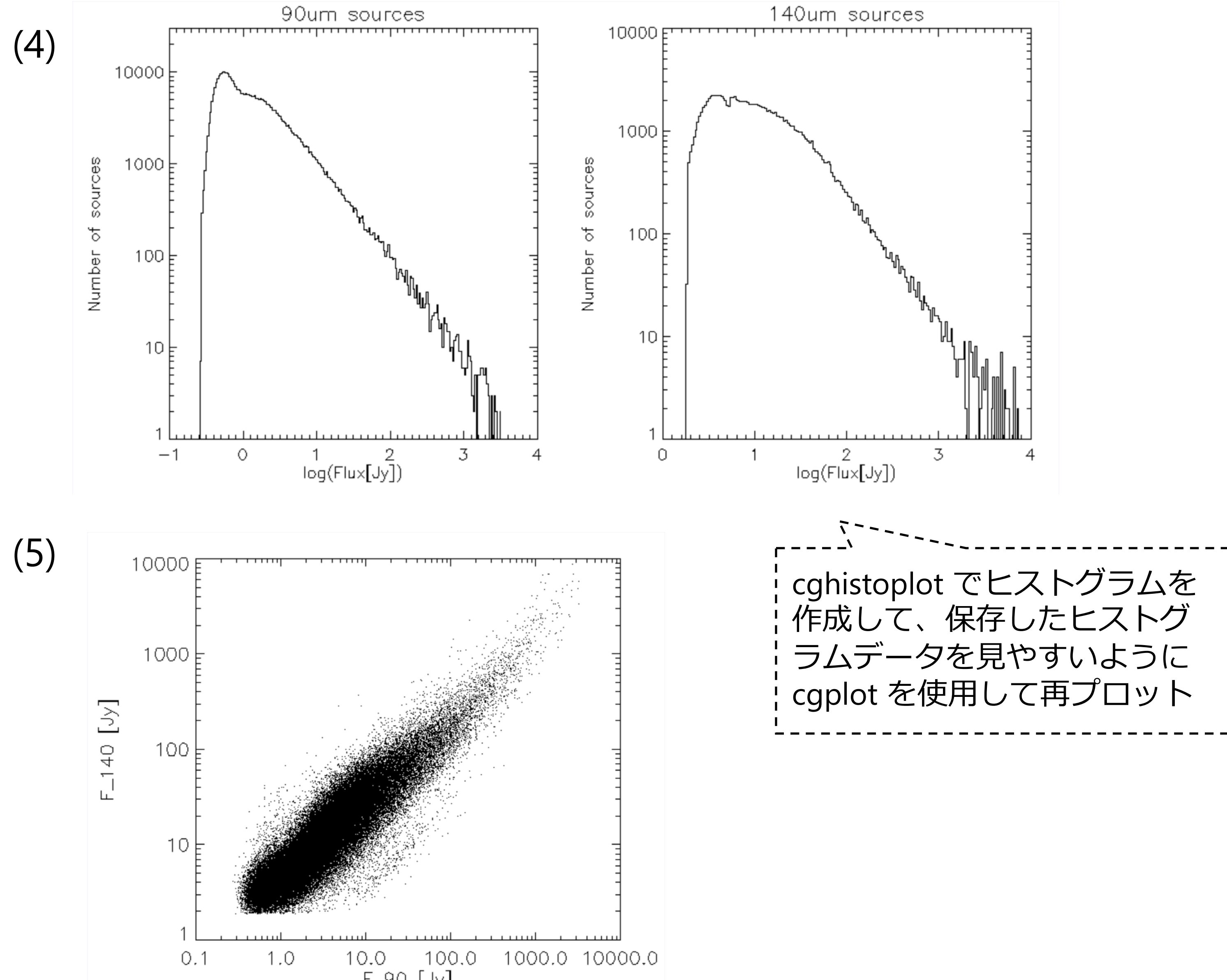
4. 90μm バンドと 140μm バンドで、それぞれクオリティ指標が良い(FQual90=3, FQual140=3)天体データだけを取り出せ。それぞれの Flux 値から logN-logS プロット(ヒストグラム)を作成せよ。
   - ✓ カタログデータには4バンドそれぞれにクオリティ指標 (3,2,1,0) が付いている。構造体名が cat とすると、cat.fqual90, cat.fqual140 。
   - ✓ フラックスデータは cat.flux90, cat.flux140 。単位は Jy 。
   - ✓ logN-logS は、明るさS (flux)ごとの天体個数密度 N の分布を両対数のグラフにしたもの

   ヒント: where(), cghistoplot, alog10()

5. 90μm バンドと 140μm バンドの<u>両方で</u>クオリティ指標が良い(FQual90=3 AND FQual140=3)天体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: where(), AND

(参考)



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/harris/idl88/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

```
; サイン波データ作成
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)
; グラフ上で確認
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して確認
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_2000
```

■ 角度の制限

CIRRANGE

- 角度の値を 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

```
;; ウィンドウにグラフィックスを表示した状態で
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 コマンドによるライ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