Интерактивная работа с данными на языке IDL

В.В. Гречнев

Email: grechnev@iszf.irk.ru

В последние десятилетия исследователи и разработчики в различных областях науки и техники сталкиваются с лавинообразным ростом объёмов информации, требующей обработки и анализа. Быстрое совершенствование компьютерной техники обеспечивает материальную базу для решения этих проблем. Но при использовании в этих целях универсальных языков программирования – C, Pascal – и традиционного FORTRAN – возникают и традиционные трудности. Создание совершенных программ остаётся прерогативой высококвалифицированных программистов, далёких от специфики решаемых с помощью этих программ задач. А разработкой техники и методов, анализом данных и их интерпретацией занимаются специалисты в конкретных предметных областях, далёкие от высот искусства программирования. Между исследователем, с одной стороны, и компьютером, с другой, традиционно необходим «интерфейс» в виде программиста. По мере возрастания сложности компьютерной техники и решаемых исследователями задач разрыв между исследователями и программистами лишь возрастает. На этом пути затруднён активный поиск исследователем или разработчиком наиболее адекватных и возможных альтернативных методик, как это в простейших случаях можно делать в интерактивном режиме, например, с помощью «электронных таблиц». Наконец, использование универсальных языков программирования для создания очередной системы обработки и анализа данных всякий раз ведёт к воспроизводству уже многократно пройденного пути, очередному «изобретению велосипеда», а накопленный многими предшественниками опыт остаётся большей частью невостребованным.

Неоптимальность такого подхода давно заботила обе этих группы специалистов. Четверть века назад широко обсуждалось создание «квазиестественного» языка, который мог бы стать средством непосредственного общения исследователя с компьютером. С изобретением графического интерфейса острота проблемы спала, и миллионы не имеющих специального образования пользователей получили простой и наглядный доступ к компьютерным ресурсам.

Однако и программы с графическим интерфейсом имеют свои ограничения. Прежде всего, они способны делать только то, что они способны делать. «Шаг в сторону» от заложенного алгоритма проблематичен. Более того, теперь возможность активного участия исследователя в разработке и модернизации такой программы, становящейся всё более сложной, становится немыслимой. А потребности экспериментальных исследований неуклонно возрастают. И если сравнительно недавно исследователя удовлетворяли графики, построенные по сравнительно небольшому числу экспериментальных точек и проведённые на них аналитические аппроксимации, то теперь качественному и количественному анализу приходится подвергать наборы данных, состоящие из сотен и тысяч изображений, выборки из огромных баз данных и другую информацию не менее сложного характера.

В конце 70-х годов прошлого века Дэвидом Стерном были разработаны основы прототипа современного *интерактивного языка для работы с данными* – IDL[1] (Interactive Data Language). Язык оказался столь удачным, что был с воодушевлением принят в ряде областей исследований: медицине, космических исследованиях, географии и других. В частности, с 90-х годов его используют как свой стандарт те, кто занимается экспериментальными исследованиями физики Солнца. Программное обеспечение обработки и анали-

[1] IDL – торговая марка фирмы Research Systems, Inc. Адрес в Internet: http://www.rsinc.com

за данных подавляющего большинства современных солнечных обсерваторий – как наземных, так и космических – реализовано на IDL. Причин этому, по-видимому, немало, а среди основных можно выделить следующие:

- реализация IDL как интерпретатора, что позволяет, не прерываясь на компиляцию и компоновку программ, практически без ограничений испытывать различные варианты методик просмотра, обработки, анализа данных;
- встроенная работа с массивами, позволяющая без явных циклов оперировать с векторами и матрицами, изображениями и их наборами;
- разнообразие типов переменных, включая символьные переменные (строки), структуры и другие;
- богатый математический аппарат и поддержка обмена с файлами многих популярных форматов;
- встроенная компьютерная графика, предоставляющая богатейшие возможности визуализации данных, их интерактивного анализа и подготовки иллюстраций, в частности, в файлах PostScript, обеспечивая высшее качество при воспроизведении векторной графики с полноцветными изображениями;
- многоплатформенность, позволяющая использовать программы IDL на разных операционных системах;
- при внешнем сходстве с FORTRAN – более высокий уровень языка, благодаря которому в немногие слова можно вложить большое содержание;
- дружелюбие и терпимость IDL, приближающие его логику к логике человеческого мышления.

При всём этом программы, написанные на IDL, по производительности обычно не уступают оптимизированным программам на FORTRAN или C, если, конечно, приняты меры для избежания явных циклов с большим числом повторений.

13-летний опыт использования автором IDL в его исследованиях свидетельствует о высочайшей продуктивности этого языка, позволившего решать задачи обработки и анализа данных в намного более сжатые сроки, чем это достигалось использовавшимися ранее средствами, и успешно справляться с такими задачами, о которых прежде трудно было и помышлять. Но IDL пока мало известен в России. Этой брошюрой мы хотели бы привлечь внимание к IDL отечественных исследователей и разработчиков. Автор работал со многими версиями IDL от 2.4 до 6.1 на операционных системах MS Windows, нескольких версиях UNIX и LINUX.

В этом кратком пособии (особенно его первой части) мы хотим помочь читателю в освоении основ языка IDL и надеемся, что впоследствии читатель обратится непосредственно к его описанию в оригинале. Вторая часть пособия посвящена интерактивному анализу данных.

Начальные сведения об IDL

1.1 Общая характеристика IDL

Название языка IDL (Interactive Data Language) означает *интерактивный язык для работы с данными*. Однако название содержит в себе каламбур: *Idle – ленивый, Ideal – идеал*. Возможно, авторы названия намекают на то, что IDL – идеальный язык обработки данных для «ленивых»…

1.1.1 Историческая справка[2]

В 1977 году создатель IDL Дэвид Стерн работал в Колорадском Университете с группой исследователей, анализировавших данные, полученные на космических аппаратах Маринер Марс 7 и 9. В процессе этой работы стало ясно, что необходим компьютерный язык, который превосходил бы функциональные возможности языка FORTRAN и позволял бы легче и быстрее разрабатывать прикладные программы, анализировать и визуализировать данные. В качестве решения проблемы Стерн написал прототип IDL – «Mariner Mars Spectral Editor» – программный язык, который позволил исследователям успешно проверять гипотезы, не прибегая к помощи программиста всякий раз, когда им требовалось писать или модифицировать программу. На его основе возникли фирма Research Systems, Inc. и диалоговый язык для работы с данными IDL (Interactive Data Language). Вдохновлённый положительными результатами, Стерн ушёл из университета и сосредоточился на создании IDL как ключевого средства для решения широкого спектра научно-исследовательских проблем.

В начале 1980-х годов Стерн понял, что для того, чтобы успеть за лавинообразным ростом количества данных, поставляемых современными инструментами, медикам, инженерам и специалистам в области наук о Земле необходимы методы анализа и визуального представления данных. IDL был переписан заново с учётом специфики работы с большими объёмами данных. В результате IDL объединил в себе всё, что позволило создать первую практическую томографическую систему, строящую изображения, тем самым открыв новые возможности кардиологии для диагностики сердечных заболеваний. Возможности анализа данных с помощью IDL играют с тех пор определяющую роль фактически в каждой космической программе NASA и Европейского Космического Агентства (ESA).

В 1987 IDL был вновь усовершенствован для использования возможностей становившихся популярными мощных рабочих станций UNIX. Это сформировало основу современного IDL, который предоставляет возможности разработки переместимых программ, работоспособных на операционных системах UNIX, Windows, Windows NT, Macintosh, Power Macintosh и системах OpenVMS.

В результате неуклонного роста фирмы Research Systems её штат превысил 110 сотрудников. Дэвид Стерн до выхода на пенсию в 2002 г. сохранял значимую роль как в производстве и оттачивании программного продукта, так и в руководстве фирмой.

1.1.2 Почему IDL?

За наблюдаемые явления ответственны разнообразные процессы, зависящие от многих параметров. Поэтому изучение природы исследуемых объектов требует привлечения разнородной информации, полученной различными методами из разных источников. Это относится ко многим сложным объектам современных исследований, будь то живой

[2] Фрагмент перевода «History of Research Systems», приведённой в сопроводительной документации к IDL.

организм, экосистема, технический комплекс, планета солнечной системы или звезда. Например, для корректной и однозначной интерпретации данных радиоастрономических наблюдений Солнца необходимы и данные, полученные в различных других спектральных диапазонах: знание магнитных полей и их конфигураций на уровне фотосферы, информация о которых дается магнитограммами, и распределение температуры и концентрации плазмы, которые можно получить из данных наблюдений в мягком рентгеновском излучении. Таким образом, современные экспериментальные исследования предполагают привлечение разнотипных данных. Благодаря всемирной сети Internet, теперь практически доступны данные фактически всех наземных и космических обсерваторий. С другой стороны, привлечение разнородного наблюдательного материала требует применения разнообразных методов их обработки и анализа, сопоставления и наложения изображений разных форматов и т.д.

Работа с данными обычно проходит в несколько этапов. На первой стадии выполняются рутинные процедуры (отбраковка записей, построение и калибровка карт, приведение их к стандартной форме и т.д.). Последующие шаги включают предварительный просмотр, отбор важнейших записей, сравнение их с данными других инструментов, измерение параметров исследуемых объектов и оценки физических и иных условий, модельные вычисления.

Как правило, первый уровень обработки данных достаточно хорошо формализован и выполняется специализированными для данного инструмента программными средствами. Последующая работа с данными часто ведется в условиях, когда неясны не только методы работы с ними, но даже и самый подход к решению физической задачи. Эти условия предполагают интерактивную работу, в ходе которой удобно опробовать различные методики работы с данными.

Интерактивный язык для работы с данными IDL является эффективным инструментом для решения таких задач. Особенностями IDL являются:

- Высокий уровень языка.
- Модульное программирование, позволяющее легко модифицировать и развивать существующие программы.
- Гибкое использование компьютерной графики для представления данных в процессе работы с ними.
- Наличие мощного математического аппарата, позволяющего выполнять сложную обработку данных.
- Наличие в стандартной поставке IDL множества процедур для представления данных в различных формах на экране монитора и вывода на твёрдую копию, что существенно облегчает и ускоряет подготовку отчетов и статей.
- Интерактивный характер IDL ярко проявляется при разработке программ. Отсутствие процесса компоновки и работа в режиме интерпретатора существенно ускоряет процесс их отладки. По сравнению с другими языками высокого уровня, IDL позволяет более быстро и эффективно опробовать значительно большее количество вариантов различных методик и алгоритмов обработки и анализа данных.
- Совместимость программ и форматов данных IDL на различных операционных системах: MS Windows, Macintosh, UNIX, OpenVMS.

Принимая во внимание такие преимущества IDL, как эффективность, прозрачность, мощные графические средства и другие, астрономы-солнечники избрали IDL своим базовым языком программирования. Многие обсерватории и институты мира пользуются

IDL в течение ряда лет. Обычно мы иллюстрируем предпочтительность IDL универсальным языкам программирования – скажем, C или Pascal – таким примером. Пусть требуется подготовить научную статью с формулами, схемами и фотографиями. Будучи хорошо знакомым с таким универсальным языком программирования, можно, в принципе, написать текстовый редактор с требуемыми для этого возможностями. Однако вряд ли кто-либо станет этим заниматься, скорее же воспользуется какой-либо из существующих систем – LaTeX, Microsoft Word или Word Perfect. Стоит ли самим разрабатывать и систему для работы с данными? Не лучше ли воспользоваться IDL, который уже однажды был написан на том же языке C.

1.1.3 О возможностях IDL

Фирма Research Systems, Inc. справедливо утверждает, что IDL – это идеальная программная сред для анализа и визуализации данных и разработки кросс-платформенных приложений. IDL сочетает все средства, требуемые для различных типов проектов, от интерактивного экспресс-анализа и просмотра данных до крупномасштабных проектов.

С помощью IDL можно легко и быстро создавать устойчиво работающие прикладные программы, затрачивая на это намного меньше времени, чем это требуется при использовании традиционных языков программирования. Несколько строк на IDL могут выполнить функции сотен строк, написанных на языке C или FORTRAN, без потери гибкости и производительности.

Наш опыт использования IDL показал, что этот язык достаточно легок в освоении и позволяет исследователям быстро создавать свои уникальные программы, в то же время обладающие общими свойствами и совместимые по интерфейсам и форматам данных. Это позволило нам резко интенсифицировать обработку и анализ данных.

IDL организован как интерпретатор, подобно языку BASIC, то есть исполнимый модуль программы не создается, а трансляция команд осуществляется по мере их ввода. В результате этого возможна работа в диалоговом (интерактивном) режиме, подобно системам DaDisp, MathLab, Excel, Statistica, Mathematika; возможно, однако, и составление программ, как на языках BASIC, FORTRAN, PASCAL и т. д. Трансляция осуществляется очень быстро, в отличие от процесса компиляции и компоновки исполнимых модулей программ на традиционных языках-компиляторах, резко облегчается отладка программ.

Однако за все приходится платить. IDL расплачивается за предоставляемые удобства уязвимостью своего быстродействия при циклических процедурах с большим числом повторений. Разработчики IDL, имея в виду эту ахиллесову пяту языка, предусмотрели ряд средств, позволяющих, по крайней мере, в большинстве, если не во всех случаях, обойти эту проблему. Это, прежде всего, следующие:

- все встроенные процедуры, для которых это возможно, обрабатывают как скалярные аргументы, так и массивы; например, если **A** – матрица, имеющая $\mathbf{M} \times \mathbf{N} \times \mathbf{L} > 1$ измерений, то **sin(A)** является матрицей той же размерности, каждый элемент которой равен синусу соответствующего элемента исходной матрицы **A**. Эти процедуры реализованы в ядре IDL и работают быстро.

- разработаны простые и эффективные средства для манипуляции массивами, средства выбора подмассивов и т. п.

Хотя процедуры, написанные в стиле программирования на FORTRAN, C или Pascal, то есть с использованием явных циклов и без применения специальных приемов IDL, также будут работоспособны, однако целесообразно придерживаться стиля, специфичного для IDL. Эта привычка окажется полезной в дальнейшем.

Например, если переменная **y** должна содержать значения J-функции Бесселя 3-го порядка при 100 значениях аргумента от 0 до 6.28, можно получить результат стандартным способом:

```
y = fltarr(100)                          ; создание вещественного 100-
                                         ; элементного массива
for j = 0, 99 do y(j) = beselj (j/99.*6.28, 3)   ; заполнение массива значениями
                                         ; функции Бесселя
```

Обратите внимание на то, что элементы массива нумеруются с нуля.

Однако средствами IDL предпочтительнее сделать это так:

```
z = beselj ( findgen(100)/99*6.28, 3 )
```

Здесь **findgen(100)** – “индексный генератор” 100 чисел вещественного типа (**F**$_{\text{LOATING-POINT}}$ **IND**$_{\text{EX}}$ **GEN**$_{\text{ERATOR}}$), то есть 100-элементный массив чисел 0.0., 1.0, 2.0, ... 99.0

Теперь можно изобразить график полученных значений:

```
plot,  y
```

Убедимся, что результаты, полученные обоими способами, совпадают:

```
print,  y-z
```

– разности всех значений равны нулю, то есть совпадают все значения.

2 Основы языка IDL

2.1 Первые упражнения

Как уже мог заметить читатель, в качестве разделителя в IDL используется запятая. Многие символы и операторы языка достаточно очевидны, а с прочими символами мы познакомимся в следующем параграфе. Теперь же попытаемся начать работать в командной строке IDL.

Зададим переменную **a** как пустую символьную (строку), поставив один за другим два апострофа:

```
a = ''
```

В IDL не требуется описаний переменных, но, конечно, тип и размерность переменной при операциях ввода должен быть известен. Введём значение **a** с клавиатуры:

```
read, a
```

Теперь IDL будет ждать ввода значения переменной **a** с клавиатуры, о чём сигнализирует знак двоеточия слева от начала командной строки. Введём, например, 30 и нажмём *Enter*. IDL получил значение переменной **a**, о чём сигнализирует восстановление надписи «IDL» слева от командной строки. Заметим, что наличие и количество пробелов в программном тексте «**read, a**» не важно, не имеет значения и регистр (можно написать и **READ,A**).

Выясним, как нас понял IDL – попросим напечатать значение переменной **a**:

print, a

– мы увидим, что наш ввод принят. Теперь выясним, каков тип этой переменной:

help, a

– он соответствует тому, что мы задали – это символьная переменная. В данном случае команды **help** и **print** дают почти одинаковую информацию, но между ними есть существенная разница. Если переменная представляет собой большой массив, то нас могут интересовать, прежде всего, её тип и размерность. В этом случае использование команды **print** приведёт к тому, что IDL будет надолго занят поэлементным выводом всего массива, а мы – тем, что будем на этот вывод смотреть до его окончания. Если же переменная представляет собой скаляр, то команда **help** выдаёт и его значение.

Теперь преобразуем тип переменной **a**:

a = float(a)

и опять выясним, с чем оперирует IDL:

help, a

Как видим, IDL «забыл» о том, что переменная **a** только что была символьной: теперь она вещественного типа. Это иллюстрирует ту особенность IDL, что *тип и размерность переменной*, находящейся в выражении *слева от знака равенства*, полностью *определяются правой частью выражения*, и поясняет ненужность и бессмысленность предварительного описания переменных в IDL (кроме единственного случая – ввода).

Преобразуем переменную **a** в радианы:

print, a * !dtor

Умножение на системную переменную **!dtor** (Degree TO Radians) как раз и выполняет это преобразование, а обратное преобразование осуществляется умножением на системную переменную **!radeg** (RAdians to DEGree). А теперь вычислим синус этого значения:

print, sin(a * !dtor)

Аналогичным образом можно вычислять и другие прямые и обратные тригонометрические функции. Обратим внимание на функцию арктангенса (**atan**):

print, atan(1)

– получаем значение в радианах, а так:

print, atan(1)*!radeg

– получаем значение в градусах. Арктангенс в IDL может быть и двухаргументным, например:

print, atan(1,1)*!radeg

При этом первый аргумент интерпретируется как ***y***, а второй как ***x***. Это позволяет работать в четырёх квадрантах – сравните:

print, atan(-1)*!radeg

– и, с другой стороны,

print, atan(1, -1)*!radeg

print, atan(-1, 1)*!radeg

а также обходить неприятные случаи типа **±π/2**:

print, atan(1, 0)*!radeg

print, atan(-1, 0)*!radeg

Теперь пара примеров работы с массивами. Снова переопределим переменную **a**:

a = indgen(5)

b = a[1:*]^2

– эта запись означает, что переменная **b** определяется как квадрат элементов массива **a** с 1-го по последний, то есть кроме начального элемента: в IDL элементы массивов нумеруются с нуля, а не с единицы.

help, a, b

– **a** и **b** – целочисленные вектор-строки, первый из которых состоит из пяти элементов, а второй – из четырёх. Посмотрим их значения:

print, a, b

Теперь получим информацию и распечатаем значения результатов нескольких арифметических операций с массивами **a** и **b**:

help, a*b

print, a*b

– получаем поэлементные произведения. Но если массивы содержат разное количество элементов, то излишние элементы отбрасываются. Теперь введём

help, a##b, a#b

– результаты этих операций имеют более высокую точность (длинные целые), а сами они представляют собой матричное произведение и транспонированную ему матрицу:

print, a##b

– *IDL выдаст:*

```
0           0           0           0
1           4           9          16
2           8          18          32
3          12          27          48
4          16          36          64
```

print, a#b

– *IDL выдаст:*

```
0           1           2           3           4
0           4           8          12          16
0           9          18          27          36
```

```
0           16          32          48          64
```

Ещё примеры:

print, a gt 2.5

– IDL выдаст 5-элементный вектор значений выполнения условия «a больше 2.5»: для первых трёх элементов это условие не выполняется (0), для остальных выполняется (1). Заметим, что здесь логические единица и ноль отождествляются с арифметическими: в ответ на команду

print, 2^(a gt 2.5)

IDL выдаст:

```
1       1       1       2       2
```

Приведённые выше примеры можно занести и в программный файл, в конце которого должен обязательно стоять **END**. Этот файл, сохранённый под каким-либо именем (напр., **example1.pro**) можно компилировать и запускать на исполнение (“Compile” и затем “Run”, а более удобно – сначала Ctrl+F5, затем F5). Однако обратим внимание на работу именно *в интерактивном режиме в командной строке*, поскольку благодаря такому режиму у исследователя появляется практически неограниченный простор для импровизации при работе с данными. Точнее, предварительные операции обычно выполняются запуском программных файлов, затем работа происходит в командной строке, а те строки, которые успешно прошли апробацию, могут быть опять внесены в программный файл, и так далее.

Обратим внимание и на то, что эти примеры выполняются в чисто текстовом режиме и будут работать даже в том случае, если между компьютером, в котором запущен IDL, и пользователем нет графического обмена (например, через Secure Shell – ssh).

2.2 Символы языка

Продолжим знакомство с символами, используемыми в языке IDL.

Символ	**Английское название**	**Русское название**	**Назначение**
!	Exclamation mark	Восклицательный знак	системная переменная
@	At Sign		вставка файла
##			матричное умножение
#			аналог матричного умножения при порядке индексов, принятом в IDL: A ## B = B # A
$	Dollar Sign	Знак доллара	перенос; выдача команды операционной системы (в начале строки)
%	Percent	процент	отмечает сообщения компилятора
^			возведение в степень

&	Ampersand		объединение строк
*	Asterisk	звездочка	знак умножения; последний элемент массива; все элементы массива; указатель (pointer)
()	Parenthesis	круглые скобки	обозначение аргументов функций; определение порядка действий; описание индексов массивов
–	Minus	Минус	знак вычитания
+	Plus	Плюс	знак сложения; конкатенация (объединение, сшивка) строк
[]	Square Brackets	квадратные скобки	конкатенация массивов; в IDL версии старше 4-й – указание индексов массива
{}		фигурные скобки	определение структуры
:	Colon	двоеточие	завершает идентификатор метки; задание промежуточных индексов массива; определение полей структуры
;	Semicolon	точка с запятой	комментарий
.	Period	Точка	первый символ исполнимых команд; десятичный разделитель; разделитель полей структуры
,	Comma	запятая	разделитель аргументов
"	Quotation Mark	кавычка	ограничитель символьной константы; указатель восьмеричной константы
'	Apostrophe	апостроф	ограничитель символьной константы; ограничитель числовой константы в 16-ричном или восьмеричном формате
/	Slash	знак дроби	знак деления; установка ключевого слова в единицу
>		больше	выбор наибольшего
<		меньше	выбор наименьшего
=		знак равенства	знак присвоения
mod	Modulus		модуль

?	Question Mark	Вопроси-тельный знак	Получение справки

Ряд символов IDL совпадает с символами, используемыми в других языках программирования. Имеются и отличия.

! (восклицательный знак) – обозначение системной переменной. Это переменные, доступные всегда и всем программам, процедурам и функциям. Например, **!pi** – число π с одинарной точностью, **!dpi** – число π с двойной точностью.

@ (символ "at") – встречается довольно редко. В тексте программы за ним следует имя файла (если без расширения, то предполагается **.pro**), который просто вставляется в программу (размещенную в другом файле). Например, если в программе имеется текст:

.....

.....

@test1

.....

.....,

то вместо строки, в которой находится знак **@**, в текст программы будет вставлено всё содержимое файла **test1.pro**.

Символ # обозначает умножение векторов и матриц, при котором каждый элемент строки первого массива размерности **(N)** умножается на соответствующий элемент столбца второго массива размерности **(1, M)**. В результате получается матрица размерности **(N, M)**. Это действие – аналог матричного умножения при порядке индексов, принятом в IDL – транспонированном по отношению к принятому в линейной алгебре. Двойной символ ## обозначает обычное матричное произведение. При этом

A ## B = B # A

Прежде чем перейти к рассмотрению знака **$**, сделаем некоторые замечания.

В одних языках программирования принято, что каждая строка программы размещается, как правило, в одной строке программного файла (BASIC, FORTRAN). В других языках (Pascal, C) строка программы не связывается со строкой программного файла, лишь обозначается его конец – точкой с запятой.

В IDL принят первый вариант. Строка программы (в принятой терминологии – "statement" – «утверждение») может начинаться на любой позиции строки программного файла, в том числе и с самой первой позиции, и любое количество пробелов и символов табуляции не имеет значения как до или после оператора, так и между аргументами и разделителями (запятыми, скобками и т.п.). Если требуется перенос на другую строку, то в конце данной строки ставится знак доллара (**$**). Одна программная строка может располагаться в трех, четырех и более строках программного файла. Максимальная длина каждой строки не должна превышать 256 символов, что, как правило, не представляет практического ограничения.

Другое назначение знака доллара – запуск в интерактивном режиме некоторого процесса в операционной оболочке (DOS, UNIX). Наибольшее применение этот режим имеет при работе в UNIX. Например, выдача команды **$ ls** (с последующим нажатием Enter) приведет к выдаче списка файлов в текущем каталоге. Ввод одного лишь знака дол-

лара вызовет временный выход в операционную среду, где можно, как в обычном режиме работы в командной строке, выполнить ряд команд. Возврат в IDL – команда **exit**.

Обратим внимание на то, что для IDL безразличны большие (заглавные) и малые (строчные) буквы. Конечно же, это не относится к символьным константам и переменным.

При работе в MS Windows по вводу знака доллара происходит запуск процесса MS DOS (COMMAND.COM). При этом появляется окно DOS, которое после завершения процесса исчезает. Поэтому ввод команды **$dir** не имеет особого смысла: Вы не успеете прочитать информацию в окне, как оно закроется. Но можно выполнить, например, команду **$dir > files.lst**, перенаправив вывод команды **dir** в файл **files.lst**; затем открыть его в любом редакторе или прямо из IDL и просмотреть содержимое этого файла. Ввод только знака доллара запускает окно MS DOS, в котором можно работать, в том числе вызывать различные программы. Окончание работы в этом режиме – команда **exit**.

Близкое назначение и применение имеет команда **SPAWN**. Она может использоваться как в интерактивном режиме, так и в программах. Первое отличие от команды "**$**" состоит в том, что команда **SPAWN** требует аргумента символьного типа, и приведенный выше пример будет выглядеть так (заключен в апострофы или кавычки):

```
spawn, 'dir  > files.lst'
```

В UNIX возможности этой команды еще богаче. Например, список файлов в текущем каталоге может быть помещен в переменную, скажем, с именем **files**:

```
spawn, 'ls', files
```

В этом случае первым аргументом (символьного типа) является команда, вторым – имя переменной, в которую направляется результат выполнения этой команды.

Еще знак доллара можно использовать в идентификаторах (именах) переменных, но в этом случае он не может быть первым или последним в идентификаторе. Допустимы, например, имена переменных:

```
variable_$1
```

```
swq$$$n85am
```

Знак процента в программах IDL не используется, он отмечает сообщения компилятора.

Знаками +, –, *, /, ^ обозначаются, соответственно, сложение, вычитание, умножение, деление, возведение в степень.

Знак '*' имеет также другие значения. Например, если **ARRAY** – 3-мерный массив, то **ARRAY[i1:*, *, i2]** обозначает следующий подмассив массива **ARRAY:**

по 1-му измерению из массива **ARRAY** выбраны ряды элементов с номера **i1** до последнего;

по второму измерению – все ряды элементов;

по третьему – ряд с номером **i2**.

Таким образом, символ '*****' (asterisk) может обозначать также последний индекс или все индексы в данном измерении массива.

Символ "&" (ampersand) используется для объединения нескольких операторов в одну строку. Это особенно полезно в интерактивном режиме, позволяя задавать выполнение сразу нескольких операторов, то есть написать в командной строке небольшую программу в одну строчку. Однако мы не рекомендуем использовать это в программных фай-

лах, поскольку то, что не находится в левом краю строки, плохо воспринимается как левая часть выражения. Иными словами, психологически труднее заметить начало оператора, если в той же строке левее его имеется ещё какой-то программный текст.

Пример. Пусть **x** – массив, содержащий 10 кадров размером 256×256 каждый. Можно создать такой массив следующим простым способом:

```
x = bytarr(256, 256, 10)                    ; создание байтового массива

for j = 0,9 do x[*,*,j] = shift(dist(256), 10*j, -10*j)

; заполнение массива. Для этого выбрана функция DIST, возвращающая массив,
; каждый элемент которого пропорционален его частоте.
; Последующие 9 кадров получаются циклическим сдвигом первого кадра.
```

Теперь ввод строки

```
for j=0, 9 do begin & tvscl, x[*,*,j] & print,j & wait, 0.5 & end
```

позволяет просмотреть последовательность сменяющих друг друга на экране всех десяти кадров. Использование процедуры **TVSCL** вместо **TV** обеспечивает масштабирование каждого кадра по яркости от черного до белого. При этом обеспечивается максимальный контраст каждого изображения. Использование **WAIT, 0.5** обеспечивает задержку в 0,5 секунды после вывода очередного кадра для того, чтобы успеть рассмотреть его.

Круглые скобки ( ) применяются:

- для обозначения аргументов функций;
- для определения порядка действий;
- для описания индексов массивов.

В IDL версий до 5.0 для индексов массивов применялись исключительно круглые скобки. Однако в некоторых случаях это приводит к неоднозначности интерпретации программного текста: обозначение вида **A(B)** может интерпретироваться либо как функция **A** от аргумента **B**, либо как подмассив массива **A**, индексы которого заданы **B**. Такие ошибки могут встретиться либо при отладке программы, либо при работе в командной строке как результат ошибки пользователя. Заметим, что отлаженные, должным образом оформленные и размещённые программы и библиотечные функции интерпретируются без ошибок такого рода. Однако для исключения таких ошибок в IDL начиная с версии 5.0 для индексов массивов введены квадратные скобки. Однако поскольку коррекция огромного количества уже существующих программных файлов требует времени, семейства IDL 5-й и 6-й версий допускают использование для указания индексов массивов как квадратных, так и круглых скобок. Очевидно, что пользователям следует ориентироваться на использование квадратных скобок, чтобы не иметь проблем с последующими версиями IDL.

Квадратные скобки [ ] применяются:

- для описания индексов массивов.
- для объединения в массивы отдельных элементов и массивов (конкатенации).

Пример. Выборка **j**-го элемента из массива **[a, b, c, d]** может быть записана только так: **([a, b, c, d]) [j]**.

В другом примере

```
y = (transpose(x))[5]
```

y – это 5-й элемент массива, полученного транспонированием массива **x**.

Пример использования квадратных скобок для объединения нескольких скалярных величин в массив:

a = [31.5, 2, –40, !dpi] – массив **a**, состоящий из 4-х элементов, имеющий двойную точность – по максимальной точности входящих в него элементов (**!dpi** – число π с двойной точностью); это легко проверить:

help, a

– IDL выдаст:

```
A     DOUBLE     = Array(4)
```

Объединение массивов:

b = [a, a] – массив размерности 8 (двойной точности).

b = [[a], [a]] – массив размерности (4×2).

c = [[[b]], [[b]]] – массив размерности (4×2×2).

Дальнейшее каскадирование скобок невозможно.

Апострофы (') используются как ограничители символьных переменных:

a = 'Year 1947'

b = '1947'

Если внутри последовательности символов встречается также апостроф, он указывается удвоением:

c = 'John''s daughter'

Кроме того, в таком случае можно в качестве ограничителей использовать и кавычки:

d = "John's daughter"

Однако при использовании кавычек следует иметь в виду, что кавычка перед числом будет интерпретироваться как число в восьмеричной форме:

e = "75 abcd"

– IDL выдаст:

```
e = "75  abcd"
       ^
% Syntax error.
```

Однако если ввести

e = "75

help, e

– IDL выдаст:

```
E     INT  =        61
```

Апострофы и кавычки могут использоваться для ввода чисел в восьмеричном и шестнадцатеричном виде:

"**75**	8-ричное число $75_8 = 61_{10}$
'8ab0'x	16-ричное число $8AB0_{16} = 35504_{10}$
'8a'xb	16-ричная запись байта $8A_{16} = 138_{10}$
'8a'xL	16-ричная запись длинного целого $8A_{16} = 138_{10}$

Фигурные скобки ({}) используются для создания переменных типа **структур**. Поля структур могут быть любого типа (конечно, кроме **undefined**) и размерности. Простой пример неименованной структуры:

a = {a : 0, tag_b : 2*findgen(10), tag_c: 'Example'}

Как видно из примера, внутри фигурных скобок, определяющих структуру, могут быть любые выражения.

Двоеточие (:) используется:

в структуре – для определения полей (см. предыдущий пример);

для обозначения меток; при этом следует иметь в виду, что идентификаторы меток должны начинаться с буквы, а не с цифры (как в FORTRAN).

Также двоеточие обозначает все промежуточные значения индексов в массивах, например:

```
var_a[4:8]
```

это эквивалентно

```
[var_a[4], var_a[5], var_a[6], var_a[7], var_a[8]].
```

Точка (.) – обычный десятичный разделитель; например, **3.14159** – число **π**. Кроме того, точка используется для обозначения полей структур. Если воспользоваться примером, приведенным выше:

```
a = {a : 0, tag_b : 2*findgen(10), tag_c: 'Example'},
```

то

```
c = a.a + a.tag_b
```

– это 10-элементный массив.

Запятая (,) – разделитель аргументов. Примеры:

print, a

c = beselj([0.1, 5, 10], 2)

В отличие от FORTRAN, в IDL разделителем является запятая, а не пробел. Пробелы и символы табуляции не играют роли, и их можно использовать для более удобного оформления текста программы.

Буквосочетание mod – остаток от деления выражения на заданное число. Например,

print, 7 mod 5.5

– IDL выдаст:

```
1.50000
```

Обратим внимание на то, что оператор модуля имеет приоритет умножения. Например, последовательность действий в выражениях **16 mod 3*2** и **(16 mod 3)*2** одинакова.

2.3 Управление компиляцией и ходом выполнения программ

Инструкции компилятору, вводимые в командной строке, начинаются точкой. Имеются следующие инструкции (в скобках приведены некоторые из допустимых сокращений):

Инструкция	Сокращённое обозначение	Выполняемое действие
.run	(.r)	запуск компиляции
.rnew	(.rn)	запуск компиляции с удалением всех переменных из памяти
.step	(.s)	выполнить следующий шаг
.step n	(.s n)	выполнить *n* следующих шагов
.continue	(.c)	Продолжить

Инструкция **.run** выполняет компиляцию программного файла. Например, если требуется скомпилировать файл **test1.pro**, достаточно ввести **.run test1**. При этом необходимо, чтобы файл **test1.pro** находился либо в текущем каталоге, либо в одном из каталогов, “известных” для IDL (внесенных в системную переменную **!path**). При наличии синтаксических ошибок в тексте программы, записанном в этом файле, компилятор сообщит об этом. Нередко встречающиеся, но неочевидные сообщения компилятора об ошибках:

```
% End of file encountered before end of program.
```

возникают как в случаях, когда просто отсутствует “**end**” в конце программного текста, так и в случаях, когда не замкнута операторная скобка “**end**” в какой-либо из петель типа:

If … then begin … endif

For … do begin … endfor и т. п.

Скомпилированные процедуры и функции сохраняются в оперативной памяти на все время данной сессии.

Если программный файл не имеет заголовка “**function**” или “**pro**”, то есть интерпретируется как программа **$MAIN$**, по инструкции **.run** он и компилируется, и запускается на выполнение. Иная ситуация с процедурами и функциями: они запускаются на выполнение при их непосредственном вызове. При этом их не требуется предварительно компилировать: IDL сам это сделает при вызове. Если же текст файла, содержащего процедуру или функцию, изменен, следует его еще раз скомпилировать. Об этом следует помнить особенно в тех случаях, когда редактирование осуществляется не во встроенном редакторе IDL.

Инструкция **.rnew** удобна тем, что освобождает память от занимавших ее переменных. При работе в интерактивном режиме на уровне **$MAIN$** часто возникает ситуация, когда часть переменных задана в программном файле, а часть – при интерактивной работе в командной строке. Простой повторный запуск программы не влияет на те переменные,

которые вводились “вручную”. Использование инструкции **.rnew** обеспечивает очистку памяти от переменных, кроме тех, которые находятся в COMMON блоках. Это бывает полезно и при обработке громоздких массивов для освобождения памяти.

О состоянии памяти информирует команда

help, /memory

А команда **help** без аргументов влечет вывод информации о всех содержащихся в памяти переменных, скомпилированных процедурах и функциях. Еще одна опция команды **help**:

help, /routines

позволяет получить минимальную информацию об аргументах, используемых скомпилированными процедурами и функциями. Позиционные аргументы обозначаются строчными, а ключевые слова – заглавными буквами в кавычках.

Инструкции **.step** и **.continue** полезны при отладке, когда в программном файле имеются команды **stop**. Они позволяют продолжить выполнение программы после останова.

2.4 Типы данных и функции преобразования типа

Следующие функции обеспечивают преобразование типа переменных и констант. С их помощью возможно также экстрагировать массивы требуемого типа и размерности из входного байтового массива.

Имя функции	Преобразование в тип:	Диапазон значений:
BYTE	Байт	0…255
FIX	короткое целое (2 байта)	–32768…32767
LONG	длинное целое (4 байта)	–2147483648…2147483647
LONG64	64-разрядное целое (8 байт)	
FLOAT	вещественное (4 байта)	–3.40282e38…3.40282e38
DOUBLE	вещественное с двойной точностью (8 байт)	–1.7976931d308…1.7976931d308
COMPLEX	комплексное (8 байт)	
DCOMPLEX	комплексное с двойной точностью (16 байт)	
UINT	беззнаковое целое (2 байта)	0…65535
ULONG	беззнаковое длинное целое (4 байта)	0…4294967295
ULONG64	беззнаковое 64-разрядное целое (8 байт)	

Общий для этих функций вид обращения:

RESULT = FUNCTION (Expression, [Offset, [Dim1, Dim2, …, Dim*n*]])

Здесь **FUNCTION** – общее обозначение этих функций; аргументы:

Expression – входное выражение

Offset – смещение (в байтах)

Dim1, Dim2, …, Dim*n* – размерность выходного массива.

Возможна работа и со скалярными аргументами, например:

```
P = fix(!pi)

print, P
```

IDL выдаст:

```
3
```

Обратим внимание на встречающуюся иногда ошибку при задании константы с двойной точностью:

~~dPi = double(3.141592653589793)~~

В этом выражении аргумент в скобках – вещественное число с одинарной точностью, и избыточные десятичные знаки игнорируются:

```
print, dPi, format = '(d)'
```

IDL выдаст:

```
3.1415927410125732
```

Корректное же задание числа π с двойной точностью:

```
dPi = 3.141592653589793d0
```

или !dpi – эта системная переменная и содержит число π с двойной точностью.

Функции преобразования типа используются в одном из двух вариантов. В первом случае выполняется просто преобразование типа при сохранении размерности массива (скаляра). Для этого функция вызывается с *единственным* аргументом. Во втором варианте из входного байтового массива можно экстрагировать массивы требуемых типов и размерностей. В этом случае функция вызывается *как минимум с тремя* аргументами. Первый аргумент задает выражение, которое требуется преобразовать. Второй (**Offset**) указывает количество байт от начала массива, которое следует пропустить. Третий аргумент (и, при их наличии, последующие аргументы) определяют размерность массива, который должен быть получен в результате преобразования. Каждый из аргументов может быть некоторым выражением.

Константы соответствующих типов задаются следующим образом:

21B – байт

21 – короткое целое

21.0 – вещественное

21L – длинное целое

21LL – 64-разрядное длинное целое

21U – беззнаковое целое

21UL – длинное беззнаковое целое

Функция **STRING** имеет ряд особенностей, рассмотренных ниже. Здесь мы обратим внимание на то, что при преобразовании байтового массива в символьный его раз-

мерность уменьшается на единицу, и наоборот – при обратном преобразовании. Например, байтовый вектор преобразуется в скалярную строку.

3 Операторы языка IDL

3.1 Операции отношения

Как и в FORTRAN, в IDL имеются операции отношения:

Gt	больше
Ge	больше или равно
Lt	меньше
Le	меньше или равно
Eq	равно

В отличие от FORTRAN, символы операций отношения не заключаются в точки.

Результат имеет байтовый тип.

Аргументами могут быть любые выражения, в том числе массивы; в последнем случае результат также будет массивом. Например,

a = [7, !pi] gt 4

help, a

– IDL выдаст:

```
A    BYTE = Array(2)
```

print, a

– IDL выдаст:

```
1   0
```

Чтобы у Вас не возникало затруднений, лучше позаботиться о том, чтобы аргументы имели одинаковую размерность, если они оба являются массивами, либо один из них был скаляром.

3.2 Логические функции – AND, OR, NOT, XOR (исключающее ИЛИ).

Напомним диаграммы истинности для этих функций:

X1	X2	X1 ***AND*** X2	X1 ***OR*** X2	X1 ***XOR*** X2	***NOT*** X1
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Если логические функции выполняются над многоразрядными числами, то каждый разряд обрабатывается независимо. Например:

print, 5 and 3

– IDL выдаст:

```
1
```

Результат получается таким образом:

Десятичная форма	**Двоичная форма**		
5	1	0	1
3	0	1	1
Результат:	0	0	1

3.3 Условные операции

В IDL имеются условные операторы, позволяющие выбрать один из двух или более вариантов выполнения требуемого фрагмента программы.

1.

a) **IF <скалярное выражение> THEN <любое выражение>**

b) IF <скалярное выражение> THEN begin

<любое выражение>

<любое выражение>

……

<любое выражение>

ENDIF

Иными словами, если после **THEN** следует единственное выражение, то всё условное утверждение начиная с **IF**… может быть записано в одну строку (в случае необходимости – с переносами). Если же после **THEN** следует несколько утверждений, то они заключаются в операторные скобки (**begin, end**). Закрывающая операторная скобка может иметь вид **END** или **ENDIF**, это безразлично; однако если условное выражение достаточно разветвленное, то использование формы **ENDIF** упрощает чтение программы. При работе интерпретаторов более ранних версий IDL встречались ошибки неоднозначности интерпретации последовательности вложенных условных выражений, и использование явно указанного типа закрывающей операторной скобки позволяло избежать таких ошибок. Слово **ENDIF** пишется без пробелов.

Обратим внимание на то, что выражение после **IF** должно быть обязательно *скалярным*, так что следующий пример:

```
IF findgen(10) GT 4 THEN a = 10
```

вызовет останов интерпретатора по ошибке. IDL выдаст:

```
% Expression must be a scalar in this context: <BYTE Array(10)>.
% Execution halted at <имя процедуры (программы)>.
```

Поэтому использование конструкции типа **IF < …> THEN <…>** при работе с массивами вынуждает применять поэлементный перебор, что резко замедляет работу программы. Однако в большинстве случаев удаётся обойти поэлементный перебор с помощью операций отношения, применимых к массивам. В качестве примера рассмотрим такую задачу.

Имеется массив географических координат некоторых точек, причём долготы **LON** заданы в пределах от 0 до 360°, например

```
lon = (sin(findgen(512)/511*!pi*2)+1)*180
```

Нам же в данном примере предпочтительно измерять их от –180° до +180°. Стандартное решение задачи:

```
for j = 0, n_elements(LON)-1 do if LON[j] gt 180 then LON[j] = LON[j]–360
```

Понятно, что если точек не 512, а намного больше, то счёт может оказаться достаточно продолжительным. Однако можно эту задачу решить по-другому:

```
LON = LON - 360*(LON gt 180)
```

Этот пример показывает стандартный приём, применяемый в IDL для избежания использования скалярных операций при работе с массивами.

Условные выражения (как, впрочем, и безусловные) могут содержать оператор безусловного перехода GOTO:

IF a GE 7 THEN GOTO, Label1

……

……

Label1:

……

……

Слово **GOTO** пишется без пробелов.

Обратим внимание на то, что идентификатор метки должен обязательно начинаться с буквы.

Переходы и метки используются в IDL намного реже, чем в языке FORTRAN. Это объясняется хорошей структурированностью языка.

2. **IF <скалярное выражение> THEN <любое выражение> ELSE $**

<любое выражение>

И в данном случае **<любое выражение>** может включать в себя как одиночное выражение, так и совокупность любого числа выражений, заключенных в операторные скоб-

ки **BEGIN** и **END**. Если операторная скобка **END** завершает последовательность операторов, следующих после **ELSE**, она может иметь форму **END** или **ENDELSE**.

3. Если требуется выбрать одну из нескольких возможностей, удобно использовать оператор **CASE**. Например, если в зависимости от значений **a**, равных **b**, **c** или **d**, требуется выполнить различные действия, а при прочих значениях **a** – некоторые другие действия, это записывается так:

```
      CASE  a  OF

b:    <любое выражение>
c:    <любое выражение>
d:    <любое выражение>

ELSE:  <любое выражение>

      ENDCASE
```

Как и в случае с **IF** и **ELSE**, **<любое выражение>** может быть любой совокупностью выражений, заключенной в операторные скобки **BEGIN** и **END**. В данном случае закрывающая операторная скобка **END** не имеет дополнительных суффиксов (как, например, в случае **ENDIF**).

Последовательность операторов **CASE** завершается **ENDCASE** (**END**). Возможность **ELSE** может отсутствовать. Однако удобно вводить **ELSE** в многопозиционных переключателях: при отладке это позволит избежать остановов программы из-за отсутствия подходящих возможностей.

Возможен и часто бывает удобен другой вариант использования оператора **CASE**. Он основан на том, что **IDL** *отождествляет логические единицу (истину) и ноль (ложь) с арифметическими*. Например, вполне реализуемо следующее действие:

```
print, 3^( [5, 2] gt 4)
```

Результат выполнения этого действия – массив [3, 1]. Действительно, **([5, 2] gt 4)** имеет результатом массив [1, 0] в логическом смысле. При возведении тройки в степени, указываемые элементами этого массива, эти элементы интерпретируются как арифметические числа. Это делает понятным результат.

С учетом сказанного мы можем записать, например, такую последовательность:

```
      CASE 1 OF

a gt b:     <любое выражение>
a^b le c:   <любое выражение>
d ne 0:     <любое выражение>

ELSE:       <любое выражение>

      ENDCASE
```

– То есть переключатель строится «наоборот»: перед двоеточием в качестве рассматриваемых возможностей могут быть любые скалярные выражения.

Если оператор CASE нашел подходящую возможность, то последующие возможности не рассматриваются. Это позволяет существенно упростить текст. Например, мы хотим присвоить некоторое значение переменной A в зависимости от того, в каком диапа-

зоне находится величина B. Строгая запись программного фрагмента, выполняющего такой выбор:

```
      CASE 1 OF

b lt b1:                a = a1
b lt b2 and b ge b1:    a = a2
b lt b3 and b ge b2:    a = a3
b lt b4 and b ge b3:    a = a4

ELSE:                   a = a5

      ENDCASE
```

Этот программный фрагмент можно записать и существенно короче:

```
      CASE 1 OF

b lt b1:    a = a1
b lt b2:    a = a2
b lt b3:    a = a3
b lt b4:    a = a4

ELSE:       a = a5

      ENDCASE
```

4 Командная строка и программные файлы

В отличие от обычных языков программирования-компиляторов, IDL является интерпретатором. При использовании традиционных компиляторов на каждом шаге разработки и отладки методики происходит прерывание работы с данными. В отличие от традиционного пути, при использовании IDL этот процесс непрерывный, и результат каждой операции можно проверить и скорректировать непосредственно после её выполнения. Организация IDL как интерпретатора позволяет работать с данными в интерактивном режиме из командной строки, даже без написания программного файла. Это делает IDL непревзойдённо удобным инструментом для обработки и анализа данных. Более того, это радикально ускоряет и упрощает разработку и отладку методик и позволяет опробовать существенно больше методик и алгоритмов по отношению к другим языкам высокого уровня. Интерактивный характер языка значительно упрощает разработку программ для рутинных операций.

Используемый нами подход имеет достаточно общий характер из-за необходимости постоянного совершенствования методик и средств работы с данными, а также появления данных новых типов. Подход состоит в следующем. Разработка методик и программ выполняются параллельно с многостадийной обработкой и анализом данных в интерактивном режиме, что возможно в IDL:

- В естественном интерактивном режиме выполняются формализация, разработка и отработка новой методики или методики работы с новыми данными.
- Процедура, реализующая разработанную методику, опробуется на различных наборах данных. В процессе получения конкретных результатов методика также корректируется.

- Отработанная методика реализуется в виде программы автоматической обработки данных или приложения с графическим интерфейсом. Последний вариант предпочтителен, если требуется активное участие исследователя. IDL позволяет создавать и такие программы. Поскольку всё выполняется в рамках единой языковой базы, разработанные процедуры и фрагменты просто включаются в главную программу.

Для измерения каждой конкретной величины или её зависимости от других параметров пишется отдельный программный файл. Обычно результатами его выполнения являются некоторые количественные значения и графическое представление соответствующих характеристик, зависимостей, пространственной структуры и т.д.

Существенная часть работы при этом выполняется в командной строке. Удовлетворившись после ряда попыток результатами наших действий, мы можем копировать команды из строки ввода команд IDL в программный файл. Таким способом мы последовательно развиваем программный файл в ходе наших экспериментов с данными.

4.1 Программы, процедуры и функции IDL

Имеется три типа программных кодов IDL. Программа наивысшего уровня называется **$MAIN$**. На этом верхнем уровне выполняются программы, записанные в программных файлах, *не имеющих заголовка* **PRO** или **FUNCTION**. Такой программный файл может иметь любое имя. По завершении выполнения программы **$MAIN$** все переменные доступны, и мы можем продолжить работу с ними в интерактивном режиме в командной строке. Этот режим мы рассматриваем как основной, поскольку именно в нём мы можем в полной мере использовать достоинства IDL как интерпретатора.[3]

В отличие от языка C, IDL различает процедуры и функции. Функции возвращают некоторые значения, которые могут быть использованы в выражениях. Процедуры такой возможности не предоставляют. Вызов пользовательских процедур (функций) ничем не отличается от вызова встроенных процедур (функций) IDL.

Процедуры, равно как и функции, которые мы хотим использовать в программе, должны помещаться или в нашем программном файле *до* основной программы, поскольку они должны быть уже скомпилированы до ее запуска, или в отдельных файлах. В последнем случае имена файлов должны точно совпадать с именами процедур (функций) и быть в нижнем регистре (без заглавных букв). Если мы будем следовать этим правилам, у нас не будет проблем ни при работе в MS Windows, ни при работе в UNIX (LINUX).

Процедуры IDL должны иметь заголовок **PRO**, за которым следует имя этой процедуры и список аргументов, например:

```
PRO MY_ROUTINE, arg1, arg2, … argN, …, $
      keyword1 = keyword1, keyword2 = keyword2, …, keywordM = keywordM

      Statement1
      Statement2
      ............
      StatementN
```

[3] Иногда IDL выдаёт сообщение: “Stopped on unknown instruction…” («останов по неизвестной инструкции …»). Чтобы выйти из этой ситуации, следует *очистить все точки останова* (*clear all breakpoints*). Самый быстрый способ сделать это в IDL версий до 5.3 включительно – следует ввести **Alt–R–E**. В IDL более высоких версий это можно сделать с помощью клавишной комбинации **Ctrl–R**.

```
END
```

Если эта процедура размещается в отдельном файле, то он должен иметь имя **my_routine.pro**. Для вызова процедуры нужно просто написать ее имя и задать аргументы, разделив из всех между собой запятыми, в данном случае, например:

```
MY_ROUTINE, x, y
```

Совершенно необязательно процедуру вызывать со всеми возможными аргументами. Это определяется конкретной процедурой и конкретной ситуацией.

Пример функции:

```
Function MY_FUNCTION, arg1, arg2, … argN, …, $
      keyword1 = keyword1, keyword2 = keyword2, …, keywordM = keywordM

      Statement1
      Statement2
      ............
      StatementN

      RETURN, RESULT
END
```

Если эта функция размещается в отдельном файле, то он должен иметь имя **my_function.pro**. В отличие от процедур, всякая функция должна возвращать единственную переменную. Однако она может иметь любой тип и размерность, быть скаляром, массивом или структурой.

Результат вызываемой функции, в отличие от процедуры, может быть использован в некотором выражении, например:

```
alpha = sin(2*MY_FUNCTION(x, y)) + !pi*cos(MY_FUNCTION(z, t))
```

Для передачи параметров между процедурами и функциями мы можем использовать *позиционные аргументы* и/или *ключевые слова*, а также *common*-блоки.

Значение *позиционных аргументов* определяется их положением.

Значение *ключевых слов* определяется их именами. Имена ключевых слов можно сокращать при вызове процедуры (функции) до тех пор, пока сохраняется однозначность в их распознании.

Common-блоки аналогичны ФОРТРАНовским, однако всякий *common*-блок в IDL должен быть именованным.

Слабым местом ряда версий IDL является ситуация, при которой из функции возвращается одноэлементный вектор символьного типа. И если IDL версии 3.0.1 в таком случае выдавал сообщение о внутренней ошибке и, помимо останова программы, ничего фатального не происходило, то IDL 5.3 под Microsoft Windows просто «разваливался» («Программа выполнила недопустимую операцию…»). Речь идёт о том случае, когда о возвращаемый величине **OUTPUT** IDL выдаёт следующую информацию:

```
help, output
```

```
OUTPUT          STRING    = Array[1]
```

Во избежание такой проблемы можно порекомендовать преобразовывать одноэлементный символьный вектор в скаляр перед его возвращением из функции:

```
function ...
............
............
if n_elements(OUTPUT) eq 1 then OUTPUT = OUTPUT[0]
return, OUTPUT
end
```

или

```
function ...
............
............
if n_elements(OUTPUT) eq 1 then return, OUTPUT[0] else return, OUTPUT
end
```

4.2 Программа $MAIN$, процедуры, функции

Поскольку IDL является интерпретатором, все определенные в командной строке переменные содержатся в памяти. При этом считается, что работа идет на верхнем программном уровне, в программе **$MAIN$**. Возможен и другой вариант работы на верхнем программном уровне: запускается на выполнение некоторый программный файл, имеющий расширение "**.pro**". Такой файл не содержит заголовка (**pro** или **function**) и завершается нормальным образом – строкой, в которой записан **END**. Например, если в файле **test1**.**pro** содержатся следующие строки:

```
   ; test1                              ; закомментированная строка,
                                        ; игнорируется интерпретатором

a = bindgen(256)#replicate(1b, 100)     ; создание двумерного массива a
                                        ; размерности 256×100, в 1-м измерении
                                        ; этого массива элементы линейно
                                        ; возрастают от 0 до 255, а все ряды
                                        ; одинаковы

Window, 0                               ; создание графического окна № 0

tvscl, a                                ; выдача массива a на графическое окно
                                        ; в яркостном представлении

print, a[n_elements(a)-1]               ; печать последнего элемента массива a

   End                                  ; конец программного файла
```

то можно его запустить на выполнение, напечатав в командной строке:

```
.run test1
```

и нажав клавишу **Enter**.

В результате на экране появится графическое окно № 0, на которое будет выдан "серый клин", соответствующий выбранной цветовой таблице, или палитре. При выборе линейной черно-белой палитры № 0, которую можно загрузить командой **loadct, 0** (она

устанавливается в режимах PseudoColor и HighColor по умолчанию), клин действительно будет серым. В окне протокола (COMMAND LOG) будет выдано последнее значение массива **a** [255].

Для совместимости с программными файлами, разработанными для более ранних версий IDL, при работе в операционных системах MS Windows имеется возможность использования программных файлов с короткими именами (это устанавливается соответствующей опцией в установках IDL: File–Preferences–General–Clip long filenames). В таком случае точно совпадать с именем файла без расширения должны первые 8 букв имени процедуры (функции). При разработке же новых программных файлов следует использовать только длинные имена файлов и следить за тем, чтобы имя файла в точности соответствовало имени процедуры или функции.

Следует иметь в виду, что в операционной системе **UNIX** регистр букв (строчный или прописной) строго учитывается, поэтому файлы

test1.pro

Test1.pro

TEST1.pro

tEst1.pro

с точки зрения системы **UNIX** – различные и вполне могут располагаться внутри одного каталога. IDL же просматривает при компиляции процедур и функций только те файлы, имена которых содержат лишь строчные буквы. Поэтому мы рекомендуем не присваивать имена файлам процедур и функций в верхнем регистре. Для файлов же, содержащих только программы верхнего уровня ($MAIN$), это не имеет значения.

Пример пользовательской функции. Функция **sign(x):**

$$\mathrm{sign}(x)=\begin{cases}-1 & \text{при } x<0\\ 0 & \text{при } x=0\\ 1 & \text{при } x>0\end{cases}$$

может быть реализована в виде:

```
function sign, x

return, fix(x gt 0) – fix(x lt 0)

end
```

Файл, содержащий эту функцию, должен называться **sign.pro**.

Функция должна возвращать какое-либо значение. В данном случае возвращается результат выполнения выражения **fix(x gt 0) – fix(x lt 0)**. Тогда функция может входить в какое-либо выражение, например:

```
a = x * sign(x)
```

Результат будет аналогичен результату выполнения такой функции:

```
a = abs(x)
```

Выражения в скобках преобразованы операторами **fix** в целый тип. Без этого преобразования они были бы байтового типа, и для отрицательных аргументов возникала бы недопустимая ситуация. Пусть **x = –5**, тогда разность была бы равна

```
0B – 1B = 255B.
```

Преобразование в целый тип обеспечивает надлежащие значения результата.

Таким образом, функция оформляется в виде последовательности строк, содержащих требуемые операторы. Первая из непустых (не закомментированных) строк содержит слово “**function**”, через пробел – имя этой функции, и далее – перечень аргументов, разделенных запятыми.

Последняя непустая строка должна содержать слово “**end**”. Внутри этой последовательности должен быть как минимум один оператор “**return**” с отделенным от него запятой возвращаемым значением (это может быть скаляр, массив любого типа или структура), с помощью которого выполняется переход на вышестоящий (вызывающий) программный уровень, куда передается значение оператора или переменной при операторе **return**. Возможен возврат из разных мест функции в зависимости от каких-либо условий. Соответственно операторов **return** может быть несколько. Функции могут, в принципе, вызываться вообще без аргументов, однако скобки необходимы. Пример:

```
print, n_params()
```

– так определяется количество позиционных аргументов, заданных при вызове функции или процедуры.

В ряде случаев от подпрограммы (процедуры) в принципе не требуется возвращать никаких значений. Это процедуры вывода, управления и т.д. В качестве примеров можно привести следующие стандартные процедуры:

```
Tvscl, dist(200)          ; выдать на графическое окно двумерный
                          ; массив в яркостном представлении
Print, !pi                напечатать число π
```

По оформлению процедуры отличаются от функций тем, что в их первой строке вместо слова “**function**” помещается слово “**pro**”. Возвращать значения из процедур с помощью оператора “**return**” не допускается, хотя сами эти операторы могут использоваться для возврата на вызывающий уровень из тела процедуры (например, при каком-либо условии).

При работе в командной строке часто используется оператор **retall** (**return + all**), осуществляющий возврат на верхний программный уровень **$MAIN$**.

Подчеркнем следующие отличия в использовании процедур и функций между FORTRAN и IDL:

1. *Вызов процедур, написанных пользователем, в IDL никак не отличается от вызова стандартных процедур.*

2. *Если в некотором программном файле имеются используемые в нем процедуры или функции, они должны помещаться **до** программного текста, который их использует (то есть в начале этого программного файла), а не после него.* Эта особенность понятна, если вспомнить, что IDL является интерпретатором: в каждой последующей программной строке должны содержаться либо процедуры или функции, имеющиеся на диске в программных файлах, либо уже скомпилированные.

4.3 Обмен аргументами с процедурами и функциями

В IDL используются два типа указания аргументов:

- позиционное,
- ключевыми словами.

Очевидное удобство ключевых слов состоит в том, что их названия можно унифицировать для различных процедур и функций. Кроме того, допускается сокращенное указание ключевых слов (до длины, еще исключающей неоднозначность).

Например,

routine1, arg1, arg2, arg3, keyw1 = arg4, keyw2 = arg5

Первый способ, обычный для FORTRAN, предполагает указание аргумента его положением в вызывающей последовательности, как в данном случае для аргументов **arg1, arg2, arg3**. Положение аргументов, указываемых ключевыми словами, не имеет значения. Так, вызов процедуры в приведенном примере будет выполнен точно с теми же аргументами и в таком виде:

routine1, keyw2 = arg5, arg1, keyw1 = arg4, arg2, arg3

Это позволяет не запоминать требуемые позиции аргументов для процедур и функций, которые имеют большое количество аргументов. Дополнительно введено соглашение:

запись

routine2, arg1, …, keyw1 = 1

эквивалентна записи:

routine2, arg1, …, /keyw1

5 Циклы

Повторяющиеся циклы выполняются операторами **FOR**, **WHILE**, **REPEAT**. Поскольку IDL – интерпретатор, трансляция программы внутри цикла повторяется с числом входов в цикл. Поэтому сколь возможно *следует избегать применения циклов с большим числом повторений*, иначе производительность резко упадет. С другой стороны, IDL содержит эффективные средства, позволяющие избежать использования повторяющихся циклов. Например, вместо стандартного для других языков программирования приема вроде

```
X = FLTARR(N)

FOR J = 0, N-1 DO X[J] = SIN(J/!PI)
```

который работает очень долго при N >> 1, обычный способ в IDL таков:

```
X = SIN( FINDGEN(N)/!PI )
```

По-видимому, полностью исключить использование циклов невозможно. IDL предоставляет три вида циклов:

- циклы с фиксированным числом повторений,
- циклы, вход в которые выполняется до тех пор, пока соблюдается некоторое заданное условие, и
- циклы, выход из которых происходит тогда, когда заданное условие будет выполнено.

Различие второго и третьего типов циклов состоит в том, что во втором случае условие проверяется на входе в цикл, а в третьем – при выходе из цикла. Поэтому операции, заданные в теле цикла, для второго типа могут ни разу не выполниться, если условие не соблюдается. Если же используется третий тип цикла, то последовательность операторов внутри цикла будет выполнена хотя бы один раз.

5.1 Циклы с фиксированным числом повторений

Эти циклы организуются таким образом:

FOR j = N0, N do <Expression>

или

FOR j = N0, N, Step do <Expression>

Если тело цикла содержит несколько операторов, используется следующая форма:

```
FOR j = N0, N do begin
<Expression>
……
……
<Expression>
endfor
```

или

```
FOR j = N0, N, Step do begin
<Expression>
……
……
<Expression>
endfor
```

Здесь **N0, N** – произвольные скалярные выражения, задающие начальное и конечное значения переменной цикла. Если задан аргумент **Step**, то приращение переменной цикла задается этим аргументом. Если он не задан, приращение равно единице.

Часто переменная цикла изменяется от нуля до некоторого значения, например, числа элементов в массиве минус единица:

FOR j = 0, N–1 do <Expression>

Следует иметь в виду, что при большом числе повторений (свыше 32767) может оказаться недостаточным задание границ переменной цикла как короткого целого числа. Тип этой переменной (здесь **j**) определяется первым аргументом. Поэтому в таких случаях нижнюю границу для переменной цикла в операторе FOR следует задать *длинным целым*:

FOR j = 0L, N–1 do <Expression>

Таким образом Вы гарантируете себя от возможного целочисленного переполнения **j**. В противном случае выдается сообщение об ошибке. В более ранних версиях IDL это приводило к неправильному результату (в результате переполнения двухбайтового короткого целого происходило опрокидывание переменной цикла в отрицательные значения, что приводило к неожиданным результатам).

5.2 Циклы с проверкой условия при входе в цикл

Вход в такие циклы выполняется до тех пор, пока соблюдается заданное условие. Циклы этого типа организуются с помощью оператора **while**:

WHILE <Условие> do <Expression>

Если тело цикла содержит несколько операторов, используется форма:

WHILE <Условие> do begin

<Expression>

……
……
<Expression>
endwhile

Выражение **<Условие>** должно быть скалярным. Его значение проверяется всякий раз ***перед*** выполнением последовательности операторов, составляющих тело цикла**.**

5.3 Циклы с проверкой условия при выходе из цикла

Такие циклы выполняются до тех пор, пока заданное условие не начнет выполняться. Циклы этого типа организуются с помощью операторов **repeat** и **until**:

REPEAT <Expression> until <Условие>

Если тело цикла содержит несколько операторов, используется форма:

REPEAT begin

<Expression>

……
……
<Expression>
endrep until <Условие>

Выражение **<Условие>** должно быть скалярным. Его значение проверяется всякий раз ***после*** выполнения последовательности операторов, составляющих тело цикла**.**

6 Действия с массивами

По существу, все массивы в IDL – динамические. Обратим внимание на то, что в IDL в принципе **отсутствуют** объявления переменных, а тип и размерность переменной в левой части выражения всегда определяется его правой частью. Одна и та же переменная может неоднократно изменять свои размерность, тип и структуру в процессе выполнения одной и той же программы. Однако есть в общем случае две ситуации, когда массивы (или скалярные величины) должны быть предварительно определены. Первый случай – ввод (например, из файла, с клавиатуры и т.п.). Очевидно, что, например, при чтении массива данных из файла прежде всего необходимо знать, каким образом он был записан, какие

его тип и размерность. Второй случай – заполнение отдельных элементов массива. По существу, этот случай можно рассматривать как частный случай ввода.

IDL предоставляет ряд функций для динамического создания массивов. Наиболее общей является **MAKE_ARRAY**. И размерность, и тип создаваемого ею массива могут задаваться результатами выполнения некоторых выражений.

Массивы можно индексировать многомерными или одномерными индексами. Элементы массивов нумеруются не с единицы, а с нуля, то есть первый элемент одномерного массива A имеет номер 0, второй – 1, третий – 2, и т. д.; последний элемент имеет номер **n_elements(A) – 1**. Нумерация элементов массивов транспонирована по отношению к тому, как это принято в линейной алгебре. В частности, в двумерном массиве первый индекс – это номер столбца, а второй – номер строки. Эта нумерация соответствует порядку построчной развертки изображения. Например, [50,30]-й элемент массива размерности (100, 200) можно задать или как двумерный индекс [50,30] или как одномерный индекс 50+30*100 = 3050. Во втором случае одномерный индекс трактуется как порядковый номер при «развёртке» с начального элемента массива (с номером 0) до последнего элемента (в данном случае 100*200–1 = 19999). Одномерный индекс – полный аналог построчный развертки. В таблице приведены примеры того, как элементы массива выводятся на экран, и соответствующие значения двумерных и одномерных индексов.

Двумерные индексы

a[0,3]	a[1,3]	a[2,3]	a[3,3]
a[0,2]	a[1,2]	a[2,2]	a[3,2]
a[0,1]	a[1,1]	a[2,1]	a[3,1]
a[0,0]	a[1,0]	a[2,0]	a[3,0]

Одномерные индексы

a[12]	a[13]	a[14]	a[15]
a[8]	a[9]	a[10]	a[11]
a[4]	a[5]	a[6]	a[7]
a[0]	a[1]	a[2]	a[3]

6.1 Выделение фрагментов массивов

Поясним использование в IDL индексов массивов несколькими примерами. Пусть **i1** and **i2** – скаляры. Тогда:

ARRAY[i1, i2] – элемент массива **ARRAY**, расположенный в колонке **i1** и строке **i2**;

ARRAY[[i1, i2]] – два элемента массива **ARRAY**, имеющие одномерные индексы **i1** и **i2**;

ARRAY[i1 : i2] – вектор-строка, содержащая все элементы массива **ARRAY**, имеющие одномерные индексы начиная с **i1** и кончая **i2**;

ARRAY[i1, *] – вектор-столбец, содержащий все элементы колонки **i1** массива **ARRAY**;

ARRAY[*, i2] – вектор-строка, содержащая все элементы строки **i2** массива **ARRAY**;

ARRAY[i1:*, *] – двумерный массив, содержащий в первом измерении все элементы массива **ARRAY** с **i1**-го до последнего, и все элементы массива **ARRAY** – во втором измере-

нии (знак звёздочки «*asterisk*» ***** означает для данного измерения либо все элементы, либо последний элемент);

ARRAY[*, i1 : i2] – двумерный массив, содержащий в первом измерении все элементы массива **ARRAY**, и все элементы массива **ARRAY** с **i1**-го до **i2**-го – во втором измерении.

6.2 Вставка одного массива в другой

Если имеется два массива **X** и **Y**, и размеры массива **Y** превышают соответствующие размеры массива **X**, то в результате выполнения следующего действия

```
Y[ix, iy] = X
```

массив **X** будет вставлен в массив **Y** начиная с элемента [**ix**, **iy**]. Например:

```
X = DIST(100)
Y = DIST(200)
Y[30, 70] = X
TVSCL, X
```

6.3 Изменение размеров

- Функция **REBIN** преобразует размерность (и увеличивает, и уменьшает) вектора или массива любой размерности вплоть до 8-мерных в новую, заданную параметрами **Di**. Новые измерения должны быть целыми кратными или множителями первоначальных измерений. Расширение или сжатие по каждому измерению не зависит от прочих, так что каждое измерение может быть растянуть или сжато с разными коэффициентами:

```
Result = REBIN(Array, D1, ..., Dn)
```

- Функция **CONGRID** сжимает или расширяет размеры массива размерности 1, 2 и 3 *с произвольным коэффициентом*. **CONGRID** аналогична **REBIN** в том, что обе этих функции могут изменять размеры одно-, двух- и трехмерных массивов, но **CONGRID** изменяет размеры в произвольное (*нецелое*) количество раз. Однако функция **REBIN** работает несколько быстрее. Возвращаемый функцией массив имеет то же число измерений, как и исходный массив, и тот же самый тип (вещественный, целый и т.д.)

```
Result = CONGRID(Array, Nx, Ny, Nz)
```

где **Nx, Ny, Nz** – новые размеры массива.

6.4 Другие преобразования массивов

Имеются другие встроенные функции IDL, выполняющие различные преобразования массивов.

- Функция **ROTATE** поворачивает и/или транспонирует двумерные массивы. **ROTATE** может вращать массивы только на углы, кратные 90°.

- Функция **ROT** выполняет поворот, увеличение или уменьшение и сдвиг двумерного массива на произвольные вещественные значения:

```
ARRAY1 = ROT(ARRAY, ANGLE, MAGNIFICATION, Xcen, Ycen)
```

Здесь **ANGLE** – угол поворота, **MAGNIFICATION** – масштабный коэффициент, **Xcen** и **Ycen** – координаты точки, которая переместится в центр массива после преобразования.

Если требуется только поворот на угол, кратный 90° и/или транспонирование массива, то более эффективно применение функции **ROTATE**.

- Функция **REVERSE** обращает порядок строк или столбцов в одно-, двух- или трехмерном массиве. Например,

```
ARRAY1 = REVERSE(ARRAY, 2)
```

изменяет порядок элементов на противоположный *во втором измерении*.

- Назначение функции **TRANSPOSE** очевидно.

- Функция **REFORM** изменяет измерения массива без изменения полного количества его элементов. Например, если **ARRAY** имеет размерность 100 × 200 (двумерный), то выражение **REFORM(ARRAY, 10, 20, 100)** – это трехмерный массив размерностью 10 × 20 × 100.

Кроме того, если новые измерения не указаны (функция вызвана с одним аргументом), то **REFORM** удаляет все первые «паразитные» измерения величиной в единицу. При использовании функции **REFORM** изменяются лишь измерения массива **ARRAY**, но содержащиеся в нем данные остаются неизменными. Например, если массив **ARRAY** имеет размерность 1 × 100 × 200, то **REFORM(ARRAY)** будет иметь размерность 100 × 200.

Иногда мы получаем вектор-столбцы (например, размерности [1, 100]), и это бывает неудобным. Например, попытка выполнить медианную фильтрацию такого массива вызовет ошибку, поскольку функция **MEDIAN** пытается выполнить двумерное сглаживание, но не находит соседних элементов в первом измерении. С помощью функции **REFORM** проблема устраняется: **MEDIAN(REFORM(VECTOR, N_MED))**. Заметим, что в данном случае тот же эффект достигается с помощью функции **TRANSPOSE**.

- **WHERE** является мощной функцией, помогающей избежать циклов. Эта функция возвращает вектор одномерных индексов элементов массива, для которых выполняется заданный критерий (является истинным). Если не существует ни одного элемента, удовлетворяющего этому критерию, результатом функции **WHERE** является скалярная величина –1. Пример использования этой функции:

```
help, where(findgen(100) gt 50, count), count
IDL выдаст:       <Expression>    LONG      = Array[49]
                  COUNT           LONG      =          49
```

Переменная **COUNT** содержит количество элементов, для которых критерий выполняется. Довольно часто требуется определить первый из элементов массива, для которых истинен критерий. Можно сделать это таким образом:

```
help, (where(findgen(100) gt 50))[0]
IDL выдаст:       <Expression>    LONG      =          51
```

- Другая мощная функция, позволяющая избегать применения циклов – **SORT**. Результатом выполнения этой функция является вектор одномерных индексов, позволяющий выстроить элементы массива в возрастающем порядке. Эта функция в равной степени работает с числами и текстовыми величинами. В последнем случае результатом является массив, отсортированный в алфавитном порядке (включая буквы русского алфавита).

Примеры использования функции **SORT**.

Отсортировать массив по возрастанию:

```
ARRAY = ARRAY[SORT(ARRAY)]
```

Отсортировать символьный массив по алфавиту:

```
array = ['Анна', 'Петр', 'Александра', 'Анисья', 'Захар']
```

```
print, array[sort(array)]
```

IDL выдаст:

```
Александра Анисья Анна Захар Петр
```

- Функция **UNIQ** удаляет из массива повторяющиеся элементы. Заметим, что в результате ее выполнения возвращается индекс *последнего* из повторяющихся элементов в каждом из «цугов». Массив, над которым производится действие, должен быть монотонным. Если это не так, то функцию **UNIQ** следует вызвать с двумя аргументами:

```
ARRAY1 = UNIQ(ARRAY, SORT(ARRAY))
```

- Функция **SHIFT** выполняет циклический сдвиг элементов векторов или массивов по любому измерению на любое целое число элементов:

```
ARRAY1 = SHIFT(ARRAY, i1, i2)
```

Величины сдвигов **i1** и **i2** могут иметь любой знак (то есть направление сдвига). Все сдвиги циклические. Если сдвиг требуется на нецелую величину, следует воспользоваться функцией **ROT**. Значения сдвигов **i1** и **i2** должны быть скалярами.

- Имеется очень мощная функция **POLY_2D** для общего преобразования двумерных массивов (изображений). Эта функция выполняет полиномиальное (в общем случае неконформное) преобразование изображений. В этом преобразовании результирующий массив определяется выражением:

$$g(x,y) = f(x',y') = f[a(x,y),b(x,y)],$$

где $g(x,y)$ представляет пиксел выходного изображения с координатой (x,y), а $f(x',y')$ – пиксел с координатой (x',y') во входном изображении, который используется для получения $g(x,y)$. Функции $a(x,y)$ и $b(x,y)$ – полиномы степени N от x и y, коэффициенты которых P и Q определяют пространственное преобразование:

$$x' = a(x,y) = \sum_{i=0}^{N}\sum_{j=0}^{N} P_{ij} x^j y^i ,$$

$$y' = b(x,y) = \sum_{i=0}^{N}\sum_{j=0}^{N} Q_{ij} x^j y^i .$$

Здесь P and Q – массивы, содержащие полиномиальные коэффициенты. Каждый из этих массивов должен содержать $(N+1)^2$ элементов. P_{ij} содержит коэффициент, используемый для определения x', и является весом члена $x^j y^i$.

Функции **ROT** и **CONGRID** (в двумерном случае) написаны на базе функции **POLY_2D**. Эти функции выполняют частные преобразования из тех многих, которые могут быть выполнены с помощью функции **POLY_2D**. Если заданы два многоугольника, соответствующие исходному и преобразованному массивам, то для нахождения массивов коэффициентов P и Q можно воспользоваться процедурой **POLYWARP**, которая аппроксимирует (x',y') как функцию (x,y).

6.5 Численное задание функций

Имеется возможность задать функцию численно. Например, если некоторое изображение имеет широкий динамический диапазон, мы можем понизить его контраст, используя логарифмическую функцию. Рассмотрим пример:

```
X = 1.2^DIST(200)
TVSCL, X
```

В графическом окне видна лишь самая яркая центральная часть изображения. Однако можно резко уменьшить контраст при помощи логарифмического масштабирования массива при его отображении:

```
TVSCL, alog(X)
```

Однако возможно задать это преобразование и численно. Разумеется, нам не избежать ограничений из-за ограниченности имеющейся памяти у компьютера. Но динамический диапазон массива **X** действительно огромен:

```
print, min(X), max(X)
IDL выдаст: 1.00000 1.57733e+011
```

Поэтому следует перенормировать массив **X**:

```
Y = (X-min(X))/(max(X)-min(X))*2000.
```

и затем создать новый массив **A**:

```
A = alog(findgen(2000))
```

Теперь мы можем выдать массив **X** на экран, применив к нему **A** как численно заданную функцию:

```
TVSCL, A[X]
```

Заметим, что здесь массив **A** одномерный, а массив **X** – двумерный. В принципе, массив **X** может иметь любую размерность. Отметим также, что синтаксис этого выражения такой же, как и для любой функции, применённой к массиву, с точностью до квадратных скобок.

Этот приём можно рассматривать и как *преобразователь кода*.

6.6 Временные переменные

Работая с массивами данных, мы вводим всё новые и новые переменные. Однако память, имеющаяся в наличии, ограничена. Наконец, мы станем замечать, что компьютер стал работать медленно из-за свопинга с виртуальной памятью. Во многих случаях можно улучшить положение, освободив некоторую часть памяти с помощью исключения ставших ненужными переменных. Более того, можно освобождать память в процессе вычислений с помощью функции **TEMPORARY**. Предположим, переменная **A** – большой массив[4]. В процессе выполнения строки

```
A = A + 1
```

в памяти создается новый массив для результата сложения, сумма помещается в этот новый массив, его значение присваивается **A** и затем прежняя область памяти, занимавшаяся массивом **A**, освобождается. Размер полной используемой при этом памяти равен удвоенному размеру массива **A**. Строка же

```
A = TEMPORARY(A) + 1
```

не требует дополнительного места в памяти.

---

[4] Этот пример и ряд описаний здесь и далее заимствованы из HELP для IDL.

7 Работа с символьными переменными

Задать символьную константу несложно – нужно просто заключить ее значение в апострофы или кавычки (см. выше). Для преобразования переменной численного типа в символьную можно воспользоваться функцией **STRING**. Если не применяется спецификация формата, преобразование выполняется с использованием соглашений, принятых по умолчанию.

Имеют особенности преобразование из символьного типа в байтовый и обратное. Как отмечалось, величина символьного типа – константа или переменная (строка) – может иметь произвольную длину, то есть состоять из произвольного числа символов. Массив символьного типа также может состоять из строк различной длины. При преобразовании символьного массива в байтовый число измерений увеличивается на единицу (даже байт от единственного символа интерпретируется как массив размерности (1)), а длина массива в этом измерении становится равной максимальной длине строки. Обратное преобразование производится аналогично; оно уменьшает размерность массива на единицу.

Часто требуется конкатенация, то есть сшивка, символьных переменных или констант. Она осуществляется с помощью знака “+”. Если действие производится с символьным массивом, то оно распространяется на все его элементы:

a = [′10′, ′18′, ′24′]

b = ′ Mar 1994 ′

print, a+b

– будет напечатано:

10 Mar 1994 18 Mar 1994 24 Mar 1994

7.1 Функции и процедуры для работы с символьными переменными

Функции **STRLOWCASE** и **STRUPCASE** позволяют преобразовать все символы данной переменной (в том числе массива) в строчные или заглавные, соответственно.

Функция **STRLEN** возвращает длину строки (и каждой строки из массива строк) в виде длинного целого (массива).

Функция **STRTRIM** позволяет отсечь пробелы, предшествующие отличным от пробела символам или/и последующие пробелы до конца строки. Соответственно имеются следующие опции этой функции: **STRTRIM(var, *n*)** при ***n* = 0** отсекает пробелы в конце строки, при ***n* = 1** – в начале, и при ***n* = 2** – те и другие. Если функция вызвана только с одним аргументом **STRTRIM(var)**, предполагается ***n* = 0**, то есть отсекаются пробелы в конце.

Функция **STRCOMPRESS** уменьшает количество всех стоящих подряд пробелов в символьной переменной или константе до единственного в каждой группе. Если ключевое слово **REMOVE_ALL** установлено в единицу, удаляются все пробелы:

A = STRCOMPRESS(B, /REMOVE_ALL)

Функция **STRMID(A, m, n)** возвращает фрагмент строки (массива строк) **A** начиная с позиции **m** и длиной **n** символов. Если **m + n** превышает длину строки, ошибок не происходит, а возвращается фрагмент с позиции **m** до конца строки.

Функция **STRPOS(A, B, m)** возвращает номер позиции первого слева символа или группы символов **B** в строке **A**. Если задано **m > 0**, предшествующие **m – 1** позиций игнорируются, и анализ начинается с позиции **m**. Если символ **B** в строке **A** не встречается, возвращается значение (**–1**).

STRPUT – контекстная замена – является процедурой, а не функцией. Она используется следующим образом:

STRPUT, A, B, n

В результате содержимое строки **A** начиная с позиции **n** на длину строки **B** заменяется содержимым **B**. Например:

A = 'To-day is 18 May 1977'

B = '02 Apr'

STRPUT, A, B, 10

print, A

– Будет напечатано:

```
To-day is 02 Apr 1977
```

7.2 Ввод-вывод с явным указанием формата

Спецификация формата в общем аналогична принятой в FORTRANe, однако её возможности расширены. Следует помнить, что возможности ввода-вывода с явным указанием формата ограничены количеством строк 1024.

Напомним некоторые основные соглашения этой спецификации.

- поскольку описание формата имеет символьный тип, оно должно быть заключено в апострофы либо в кавычки. Внутри них вплотную к ним должны быть круглые скобки: '**(....)'.**
- Наиболее часто используются следующие форматы (хотя IDL предоставляет и ряд других):

Обозначение	русское название	английское название
I	целый	integer
F	вещественный	floating-point
D	вещественный двойной точности	double precision floating-point
E	экспоненциальный	exponential
G	обобщённый (применяет формат **F** или **E** в зависимости от величины данных)	generalized
A	символьный	alphabetical
Z	шестнадцатеричный	hexadecimal
O	восьмеричный	octal
X	Пробел	space

Регистр (верхний или нижний) не имеет значения. Если число не может быть представлено в заданном формате, IDL информирует об этом, заполняя отведенные под число знакоместа звездочками (*****).

7.2.1 Целочисленный формат

I*n* – *n* десятичных цифр целого числа;

I*n.m* – *n* десятичных цифр целого числа с заполнением нулями пустых знакомест слева. *n* не должно быть меньше *m*. Если целое число содержит количество цифр, равное *l*, меньшее, чем *m*, то (*m*– *l*) старших цифр числа будут заполнены нулями, а (*n*–*m*) – пробелами.

I – представление целого числа соответственно соглашениям IDL, используемым по умолчанию.

Пример:

```
print, 35, format = '(i4.4)'
0035
print, 35, format = '(i4.3)'
 035
print, 35, format = '(i4.0)'
 35
```

Еще один пример. Число **137.91** в формате **I3** будет выглядеть как 138, в формате **I5.5** – как 00138.

7.2.2 Вещественные форматы с фиксированной точкой

F*n.m* – *n* знаков числа, включая точку и возможный знак “**–**”; *m* цифр после точки. Например, в формате **F5.2** число π будет выглядеть так: **3.14**.

Если формат **F** указан без параметра *m***,** то число просто представляется *n* цифрами, причём положение правой цифры определяется точностью числа (7 цифр после точки для одинарной точности и 16 цифр после точки для двойной точности). Если указанное число цифр в формате недостаточно, то вместо всех цифр числа выдаются звёздочки (*******). Если формат **F** указан без параметров *n* и *m***,** то число представляется соответственно соглашениям IDL, используемым по умолчанию. *Всё это относится и к прочим форматам, упоминаемым ниже*.

D*n***.***m* – *n* цифр числа, включая точку и возможный знак "**–**"; *m* цифр после точки. Если формат **D** указан без параметров *n* и *m***,** то число представляется соответственно соглашениям IDL, используемым по умолчанию:

```
IDL> print, !dpi, format = '(D)'
```

IDL выдаст:

```
 3.1415926535897931
```

E*n***.***m* – экспоненциальный формат: *n* цифр, включая точку, возможный знак числа "**–**" и 5 знаков на символ "**e**", знак порядка и три цифры порядка; *m* цифр после точки. Например,

```
IDL> print, !dpi, format = '(e10.2)'
```

IDL выдаст:

```
3.14e+000
```

Если формат **E** указан без параметров *n* и *m***,** то число представляется соответственно соглашениям IDL, используемым по умолчанию.

G *n***.***m* – обобщённый формат. Он автоматически применяет формат **F** или **G** в зависимости от величины данных. Однако при использовании этого формата выдаётся на одну значащую цифру после точки меньше, чем при использовании форматов **F** или **G**.

7.2.3 Символьный формат

Применяется при вводе-выводе данных символьного типа (STRING).

A*n* – *n* символов.

7.2.4 Восьмеричный формат

O*n* – *n* восьмеричных цифр;

O*n***.***m* – *n* восьмеричных цифр с заполнением нулями пустых знакомест слева. *n* не должно быть меньше *m*. Если число содержит количество цифр, равное *l*, меньшее, чем *m*, то (*m*– *l*) старших цифр числа будут заполнены нулями, а (*n*–*m*) – пробелами. Аналогичен формату **I**.

O без параметров *n* и *m* – представление целого числа соответственно соглашениям IDL, используемым по умолчанию.

7.2.5 Шестнадцатеричный формат

Z*n* – *n* шестнадцатеричных цифр;

Z*n***.***m* – *n* шестнадцатеричных цифр с заполнением нулями пустых знакомест слева. *n* не должно быть меньше *m*. Если число содержит количество цифр, равное *l*, меньшее,

чем *m*, то (*m*– *l*) старших цифр будут заполнены нулями, а (*n*–*m*) – пробелами. Аналогичен формату **I**.

Z без параметров *n* и *m* – представление целого числа соответственно соглашениям IDL, используемым по умолчанию.

7.2.6 Формат X

– аналогично FORTRAN, **"X"** с числом впереди указывает количество пробелов. Наличие числа обязательно, даже если оно равно 1. Пробелы могут указываться в спецификации формата и непосредственно.

Описание формата может также содержать любые символы.

Пример.

Day = 8

Month = 'May'

Year = 1984

print, string(day, month, year, format = ' (i2.2, 1X, A3, " ", i4) ')

– IDL выдаст:

```
08 May 1984
```

8 Графика

8.1 Манипуляции с графическими окнами

При работе в среде IDL для вывода графики доступен ряд графических устройств. Получить справку о доступных устройствах можно с помощью команды

`help,/device`

Обычно это следующие графические устройства.

- Графические окна: а) Microsoft Windows (WIN) или б) X Window System (X), если работа осуществляется в системе UNIX (LINUX);
- CGM (Computer Graphics Metafile) – файл, позволяющий передать графику, создаваемую IDL, текстовому процессору (WinWord). Более современная версия метафайла – WMF;
- WMF (Windows MetaFile) – стандартный графический метафайл Microsoft Windows (начиная с версии IDL 5.3);
- HP (Hewlett-Packard Graphics Language) – файл в командах языка графопостроителя HP-GL;
- NULL – псевдоустройство: отсутствие графического вывода;
- PCL (Hewlett-Packard Printer Control Language) – файл на языке управления, применяемом для управления недорогими принтерами фирмы Hewlett-Packard;
- PRINTER – системный принтер;
- PS (файл PostScript) – файл на языке PostScript для управления высококачественной полиграфической техникой;

- Z – графический буфер.

Далеко не каждое из этих устройств имеет возможность работы с графическими окнами – лишь **WIN** и **X**.

- Чтобы создать новое графическое окно, следует ввести

```
WINDOW, N_WINDOW
```

где **N_WINDOW** – его номер (напр., 0, 1, 2, 4, 6 и т.д.). Кроме того, графическое окно номер 0 создается на экране автоматически, если до того ни одного графического окна не существовало (то же и при вводе **WINDOW** без аргумента). Если на экране уже имеется несколько окон, и мы не хотим изменять их содержимого, полезна опция /FREE:

```
WINDOW, /FREE
```

В этом случае IDL создаёт новое графическое окно с незанятым номером (начиная с 32 и в возрастающем порядке).

- Команда **WSHOW, N_WINDOW** выводит указанное графическое окно на передний план. Если номер окна не указан (просто **WSHOW**), поверх остальных окон Windows показывается то графическое окно, которое в данный момент активно. Ряд версий IDL для Microsoft Windows имеет неудобную особенность: когда создается новое графическое окно, на него переключается фокус ввода. Разумеется, нам требуется переключить его обратно на строку ввода команд (что можно сделать с помощью клавишной комбинации **Ctrl–W**). Но в результате новое графическое окно скрывается позади главного окна IDL. Наилучший способ вывести его на передний план – применить процедуру **WSHOW**. Иначе, если просто нажать на значок этого окна на панели задач, следующая попытка перейти в строку ввода команд приведёт к тому, что окно опять скроется. *Существенно, что эта процедура не активизирует графическое окно, а лишь показывает его поверх прочих*.

- Команда **WSET, N_WINDOW** активизирует окно с номером **N_WINDOW** (переключает на него ввод/вывод). *Существенно, что эта процедура не показывает графическое окно поверх прочих, а лишь активизирует его*.

- Команда **WDELETE, N_WINDOW** удаляет указанное окно (окна).

- Команда

```
ERASE, COLOR_INDEX ; (напр.: ERASE, 100)
```

вызывает равномерное заполнение выбранного графического устройства (графического окна – WIN в системе Microsoft Windows или X Window System) цветом, номер которого задан значением **COLOR_INDEX**. Эта же процедура, вызванная без аргумента, стирает графическое устройство тем цветом, который является цветом фона по умолчанию (он задан полем **!P.BACKGROUND** системной переменной **!P**). При стандартных установках IDL фоновый цвет – чёрный.

Некоторые процедуры, работающие с графическими окнами (**RDPIX** – чтение координат и значений пикселей массива, **ZOOM** – просмотр увеличенного фрагмента массива, **PROFILES** – профили массива по строкам и столбцам и другие) предполагают ввод координат с графического окна с помощью “мыши”. Для окончания работы такой процедуры (выхода из процедуры) обычно требуется нажать правую кнопку “мыши”, когда ее курсор располагается на том графическом окне, в котором изображен исследуемый массив. В результате дополнительное графическое окно исчезает, и работа процедуры заканчивается. **Но не следует удалять (закрывать) графические окна, не закончив работы процедуры!** В этом случае IDL будет продолжать ожидать ввода координат курсора. В последних версиях IDL в таком случае создаёт новое, пустое графическое окно. (В более

ранних версиях IDL ожидал ввода координат курсора с уже несуществующего окна, нормальное завершение работы процедуры было невозможно, и единственным выходом в такой ситуации была перезагрузка IDL).

8.2 Координаты

IDL оперирует с тремя координатными системами: **DATA**, **DEVICE** и **NORMAL**. Система координат **DATA** соответствует *последнему выводу на графическое окно, устанавливающему координатную систему* (даже если мы активизируем или создадим другое графическое окно): **PLOT, SURFACE, CONTOUR, MAP_SET** и т.д. Отметим, что процедура **TV** (**TVSCL**) таковой не является: при выводе массива, устанавливая значение каждого пиксела графического устройства в соответствии со значением элемента массива, она не устанавливает координатной системы.

Координатная система **DEVICE** относится к пикселам графического устройства. Например, если мы работаем с графическим окном на экране, координаты в системе **DEVICE** – это просто двумерные номера точек от нуля до размера окна минус единица, то есть, например, [(800–1), (600–1)] для окна размером 800 × 600. Начало координат [0, 0] соответствует нижнему левому углу графического окна.

Координаты нормальной системы **NORMAL** изменяются от [0, 0] в нижнем левом углу до [1, 1] в верхнем правом углу выбранного графического устройства (например, графического окна на экране).

Для работы с координатами полезны **CONVERT_COORD** и **CURSOR**.

- Функция **CONVERT_COORD** выполняет преобразования между различными координатными системами – **DATA**, **DEVICE** и **NORMAL**, например:

```
NORM_COORD = CONVERT_COORD(X, Y, /DATA, /TO_NORMAL).
```

Эта функция всегда возвращает три координаты для каждой точки. Во многих случаях, когда нет третьего измерения, координата **Z** не используется, так что соответствующие величины равны нулю. Эта функция работает корректно и в тех случаях, когда установлена какая-либо географическая проекция.

- Процедура **CURSOR** имеет много возможностей, но, на наш взгляд, неудобна в использовании при работе в командной строке. По этой причине мы разработали *функцию* **CURSORPOS***[5], которая выдаёт координаты текущей позиции курсора в левый нижний угол выбранного графического окна. Эта функция тоже работает со всеми тремя координатными системами (по умолчанию предполагается **DATA**). Функция **CURSORPOS** так же корректно обрабатывает координаты, если установлена какая-либо географическая проекция. Другие опции этой функции:

/TOP – выводит координаты текущей точки в *верхнем правом* углу графического окна;

/TIME – выводит координаты по горизонтальной оси в форме «*часы*, *минуты*, *секунды*». При этом предполагается, что величины по горизонтальной оси отложены в *секундах*.

/HOURS – то же самое, но для случая градуировки горизонтальной оси в *часах*.

/YTIME – то же самое, как и **/TIME**, но время отложено по *вертикальной* оси.

---

[5] Процедуры и функции IDL, помеченные звёздочкой (*), разработаны автором и доступны в Internet по адресу http://ssrt.iszf.irk.ru/idl

Пример:

```
PRINT, CURSORPOS()
```

Чтобы завершить выполнение этой функции, следует нажать либо левую, либо правую кнопку «мыши», когда курсор находится внутри анализируемого графического окна. Как мы уже предупреждали, *не удаляйте графическое окно, пока Вы не выйдете из процедуры*, которая работает с координатами в графическом окне (**PROFILE, PROFILES, CURSOR, CURSORPOS, ZOOM, RDPIX** и т.д.).

8.3 Цвета

8.3.1 Режимы цветопередачи

В IDL есть возможность работать с изображениями в псевдоцветном режиме (*Pseudo Color*), цветном режиме *High Color* и истинно цветном (полноцветном) режиме (*True Color*).

В режиме *True Color* цвет каждого пиксела задаётся триадой базовых цветов. Наиболее часто используется цветовая система *красный*, *зелёный*, *синий* (*red*, *green*, *blue – RGB*).[6] Комбинируя три базовых цвета с соответствующими весами, можно получить все прочие цвета. Например, задавая всем цветам равные интенсивности, мы получим серую шкалу от чёрного до белого.

Интенсивность каждого из трёх базовых цветов независимо задаётся одним байтом; следовательно, возможное количество цветов в режиме True Color равно $2^{3\times 8} = 2^{24} = 16777216 \approx 16$ миллионов. Отметим, что каждое истинно-цветное изображение представляется *трёхмерным массивом*, имеющим два пространственных измерения и одно цветовое. Ниже приведён простой пример того, как создать и визуализировать истинно-цветное изображение.

```
device, decomposed = 1        ; Переключение в истинно-цветной режим

N = 200

x = bytscl(dist(N))           ; Подготовка трёхмерного массива
y = bytarr(N, N, 3)           ; под изображение

for j=0,2 do y[*,*,j] = shift(x, 40*j, -45*j)  ; Формирование изображения

window, 0, xs = N, ys = N

tvscl, y, true = 3            ; Визуализация в режиме True Color массива,
                              ; содержащего цветовую информацию
                              ; в 3-ем измерении
```

Хотя этот цветовой режим имеет наибольшие возможности цветопередачи, в нём не так просто работать с изображениями; кроме того, требуется втрое больше памяти, чем для изображений, цветовая информация в которых может быть представлены одним байтом на пиксел. Упрощённый режим, называемый *псевдоцветным* (*Pseudo Color*), использует *цветовую таблицу*. Эта таблица имеет длину 256 и ставит в соответствие каждому из

[6] Используется не только система RGB: например, система «синий, пурпурный, жёлтый» (cyan, magenta, yellow) также иногда применяется, но она не поддерживается в IDL.

возможных значений от 0 до 255 некоторую тройку RGB. Например, в случае линейной серой шкалы каждый цветовой индекс в цветовой таблице соответствует красному, зелёному и синему цветам, взятыми с одинаковой интенсивностью, равной значению цветового индекса. На рис. 2 показаны эта цветовая таблица и таблица «красной температурной шкалы», имитирующей тепловое излучение чёрного тела.

Отметим, что в однобайтном псевдоцветном режиме *одна* заданная цветовая таблица действует для *всех* изображений во *всех* графических окнах.

Чтобы выделить относительно слабые особенности в изображении, иногда пользуются немонотонными цветовыми таблицами. Это действительно помогает, но часто дезориентирует. К тому же при выдаче представленного таким образом изображения на нецветной принтер трудно понять, что же в действительности ярче, а что темнее. По этим причинам мы предпочитаем использовать только монотонные цветовые таблицы (главным образом те, что только что обсуждались).

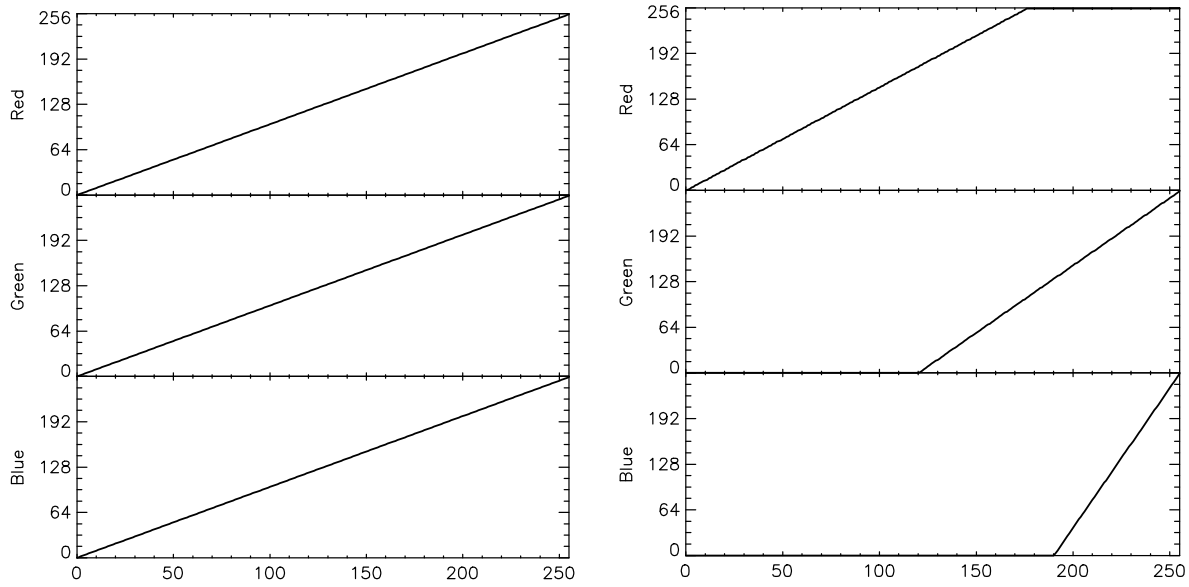


Fig. 2. Цветовые таблицы линейная чёрно-белая (№ 0, слева) and красная температурная (№ 3, справа).

16-разрядный цветовой режим *High Color* является комбинированным: предполагается, что на одном графическом окне (или на экране) одновременно возможно отображение до 256 изображений, каждое из которых выводится со своей цветовой таблицей (тоже 256) – отсюда максимальное количество цветов $2^{2\times 8} = 2^{16} = 65536$. Вывод изображения всегда предполагает следующую последовательность: вначале загружается цветовая таблица, затем выдаётся изображение. Чтобы переключить IDL в режим *High Color*, следует ввести:

```
device, decomposed = 0
```

В IDL этот цветовой режим называется *True Color not using decomposed color* – истинно цветной (*True Color*), но не использующий разложения на базовые цвета.

8.3.2 Манипуляции с цветовой таблицей

В IDL имеется ряд процедур для работы с цветовой таблицей. Мы рассмотрим лишь три из них.

- **TVLCT** – непосредственная загрузка таблицы трансляции цветов:

```
TVLCT, R, G, B
```

Здесь **R, G, B** – векторы, задающие значения базовых цветов, соответствующие цветовым индексам (обычно от 0 до 255). Они могут быть получены, например, в результате чтения графического файла. **R, G, B** могут быть не только численными векторами, но и соответствующими выражениями. С помощью этой процедуры можно также получить текущую цветовую таблицу:

```
TVLCT, R, G, B, /GET
```

- Процедура **LOADCT** загружает одну из 41 цветовых таблиц, записанных в файле **colors1.tbl**, расположенном в главном каталоге IDL. Если текущее графическое устройство имеет менее 256 цветов, цветовая таблица интерполируется.

- Процедура **XLOADCT** показывает загруженную цветовую таблицу и предоставляет графический интерфейс к процедуре **LOADCT**. Она показывает список имеющихся в файле **colors1.tbl** доступных цветовых таблиц. Над текущей цветовой таблицей может выполняться ряд действий – гамма-коррекция, растяжка границ, управление передаточной функцией.

Результаты работы этой процедуры существенно отличаются для разных цветовых режимов. В режиме Pseudo Color производимые изменения сразу же отражаются на содержимом всех графических окон. В режиме High Color изменения цветовой таблицы проявятся лишь при последующей загрузке изображения. Это даёт возможность изобразить даже на одном графическом окне изображения с разными цветовыми таблицами. В режиме True Color с использованием разложенных цветов вообще не приходится ожидать никаких изменений.

- Процедуры, работающие с цветовой таблицей, используют цветовой **COMMON** блок. Если мы хотим воспользоваться содержащимися в нем переменными, в начале текста нашей процедуры или функции или – при работе в интерактивном режиме – в командной строке следует ввести

```
COMMON colors, r_orig, g_orig, b_orig, r_curr, g_curr, b_curr
```

Первые три переменные содержат начальную цветовую таблицу, а последующие три – текущую.

- Иногда требуется инвертировать изображение, содержащееся в графическом окне, то есть преобразовать его в негатив. Это можно сделать следующим образом:

```
z = tvrd()        ; чтение содержимого графического окна в память

tvscl, max(z)-z
```

В режиме True Color негативное изображение получается аналогичным образом:

```
z = tvrd(tru = 3)

tvscl, max(z)-z, tru = 3
```

8.4 Графические ключевые слова

Стандартная процедура вывода графиков **PLOT** имеет множество возможных опций и аргументов. Зададим массив

x = findgen(400)/100 * !pi – набор 100 последовательных значений от 0 до 4π.

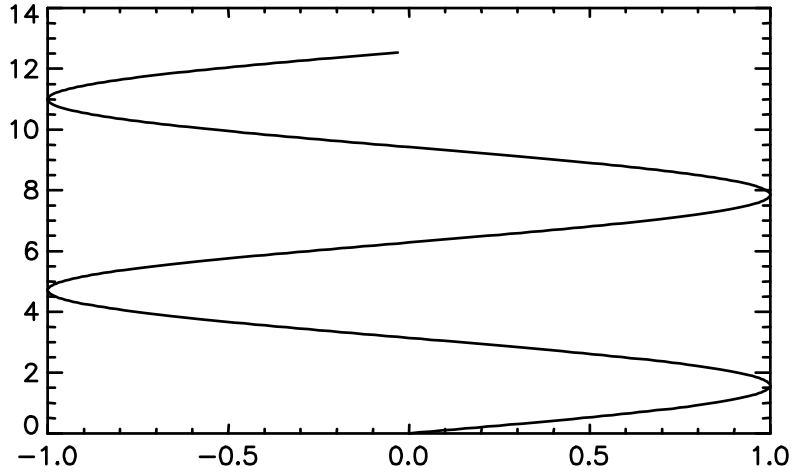
Процедура **plot** может быть вызвана с одним или двумя позиционными аргументами. В первом случае по горизонтальной оси откладываются номера элементов в массиве, во втором – значения первого аргумента.

Вызов с одним аргументом: **plot, sin(x)**

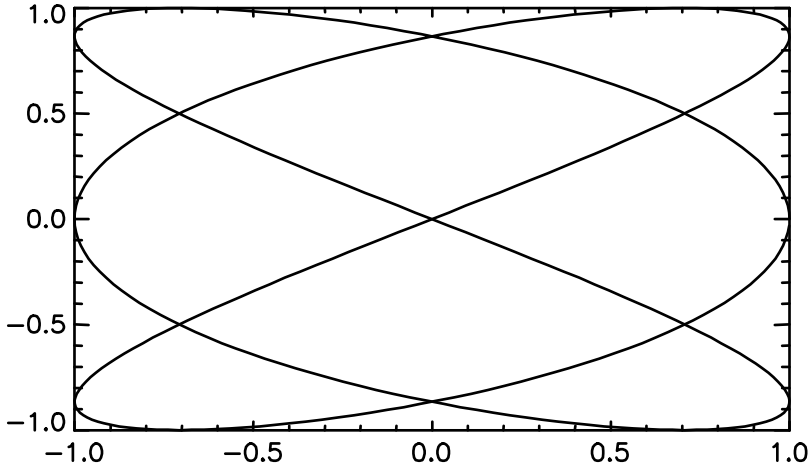
Вызов с двумя аргументами: **plot, x, sin(x)**

Обратим внимание в обоих случаях на горизонтальную ось. Вызов процедуры **plot** с двумя аргументами позволяет легко изображать параметрически заданные функции, неоднозначные функции:

plot, sin(x), x



plot, cos(1.5*x), sin(x)



Можно также изобразить различные модификации этих графиков, например:

plot, cos(3*x), sin(2*x), xstyle = 8, ystyle = 8, xticklen = -0.02, $

yticklen = -0.02, position = [0.5, 0.2, 0.9, 0.8]

В этом случае график будет смещен в правую часть графического окна так, что область, ограниченная координатными осями, будет располагаться от 0.5 до 0.9 от размера

окна по горизонтали и от 0.2 до 0.8 по вертикали (ключевое слово **position** указывает эти значения в нормальных координатах). Координатные оси будут изображены только слева и снизу, но не сверху и справа (**{xy}style = 8**). *Здесь и далее обозначение вида* **{xy}style** *означает* **xstyle** *либо* **ystyle**. Риски на координатных осях будут направлены наружу, на что указывают отрицательные значения аргументов **{xy}ticklen**. Длина рисок указывается в долях размера области графика в соответствующем направлении.

Очевидное удобство ключевых слов состоит в том, что их названия можно унифицировать для различных процедур и функций. Кроме того, допускается сокращенное указание ключевых слов (до длины, еще исключающей неоднозначность). Так, предыдущий пример может быть записан и в виде:

```
plot, cos(3*x), sin(2*x), xst = 8, yst = 8, xtickl = -0.02, ytickl = -0.02, $
pos = [0.5, 0.2, 0.9, 0.8]
```

{xyz}range указывает диапазон граничных значений по соответствующей координатной оси. Это двухэлементный массив. Первое значение может быть и больше второго; в этом случае отсчет значений будет направлен в обратную сторону. Этот массив не указывает жестко диапазон значений, и IDL сам определяет их вблизи заданного диапазона. Для того чтобы IDL строго следовал заданному диапазону, следует также указать **{xyz}style = 1**.

{xyz}style – стиль координатной оси X, Y или Z. Указывается целым числом, каждый из составляющих двоичных разрядов которого имеет следующее значение:

Двоичный разряд	Десятичное значение	Опция
0	1	точно выдерживать диапазон значений
1	2	расширенный диапазон значений
2	4	не изображать координатную ось
3	8	не изображать рамочные оси (ось X только внизу, ось Y только слева)
4	16	запрещает установку минимального значения оси Y в ноль (чтобы более детально показать график) – только для оси Y

Например, **ystyle = 9** = (1 + 8) указывает на то, что ось Y должна быть изображена только с левой стороны графика, и диапазон откладываемых на оси значений должен точно соответствовать диапазону значений Y (если не указан другой диапазон ключевым словом **yrange**). Дело в том, что по умолчанию IDL пытается сам выбрать подходящий диапазон значений, откладываемых на оси, и указание **ystyle = 1** позволяет изменить его требуемым образом.

Одно лишь указание **ystyle = 1** (**xstyle, zstyle**) может и не привести к желаемому результату. Например, если значения **y** изменяются от –0.999 до +0.999, то в этом случае граничные значения по оси и составят [–0.999, +0.999]. Если же требуется, чтобы диапазон значений по оси был [–1, 1], то следует дополнительно задать другой аргумент: **yrange = [–1, 1]**. В этом случае в совокупности с **ystyle = 1** это обеспечит точный диапазон значений по оси Y от **–1 до +1**. **{xyz}range** должен быть 2-элементным массивом.

{xyz}type: 0 – линейный масштаб, **1** – логарифмический масштаб. Изобразить график в полулогарифмическом или двойном логарифмическом масштабе можно и другим способом:

Тип графика	***Указание с помощью {xy}type***	***Модификация имени процедуры plot***
X линейный, Y линейный	plot, x, y, xtype = 0, ytype = 0	plot, x, y
X линейный, Y логарифмический	plot, x, y, xtype = 0, ytype = 1	plot_io, x, y
X логарифмический, Y линейный	plot, x, y, xtype = 1, ytype = 0	plot_oi, x, y
X логарифмический, Y логарифмический	plot, x, y, xtype = 1, ytype = 1	plot_oo, x, y

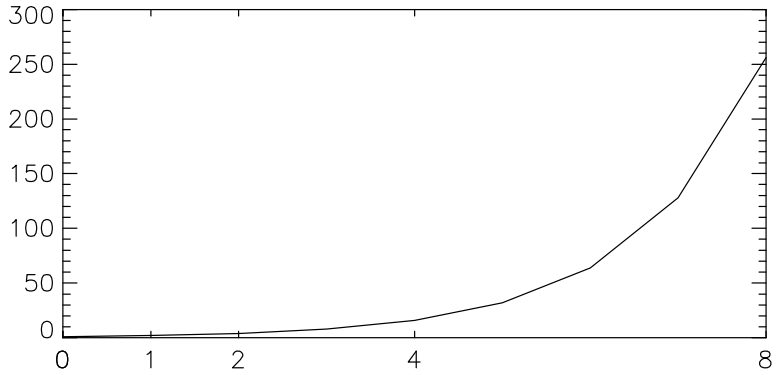
{xyz}ticks – количество рисок на оси (равное количеству делений плюс единица)

{xyz}tickv – значения, у которых будут проставлены риски на координатной оси. Для того, чтобы этот аргумент действовал должным образом, следует одновременно указать {xyz}ticks.

Пример:

```
xtickvalues = [0, 1, 2, 4, 8]
plot, 2^findgen(9), xtickv = xtickvalues, xticks = n_elements(xtickvalues)-1
```

Результат показан на рисунке.



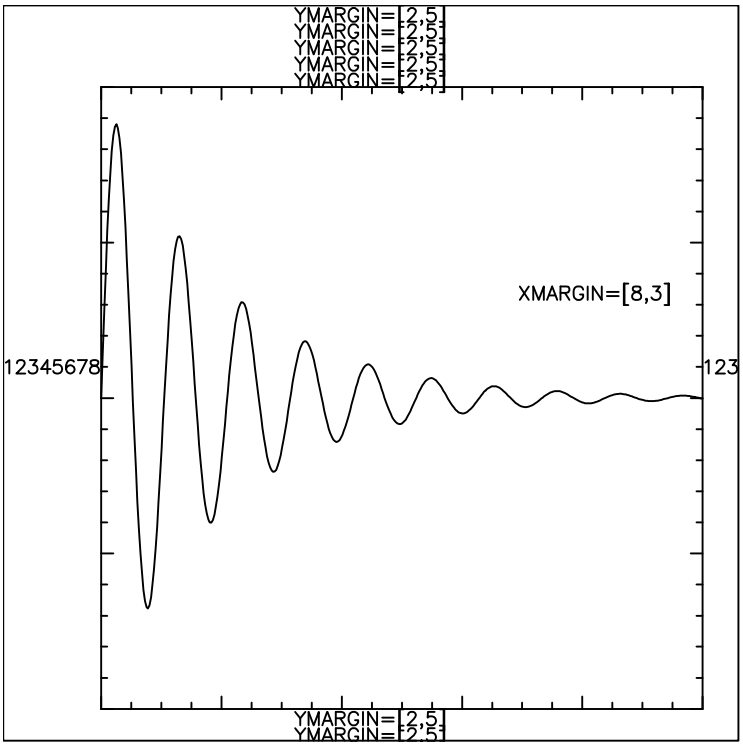
{xyz}margin указывает поля по соответствующей оси. Этот аргумент должен быть 2-элементным массивом. Величина полей указывается в единицах размеров символов по

соответствующему направлению (горизонтали и вертикали). Например, **ymargin = [2, 5]** задает поля по высоте, равные 2 высотам символов снизу и 5 2 высотам символов сверху от области графика, ограниченной координатными осями.

{xyz}tickname – “имена” меток (рисок) на координатной оси, то есть текст, который должен быть помещен у каждой риски. Это должен быть массив символьного типа длиной до 30 элементов. Если длина его меньше числа меток, то будут изменены лишь имена тех делений (начиная с младшего – левого или нижнего), количество которых равно длине этого массива. Последующие останутся без изменения. Также останутся без изменения те деления, именам которых соответствуют пустые символы (″). Если требуется просто подавить маркировку каких-либо меток, например, 2-й и 5-й, следует для них задать в качестве “имен” пробелы, а остальные задать пустыми символами:

xtickname = ['', '', '_', '', '', '_**]**

Если задан аргумент **{xyz}tickformat**, то значения аргумента **{xyz}tickname** игнорируются.



{xyz}ticklen – длина меток на оси, выраженная в нормальных координатах. Положительные значения – риски направлены внутрь, отрицательные – наружу. Если задано нулевое значение, выбирается длина, заданная по умолчанию (+0.02).

{xyz}minor – количество малых делений между основными метками.

{xyz}tickformat – формат надписи у меток координатной оси. Этот аргумент должен быть символьного типа. В простейшем случае можно использовать спецификацию формата, аналогичную принятой в FORTRANe.

8.4.1 Другие часто встречающиеся ключевые слова для графических процедур

linestyle – стиль линии, которой изображается график. Приняты следующие соглашения:

номер	тип линии
0	сплошная
1	точечная
2	штриховая
3	штрих–пунктирная
4	штрих–три точки
5	длинные штрихи

color – цвет.

Узнать характеристики графического устройства (в данном случае – экрана) несложно. Достаточно ввести следующую команду:

help, /device

В ответ выдается сообщение следующего вида:

```
Available Graphics Devices: CGM HP NULL PCL PRINTER PS WIN Z
Current graphics device: WIN
Screen Resolution: 1152x864
Simultaneously displayable colors: 65536
Number of allowed color values: 16777216
System colors reserved by Windows: 0
IDL Color Table Entries: 256
NOTE: this is a TrueColor device
NOT using Decomposed color
Graphics Function: 3 (copy)
Current Font: System,    Current TrueType Font: <default>
Default Backing Store: None.
```

В псевдоцветном режиме цвет зависит от выбранной палитры. Палитра (palette), или цветовая таблица (color table) – это совокупность (R, G, B)-триад (то есть красного, зеленого и синего), поставленных в соответствие каждому из 236 индексов цвета (или их иного заданного числа). Наиболее простой способ задать или сменить цветовую таблицу – воспользоваться процедурой **LOADCT** или **XLOADCT.** Обе эти процедуры позволяют загрузить одну из 38 имеющихся цветовых таблиц, но **XLOADCT** дополнительно имеет широкие возможности их подстройки в интерактивном режиме. Вызов процедуры **LOADCT**:

LOADCT, *n* *n* = 0…37 – номер палитры

При вызове процедура сообщает название палитры. Первоначально загружается черно-белая линейная палитра, в которой цветовой индекс **0** соответствует черному, а максимальный (**235**) – белому цвету. Если требуется, чтобы график на экране был светло-серым на черном фоне, задайте, например,

plot, findgen(10), color = 160

Если, наоборот, Вы хотите получить черный график на белом фоне, можно сделать так:

plot, findgen(10), color = 0, background = !d.n_colors–1

!d – это системная переменная (“**!DEVICE**”), которая содержит информацию о текущем графическом устройстве. О ее структуре и некоторых значениях можно узнать с помощью команды:

help, !d, /structure

В ответ выдается следующая информация:

```
** Structure !DEVICE, 17 tags, length=80:
   NAME            STRING    'WIN'
   X_SIZE          LONG               576
   Y_SIZE          LONG               432
   X_VSIZE         LONG               576
   Y_VSIZE         LONG               432
   X_CH_SIZE       LONG                 9
   Y_CH_SIZE       LONG                12
   X_PX_CM         FLOAT            47.4074
   Y_PX_CM         FLOAT            47.4725
   N_COLORS        LONG          16777216
   TABLE_SIZE      LONG               256
   FILL_DIST       LONG                 1
   WINDOW          LONG                -1
   UNIT            LONG                 0
   FLAGS           LONG            328124
   ORIGIN          LONG      Array[2]
   ZOOM            LONG      Array[2]
```

Более подробно мы познакомимся с ней позднее, а пока для нас существенно то, что в ней содержится информация о количестве доступных цветов (поле **n_colors**). Вспомнив, что в IDL нумерация индексов производится не с единицы, а **с нуля**, мы увидим, что **!d.n_colors–1** и есть максимальный цветовой индекс (в случае цветовой палитры № 0 – белый).

Еще используемые графические ключевые слова:

thick – толщина (1, 2, 3,…); 0 – знчение по умолчанию (1)

charthick – толщина символов надписей на графике

charsize – размер символов для надписей на графике

При одновременном использовании ключевых слов **linestyle** и **thick** > 1 в случае вывода на экран в старых версиях IDL для MS Windows линии все равно изображаются сплошными (как в случае **linestyle** = 0). Это проблема данной версии IDL.

nsum – количество суммируемых точек при выводе на графическое устройство. В некоторых случаях удобно показывать не все множество точек массива, а меньшее число *усредненных* – например, при очень большом их количестве, заведомо превышающем разрешение экрана. Этот режим вывода аналогичен скользящему усреднению массива.

psym (plotting symbol) – символ, которым отмечаются точки изображаемого массива. Точки соединяются между собой отрезками с помощью линейной интерполяции при *отрицательном* значении этого аргумента, при положительном – точки изображаются без соединяющих отрезков. Чтобы установить размер символов, используется ключевое слово **symsize**.

Приняты следующие обозначения:

PSYM	Значение символа
1	Знак плюс (+)
2	Звездочка (∗)
3	Точка (·)
4	Ромб
5	Треугольник
6	Квадрат
7	Крест (×)
8	Символ, определяемый пользователем. См. процедуру **USERSYM**
9	Не определен
10	Режим гистограммы. Горизонтальные и вертикальные линии соединяют изображаемые точки

В случае использования символа № 8 предварительно следует определить его форму с помощью процедуры **USERSYM**. Эта процедура позволяет задать символ, определив его вершины, соединяемые линией заданной толщины, с заливкой (заполнением) или без нее.

Пример. Весьма популярно применение на графиках символа, отсутствующего в стандартном наборе символов IDL – окружности или кружка. Для его задания можно использовать следующую процедуру:

pro circ, fill = fill, thick = thick

; Defines user symbol as a filled or non-filled circle.

```
if n_elements(fill) le 0 then fill=0
if n_elements(thick) le 0 then thick = !P.thick

N = 16

a = findgen(N)/(N-1)*!Pi*2

usersym, cos(a), sin(a), fill = fill, thick = thick

    end
```

Как указывалось, **symsize** – размер символов, используемых для маркировки точек. Это может быть любая положительная величина, целая или дробная. По умолчанию принимается **symsize** = 1.

При изображении графиков на экране качество символов (как, впрочем, и надписей) часто вызывает неудовлетворенность. Однако следует иметь в виду, что причина этого – в недостаточном разрешении экрана. Если вывод графика производится, например, в PostScript, то качество не вызывает претензий.

Еще два ключевых слова, которые используются при построении более сложных графиков:

noerase – не выполнять стирание экрана. В обычной ситуации перед выводом графика выполняется полное стирание экрана. Вызов процедуры **plot** (**contour**) с опцией **/noerase** позволяет подавить это стирание.

nodata – не выводить данные. В некоторых случаях процедура **plot** (**contour**) используется только для установки (задания) системы координат, а вывод данных на экран этой процедурой не требуется (он выполняется последующими процедурами), что и позволяет выполнить опция **/nodata**.

title – генерирует основное название графика, расположенное выше области графика и центрированное. Размер этого названия больше, чем другой текст, в 1.25 раза.

{xyz}title – строка, которая содержит название для определенной оси.

noclip – не обрезать график. По умолчанию все, что выходит за область, ограниченную координатными осями, отсекается. Иногда это неудобно. Установка опции **noclip = 1** (**/noclip**) позволяет избежать этого обрезания.

Пример.

plot, findgen(10), psym = –2, /xstyle

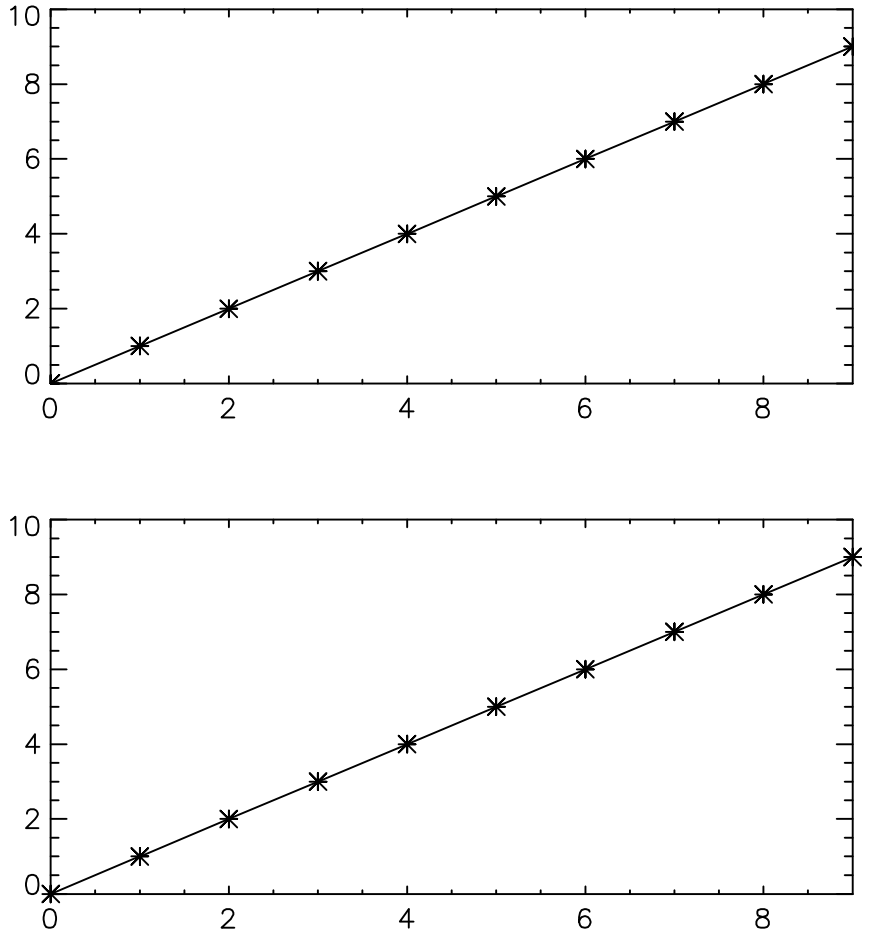
Обратите внимание на первую и последнюю точки графика (см. рисунок). Теперь воспользуемся опцией **/noclip**:

plot, findgen(10), psym = –2, /xstyle, /noclip

В этом случае полностью видны все точки графика.

Прямоугольная область, за пределами которой выполняется обрезание, может быть точно задана с помощью аргумента **clip**. Этот аргумент задается в виде **[X0, Y0, X1, Y1]**, определяя координаты левого нижнего и верхнего правого углов, соответственно. По умолчанию прямоугольник области обрезания задается окном графика – областью, заключенной в пределах осей последнего из выводившихся графиков. Координаты определяются в единицах данных, если это не оговорено другое соответствующим ключевым словом (то есть NORMAL или DEVICE).

Заметим, что, по умолчанию, результат исполнения процедур **PLOTS** и **XYOUTS** не обрезается. Чтобы разрешить обрезание, следует указать **NOCLIP = 0**.



Использование ключевого слова *noclip*. Вверху: *noclip* не задано, внизу: *noclip = 1*

9 Работа с файлами

В ряде случаев нет нужды заботиться о том, каким образом осуществляется запись или чтение данных при обмене с дисковыми файлами: ряд стандартных форматов обслуживается с помощью соответствующих процедур или функций. Одна из часто встречающихся задач – сохранение и восстановление промежуточных и конечных результатов работы.

9.1 Обмен данными. Форматы файлов

IDL стал языком программирования, удобным для обмена *программами* между различными исследовательскими группами. Но как обмениваться данными?

Процедура SAVE предоставляет простой способ сохранения и восстановления данных, полученных в сеансе IDL. Пользоваться ею просто:

```
SAVE, FILENAME = 'IDLSAVE.XDR', var1, var2, var3, … varN
```

Здесь **var1, var2, var3, … varN** – переменные, которые требуется сохранить, **FILENAME** – символьная переменная, содержащая имя файла, в который должны быть записаны сохраняемые данные. Для этой цели используется машинно-независимый формат XDR (eXternal Data Representation). Когда переменные сохранены, мы можем послать этот файл коллеге, например, в другую страну. Адресату же достаточно просто ввести

```
RESTORE, 'IDLSAVE.XDR'
```

– и все сохранённые переменные появятся в оперативной памяти его компьютера.

Этот способ удобен для того, чтобы данными могли пользоваться исследователи, работающие с IDL. Однако формат XDR может быть неизвестен другим пользователям. Имеется другой формат файлов, разработанный для обмена данными между астрономами. Он называется FITS (Flexible Image Transport System – гибкая система передачи изображений). Файлы в формате FITS могут содержать массивы данных любой размерности и точности, например, байтовые, двухбайтовые короткие целые или четырёхбайтовые длинные целые, вещественные одинарной или двойной точности. FITS-файлы содержат двоичные данные, которым предшествует текстовый заголовок, содержащий описание данных и различную сопроводительную информацию – дополнительные параметры и комментарии. Более сложные FITS-файлы после первичного заголовка и массива данных могут содержать расширения.

Мы рассмотрим простейший тип FITS-файлов, которые содержат единственный регулярный массив. Заголовок должен содержать 2880 байтов (36 строк по 80 знаков) или быть кратным этой величине. В левой части заголовка располагаются ключевые слова. Правее указываются значения параметров, соответствующие этим ключевым словам. Ещё правее, после знака дроби, могут присутствовать комментарии о соответствующих параметрах. Заголовок FITS должен содержать некоторые *обязательные* ключевые слова; могут присутствовать ключевые слова, имя и назначение которых стандартизованы; кроме того, могут присутствовать и некоторые произвольные ключевые слова.

Типичный пример заголовка FITS показан ниже (в сокращённом виде). Это заголовок файла, содержащего двумерное радиоизображение Солнца, полученное на Радиогелиографе Нобеяма в Японии.

```
SIMPLE  =                    T / file does conform to FITS standard
BITPIX  =                  -32 / number of bits per data pixel
NAXIS   =                    2 / number of data axes
NAXIS1  =                  128 / length of data axis   1
```

```
NAXIS2  =                  128 / length of data axis   2
DATE-OBS= '1999-12-22'         /
TIME-OBS= '02:12:31.333'       /
STARTFRM=                11528 /
ENDFRM  =                11528 /
ATT-10DB= '00dB    '           /
OBS-FREQ= '17GHz   '           /
DATA-TYP= 'cleaned_map'        /
OBJECT  = 'sun     '           /
TELESCOP= 'radioheliograph'    /
ORIGIN  = 'nobeyama radio obs' /
POLARIZ = 'r+l     '           /
CRVAL1  =              -402.70 / arcsec
CRVAL2  =               279.93 / arcsec
CRPIX1  =                64.50 /
CRPIX2  =                64.50 /
CDELT1  =              2.45552 / arcsec
CDELT2  =              2.45552 / arcsec
CTYPE1  = 'solar-west'         /
CTYPE2  = 'solar-north'        /
SOLR    =               977.11 / optical solar radius (arcsecond)
SOLP    =               7.1330 / solar polar angle (degree)
SOLB    =              -1.7515 / solar b0 (degree)
DEC     =             -23.4374 / declination (degree)
HOURA   =             -1905.14 / hour angle (second)
BUNIT   = 'K       '           / disk = 10000K
END
```

<u>В этом заголовке</u>

Обязательные ключевые слова:

SIMPLE – Признак формата FITS. Должен быть равен «T» («True»)

BITPIX – указывает кодировку (точность) данных

NAXIS – указывает размерность массива данных (*n*)

NAXIS1 – число элементов в первом измерении

…

NAXIS*n* – число элементов в *n*-ом измерении

END – конец заголовка

Стандартные ключевые слова:

CDELT*n* – определяет значение пиксела (масштаб) в *n*-ом измерении

CRPIX*n* – положение точки, относительно которой указывается координата по оси *n*

CRVAL*n* – значение координаты точки CRPIX*n*

TIME-OBS – время наблюдения

DATE-OBS – дата наблюдения

Нестандартные ключевые слова:

SOLR – радиус Солнца

SOLP – позиционный угол Солнца (P0)

SOLB – широта центра солнечного диска (B0)

DEC – склонение

HOURA – часовой угол

9.2 Стандартный способ сохранения – восстановления данных

В IDL предусмотрена техника сохранения–восстановления данных, содержащихся в памяти, в файлы стандартизованного формата XDR (E**X**ternal **D**ata **R**epresentation – внешнее представление данных). Эта техника очень удобна. Файлы в формате XDR машинно-независимы, и данные, записанные, например, на рабочей станции UNIX, корректно воспроизводятся без дополнительных манипуляций, например, в MS Windows. Выполняется это следующим образом.

В процессе работы в оперативной памяти компьютера формируются некоторые переменные (как скалярные, так и векторные). Для сохранения их в файле следует использовать процедуру **SAVE** и указать имена переменных, которые Вы желаете сохранить, и имя файла для их записи:

```
SAVE, var1, var2, var3, var4, filename = filename
```

Здесь ***var1, var2, var3, var4*** – имена сохраняемых переменных (в данном примере их четыре, однако реально количество их произвольно; **filename** – имя файла, в который записываются сохраняемые переменные. Если имя файла не задается, то по умолчанию используется имя файла **idlsave.dat**.

Для последующего воспроизведения в оперативной памяти записанных в файле переменных достаточно вызвать процедуру **RESTORE**:

```
RESTORE, filename
```

Поясним это на конкретном примере.

Создадим в памяти следующие переменные:

```
x = dist(200)
y = findgen(100)
z = 'Test'
```

Сохраним их в файле:

```
SAVE, x, y, z, filename = 'probe.sav'
```

Для того чтобы быть уверенными в "чистоте эксперимента", завершим текущий сеанс работы с IDL и затем запустим его снова. Убедимся, что в памяти нет определенных нами переменных:

```
help, x, y, z
```

– IDL выдаст:

```
X               UNDEFINED = <Undefined>
```

```
Y               UNDEFINED = <Undefined>
Z               UNDEFINED = <Undefined>
```

После этого выполним восстановление данных из файла:

RESTORE, ‘probe.sav’

и повторно введем

help, x, y, z

– IDL выдаст:

```
X               FLOAT     = Array(200, 200)
Y               FLOAT     = Array(100)
Z               STRING    = 'Test'
```

9.3 Как открыть файл

Тем не менее, редко дело обходится без обмена с некоторыми файлами не стандартизованных форматов, либо специфичными для инструмента, либо требуемыми для Вас в процессе Вашей работы. Рассмотрим работу с файлами в общем случае. Прежде всего файл нужно открыть, то есть установить связь с ним по некоторому логическому каналу. Максимальное количество одновременно открытых файлов ограничено 20-ю, однако автору еще не приходилось встречать ситуаций, когда этого было бы действительно недостаточно. Открытие файла осуществляется с помощью соответствующей процедуры, имеющей три модификации:

ORENR, logic_unit_number, filename – открытие существующего файла только для чтения

ORENW, logic_unit_number, filename – открытие файла для записи (если этот файл существовал ранее, он перезаписывается без предупреждения)

ORENU, logic_unit_number, filename – открытие существующего файла как для чтения, так и для записи (в частности, для дополнения, на что указывает буква U – “Update”).

Переменная **logic_unit_number** имеет смысл номера логического канала. Если он непосредственно указывается Вами, он должен иметь значение в пределах от 1 до 20. Этот способ обычно используется при работе в интерактивном режиме. Смысл этого аргумента прозрачен: первый открываемый Вами файл Вы связываете с логическим каналом № 1, второй (когда первый еще не закрыт) – с каналом № 2 и т. д. После того, как Вы закрыли файл, Вы можете снова использовать этот же логический канал для связи с другим, вновь открываемым файлом. Переменная **logic_unit_number** – целое число.

Если Ваша работа с файлом осуществляется не в интерактивном режиме, а из программы, хотя бы самой небольшой, старайтесь использовать другой вариант:

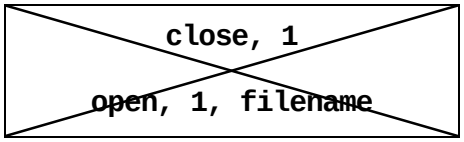
OPEN(R, W, U), logic_unit_number, filename, /get_lun

В данном случае значение **logic_unit_number** не определяется Вами, а отдается “на откуп” IDL. Это просто имя переменной. IDL сам присваивает логическому каналу надлежащее значение. Достаточно очевидное преимущество этого способа состоит в сле-

дующем. Ваша программа может обращаться к разным процедурам и функциям, некоторые из которых, возможно, сами производят обмен с файлами. Было бы весьма неудобно и затруднительно за каждой из них закрепить свой номер логического канала (да и число их ограничено). Если же оперировать с одним и тем же номером (например, 1) в разных процедурах, вряд ли удастся избежать ситуации, когда файл, который должен быть открыт в данной процедуре, уже открыт другой процедурой. Об этом IDL немедленно извещает остановом программы с выдачей сообщения:

```
% OPENR: File unit is already open: 1.
% Execution halted at $MAIN$   (OPENR).
```

Чтобы избежать этого, неопытные пользователи прибегают к такому ухищрению: перед открытием логического канала с явно заданным номером закрывают его:



Этот способ трудно назвать удачным. Гораздо лучше возложить выбор номера логического канала на IDL, что и предоставляет опция **/get_lun**. В этом случае логическим каналам присваиваются номера от 100 до 128. Переменная **logic_unit_number** – длинное целое число.

Заметим, что логические каналы с номерами 0, –1, –2 закреплены за стандартными устройствами ввода–вывода (клавиатурой, терминалом).

Filename – переменная или константа, содержащая имя файла. Если файл находится в текущем каталоге, достаточно указать краткое имя файла (без указания полного пути – в рамках обычных соглашений, принятых в данной операционной системе). Можно указать и полное имя файла с возможностью перенастройки пути:

```
        if (!version.OS eq 'windows') or (!version.OS eq 'Win32') then begin

Delimiter = '\'
path='C:\USER\MYDIR\IDL'

        endif else begin

Delimiter = '/'
path = '/home/mydir/IDL'

        endelse

fname = path + Delimiter + 'proba.dat'
openr, lun, fname, /get_lun
```

Можно две последних строки совместить в одну:

openr, lun, path + Delimiter + ‘proba.dat’, /get_lun

После того, как Вы выполнили с файлом все необходимые операции (что будет рассмотрено ниже), его нужно закрыть. Это возможно осуществить любым из следующих способов:

close, logic_unit_number

close, /all

free_lun, logic_unit_number

В первом случае закрывается указанный логический канал. Во тором – все логические каналы. Процедура **free_lun**, во-первых, закрывает указанный логический канал; во-вторых, освобождает номер, который был ему присвоен.

9.4 Форматные и бесформатные записи

Записи в файлах бывают двух типов:

- двоичные (бесформатные) и
- текстовые (форматные, или ASCII[7])

При записи данных в текстовом виде на каждую цифру приходится один символ (байт) плюс некоторое количество пробелов на все число. Например, число 512 содержит три цифры, и для его записи требуется как минимум 3 байта. Однако максимальное значение короткого целого числа (то есть представляемого в двоичном виде двумя байтами) равно

$2^{(2*8-1)} - 1 = 32767$, поэтому для универсального представления целого числа с учетом знака требуется 6 байт. Как видно, даже без учета разделяющих пробелов запись чисел в текстовом виде очень неэкономична. У нее есть всего два преимущества:

– такие записи легко читаемы “глазами”

– форматные записи переместимы между различными платформами (UNIX, MS Windows).

Вследствие этих преимуществ форматные записи широко используются теми, кто выполняет расчеты на языке FORTRAN.

Однако записи в двоичном виде не только быстрее обрабатываются и более компактны. Они обеспечивают бóльшие возможности, и работа с ними проще. IDL практически не имеет ограничений при обмене данными с файлами; он оснащен эффективными средствами графического представления информации, что в большинстве случаев снимает надобность в непосредственном чтении исследователем численных значений. Поэтому здесь преимущественно используется бесформатный способ записи данных. Для тех же исследователей, которые для вывода больших массивов данных в текстовые файлы активно используют программы, написанные на FORTRANe, можно порекомендовать также

[7] ASCII (American Standard Code for Information Interchange – Американский стандартный код для передачи информации) – семиразрядный код для представления текстовой информации

попытаться освоить бесформатный обмен с FORTRANовскими программами. Так как часто такие массивы данных сопровождаются разнообразной информацией о значениях параметров и т. п., часто возникает необходимость сопровождать эти массивы данных заголовками. Весьма удобен для этих целей формат FITS[8], который рассмотрен в соответствующем разделе.

Тем не менее, IDL позволяет работать как бесформатными, так и с форматными записями, в том числе и в пределах одного и того же файла.

9.5 Получение сведений о файле

В ряде случаев работа с файлами упрощается и становится более гибкой, если возможно из программы получать информацию о длине данного файла, текущей позиции в файле и т. д. Функция **FSTAT(logic_unit_number)** возвращает переменную типа структуры, содержащую следующие поля:

Имя поля	Тип	Значение	Примечание
UNIT	LONG	100	Номер логического канала
NAME	STRING	'D:\FF\050796.PS'	Полное имя файла
OPEN	BYTE	1	Файл открыт
ISATTY	BYTE	0	Не терминал
READ	BYTE	1	Чтение разрешено
WRITE	BYTE	0	Запись не разрешена
TRANSFER_COUNT	LONG	0	Число переменных IDL, переданных при последней операции ввода–вывода по данному каналу. Может использоваться для исправления ошибок обмена
CUR_PTR	LONG	120	Текущее положение указателя (позиция относительно начала файла в байтах)
SIZE	LONG	31300	Длина файла в байтах
REC_LEN	LONG	0	В VMS это поле содержит длину записи, при работе на других платформах – 0.

Например, если известно, что некоторый двоичный (не текстовый) файл имеет регулярную структуру без заголовка и содержит какое-то количество кадров формата 256× 256, записанных короткими целыми числами – по два байта на каждый элемент изображения (пиксель), – то “прочитать его содержимое в переменную” **frames** можно следующим образом:

[8] FITS (Flexible Image Transport System) – “Гибкая система транспортировки изображений”

```
openr, lun, filename, /get_lun                    ; открываем файл с именем filename
status = fstat (lun)                              ; переменная status теперь содержит
                                                  ;       информацию о файле
frames = intarr (256, 256, status.size/(256L*256*2))  ; создаем трехмерный массив
                                                  ;       для ввода данных
readf, lun, frames                                ; читаем содержимое файла
free_lun, lun                                     ; закрываем файл
```

Для чтения содержимого двоичного файла используется процедура **READU** – один из вариантов процедуры ввода (**READU**$_{\text{NFORMATTED}}$ – читать бесформатную запись).

Вторая процедура используется для позиционирования по файлу и, наоборот, для получения текущей позиции указателя в файле. Ее вид:

point_lun, lun, pointer

Здесь **lun** – номер логического канала, по которому осуществляется связь с интересующим файлом. Если **lun** положителен, то в результате выполнения этой процедуры текущая позиция в файле будет изменена на ту, которая задана аргументом **pointer**. Это может быть любое скалярное выражение, значение которого указывает позицию от начала файла в байтах.

Если **lun** отрицателен, то **pointer** интерпретируется как имя переменной, в которую передается текущая позиция в файле, открытом по логическому каналу $|\text{lun}|$.

9.6 Работа с бесформатными (двоичными) записями

****

Перестановка байт при переходе на другую операционную систему!!!

****

9.7 Работа с форматными (текстовыми) записями

Напомним о следующих особенностях IDL.

1. Элементы массивов нумеруются с нуля, то есть первый элемент массива A имеет номер 0, второй – 1, третий – 2, и т. д.; последний элемент имеет номер **n_elements(A) – 1**.

2. Обозначение элементов массивов транспонировано по отношению к тому, как это принято в линейной алгебре: первый индекс – это номер столбца, второй – номер строки.

3. Нумеровать массивы можно раздельно по каждому измерению (2-мерными, 3-мерными и т.д. индексами) или использовать сквозную нумерацию одномерными индексами.

4. В IDL невозможно выполнить считывание в элемент массива; можно считывать только переменную, которая может быть как скаляром, так и массивом.

9.7.1 Запись и чтение массива при его известной длине

Зададим массив **array = findgen(36, 18)**. Откроем файл для записи:

OPENW, lun, ‘probe2.dat’, /get_lun

И теперь выведем массив в этот файл, используя установки IDL, применяемые по умолчанию:

printf, lun, array

Закроем файл:

free_lun, lun

Теперь можно просмотреть содержимое файла, используя любое средство просмотра, имеющееся в системе. Универсальный, хотя и далеко не самый удобный вариант, который можно использовать как в UNIX, так и системах Microsoft (в DOS-моде):

more probe2.dat

Для просмотра следующей страницы нажмите пробел. Прерывается просмотр комбинацией клавиш **Ctrl–C**.

В MS Windows можно просматривать файл в редакторе NotePad (“Блокнот”), если файл не очень велик. Если у Вас имеется Norton Commander или подобная программная оболочка для работы с файлами в среде MS DOS, Вы можете запустить ее прямо из IDL процедурой[9]

spawn, ‘nc’

Или вместо **‘nc’** ввести имя другого соответствующего программного файла (если необходимо, то с указанием пути). Здесь Вы можете просматривать и редактировать файлы обычным в Norton Commander способом, по нажатию клавиш F3 (Alt– F3) и F4 (Alt–F4). Выход из сеанса в оболочке (DOS или UNIX) – по команде **exit**.

В UNIX можно использовать, например, редактор XEDIT, если Вы работаете в оболочке X Windows:

xedit probe2.dat &

Здесь знак & (ampersand) указывает на то, что после запуска программы (в этом случае – xedit) в данном окне могут вводиться и последующие команды до окончания работы с редактором. Иначе до выхода из xedit эта возможность будет заблокирована, и Вам придется запускать еще одно командное окно.

В OpenWindows Вы можете также, выделив интересующий Вас файл (probe2.dat), выбрать в МЕНЮ File Manager “File–Open in Editor”

Теперь прочитаем содержимое файла probe2.dat в IDL. Вообще говоря, для того, чтобы прочитать какой-либо файл, необходимо знать, как он устроен. Либо с этим его “устройством” должна разбираться используемая Вами для чтения программа, либо его должны знать Вы. В данном случае нам известно, что в файле записан вещественный массив размерностью 36×18. Поэтому возможно выполнить считывание таким образом:

OPENR, lun1, ‘probe2.dat’, /get

x = fltarr(36, 18)

---

[9] Процедура **SPAWN** может не работать в IDL версии 5. Если Вы столкнетесь с этой проблемой, скопируйте файл idlspawn.pif из корневого каталога IDL в Ваш текущий каталог.

```
readf, lun1, x

free_lun, lun1

surface, x
```

Чтение форматной записи выполняется процедурой **READF**. Последняя процедура просто выводит на экран содержимое считанного массива.

9.7.2 Запись и чтение нескольких массивов при их известной длине

Пусть **x**, **y** и **z** – три одномерных массива одинаковой длины по 130 элементов, причем **x** – целый, **y** – вещественный, **z** – вещественный двойной точности. Для определенности зададим:

```
x = indgen(130)

y = x*2. + 100

z = y^2.d0
```

Используя применяемые по умолчанию установки IDL, запишем данные в файл в три колонки:

```
OPENW, lun, 'probe3.dat', /get

for j = 0, n_elements(x) – 1 do printf, lun, x(j), y(j), z(j)

free_lun, lun
```

Чтение данных из такого файла возможно несколькими способами.

Способ 1

```
N0 = 130
xr = intarr(N0)
yr = fltarr(N0)
zr = dblarr(N0)
xr1 = 0
yr1 = 0.
zr1 = 0.d0
OPENR, lun, 'probe3.dat', /get
for j = 0, n_elements(xr) – 1 do begin
readf, lun, xr1, yr1, zr1
xr[j] = xr1
yr[j] = yr1
zr[j] = zr1
endfor
free_lun, lun
```

Еще раз обратим внимание на то, что непосредственное чтение из файла в элемент массива невозможно:

~~readf, lun, xr[j]~~

Можно считывать либо скаляры, как в данном примере, либо массивы целиком, как в предыдущем случае.

Способ 2

```
array = dblarr(3, 130)

OPENR, lun, 'probe3.dat', /get

READF, lun, array

free_lun, lun

array = transpose(array)

x_r = fix(array [*, 0])

y_r = float(array[*, 1])

z_r = array[*, 2]
```

IDL выполняет считывание массивов, используя соглашения, применяемые по умолчанию. В форматных записях пробелы интерпретируются как разделители чисел, находящихся в одной строке, а управляющие коды “Cartridge Return” (CR, ′0A′xB) – как разделители строк.

Задав переменную для считывания в виде вещественного массива двойной точности, мы застраховали себя от потери точности при вводе. После считывания можно было бы сразу вычленить из этого массива требуемые подмассивы **x_r1 = fix(array[0, *])**, однако в этом случае **x_r1** получился бы вектор-столбцом, а не вектор-строкой, каким был исходный массив **x**. В этом случае он рассматривался бы IDL как двумерный массив размерности $\mathbf{1 \times 130}$, а это может привести к некоторым ограничениям: например, операторы скользящего усреднения **SMOOTH** и медианного сглаживания **MEDIAN**, если их входные аргументы – двумерные массивы, выполняют **двумерное** сглаживание. В таком случае, задав минимальную ширину 3, мы тут же столкнемся с ошибкой: размерность массива в 1-м измерении (1) окажется меньше ширины сглаживания (3). Поэтому проще всего вначале транспонировать массив, а затем выделить из него требуемые подмассивы.

9.7.3 Общий случай: файл с заголовком, длина массивов неизвестна.

Рассмотрим общий случай. Вначале создадим файл, требуемый для нашей задачи. Воспользуемся теми же массивами

```
x = indgen(130)

y = x*2. + 100

z = y^2.d0
```

Сформируем заголовок, например, таким образом:

```
Header = [′test file′, ′x = indgen(130) ′, ′y = x*2. + 100′, ′z = y^2.d0′]
```

Убедимся в том, что заголовок – действительно 4-элементный текстовый массив:

help, Header

Откроем для записи файл **probe4.dat**:

OPENW, lun, ′probe4.dat′, /get

Запишем заголовок (по одному элементу в каждой строке):

printf, lun, Header, format = ′ (A) ′

printf, lun, format=′(40("*"), /)′

Теперь запишем данные, указав формат явно:

for j = 0, n_elements(x) – 1 do $

printf, lun, x[j], y[j], z[j], format = ′ (i3, 2X, f5.1, 2X, f10.2) ′

free_lun, lun

Теперь мы можем просмотреть файл с помощью любой программы просмотра.

Будем считать, что количество записей (строк) нам неизвестно. В таком случае можно выполнить чтение в цикле до достижения маркера конца файла.

Теперь попробуем прочитать содержимое файла **probe4.dat**.

```
file = 'probe4.dat'
add_eof, file
a1 = 0
b1 = 0.
c1 = 0.d0
head = strarr(4)
OPENR, lun, file, /get
readf, lun, head
j = – 1L
     while not EOF(file) do begin
READF, lun, a1, a2, a3
if j lt 0 then begin
a = a1
b = b1
c = c1
endif else begin
a = [a, a1]
b = [b, b1]
c = [c, c1]
endelse
j = j + 1
     endwhile
free_lun, lun
help, a, b, c
print, head
```

Этот способ достаточно универсален, но при большой длине файла работает очень долго.

Еще один возможный вариант: файл считывается построчно в память с помощью процедуры **READF**, а затем элементы массивов, расположенные по несколько в строке, выделяются с помощью процедуры **READS**, имитирующей считывание из файла при считывании из оперативной памяти.

file = 'probe4.dat'
OPENR, lun, file, /get

```
function readform, file

;+ Performs formatted reading
; of the string-type array from a file
;–

if n_elements(file) le 0 then $                ; Диалоговый выбор файла, если он
        file = pickfile(/read)                 ; не задан как входной аргумент
if file eq '' then return, ''                  ; Возврат, если файл не выбран

widget_control, /hour                          ; Форма курсора меняется на “часы”

  openr, lun, file, /get                       ; Открыть файл

st = fstat(lun)                                ; В переменную st передается
                                               ; информация о файле
data = bytarr(st.size)
readu, lun, data

point_lun, lun, 0

iii = where(data eq '0A'xB)
N = n_elements(iii)

data = strarr(N)
```

```
readf, lun, data

st = fstat(lun)                          ; В переменную st передается
                                         ; информация о файле

  if (st.cur_ptr + 1) lt st.size then begin
tmp = ''
readf, lun, tmp
data = [data, tmp]
  Endif

  free_lun, lun

return, data

  End
```

10 Общие замечания об обработке и анализе данных с использованием IDL

В исследованиях по солнечной физике мы привыкли иметь дело с большими объемами данных. Если мы занимаемся теоретическим или численным моделированием, мы должны представлять наши результаты. В настоящее время многие солнечные инструменты на космических и наземных обсерваториях поставляют огромные количества данных. Поэтому нам приходится работать с данными в любой области солнечных исследований. В этих лекциях мы хотим представить вам прежде всего общие методы и методики обработки и анализа данных. С помощью этих методик и пользуясь преимуществами языка обработки данных *IDL* вы сможете решать различные задачи солнечной физики.

В этих разделах мы обсудим:

в разделе 1 – общие вопросы обработки и анализа данных и некоторые особенности использования IDL;

в разделе 2 – анализ одномерных массивов;

в разделе 3 – анализ двумерных массивов;

раздел 4 посвящен анализу трехмерных массивов;

в разделе 5 мы намереваемся показать, как строить программы с графическим интерфейсом (приложения);

и, наконец, в разделе 6 обсудить современное состояние анализа данных с использованием IDL и высветить некоторые проблемы.

При рассмотрении процедур и функций IDL мы будем останавливаться лишь на их важнейших возможностях и особенностях, полагая, что информацию о дальнейших деталях читатель сможет получить сам из справочной системы IDL и соответствующей документации.

10.1 Типы данных

Особенности объектов, с данными о которых мы имеем дело, инструментов, на которых они получены, физики исследуемых объектов и процессов, как и другие факторы, существенно влияют на задачи обработки данных и их анализа, а также на используемые методы. Например, изображения фотосферы и хромосферы Солнца содержат солнечный диск с потемнением к его краю; микроволновые изображения Солнца – тоже подложку солнечного диска, но почти равномерной яркости. Динамические спектры солнечного радиоизлучения содержат продольные полосы радиопомех от радиовещательных и телевизионных станций на фиксированных частотах. В картах распределения геофизических параметров присутствуют контуры континентов и т.д. Поэтому читателю следует самому выделить такие особенности в данных, с которыми он работает, и подумать о том, как эти особенности использовать при работе с этими данными. Автор имел дело, главным образом, с солнечными данными, что определяет специфику ряда рассматриваемых методик и примеров, а также содержание некоторых параграфов этой книги.

Помимо особенностей, присущих данным различного происхождения, есть и некоторые общие признаки. В частности, характер и реализация методик зависят от размерности массивов данных. Мы различаем три главных типа наборов данных по их размерности: одномерные, двумерные и трёхмерные.

Одномерные наборы данных – это, например, временные профили, представляющие изменения некоторого параметра (интегрального потока излучения, записанного радиометром), или пространственные профили, выдаваемые инструментами с одномерным пространственным разрешением (например, линейными интерферометрами).

Солнечные изображения или карты, динамические спектры или последовательности одномерных профилей представляют примеры *двумерных наборов данных*.

Трехмерные наборы данных – это наиболее типичный тип данных современных наблюдений. Множество космических и наземных астрономических инструментов поставляет наборы двумерных изображений, например, серии кадров, снятых во время солнечной вспышки. В этом случае мы имеем дело с последовательностью двумерных изображений, две координаты которых X и Y относятся к пространственным измерениям, а номер изображения связан со временем. Таким образом, мы можем рассматривать такой набор данных как *трехмерный куб данных*, характеризующийся тремя координатами X, Y и t.

X и Y – это в действительности номера пикселов вдоль горизонтальной оси (номер колонки) и вертикальной оси (номер строки), а t – номер слоя по глубине куба данных.

Разумеется, существуют данные и большей размерности, но нам неизвестны развитые методы для их обработки. Единственный общий метод, который мы могли бы порекомендовать – *понижение числа измерений* набора данных. Это можно сделать, например,

- рассматривая выборки данных раздельно,

- используя интегральные характеристики (например, интегральный поток[10], средние параметры по изображению и т.д.)

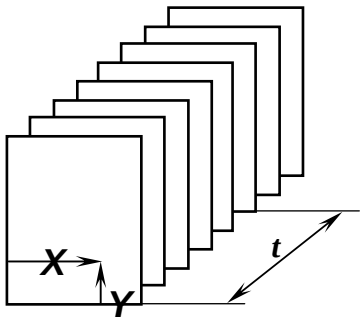


Рис 1. Куб данных

Помимо этого, для эффективного анализа наборов данных можно воспользоваться удобным представлением данных (например, просмотр в режиме «кино»).

Эти типы не исчерпывают всех существующих типов данных. Например, базы данных содержат наборы данных, организованные в многоуровневые, иерархические структуры. Этот тип данных хорошо изучен, методы работы с такими данными детально разработаны, и вы можете освоить их с помощью существующей литературы.

10.2 Общие задачи обработки и анализа данных

Для всех типов данных существуют следующие общие задачи обработки и анализа:

- Просмотр данных. Графическое представление помогает нам обнаружить и изучить исследуемые особенности. Во многих случаях не так просто представить наши данные и сделать наши выводы убедительными для нас же самих и наших коллег. Поэтому визуализация является важной задачей.
- Коррекция инструментальных искажений и поиск дефектов. Реальные записи данных часто неидеальны. Иногда дефекты, присутствующие в данных, искажают результаты или даже препятствуют успешному анализу данных. В некоторых случаях такие проблемы можно скорректировать, иногда – нет. Ограниченное качество данных определяет круг задач, которые могут решаться с использованием этих данных.
- Устранение дефектов. В общем случае возможна одна из двух ситуаций:
    - а) записи данных столь некачественны, что их нельзя использовать. В этом случае мы должны удалить плохие записи, не поддающиеся коррекции;
    - б) иная ситуация – качество данных оставляет желать лучшего, но его можно повысить. В таком случае нужна коррекция плохих записей. Пример – набор изображений с большими дрожаниями между кадрами, но без других дефектов.
- Поиск интересующих особенностей. Это общая задача, однако иногда эти особенности непросто обнаружить, например, из-за широкого динамического диапазона данных, или если наш набор данных столь велик, что непростой задачей является даже его просмотр.

[10] Например, вычисление временного профиля интегрального потока понижает количество измерений куба данных с 3 до 1.

- Калибровки и нормировки. Говорят, что «наука начинается с измерений». Если данные не калиброваны, трудно проводить количественные оценки. Иногда калибровки выполняются с использованием опорных (эталонных) источников; в других случаях можно откалибровать данные, используя их собственные свойства (так называемая «самокалибровка»).
- Измерение параметров исследуемых особенностей. Эта задача представляется очевидной.
- Вычисление общих характеристик. Когда мы нашли качественные и количественные параметры наших исследуемых особенностей, следующий шаг – описать характеристики анализируемого объекта, события или явления как целого. При этом можно ожидать зависимостей определённых параметров от некоторых условий и пытаться найти важнейшие величины с помощью аппроксимации ожидаемых функций к экспериментальным наборам данных.
- Интерполяция и аппроксимация. Задачу, сформулированную в конце предыдущего абзаца, позволяет решить *аппроксимация* ожидаемых функций к экспериментальным точкам *методом наименьших квадратов*. Найти коэффициенты для наилучшей аппроксимации некоторой величины, зависящей от линейной комбинации нескольких независимых параметров, позволяет линейная *регрессия*. Различные методы *интерполяции* позволяют отыскать значения некоторой величины в промежуточных точках между существующими экспериментальными отсчётами. Другим полезным приложением интерполяции является вычисление значений параметра в узлах новой сетки отсчётов.
- Совместный анализ различных данных, полученных в наблюдениях на различных инструментах в разных спектральных диапазонах – одна из важных задач в экспериментальной солнечной физике. В частности, мы рассмотрим, как сравнить разные изображения.

В основном мы будем обсуждать различные методики представления и анализа данных.

11 Одномерные массивы данных

11.1 Визуализация данных

Одна из важнейших манипуляций с данными – их визуализация, то есть представление в графическом виде на экране и/или в виде твёрдой копии. Те, кто привык работать с данными на FORTRANe, обычно просматривают численные значения, сгруппированные в колонки и строки. Однако этот способ не нагляден, занимает много времени и поэтому неэффективен. Разумеется, иногда можно и распечатать некоторые значения, но это требуется не так уж часто. Намного более эффективен просмотр наборов данных на экране в графическом представлении. IDL предоставляет для этого следующие возможности.

Для того, чтобы вывести одномерный массив **Y** на экран, следует ввести команду:

```
PLOT, Y
```

В этом случае мы увидим изменения переменной **Y** как функцию от номера элемента в массиве. Однако если частота снятия отсчётов неравномерна, истинный график будет искажён. В этом случае лучше воспользоваться другим способом, свободным от этого недостатка:

```
PLOT, X, Y
```

Например, **Y** представляет некоторый временной профиль, а **X** – массив значений моментов времени, в которые сняты отсчёты. Даже если частота снятия отсчётов неравномерна, мы увидим правильный график.

Отметим, что вызов процедуры **PLOT** *с двумя аргументами – наиболее общий и гибкий способ* использования процедуры, позволяющий просто строить корректные графики в более сложных случаях:

графики с осями, размеченными в часах, минутах и секундах;

графики неоднозначных функций (у которых одному значению **X** соответствует более одного значения **Y**):

PLOT, X, Y и **PLOT, Y, X**

изображение нескольких величин с различными частотами отсчётов на одном и том же графике:

```
PLOT, time1, value1
OPLOT, time2, value2
```

и так далее.

Если визуализируемая величина изменяется в очень широких пределах, можно воспользоваться графиками с логарифмическими или полулогарифмическими осями:

PLOT_IO, X, Y линейная горизонтальная ось, логарифмическая вертикальная ось

PLOT_OO, X, Y обе оси логарифмические

PLOT_OI, X, Y логарифмическая горизонтальная ось, линейная вертикальная ось

Если изображаемая величина изменяется в небольших пределах относительно некоторого большого значения, можно подчеркнуть эти изменения с помощью

```
PLOT, X, Y, /ynozero
```

Если мы хотим ограничить набор данных некоторым значением **LEV** снизу, можно сделать так:

```
PLOT, X, Y > LEV
```

Знак “больше” (>) – оператор максимума в IDL. Величина "**A > B**" равна наибольшей из **A** и **B**. Аналогично, знак “<”– оператор минимума в IDL. Оба этих оператора можно применять к массивам любой, но одинаковой, размерности.

Если мы хотим рассмотреть более детально некоторую часть графика, можно задать диапазон для **X** и/или **Y**:

```
PLOT, X, Y, xrange = [X0, X1], yrange = [Y0, Y1]
```

Для изображения более одного графика на графическом устройстве можно установить поле **MULTI** системной переменной **!P**. Это 5-элементный массив, элементы которого задают:

Элемент	Значение
!P.MULTI[0]	Полное количество остающихся графиков
!P.MULTI[1]	Число графиков по горизонтали (вдоль оси **X**)
!P.MULTI[2]	Число графиков по вертикали (вдоль оси **Y**)
!P.MULTI[3]	Число графиков вдоль оси **Z**
!P.MULTI[4]	0: вначале слева направо, затем сверху вниз; 1 – обратный порядок: вначале сверху вниз, затем слева направо

Когда **!P.MULTI[0] = 0**, производится стирание страницы.

Например, установка **!P.MULTI = [4, 2, 3, 0, 1]** задаёт 6 графиков на страницу с выводом сначала по столбцам, двух по горизонтали и трёх по вертикали. Также эта установка отключает стирание графического устройства и устанавливает следующий вывод начиная с позиции третьего графика (с левого нижнего угла). Последующая позиция вывода будет в правом верхнем углу, и затем в позиции ниже. Чтобы вернуться к выводу одиночного графика, следует установить просто **!P.MULTI = 0**.

Имеется специфика при выводе графики в файл формата PostScript. Когда число графиков превысит их полное заданное количество, вывод будет производиться на следующую, новую страницу. Например, если **!P.MULTI = 0** и мы введём

```
PLOT, FINDGEN(100)
PLOT, FINDGEN(100)
```

– появится вторая страница.

11.1.1 Графики временных рядов

Если аргумент в ключевом слове **{xyz}tickformat** не начинается с открытый скобки, "(", он интерпретируется как имя функции, генерирующей формат требуемой спецификации. В этом случае функция должна воспринимать 3 аргумента: **axis**, **index**, **value**:

axis – номер оси: 0 для оси X, 1 для оси Y, 2 для оси Z;

index – индекс метки, начинающийся с нуля;

value – величиной маркера метки (вещественное число).

Пример. Часто требуется пометить деления оси времени с форматом HH:MM:SS. Для этого можно использовать следующую функцию, поместив ее либо в начале программного файла, либо в отдельном файле. Имя этого файла также должно быть **timeticks.pro**.

```
      FUNCTION TIMETICKS, axis, index, value

hour = long(value)/3600

minute = long(value-3600 * hour) / 60

sec = value mod 60

RETURN, STRING (hour, minute, sec,  FORMAT="(i2.2, ':', i2.2, ':', i2.2)")

      END
```

Затем следует задать аргументы, например:

```
T = DINDGEN(100)*4 + 3600*5

Y = EXP(((T-3600.*5)/300)^2)*SIN(T/10)
```

И использовать вызов:

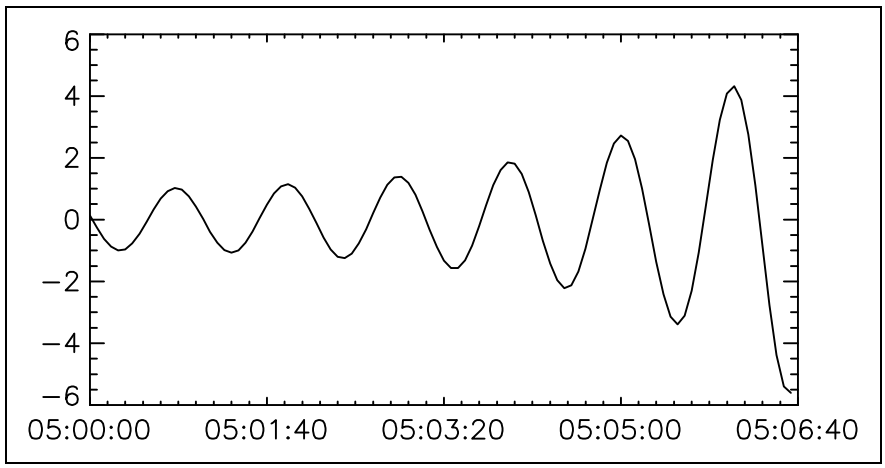
```
PLOT, T, Y, XTICKFORMAT = 'TIMETICKS', xmargin = [10, 8]
```

```
      end
```

Последняя строка содержит **end** – конец программы.

Увеличенное поле слева **xmargin = [10, 8]** задано для того, чтобы крайняя правая надпись на оси X (*05:06:40*) поместилась в пределах области графика.

Результат показан на рисунке.



Обратим внимание читателя на то, что графики временных реализаций (подобно данному примеру) удобно изображать, используя вызов процедуры **PLOT** с двумя аргументами. В этом случае, если даже отсчеты расположены неравномерно по времени, не возникнет проблем. В качестве примера приведен следующий график. На нем изображаемые точки помечены крестиками.

Здесь для получения неравномерной сетки отсчетов на начальном участке графика использованы “прореженные” значения времени:

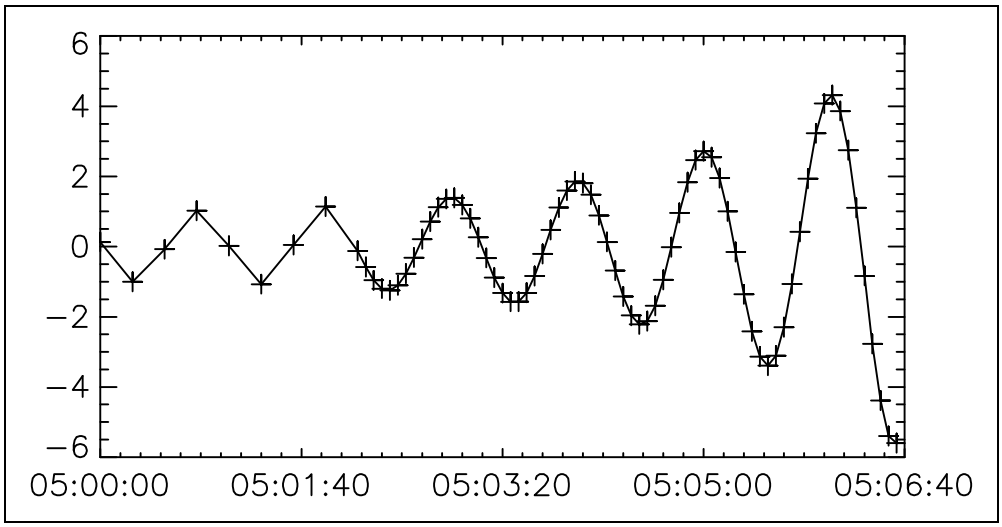
```
T = DINDGEN(100)*4 + 3600*5

T=[T[INDGEN(8)*4], T[32:*]]

Y = EXP(((T-3600.*5)/300)^2)*SIN(T/10)

PLOT, T, Y, XTICKFORMAT = 'TIMETICKS', xmargin = [10, 8], PSYM = –1
```

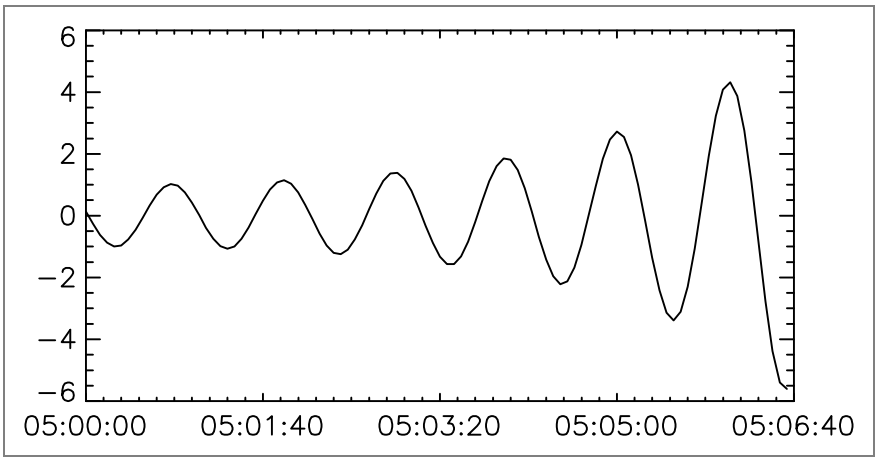
В средней строке из исходного массива для значений времени отобраны 0-е, 4-е, 8-е, 12-е, 16-е, 20-е, 24-е, 28-е, 32-е, а далее – все подряд (33-е, 34-е,… 100-е).



Также остановимся еще на одном специальном примере.

В ряде случаев известны значения не во всех точках, и график должен быть изображен с разрывами. Для этого удобно использовать ключевое слово **MAX_VALUE**. Если присутствует это ключевое слово, данные, превышающие **MAX_VALUE**, обрабатываются как отсутствующие и не изображаются. Таким образом, для изображения графика с разрывами необходимо, во-первых, присвоить массиву данных в тех точках, где они отсутствуют, значения, заведомо превышающие разумные. Во-вторых, при вызове графической процедуры нужно указать соответствующее пороговое значение.

Пример такого графика приведен на следующем рисунке. Для его изображения использован следующий программный фрагмент:




```
T = DINDGEN(100)*4 + 3600*5

Y = EXP(((T-3600.*5)/300)^2)*SIN(T/10)

Y([15, 16, 17, 18, 19, 50, 51, 80]) = 1e6

PLOT, T, Y, XTICKFORMAT = 'TIMETICKS', $
xmargin = [10, 8], max_value = 100
```

В третьей строчке массиву **Y** в **15, 16, 17, 18, 19, 50, 51, 80** точках присвоено значение в один миллион. Установленный порог составляет 100.

11.2 Анализ одномерных массивов данных

- Функция **TOTAL(Y)** возвращает скалярную величину, равную *сумме* всех элементов массива **Y**.
- Функция **MEAN(Y)** возвращает скалярную величину, равную *среднему арифметическому* всех элементов массива **Y**.
- Функция **STDDEV(Y)** возвращает скалярную величину, равную *среднеквадратичному отклонению* элементов массива **Y.**
- *Экстремумы.* Функции **MIN(Y)** и **MAX(Y)** возвращают скалярные значения, равные, соответственно, минимальному и максимальному значениям массива **Y**. Эти функ-

ции имеют дополнительные удобные возможности. Если нам требуются и минимум, и максимум, то, вместо выполнения повторного сканирования массива, мы можем ввести

```
AMIN = MIN(Y, IMIN, MAX = AMAX)
```

Переменная **AMIN** будет содержать минимальное значение, **AMAX** – максимальное, а **IMIN** – номер элемента массива, в котором достигается минимальное значение **AMIN**.

- *Средневзвешенный центр.* Иногда предпочтительнее вычислить *средневзвешенный центр*, аналогичный центру тяжести, нежели положение максимума. Такая необходимость возникает, например, при анализе несимметричных кривых; в случаях, когда требуется нецелое значение центра и т.д. Средневзвешенный центр определяется выражением

$x_{wgt} = \frac{\sum_i x_i y_i}{\sum_i y_i}$. В IDL это выражение вычисляется таким образом:

```
Xwgt = total(findgen(n_elements(Y))*Y)/total(Y)
```

- *Производная.* Можно вычислить производную кривой **X** с помощью функции **DERIV(X)**. Однако если анализируемая кривая зашумлена, производная будет содержать множество знакопеременных коротких пиков большой величины и будет неудовлетворительной. В этом и подобных случаях требуется фильтрация сигнала.

- *Определённое интегрирование* численных массивов. Определённый интеграл от одномерного массива **X** вычисляется с помощью функции **TOTAL**. Разумеется, при этом необходимо учитывать шаг интегрирования. Ниже приведён пример вычисления определённого интеграла $\int_0^{\pi} \sin x dx$:

```
N = 200
x = findgen(N)/(N-1)*!pi*2          ; аргумент
y = sin(x)
plot, x, y                          ; визуализация для контроля
print, total(y[0:99])*mean(deriv(x))     ; mean(deriv(x)) – стандартный
                                         ; способ вычисления среднего
                                         ; интервала между отсчётами
```

IDL выдаст: **1.99996**

Как известно, точное значение этого интеграла равно 2. Увеличивая **N**, мы будем приближаться к этому значению (попутно заметим, что число π с двойной точностью – **!DPI**).

- В IDL версий 5.3 и выше неопределённое интегрирование численного массива **X** выполняется следующим образом:

```
TOTAL(X, /CUMULATIVE)
```

В более ранних версиях **IDL** нет аналога этой функции.

11.3 Фильтрация

Ниже рассмотрены три примера простейших фильтров. Они могут использоваться и в одномерном, и в двумерном случаях.

1. Скользящее усреднение, или *сглаживание*, с шириной m получается вычислением каждого элемента нового массива как среднего по m соседним элементам:

$a_k = \frac{1}{m} \sum_{j=0}^{j=m-1} a_{k+j-\frac{m}{2}}$ в одномерном случае.

Эта фильтрация производится функцией **SMOOTH**:

```
YSM = SMOOTH(Y, N_POINTS)
```

Обязательный параметр **N_POINTS** указывает число соседних элементов массива, используемых для усреднения (ширина = m = **N_POINTS**). Существуют следующие очевидные ограничения: **N_POINTS** должно быть больше двух, а оба края массива шириной **N_POINTS/2** не сглаживаются. Однако, если установлено ключевое слово **EDGE_TRUNCATE**, то края массива также обрабатываются. При этом отсутствующие элементы за краями массива заменяются на элементы вблизи краёв, взятые с требуемым числом повторений.

2. Медианная (серединная) фильтрация

```
YMED = MEDIAN(Y, N_POINTS)
```

работает в два этапа. На первом этапе **N_POINTS** соседних элементов массива сортируются в возрастающем порядке. На следующем шаге выбирается элемент, расположенный посередине отсортированной порции массива. Например, серединная величина для массива из трёх элементов [1, 3, 100] равна 3. Этот тип фильтрации удобен для подавления отдельных точек массива с очень высокими или низкими значениями. Примером таких дефектов являются яркие компактные белые и чёрные точки, присутствующие в некоторых изображениях.

Если функция **MEDIAN** вызвана с единственным аргументом – массивом, то возвращается скалярная серединная величина, вычисленная по всему массиву.

3. Фурье-фильтрацию можно выполнить с помощью функции быстрого преобразования Фурье (**FFT**). Ниже приведён пример простейшего Фурье-фильтра. Параметр **N_POINTS** задаёт количество сохраняемых гармоник, а все более высокие зануляются. Например, **FFT_FILTER(Y, 1)** возвращает массив, в спектре которого присутствует только первая гармоника. Чем больше величина **N_POINTS**, тем ближе отфильтрованный массив к исходному. Напротив, если установлено и не равно нулю ключевое слово **HIGH_PASS**, в спектре возвращаемого массива сохраняются лишь высшие гармоники. Однако параметр **N_POINTS** должен быть меньше половины количества элементов в массиве, и для того, чтобы избежать останова по ошибке, в функции вводится внутренний параметр **N_p**, удовлетворяющий этому ограничению.

```
      function fft_filter, x, N_points, high_pass = high_pass

ft = fft(x, -1, /double)
N = n_elements(x)
N_p = N_points < (N/2-1)

      if keyword_set (high_pass) then begin
```

```
ft(0:N_p) = 0
ft(N-N_p-1:*) = 0
      endif else ft(N_p:N-N_p-1) = 0

return, float(fft(ft, 1))

      end
```

11.4 Аппроксимация одномерных массивов аналитически заданными функциями

Нередко, анализируя экспериментальную кривую, мы интерпретируем её как проявление некоторого процесса, который мы можем описать количественно. В таком случае, если мы найдём удовлетворительную аппроксимацию экспериментальных точек ожидаемой функцией, то параметры аппроксимированной кривой дадут нам требуемую информацию об исследуемом процессе. Можно попытаться найти такую удовлетворительную аппроксимацию вручную, варьируя параметры аналитической функции и проводя график полученной кривой на рисунке, на котором изображены экспериментальные точки.[11] Однако более эффективный и точный способ – использовать методы статистики. Функция IDL **CURVEFIT** решает эту задачу с помощью *метода наименьших квадратов*. По умолчанию, если мы не задаём свою функцию для аппроксимации, для аппроксимации используется комбинация полинома второй степени с гауссианой $A_1 + A_2 x + A_3 x^2 + A_4 e^{-A_5 (x - x_0)^2}$. С помощью такой функции можно аппроксимировать многие виды экспериментальных кривых. Однако множество кривых невозможно удовлетворительно аппроксимировать этой зависимостью, и в большинстве случаев мы сами задаём аппроксимирующие функции. Чтобы это сделать, мы должны написать процедуру IDL, которая принимает независимый аргумент и выдает соответствующие массивы функции и её частных производных по искомым параметрам.

Рассмотрим пример аппроксимации экспериментальных данных полиномом второй степени $y = ax^2 + bx + c$. Частные производные по параметрам равны:

$$\frac{\partial Y}{\partial a} = X^2, \quad \frac{\partial Y}{\partial b} = X, \quad \frac{\partial Y}{\partial c} = 1$$

Создаём программный файл, называемый, например, “**fit1.pro**”:

```
pro fit1, X, params, F, Pder
N = n_elements(X)
Pder = fltarr (N, 3)

a = params(0)
b = params(1)
c = params(2)

F = a*X^2 + b*X + C

Pder(*, 0) = X^2
Pder(*, 1) = X
Pder(*, 2) = 1
```

[11] Заметим, что такой путь при работе в среде IDL оказывается заманчиво и даже провокационно лёгким, однако лучше всё же прибегнуть к услугам математической статистики для получения именно наилучшей аппроксимации и – попутно – оценки качества этой аппроксимации.

```
END
```

Затем пишем простой программный файл, в котором производится вызов функции **CURVEFIT** (или же эта часть программы может находиться в том же самом программном файле **fit1.pro**):

```
N = 200
x = findgen(N)/(N-1)*!pi
y = sin(x)
weights = replicate(1, N)
PARAMS = [1., 1., 1.]
Result = CURVEFIT(X, Y, Weights, PARAMS, Sigma, funct = 'fit1')
END
```

Вектор **PARAMS** должен содержать начальное приближение. Качество этого начального приближения определяет количество итераций. При производительности современных компьютеров мы не встречались с ситуациями, в которых качество начального приближения было бы существенным. При любом начальном приближении мы получим один и тот же результат, но лишь если итерационный процесс сходится, то есть если экспериментальный набор данных и аппроксимирующая кривая на самом деле схожи. Например, если мы попытаемся аппроксимировать точки кривой $\exp(-x)$ функцией Ax^2, то нам в этом случае вряд ли удастся найти параметр *A*. Скорее всего, функция **CURVEFIT** выдаст сообщение:

```
Failed to converge
```

Если набор данных, которые мы хотим аппроксимировать, имеет широкий диапазон, может оказаться полезным работать не прямо с экспериментальными точками, а с их логарифмами.

11.5 Полиномиальная аппроксимация

Нередко встречается ситуация, когда экспериментальный набор данных может быть аппроксимирован некоторым полиномом. Как раз этот случай рассматривался в предыдущем примере. Наиболее общий случай:

$$y = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0$$

Полиномиальная аппроксимация выполняется функцией IDL **POLY_FIT**:

```
Result = POLY_FIT(X, Y, N, Yfit)
```

Здесь **N** – степень полинома, который мы хотим использовать для аппроксимации. **Yfit** – вектор значений аппроксимирующего полинома, соответствующих входному вектору заданных точек **X**.

11.6 Регрессия

Специальная, но достаточно часто встречающаяся ситуация, когда ожидается, что набор экспериментальных точек представляет собой линейную комбинацию значений некоторых других параметров, имеющих то же количество точек:

$y = a_0 + a_1 x_1 + a_2 x_2 + \ldots + a_n x_n$.

Это – уравнение многопараметрической *линейной регрессии*.

Параметры $a_0, a_1, a_2, \ldots a_n$ можно найти с помощью функции IDL **REGRESS**:

```
RESULT = REGRESS(Y, X, Weights, Yfit, a0)
```

В наиболее часто встречающемся случае однопараметрической регрессии следует задать:

```
RESULT = REGRESS(Y, transpose(X), replicate(1, n_elements(Y)), Yfit, a0)
```

Заметим, что **RESULT** – это вектор, даже в однопараметрическом случае. Поэтому, если мы хотим использовать его значение в каком-либо выражении, его следует преобразовать в скаляр (в однопараметрическом случае!):

```
Yfit1 = a0 + RESULT[0]*X
```

11.7 Интерполяция

Наиболее часто используются *линейная* (*билинейная* в двумерном или *трилинейная* в трёхмерном случае) и сплайн-интерполяция.

11.7.1 Интерполяция типа линейной

Интерполяция линейного типа выполняется с помощью функции **INTERPOLATE.** Чтобы получить интерполированные значения, необходимо задать положения (то есть номера в массиве), для которых должны быть получены эти значения. Номера новых элементов могут быть нецелыми. Эта задача может оказаться нетривиальной, например, если требуется интерполяция на нерегулярную сетку. В этом случае помогает другая функция – **INTERPOL**. Однако она может работать только с одномерными массивами.

11.7.2 Сплайн-интерполяция

Функция **SPLINE** выполняет интерполяцию кубическими сплайнами:

```
SPL_INT = SPLINE(X, Y, T)
```

Здесь **X** – вектор абсцисс (*значения должны быть монотонно возрастающими!*), **Y** – вектор значений ординат, соответствующих **X**, **T** – вектор значений абсцисс, для которых требуется получить интерполированные значения. Значения **T** также *должны быть монотонно возрастающими*. Сплайн-интерполяция весьма удобна, поскольку она обеспечивает непрерывность значений интерполированной функции и её производных. Однако такая интерполяция чревата артефактами, если экспериментальные точки имеют разброс.

11.7.3 Вычисление сдвига между двумя кривыми

Задача может быть сформулирована следующим образом. Имеются две относительно схожих кривых, сдвинутых по горизонтальной оси одна относительно другой. Требуется найти величину этого сдвига. Эта задача решается с помощью функции *кросс-корреляции*. Ниже приведён пример программного текста, иллюстрирующий применение кросс-корреляционной функции.

```
x = findgen(100)/99
```

```
dy1 = 0.3               ; Центр первой кривой
dy2 = 0.6               ; Центр второй кривой
wy1 = 0.16              ; Ширина первой кривой
wy2 = 0.09              ; Ширина второй кривой

y1 = exp(-(x-dy1)^2/wy1^2)
y2 = exp(-(x-dy2)^2/wy2^2)

!p.multi = [0,1,2]      ; Располагаем два графика в окне по вертикали

plot, y1
oplot, y2, lines = 1

lag = 100*(findgen(40)/40-0.5)
cor = c_correlate(y1, y2, lag, /double)

plot, lag, cor

!p.multi = 0            ; Восстанавливаем установку на один график в окне

amax = max(cor, imax)
print, 'Shift: ', lag(imax)

end
```

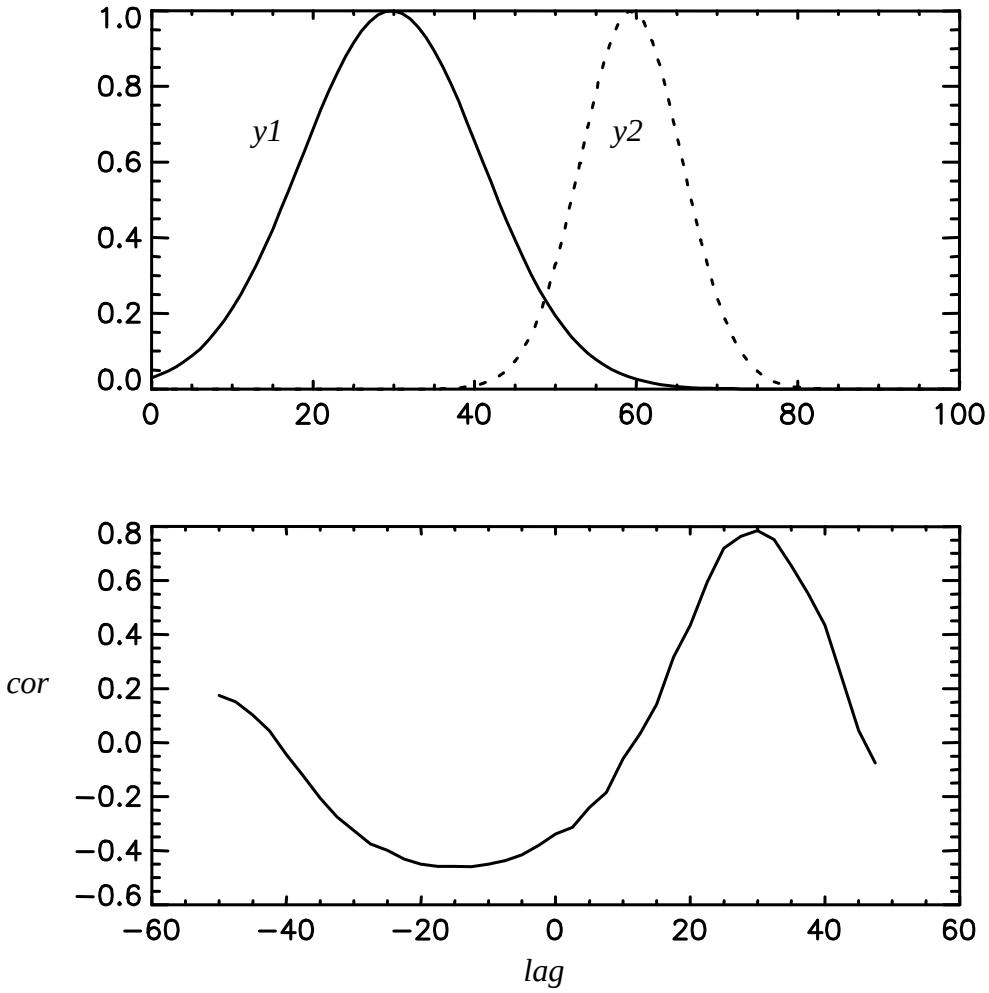


Рис. 4. Кросс-корреляционная функция

Это задача имеет множество приложений: иногда требуется измерить задержку между двумя сигналами; зная величину сдвига, можно совместить две кривых и т.д.

11.8 Поиск дефектных записей

Наличие дефектных записей (кадров) может быть установлено в процессе просмотра данных. Однако при большом их количестве выделение дефектных записей может оказаться непростой задачей.

Общий способ выявления дефектных записей – нахождение некоторого параметра или характеристики данных, которая на дефектной записи испытывает достаточно резкое или просто заметное изменение. Например, это могут быть:

- достижение порогового уровня при насыщении большим сигналом;
- резкое увеличение ширины отклика при перегрузке;
- дрожание или скачки центров ярких источников при погрешностях пространственной привязки;
- большие скачки, приводящие к выходу источника за пределы кадра, отражаются на суммарной величине сигнала по кадру;
- сбои амплитудной калибровки проявятся как импульсы на интегральном профиле потока и т.д.

При любой размерности данных желательно найти такой показатель качества данных – параметр или характеристику, который был бы скалярной величиной (просто числом) p_i для каждой из записей. Тогда временной профиль этого показателя качества на дефектных записях будет иметь выбросы, которые несложно отфильтровать с помощью медианного сглаживания **MEDIAN**(p_i, Width). Превышение разности временного профиля такого показателя с отфильтрованной кривой заданного порога **THRESHOLD** указывает на наличие дефектной записи:

```
BAD_RECORDS = WHERE(p – MEDIAN(p, Width) GT THRESHOLD)
```

Поскольку эта методика применима к массивам любой размерности, мы далее не будем возвращаться к этому вопросу.

Существует ряд других способов обработки и анализа одномерных наборов данных, возможно, известных Вам. Большинство их реализовано в IDL.

12 Двумерные массивы

Вначале – одна часто встречающаяся задача: определить элементы двумерного массива, расположенные внутри заданного многоугольника. Эта задача решается с помощью функции **POLYFILLV**, которая возвращает вектор одномерных индексов элементов массива, удовлетворяющих этому условию.

Result = POLYFILLV(X, Y, Sx, Sy)

12.1 Визуализация двумерных массивов

Для визуализации двумерных массивов используются три основных способа представления: яркостный («телевизионный»), контурный и в виде поверхности. Яркостное (полутоновое) представление показывает массив как изображение подобно телевизору, отображая значения пикселов (элементов массива) соответствующей яркостью или цветом. Этот способ представления наиболее удобен и информативен для быстрого просмотра массива с целью получения о нём представления на качественном уровне. Он также удобен для указания координат пикселов (элементов массива) для их различного анализа. Для изображения массива на экране «пиксел в пиксел» имеется две процедуры IDL – **TV** и **TVSCL**:

```
TV, ARRAY
```

```
TVSCL, ARRAY
```

Процедура **TV** изображает двумерный массив на графическом устройстве (например, в графическом окне на экране монитора) с цветами и яркостями точек в соответствии с их величинами и текущей цветовой таблицей. Если мы работаем с линейным серым клином, то цветовой индекс 0 соответствует чёрному, а 255 – белому (в простейшем случае). Элементы, значения которых превосходят 255, будут иметь яркость (цвет) соответственно выражению **VAL mod 256**, так что те элементы, значение которых составляет 256, будут снова чёрными. Если мы хотим отобразить массив с помощью процедуры **TV**, масштабируя значения массива от чёрного до белого (минимальная величина соответствует чёрному, максимальная – белому), следует ввести:

```
TV, BYTSCL(ARRAY, TOP = !d.n_colors - 1)
```

То же самое выполняется процедурой **TVSCL**, то есть визуализация массива в «телевизионном» представлении с масштабированием яркости.

Также нами разработана процедура **TVCON**[*] (**TVscl + CONgrid**). Эта процедура масштабирует размеры массива, подстраивая его под область графика на графическом устройстве, и изображает координатные оси вокруг изображения. Она подобна стандартной процедуре **IMAGE_CONT** (но без контуров) и позволяет применять различные ключевые слова для модификации вывода. Вызов это процедуры, её аргументы и ключевые слова подобны использованию их в процедуре **CONTOUR**.

Заметим, что иногда могут возникнуть проблемы, если массивы имеют паразитные вырожденные измерения (например, [1, M, N] или [M, 1, N]). В таком случае предпочтительнее удалить паразитные измерения перед визуализацией с помощью функции **REFORM**, вызванной с единственным аргументом:

```
TVCON, REFORM(ARRAY)
```

Контурное представление двумерных массивов позволяет просматривать массив, используя линии равного уровня («изолинии»). Этот способ представления показывает количественную информацию и выделяет формы объектов, имеющихся в изображении, но даёт довольно бедную информацию по сравнению с «телевизионным» представлением и скрывает множество деталей. Фактически исследователь решает, какие детали подчеркнуть, а какие оставить скрытыми. Кроме того, «холмы» и «впадины» на контурном представлении выглядят очень схоже. Так что изображения, представленные контурами, иногда могут выглядеть обманчиво, и следует обращать внимание на уровни контуров, их число, и отображение «выпуклостей» и «вогнутостей». Чтобы показать контуры массива ARRAY на графическом устройстве, нужно ввести:

```
CONTOUR, ARRAY
```

Эта процедура имеет множество опций. Можно показать контуры, заполненные разными цветами; выполнить различные типы координатной сетки соответственно заданным аргументам; изобразить контуры разных уровней различными цветами (одинаковыми или разными), толщиной и стилями линий и т.д. В принципе, контурное представление тоже можно использовать для указания координат интересующего элемента, но в координатной системе **DATA**.

Представление в виде ***поверхности*** позволяет показать массив как квазитрёхмерную фигуру, у которой «высоты» соответствуют значениям пикселов (элементов массива). Области пониженной или повышенной величины хорошо видны как впадины и холмы. Имеется возможность показать массив в виде сетчатой или оттенённой поверхности:

```
SURFACE, ARRAY
```

```
SHADE_SURF, ARRAY
```

Понятно, что некоторые детали массива будут затеняться холмами или впадинами, расположенными перед ними. Чтобы рассмотреть последовательно весь массив, можно изменять ориентацию осей X и Z с помощью задаваемых ключевыми словами параметров **AX** и **AZ** (в градусах). Указание какого-либо элемента в массиве в интерактивном режиме с помощью этого представления неудобно.

Используются также комбинации этих трёх основных типов представления двумерных массивов: «телевизионный» + контурный, контуры + цвета, поверхность + цвета (оттенённая поверхность). Однако наиболее гибким, наглядным и удобным для анализа является «телевизионный» способ представления.

12.1.1 «Телевизионное» представление

Процедуру **TV** (**TVSCL**) можно вызвать с одним, двумя или тремя аргументами. Первый аргумент – это всегда массив, который требуется вывести на экран или другое графическое устройство.

Если указан только *один* аргумент, массив изображается начиная с левого нижнего угла графического устройства:

```
TVSCL, ARRAY
```

Если указаны *три* аргумента, то второй и третий трактуются как координаты **X** и **Y** левого нижнего угла области, в которую будет помещён изображаемый массив:

```
TVSCL, ARRAY, X0, Y0
```

Если указаны *два* аргумента, то второй аргумент определяет номер позиции, в которую будет помещён массив, начиная с верхнего левого угла слева направо и сверху вниз (Рис. 2). Результат, показанный на рисунке, получен следующим образом:

```
FOR j = 0,5 DO TVSCL, SHIFT(DIST(200), 100, 100), j
```

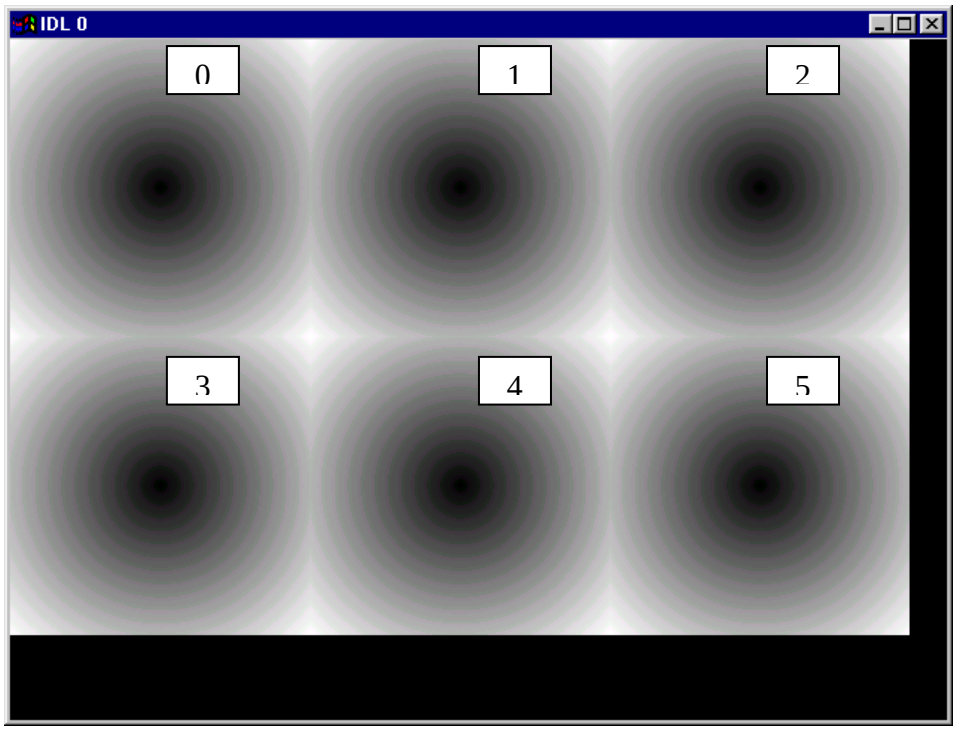


Рис. 2. Позиции, в которые помещается отображаемый массив при вызове процедуры **TV** (**TVSCL**) с двумя аргументами. Номера в рамках указывают соответствующие позиции.

Довольно просто просмотреть изображение, выведенное в «телевизионном» режиме. Однако эта задача становится нетривиальной, если мы хотим увидеть слабоконтрастные детали в изображении, имеющем широкий диапазон значений (например, если этот диапазон простирается от нуля до нескольких миллионов, а нам требуется увидеть детали величиной в несколько тысяч единиц). В ряде случаев можно использовать ограничение пороговых уровней с помощью операторов минимума и максимума:

```
TVSCL, ARRAY > THRES1 < THRES2
```

Пороговый уровень **THRES1** должен быть меньше, чем уровень **THRES2**.

Шумы типа “соль и перец” (яркие белые и чёрные точки), присутствующие в некоторых изображениях, могут существенно понизить контраст, если эти локальные дефекты имеют очень высокие и низкие значения. Существенно улучшить контраст в этом случае может *медианная фильтрация*, однако при этом пострадает пространственное разрешение.

Другой мощный, но довольно грубый способ состоит в *выравнивании гистограммы* распределения яркостей. Это нелинейное преобразование изображение делает амплитудное распределение яркости по изображению ровным. Сущность этого способа состоит в удалении значений, имеющих низкую статистическую значимость или вообще отсутствующих в изображении. Динамический диапазон изображений, обработанных этим способом, резко сжимается. В результате контраст значительно уменьшается, и очень яркие и

очень слабые детали в изображении становятся видимыми одновременно. Таким образом, эта методика облегчает выявление деталей минимального контраста:

```
TVSCL, HIST_EQUAL(ARRAY)
```

Более мягкими способами являются использование логарифмического или степенного масштабирования массива:

```
TVSCL, ALOG10(ARRAY – THRES > 1)
```

```
TVSCL, (ARRAY – THRES > 0)^POWER
```

Значение **THRES** можно выбрать равным, например, минимальному значению в массиве.

Если требуется представить биполярное изображение (например, магнитограмму), можно использовать этот же способ представления, применяя степенную функцию к *модулю* значений в массиве, то есть применяя её к положительным и отрицательным значениям раздельно:

```
TVSCL, (ARRAY > 0.)^POWER -(-ARRAY > 0.)^POWER
```

Это делает функция **MAGPOWER**[*]:

```
TVSCL, MAGPOWER(ARRAY, POWER)
```

12.1.2 « упа»

Процедура **ZOOM** показывает в увеличенном масштабе в новом графическом окне часть изображения, выведенного в текущее графическое окно. Для указания центра области, показываемой в увеличенном виде, используется курсор. Чтобы вызвать процедуру **ZOOM**, следует просто ввести

```
ZOOM
```

После вызова процедуры **ZOOM** поместите указатель мыши (курсор) на изображение в графическом окне IDL. Для вывода на экран увеличенной версии изображения в новом графическом окне нажмите на левую кнопку мыши. Центр увеличенного изображения находится около пиксела, выбранного в исходном окне. Нажатие правой кнопки приводит к выходу из процедуры.

12.1.3 Выделение кра в

Этот эффект даёт операция дифференцирования. В IDL имеется две функции, выполняющие подчёркивание краёв в изображении. Функция **ROBERTS** выполняет аппроксимацию оператора Робертса для выделения краёв изображений, представляющего простой метод двумерного дифференцирования для подчёркивания и выделения краёв:

$$R_{j,k} = \left|F_{j,k} - F_{j+1,k+1}\right| + \left|F_{j,k+1} - F_{j+1,k}\right|.$$

Другая возможность – использование функции **SOBEL**. Ниже приведены примеры использования этих функций применительно к изображению квадрата:

```
X = FLTARR(150, 150)
X[30:70, 50:90] = 1
TVSCL, X, 0
TVSCL, ROBERTS(X), 1
TVSCL, SOBEL(X), 2
```

12.2 Использование географических проекций и полярных координат

При работе с солнечными или геофизическими данными часто возникает необходимость использования полярных или иных непрямоугольных координатных систем. В данном параграфе кратко рассматриваются эти вопросы.

12.2.1 Работа с географическими проекциями

IDL поддерживает множество географических проекций: азимутальную, коническую, цилиндрическую, меркаторскую, синусоидальную, стереографическую и т.д. Мы, однако, использовали почти исключительно ортографическую проекцию, поскольку именно она соответствует изображению удалённого сферического небесного тела, в нашем случае Солнца. Поэтому мы ограничимся рассмотрением этого примера.

Для установки географической проекции в IDL имеется процедура **MAP_SET**. Выбор типа проекции осуществляется соответствующим ключевым словом. IDL при работе с географическими проекциями использует системную переменную !MAP. С её помощью осуществляются корректные преобразования координат между тремя системами, причём в системе DATA первая координата – это долгота, вторая – широта. Когда географическая проекция установлена, с долготой и широтой работают и прочие процедуры, например, **CONVERT_COORD**, **CURSORPOS** и т.д.

С переходом к IDL версии 5 процедура **MAP_SET** была радикально переработана, было добавлено множество опций, однако работа с ортографической проекцией при произвольном центре и радиусе сферы не предусматривалась. Дэвид Стерн подсказал нам способ управления этими параметрами проекции. В этом способе с помощью полей системных переменных **!X.S** и **!Y.S** напрямую задаётся преобразование между нормальными координатами и координатами в системе данных. Координатная сетка выводится уже после этого. Задание обрезания области вывода графика **!P.CLIP** в конкретных случаях может и не потребоваться, однако лучше обезопаситься, задав его, поскольку в противном случае на экране может вообще ничего не появиться.

В приведённом ниже примере заданы координаты центра [200, 250], радиус 197, широта **B0** и долгота **L0** центра диска и позиционный угол **P0** могут быть заданы в широких пределах. Предполагается, что сетка накладывается на уже выведенное изображение, поэтому установлен ключ **/NOERASE**.

```
Ns = 512

window, 0, xs = Ns, ys = Ns

B0 = 15.
L0 = 20.
P0 = 10.

Center = [200., 250.]
Radius = 197.

map_set, B0, L0, P0, /ortho, /noerase, pos=[0,0,1,1], /noborder

!x.s=[Center[0], Radius] / float(!d.x_size)
!y.s=[Center[1], Radius] / float(!d.y_size)

P_clip_save=!P.clip
!p.clip = [0, 0, !d.x_size, !d.y_size]
```

```
MAP_GRID, latdel=10, londel=10, /label

!P.clip=P_clip_save

end
```

Для того, чтобы отобразить в некоторой географической проекции массив, заданный в координатах долгот и широт, используется функция MAP_IMAGE.

12.2.2 POLAR_CONTOUR

Нередко двумерные массивы геофизических данных задаются в виде наборов чисел, соответствующих немногим значениям радиус-вектора и угла. В этом случае для изображения такого массива полезна процедура **POLAR_CONTOUR**. Ниже приведён пример использования этой процедура из справочной системы IDL.

```
nr = 12     ; Число значений радиуса.
nt = 18     ; Число значений угла.

r = FINDGEN(nr)/(nr-1) ; Создаётся вектор значений радиуса.

theta = 2*!PI * FINDGEN(nt)/(nt-1) ; Создаётся вектор значений угла.
z = COS(theta*3) # (r-0.5)^2       ;Задаются значения данных.

POLAR_CONTOUR, z, theta, r, /FILL, nlevels = 15
```

12.3 Анализ двумерных массивов

Примерами двумерных массивов являются многие типы солнечных изображения или карты Солнца, динамические радиоспектры, последовательности одномерных пространственных профилей и т.д.

12.3.1 сновные ункции

TOTAL(Y, 1) вычисляет вектор сумм по первому измерению массива **Y** (одномерный вертикальный “скан”).

TOTAL(Y, 2) вычисляет вектор сумм по первому измерению массива **Y** (одномерный горизонтальный “скан”).

TOTAL(Y) вычисляет скалярную величину, равную сумме всех элементов массива **Y**.

Функции **MEAN**, **MIN** и **MAX** работают аналогично одномерному случаю. Если требуется выяснить, в котором элементе достигается максимальная величина, то следует иметь в виду, что определяемый индекс является *одномерным*. Его можно преобразовать в двумерный индекс для двумерного массива размерности (M,N) с помощью следующих выражений:

```
Xmax = Index mod M

Ymax = Index / M
```

12.3.2 олиномиал ная аппроксимация

Функция **SFIT** выполняет полиномиальную аппроксимацию массива **IMAGE** поверхностью и возвращает аппроксимирующий массив **FIT**:

```
FIT = SFIT(IMAGE, DEGREE)
```

Здесь **DEGREE** – максимальная степень аппроксимирующего полинома. Эта аппроксимирующая функция определяется выражением $f(x,y)=\sum kx_{ji}x^i y^j$. Коэффициенты многочлена от *x* и *y*, аппроксимирующего данные, можно получить с помощью ключевого слова **KX**.

Следует иметь в виду, что результат аппроксимации имеет двойную точность. Для экономии памяти можно преобразовать массив **FIT** в вещественный массив одинарной точности.

Функция **SFIT** может работать довольно долго, если размерность массива **IMAGE** велика, и к тому же степень **DEGREE** относительно высока. В таких ситуациях, а особенно при низких степенях аппроксимации, когда детальность не требуется, лучше уменьшить размерность массива перед аппроксимацией и затем перемасштабировать результат к первоначальным измерениям. Например, если размерность массива составляет 512 × 1024, можно всё это выполнить с помощью следующей последовательности операторов:

```
Sz = size(IMAGE)
FIT=REBIN(FLOAT(SFIT(REBIN(IMAGE, Sz[1]/8, Sz[2]/8), DEGREE)),Sz[1],Sz[2])
```

12.3.3 Интерактивный анализ

Процедура **RDPIX** выводит в интерактивном режиме координаты курсора **X** и Y и значение соответствующего пиксела в окне командного вывода IDL. Однако данная процедура работает лишь с координатами в системе **DEVICE**. Мы расширили возможности этой процедуры, обеспечив работу с другими координатными системами и улучшив формат вывода данных. Модифицированная процедура называется **RDPIXG**[*]. Для исследования массива **ARR** следует ввести:

```
RDPIXG, ARR
```

Для выхода из процедуры следует нажать правую кнопку мыши.

Нередко удобно анализировать таким образом увеличенное изображение массива или его фрагмента. Если размеры массива невелики (например, 64 × 64), можно вывести его увеличенное изображение с помощью процедуры **TVCON** и затем анализировать его в координатной системе **DATA**:

```
TVCON, ARR
```

```
RDPIXG, ARR, /DATA
```

Следует иметь в виду, что после вывода массива процедурой **TVCON** с тремя аргументами, устанавливающими произвольные координатные системы по осям X и Y, процедура **RDPIXG** будет давать ложные значения пикселов массива.

К сожалению, в IDL версий 5.4 и 5.5 вывод этих процедур в окно командного вывода IDL не обновляется в процессе выполнения процедуры, поэтому их невозможно использовать. Нам не удалось преодолеть эту проблему.

Не стоит забывать и про функцию **CURSORPOS**, которая хоть и не выводит значений пикселов, однако помогает определить координаты деталей в изображении.

Процедура **PROFILES** выводит в интерактивном режиме график сечения массива (профиль) по строке или колонке в отдельном графическом окне. Процедурой создаётся это новое окно, и положение курсора мыши в исходном окне используется для указания координаты точки массива, через которую проводится сечение. По мере перемещения курсора в первоначальном окне график профиля в новом окне обновляется. По нажатию левой кнопки мыши происходит переключение профиля между строками и колонками. Чтобы исследовать массив **IMAGE**, следует ввести

```
PROFILES, IMAGE
```

По нажатию правой кнопки мыши работа процедуры завершается.

Функция **PROFILE** выделяет в интерактивном режиме профиль массива **IMAGE** и возвращает вещественный вектор, содержащий значения массива вдоль линии сечения, отмеченной пользователем.

Для отметки линии сечения после вызова функции **PROFILE** следует нажать на левую кнопку мыши, чтобы пометить точки начала и конца. Координаты пикселов для отмеченных точек выводятся в окне протокола IDL.

```
PROF = PROFILE(IMAGE)
```

Результат этой функции – **PROF** – содержит вектор, составляющий профиль массива вдоль заданного отрезка. Его можно представить на графике:

```
PLOT, PROF
```

Функция **DEFROI** определяет интересующую область неправильной формы в изображении с использованием экрана монитора и указателя мыши. Результат – вектор одномерных индексов пикселов внутри заданной области. Отмеченная область на изображении выделяется цветом. Области могут иметь не более 1000 вершин.

Предостережение. Функция **DEFROI** работает некорректно в программах с графическим интерфейсом. В этом случае следует использовать «**CW_DEFROI**».

Вызов функции:

```
Result = DEFROI(Sx, Sy)
```

Здесь Sx, Sy – целые значения, определяющие горизонтальный и вертикальный размер изображения в пикселах. После вызова следует нажать на изображении левую кнопку мыши для того, чтобы отметить точки на границе интересующей области. Отмеченные точки последовательно соединяются. Возможна также обводка интересующей области непрерывным перемещением курсора при нажатой левой кнопке мыши. Для замыкания области и завершения работы функции следует нажать правую кнопку.

Например, если нас интересуют среднее, минимальное и максимальное значения пикселов в некоторой области изображения **IMAGE**, можно их найти следующим образом:

```
Sz = size(IMAGE)
window, 0, xsize = Sz[1], ysize = Sz[2]
tvscl, IMAGE
```

```
res = DEFROI(IMAGE, Sz[1], Sz[2])
FRAG = IMAGE[res]
print, mean(FRAG), min(FRAG), max(FRAG)
```

Заметим, что **FRAG** – *одномерный* массив, и его нельзя непосредственно показать на экране. Это можно сделать следующим образом:

```
IMG1 = make_array(size = Sz)
IMG1[res] = FRAG
TVSCL, FRAG
```

12.3.4 Поиск объектов

Функция **LABEL_REGION** находит топологически связанные изолированные области и отмечает каждую из них уникальным индексом («раскрашивая» отдельные области связности). Аргумент функции **LABEL_REGION** – двумерный двухуровневый (монохромный) массив целого типа, в котором существенны лишь нулевые и ненулевые значения. Выходной массив имеет значения от нуля до количества найденных областей. Нулевой индекс указывает, что исходный пиксел был нулевым и не принадлежит никакой из областей. Если переменная **REG** содержит результат функции **LABEL_REGION**, то число найденных областей равно **MAX(REG)**. Чтобы выбрать область номер **J**, следует задать **REG eq J**. Ниже приведён пример, в котором функция **GAUSS1** вычисляет гауссиану, используемую для моделирования источника излучения.

```
      function gauss1, x, width, center
z=((x-center)/width)
return, exp(-(z^2 < 20)*alog(16d0))
      end

; ****************** $MAIN$ *********************

X = (findgen(200)-100)/100

Y =   gauss1(X, 0.3, 0.0) # gauss1(X, 0.3, 0.0) + $
      gauss1(X, 0.2, 0.4) # gauss1(X, 0.2, 0.2) + $
      gauss1(X, 0.3, 0.4) # gauss1(X, 0.2, -0.2)

reg = label_region(Y ge 0.5)

tvscl, reg eq 2
end
```

12.3.5 анные, получаемые на многоканал ны система

12.3.5.1 Выравнивание усиления ме ду каналами

Пример таких данных представляют динамические спектры, выдаваемые радиоспектрографами, снимающими отсчёты с многих каналов. В типичном случае каналы не являются идеально идентичными, и имеется разброс между каналами как по усилению, так и по уровню нуля. Для нахождения значений усиления и уровней нуля по каналам обычно на вход системы подают шумовой сигнал с плоским спектром (например, уровень

неба), и затем обрабатывают выходные сигналы со всех каналов. Рассматривая статистические свойства выходных сигналов, заметим, что *средние значения* выходных сигналов равны нулевым уровням, а *среднеквадратичные отклонения* пропорциональны значениям усиления в каналах. Во избежание повторного сканирования *L*-канального массива можно использовать для вычисления среднеквадратичных отклонений следующее выражение:

$$\sigma_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N} x^2{}_{ij} - \frac{1}{N^2}\left(\sum_{j=1}^{N} x_i\right)^2} \ ,$$

где $i = 1,2,\ldots L$ – номер канала, а $j = 1,2,\ldots N$ – номер отсчёта в массиве данных, содержащем *N* отсчётов[12].

Если в радиоспектре присутствуют помехи, связанные, например, с попаданием в полосу спектрографа каналов телевизионного вещания, то среднее значение и среднеквадратичное отклонение для этих каналов спектрографа будет резко отличаться от соответствующих значений, вычисленных для других каналов. То же самое будет и в случае, если некоторые каналы неисправны. Эти различия можно использовать как критерий отбраковки тех каналов, которым нет веры.

12.3.5.2 Поиск импульсных деталей

Спектрографические записи содержат многие тысячи отсчётов, и поиск исследуемых особенностей в столь длинных записях непрост. Один из возможных путей решения задачи – параллельная запись интегрального потока отдельным радиометром. По существу, можно получить аналогичную запись, суммируя все каналы радиоспектрографа и, таким образом, заменяя массив одного отсчёта с многих каналов одной точкой. Обычно мы в состоянии обрабатывать длинные массивы лишь по частям, последовательно обрабатывая блоки разумной длины. Другой способ – суммирование нескольких точек по каждому каналу (также обрабатывая запись поблочно). Однако это способ не позволяет «поймать» короткие импульсные детали. Эту последнюю задачу можно решить, рассчитывая по каждому каналу максимальное значение по некоторой длине записи (некий аналог скользящего усреднения). В IDL нет специальной функции, выполняющей такую операцию. Можно сделать это таким образом:

```
Sz = size(BLOCK)
maxima = BLOCK [*,*,0]
for j = 1, Sz[3]-1 do maxima = BLOCK [*,*,j] > maxima
```

Задавая длину блока **BLOCK** равной некоторой разумной величине (в зависимости от задачи и длины всего массива), можно таким образом получить набор новых отсчётов, удобный для просмотра.

12.3.6 ентрирование изо ражений Солнца

Центр изображения Солнца с высокой точностью предсказуем для данных различных орбитальных телескопов (Yohkoh, SOHO, TRACE). Однако центрирование оптических данных наземных обсерваторий, а также данных радиогелиографов часто испытывает сдвиги. Мы опишем некоторые методы, используемые для центрирования изображений полного диска Солнца. Если имеется изображение лишь отдельной области, задача становится ещё более сложной.

---

[12] Эта методика выравнивания разработана С.В. Лесовым.

12.3.6.1 Простей ий случай – простей ий метод

Если изображение Солнца содержит ненулевые значение внутри диска и нулевые вне его, задача становится совсем простой. Такая ситуация возможна, если мы имеем дело с изображениями в белом свете. Для того, чтобы величины пикселов вне диска были гарантированно нулевыми, можно обрезать значения пикселов в изображении ниже некоторого порога. Далее удобно воспользоваться одномерными суммами по обеим координатам:

```
SCAN_X = TOTAL(IMAGE, 2)
SCAN_Y = TOTAL(IMAGE, 1)
```

Затем находим первое и последнее ненулевые значения вдоль горизонтальной (**SCAN_X**) и вертикальной (**SCAN_Y**) осей и берём их среднее значение. Номера ненулевых значений переменной **VARIABLE** можно найти следующим образом:

```
IND = WHERE(VARIABLE NE 0)
```

Номера первого и последнего ненулевых значений равны, соответственно, **IND[0]** и **IND[N_ELEMENTS(IND)-1]**. Дальнейшее решение не составляет труда.

12.3.6.2 Центрирование по краям

Солнечный диск имеет резкие края. Поэтому, если мы продифференцируем изображение Солнца (с помощью функции **SOBEL**), солнечный лимб выделится. Тогда можно вычислить суммы по горизонтали и вертикали, и после этого несложно найти центр изображения и его радиус. Эта методика используется в солнечной обсерватории Big Bear.

12.3.6.3 Подавление полос

Предыдущий метод не работает, если в изображении имеются яркие горизонтальные или вертикальные линии или полосы. Чтобы подавить их влияние, для грубого центрирования можно использовать первую гармонику Фурье, поскольку она нечувствительна к узким локальным пикам. После этого можно наложить на изображение маску в виде модельного диска, положение которого приближённо определено, несколько большего радиуса, чем солнечный радиус, и после этого применить более точный мтод центрирования.

12.3.6.4 Отра ение и вычитание

Если отразить изображение солнечного диска по вертикали и вычесть отражённое изображение из исходного, то в результате вычитания ошибки вертикальной центровки проявятся в виде тёмных – с одной стороны – и светлых – с другой – дуг. Чем больше сдвиг центра диска относительно центра кадра, тем больше сумма абсолютных величин этого разностного изображения. Точно таким же способом можно найти центр изображения по горизонтали. Для отражения изображений по горизонтали можно использовать функцию **ROTATE** с параметром 5 и с параметром 7 для отражения по вертикали. Ниже приведен пример программы для горизонтального центрирования этим методом.

```
N = 20                          ; Количество шагов

xval = fltarr(N)                ; Массив для результатов
arg = (findgen(N)-(N-1)/2.)     ; Аргумент: значения сдвига

      for j = 0, N-1 do begin
      im = abs(shift(IMAGE, arg(j)) - rotate(shift(IMAGE, arg(j)), 5))
```

```
xval(j) = total(im)

amin = min(xval, imin)
xoptimal = arg(imin)
endfor
```

При практическом использовании этого метода для повышения точности целесообразно сделать по крайней мере два повтора процедуры, поскольку при первом проходе, например, по горизонтали, неточность вертикальной центровки может значительно исказить результат. Кроме того, с помощью функции **ROT** можно получить более точный сдвиг, чем с помощью функции **SHIFT**, что позволяет, используя функцию **ROT** на втором проходе, центрировать изображение с точностью до долей пиксела. Наконец, если грубая центровка изображения выполнена, то для более точной центровки можно наложить на изображение маску, исключающую вклад деталей вдали от края диска, а сами края можно усилить с помощью функции **SOBEL** или **ROBERTS**.

12.3.6.5 Центрирование с помощью кросс-корреляционной функции

Если в изображении хорошо просматривается солнечный диск, но непросто выделить область вне диска (как в простейшем методе), можно использовать кросс-корреляцию с модельным диском. Этот метод используется при центрировании изображений Солнца, получаемых на радиогелиографе Нобеяма в Японии. Японские коллеги описывают эту процедуру следующим образом: «Вычисляется коэффициент корреляции между центрируемым изображением и модельным диском, сдвигая модель шаг за шагом. Когда коэффициент корреляции принимает максимальное значение, модельный диск оказывается в точном положении, которое занимает наблюдаемое Солнце».[13] Эта методика реализована в процедуре **CORREL_OPTIMIZE** из астрономической библиотеки IDL Astronomy User's Library[14].

12.3.7 али ровка по интенсивности

Для абсолютных калибровок по интенсивности используются опорные источники известной интенсивности. Однако для многих задач таких источников не существует. В таких случаях можно применять «самокалибровки», используя свойства самих экспериментальных данных. Например, коллеги из обсерватории Нобеяма для калиюровки используют солнечный диск, присутствующий в изображениях, и принимают значение яркостных температур для неба и спокойного солнечного диска равными, соответственно, 0 и 10^4 K. Затем они анализируют распределение значений яркости пикселов во всём изображении: «Яркость солнечного диска вычисляется из гистограммы – плотности распределения интенсивности по изображению. На гистограмме имеется два пика. Меньший пик соответствует яркости неба, а более яркий соответствует яркости диска»[13].

Для калибровки данных ССРТ по интенсивности мы используем эту же методику.

12.4 Совместный анализ разных изобра ений

Солнце – очень сложный объект. Наблюдая Солнце на одной-единственной частоте, мы вряд ли сможем понять процессы, происходящие на нём. В то время как тепловое излучение (мягкий рентгеновский диапазон, крайний ультрафиолет) показывают нам изо-

---

[13] Hanaoka Y., Shibasaki K., Nishio M. et al. Processing of the Nobeyama Radioheliograph data. In: The Nobeyama Radioheliograph. Nobeyama Radio Observatory Report No. 357. 1994, p. 35.

[14] http://idlastro.gsfc.nasa.gov/

бражения тонких пространственных структур, нетепловые виды излучения (прежде всего жёсткий рентгеновский диапазон) несут информацию о частицах высоких энергий, ускоренных во время солнечных вспышек. Радиоизлучение чувствительно и к тепловым, и к нетепловым процессам. Работая с радиоизлучением, мы должны помнить о том, что за излучение электронов могут быть ответственны различные механизмы. По этой причине необходимо привлекать данные различных спектральных диапазонов – мягкого и жёсткого рентгеновского излучения, магнитограммы, фильтрограммы Hα и т.д. При этом возникает вопрос, каким образом сопоставить или наложить два или более изображения для идентификации соответствующих друг другу объектов на этих изображениях.

12.4.1 Сопоставление не ол и рагментов

Простейший способ сопоставления двух или более изображений – показать рядом небольшие фрагменты этих изображений, соответствующих одной и той же области. В этом случае особенно полезны линии координатной сетки и/или какие-либо иные маркеры.

12.4.2 аложение контуры + полутон

Наиболее широко используемый способ состоит в наложении контуров одного изображения поверх другого, представленного в полутоновом или псевдоцветном виде. В IDL нет специальной процедуры, выполняющей такое наложение, поэтому мы предлагаем воспользоваться разработанной нами процедурой **IM_CON_TOG*** (*Image and CONtour TOGether*, т.е. « *зображение и онтуры вместе*»). Для представления контуров изображения **IMAGE2**, наложенных на изображение **IMAGE1** с помощью этой процедуры следует ввести

```
IM_CON_TOG, IMAGE1, IMAGE2
```

Очевидно, размерность массивов **IMAGE1** и **IMAGE2** должна быть одинаковой. Однако данная процедура позволяет выполнить наложение и для массивов с одним и тем же центром и размером пиксела, но разным полем зрения при кратной размерности массивов; то есть, например, если оба изображения соответствуют размеру пиксела 5″, и одно из них имеет размер 256 × 256, а другое 128 × 128. В таком случае процедура выбирает общее, то есть наименьшее, поле зрения. Процедура **IM_CON_TOG** воспринимает ряд ключевых слов, аналогичных процедуре **CONTOUR**. Чтобы наложить на изображение контура ещё одного (или более) массива, следует использовать процедуру **CONTOUR** с ключевым словом /**OVERPLOT**. При этом уровни контуров должны быть явно заданы.

12.4.3 Спосо «мерцания»

В ряде случаев, особенно если требуется заметить сдвиг изображений друг относительно друга или иные их различия, полезен быстрый поочередный вывод обоих изображений на экран. Это можно сделать следующим простым способом:

```
FOR j = 0,50 DO BEGIN & TVSCL,IMAGE1 & TVSCL,IMAGE2 & END
```

В данном случае будет выполнен 51 цикл последовательного вывода. В астрономической библиотеке IDL Astronomy User’s Library имеется процедура **BLINK**, позволяющая сделать такой вывод более быстрым. Для использования этой процедуры нужно вывести один массив в графическое окно номер **N0**, другой – в графическое окно номер **N1** (например, **N0 = 0, N1 = 2**) и затем ввести:

```
BLINK, [N0, N1], PERIOD
```

PERIOD – период вывода в секундах (например, 0.8). Для выхода из режима мерцания достаточно нажать любую клавишу. Эта процедура хорошо работает под UNIX, однако под Microsoft Windows работа процедуры часто завершается сразу же после её вызова. Это связано с особенностью работы функции **GET_KBRD**, используемой в процедуре **BLINK**. Имеющему некоторый опыт пользователю не составит труда переделать эту процедуру так, чтобы она работала устойчиво. Следует только заменить условие в строке

```
WHILE (get_kbrd(0) EQ '') DO BEGIN
```

на

```
WHILE (get_kbrd(0) NE ' ') DO BEGIN
```

Тогда выполнение процедуры будет заканчиваться по нажатию пробела.

12.4.4 ом инация нескол ки цветны изо ражений

Ещё один способ – цветовая комбинация разных изображений. Мы использовали этот способ для того, чтобы показать на одном рисунке несколько (реально - 7) изображений[15]. Сущность этого способа состоит в следующем:

1) формируется комбинированное изображение в графическом окне на экране с использованием различных цветовых таблиц (*без* использования разложения цветов), и затем оно считывается с графического окна в режиме *True Color* (с использованием разложения цветов);

2) для вывода каждого изображения независимо от остальных графическая функция экрана устанавливается в **XOR** (6);

3) перед выводом каждого изображения процедурой **TVSCL** загружается соответствующая цветовая таблица;

4) каждое изображение отображается через маску, пропускающую лишь данное изображение и непрозрачную для всего, не относящегося к нему;

5) наконец, комбинированное изображение считывается с графического окна в режиме *True Color* и направляется в файл PostScript также в режиме *True Color*. Ниже приведён пример вормирования такого изображения и вывода его в файл.

```
device, decomp = 0          ; Переход в 16-разрядный режим 'High Color'

N0 = 400                    ; размер первого изображения
N1 = 100                    ; размер второго изображения

X0 = 50
Y0 = 130                    ; Нижний левый угол вставки

x = bytscl(dist(N0))        ; Первое изображение
z = bytarr(N0, N0)

y = bytscl(dist(N1))        ; Второе изображение

z(X0, Y0) = y               ; Вставка второго изображения в первое

mask_z = bytarr(N0, N0)
```

[15] Grechnev V.V., Nakajima H. 2002, Astrophysical Journal, 566, 539.

```
mask_z(X0:X0+N1, Y0:Y0+N1) = 1     ; Маска для второго изображения

mask_x = 1-mask_z                  ; Маска для первого изображения

window, /free, xs = N0, ys = N0

device, set_gr = 6                 ; Перевод графической функции в XOR

loadct, 1                     ; Загрузка цветовой таблицы для 1го изображения

tvscl, x*mask_x               ; Вывод 1го изображения

loadct,3                      ; Загрузка цветовой таблицы для 2го изображения

tvscl, z*mask_z               ; Вывод 2го изображения

device, set_gr = 3            ; Перевод графической функции обратно в COPY

loadct,0

xx = tvrd(true = 1)           ; Чтение комбинированного изображения в режиме
                              ;     True Color

init_dev = !d.name            ; Сохранение имени первоначального
                              ;     графического устройства

set_plot, 'ps'                ; Переход к выводу в файл PostScript

      device, file = 'colfig1.ps', /col, $
      xs = 16, ys = 16, xof = (21.-16)/2, yof = (29-16.)/2

loadct, 0                     ; Подготовка к выводу в режиме True Color

tvscl, xx, true = 1

      device, /close

set_plot, init_dev            ; Восстановление первоначального графического
                              ;     устройства

      end
```

13 р хмерные массивы данных

Трёхмерные массивы данных – наиболее общий тип данных из тех, с которыми мы имеем дело в настоящее время. Многие наземные и орбитальные телескопы поставляют нам многочисленные наборы двумерных изображений. Например, мы работаем с длинными сериями двумерных изображений при изучении солнечных вспышек или корональных выбросов. В таком случае мы имеем дело с набором двумерных изображений, или с *трёхмерным кубом данных*. Две координаты такого куба данных, *X* и *Y*, описывают пространственное положение, а номер изображения в кубе («слой» или «плоскость»), по существу, является временной координатой *t*. Чтобы выявить существенные источники, бывшие активными в солнечной вспышке, и исследовать особенности из поведения, приходится иметь дело с наборами данных, состоящими из сотен кадров.

Мы уже упоминали два общих способа работы с трёхмерными наборами данных: *понизить размерность* набора данных, а также использовать удобное представление данных.

Пусть у нас имеется некоторое число FITS-файлов одного и того же типа, размерности и т.п. в каталоге **DIR1**. Например, там находится ряд файлов радиокарт полного диска Солнца одного и того же размера 512×512, полученных на Радиогелиографе Нобеяма (NoRH). Типичные имена файлов NoRH – **ifa990625_013404** для изображения полного диска Солнца в общей интенсивности (параметр Стокса I), полученного на частоте 17 ГГц 25 июня 1999 года в 01:34:04. Для такого же изображения в поляризации (параметр Стокса V) имя файла будет точно таким же, но вместо **ifa…** в имени файла будет **ifs…**. Эти файлы соответствуют стандарту FITS. Прежде всего, найдём все эти файлы (предполагаем, что мы работаем в Microsoft Windows):

```
I_FILES = FINDFILE(DIR1 + '\ifa*')

I_FILES = I_FILES[SORT(I_FILES)]
```

Вторая строка упорядочивает файлы соответственно монотонно возрастающему времени. Затем считаем заголовки всех этих файлов с помощью функции **RD_FHEAD***, которую мы разработали для чтения заголовков группы файлов (*заметим, что все заголовки должны иметь одну и ту же длину*):

```
HEADERS = RD_FHEAD(I_FILES)
```

Если нас интересует, например, время наблюдений для каждого из файлов, можно выделить соответствующую запись из заголовков:

```
TIMES = MSXPAR(HEADERS, 'time-obs')
```

То же самое можно сделать и для любой другой записи в заголовках, например, даты наблюдений:

```
DATES = MSXPAR(HEADERS, 'date-obs')
```

Функция **MSXPAR*** – расширение функции **SXPAR** из Astronomy User's Library для случая *многих* заголовков. Теперь можно распечатать массив времён.

Если количество файлов невелико (например, < 20), можно прочитать данные из всех этих файлов в память компьютера с помощью функции **READFITS**:

```
NFILES = N_ELEMENTS(I_FILES)
X = FLTARR(512, 512, NFILES)
FOR J=0, NFILES-1 DO X[*,*,J] = READFITS(I_FILES[J])
```

Для дальнейших манипуляций с данными удобно преобразовать время, выраженное в символьном (текстовом) представлении – в часах, минутах и секундах – во время, выраженное в секундах с двойной точностью. Можно это сделать с помощью функции **ANYTIM** из пакета *SolarSoftware*:

```
DTIMES = ANYTIM(DATES + ' ' + TIMES)
```

Функция **ANYTIM** по умолчанию вычисляет время, прошедшее с 1 января 1979 года, поэтому значения **DTIMES** будут порядка 10^8. Однако для визуализации и анализа данных более удобно иметь дело с временем в пределах от 00:00 to 23:59 – разумеется, если мы работаем в пределах одной календарной даты; здесь мы опускаем дополнительные трудности, возникающие при смене даты. Можно выполнить это преобразование, вычитая число секунд, прошедшее с 1 января 1979 года до начала даты наблюдений:

dtimes1 = dtimes-(long(dtimes[0]/86400)*86400d0)

Здесь 86400 = 60×60×24 – количество секунд в сутках, а количество дней, прошедших с 1 января 1979 года, равно **LONG(DTIMES[0]/86400**).

13.1 Просмотр тр хмерных наборов данных

Теперь можно просмотреть куб данных **X**, расположенный в оперативной памяти. Можно это сделать в режиме *кино* или попытаться показать все кадры в графическом окне на экране (или на другом графическом устройстве). Режим кино особенно удобен, когда рассматривается большое количество изображений.

13.1.1 ежим кино

Создадим новое окно размером, соответствующим размеру изображения:

```
WINDOW, 0, XS = 512, YS = 512
```

и затем последовательно выведем все кадры нашего куба данных (например, используя степенную функцию):

```
FOR J=0, NFILES-1 DO TVSCL, (X[*,*,J]>0)^0.3
```

Однако смена кадров в этом кино может быть слишком быстрой. Можно её замедлить:

```
for j=0, Nfiles-1 do begin & tvscl, (x[*,*,j]>0)^0.3 & wait, 0.5 & end
```

Символ *ampersand* (&) весьма полезен в этом случае. По существу, в строке ввода команд можно ввести короткую программу, которая умещается в *одной* строке. Символ амперсенд позволяет записать несколько операторных строк в единственной строке.

```
FOR J=0,NFILES-1 DO BEGIN&TVSCL,(X[*,*,J]>0)^0.3& $
XYOUTS,0.5,0.05,/NOR,AL=0.5,TIMES[J]&EMPTY&WAIT,0.5&END
```

При работе в MS Windows можно всё это записать в одной строке, без знака доллара – переноса. Однако при работе в LINUX это возможно лишь таким образом, поскольку длина строки в LINUX ограничена размером экрана. Из-за этого ограничения очень удобно использовать сокращения для ключевых слов. Здесь «**NOR**» – сокращение ключевого слова «**NORMAL**». Более короткое сокращение недопустимо, поскольку оно становится неоднозначным («**NOCLIP**» также подходит под такое сокращение). «**AL**» – сокращение ключевого слова «**ALIGNMENT**». Процедура **EMPTY** используется для того, чтобы при-

нудительно вывести содержимое графического буфера немедленно на экран. В противном случае до начала интервала ожидания некоторые буквы могут остаться недорисованными.

Описанный способ позволяет просмотреть все изображения с индивидуальным масштабированием по яркости. Поэтому мы не можем увидеть изменений яркости от одного кадра к другому. Чтобы показать все кадры с сохранением взаимных изменений яркости (то есть с общим масштабированием по яркости), вначале следует вычислить максимум куба данных, а затем использовать для вывода каждого изображения процедуру **TV** вместо **TVSCL**:

```
AMAX = MAX(X)

FOR J=0, NFILES-1 DO TV, (X[*,*,J]>0)/AMAX*(!D.TABLE_SIZE-1), J
```

Заметим, однако, что в ряде случаев нет возможности считывания содержимого всех файлов данных из памяти, поскольку общий объём данных может быть огромен. Например, располагая 200 файлами, содержащими данные спутника *TRACE*, каждый из которых содержит изображение из 1024×1024 двухбайтовых пиксела, мы должны отдавать себе отчёт в том, что полный объём этого набора данных составляет около 420 Мбайт. В таких случаях следует просматривать изображения одно за другим сразу после чтения данных из каждого последующего файла:

```
FOR J=0, NFILES-1 DO BEGIN
Z = READFITS(I_FILES[J], HEADER)
TVSCL, HIST_EQUAL(Z)
XYOUTS, 0.5, 0.05, /NOR, ALI = 0.5, sxpar(header, 'time-obs')
EMPTY
WAIT, 0.5
ENDFOR
```

13.1.2 Вывод ряда изо ражений в одно окно

Вначале следует создать графическое окно, в котором можно разместить все изображения, которые мы хотим просмотреть одновременно (разумеется, обычно в уменьшенном виде). Затем следует изменить размер наших данных для того, чтобы они уместились в это окно, например:

```
Y= REBIN(X, 128, 128, NFILES)
```

Наконец, выводим все изображения в это графическое окно:

```
FOR J=0, NFILES-1 DO TVSCL, (Y[*,*,J]>0)^0.3, J
```

Можно перемасштабирование данных выполнить и в процессе вывода:

```
FOR J=0, NFILES-1 DO TVSCL, REBIN(X[*,*,J]>0, 128, 128)^0.3, J
```

13.1.3 Выстраивание ку а данны

Некоторые типы данных, прежде всего, наборы изображений, полученные на радиоинтерферометрах, могут иметь сдвиги от кадра к кадру. В таком случае необходимо стабилизировать положения источников на изображениях, или выстроить изображения, совместив их друг относительно друга. Кроме того, для сравнения данных с изображениями, полученными на других инструментах необходима точная привязка кадров по абсолютному положению. При работе с радиоданными вторая задача выполняется с помощью привязки к некоторым характерным деталям изображений на кадрах до или после вспышки и сравнением их с изображениями, полученными в тепловом излучении (крайний ультрафиолет, мягкий рентгеновский диапазон и т.д.).

Существует несколько методов выстраивания кубов данных. Для взаимного совмещения кадров можно использовать некоторые реперные источники на изображениях. Такие источники должны быть стабильными. Другой метод предполагает контроль смещений источников относительно их предшествующих положений. Для уменьшения влияния структурных и т.п. изменений можно накладывать на изображения маски. Можно также использовать кросс-корреляцию для нахождения оптимального сдвига между двумя изображениями. Напомним, что это возможно с помощью процедуры **CORREL_OPTIMIZE** из астрономической библиотеки IDL Astronomy User's Library.

Для точного совмещения требуются не целые, а вещественные значения сдвигов. Их можно найти, например, с помощью взвешенных центров источников на одномерных суммах по изображениям. Для уменьшения эффектов структурных изменений можно использовать высокую степень данных (например, четвёртую). Когда величины сдвигов **SHIFTS** найдены, точное выстраивание куба данных **IN_CUBE** можно выполнить с помощью функции **POLY_2D**:

```
OUT_CUBE = IN_CUBE
Sz = size(IN_CUBE)
           FOR j=0, Sz[3]-1 DO BEGIN
p = [-SHIFTS[j,0],0,1,0]
q = [-SHIFTS[j,1],1,0,0]
OUT_CUBE[*,*,j] = poly_2d(IN_CUBE[*,*,j], p, q, cubic=-0.5, missing=0.0)
           ENDFOR
```

Здесь **OUT_CUBE** – выходной (выстроенный) куб данных, а массив **SHIFTS** имеет размерность N×2, где N = Sz[3] – глубина куба (количество изображений).

Качество совмещения можно проверить просмотром изображений в режиме кино, либо с помощью дисперсионной карты, о чём пойдёт речь несколько позже, в параграфе «Дисперсионный метод».

13.2 Анализ тр хмерных массивов данных

IDL не богат стандартными процедурами для анализа трёхмерных массивов данных. Поэтому мы сами разработали некоторые из них. В этом параграфе мы обсудим некоторые методы и процедуры.

13.2.1 Временные про или

Процедура **TIMEPROF*** строит в интерактивном режиме профиль трёхмерного массива по третьему измерению через пиксел, в котором в данный момент помещён курсор мыши. При этом предполагается, что на графическом окне выведен какой-либо из кадров, относящийся к анализируемому массиву (либо один из кадров, либо из сумма и т.п.):

```
TIMEPROF, X
```

Этот профиль выводится в отдельном вновь созданном графическом окне. При этом непрерывно выводятся значения координат **X** и **Y**. Работа процедуры оканчивается по нажатию правой кнопки мыши. При этом графическое окно, в которое выводился профиль, удаляется. Эта процедура подобна стандартной процедуре **PROFILES**, однако относится к третьему измерению куба данных.

13.2.2 Интеграл ные аракте ристики

Для анализа трёхмерных массивов можно использовать некоторые характеристики, которые фактически уменьшают количество измерений в массиве. Очевидные примеры – усреднённое по всем кадрам изображение (размерность уменьшена на единицу) или интегральный поток (размерность уменьшена на два). Мы обсудим некоторые из них ниже.

13.2.2.1 средн нные изобра ения

Построить усреднённое по всем **N** кадрам изображение можно с помощью функции **TOTAL**:

```
AVE_IMAGE = TOTAL(X, 3)/N
```

Если яркость источников в наборе изображений испытывала значительные изменения, то основной вклад в построенное таким образом усреднённое изображение дадут кадры с максимальной яркостью. Если же нас интересует усреднённое изображение, в котором вклад каждого кадра был бы с одинаковым весом, то можно построить его следующим образом:

```
Sz = SIZE(X)

AVE_IMAGE = fltarr(Sz[1], Sz[2])
for j = 0, Sz[3]-1 do AVE_IMAGE = AVE_IMAGE + X[*,*,j]/max(X[*,*,j])/Sz[3]
```

Последний делитель **Sz[3]** обеспечивает нормировку усреднённого изображения на единицу и не является обязательным.

Если нас интересует построенное аналогичным образом нормированное усреднённое изображение по биполярным кадрам (изображения в поляризации, магнитограммы и т.п.), можно построить его так:

```
Sz = SIZE(X)

AVE_IMAGE = fltarr(Sz[1], Sz[2])
for j=0,Sz[3]-1 do AVE_IMAGE= AVE_IMAGE +X[*,*,j]/max(ABS(X[*,*,j]))/Sz[3]
```

13.2.2.2 Максимальная яркость, интегральный поток и ффективная площадь

Изменчивость излучающего источника можно характеризовать *макисмальной яркостью* по каждому кадру:

```
SZ = SIZE(X)
BMAX = FLTARR(SZ[3])
FOR J = 0, SZ[3]-1 DO BMAX = MAX(X[*,*,J])
```

Хорошо известный *интегральный поток* учитывает также пространственные изменения, происходившие в анализируемом интервале. Функция **TOTAL_FLUX*** вычисляет суммы по каждому кадру трёхмерного массива. По существу, она выполняет операцию **TOTAL(TOTAL(X, 1), 1)**. Если заданы оба параметра (ключевых слова) **PIXEL_SIZE** и **FREQUENCY** (в *угловых секундах и ц*), результат возвращается в солнечных единицах потока (*1 с.е.п. = 10^{-22} Вт/м^2/ ц*):

$$S_{[с.е.п.]} = 2k\left(\frac{\nu}{c}\right)^2 \sum_i T_{Bi}\Delta\Omega = 7{,}22\cdot 10^{-11}\nu^2_{[ГГц]}\rho^2_{[угл.сек.]}\sum_i T_{Bi[K]},$$

где k – постоянная Больцмана, c – скорость света, T_{Bi} – яркостная температура пиксела (с линейным размером ρ), ν – частота, и $\Delta\Omega$ – телесный угол, соответствующий одному пикселу изображения. Пример:

```
TF = TOTAL_FLUX(Array, PIXEL_SIZE = 4.911, FREQUENCY = 17)
```

Если параметры **PIXEL_SIZE** или **FREQUENCY** не заданы, то коэффициент при сумме принимается равным единице.

Однако эти интегральные характеристики не учитывают возможных структурных изменений. Дополнительные особенности можно выявить с помощью *эффективной площади* излучающей области S_{eff}. Она пропорциональна интегральному потоку, делённому на максимальную яркостную температуру. Эта характеристика математически подобна средней яркостной температуре:

$$\overline{T}_B = \frac{\sum T_{B_i}\Delta S_i}{S},\ S_{eff} = \frac{\sum T_{B_i}\Delta S_i}{T_{B_{\max}}},$$

где ΔS_i – площадь i-го пиксела изображения, а S – площадь всего кадра. Смысл эффективной площади ясен из рассмотрения двух предельных примеров: равномерно освещённого кадра и единственной яркой точки на кадре.

13.2.2.3 Координаты

Функция **PEAK_COORD*** вычисляет координаты (**X,Y**) максимумов для каждого кадра двух- или трёхмерного массива. Эти координаты – всегда целые величины. В качестве альтернативы для получения вещественных значений можно использовать взвешенный центр.

Если излучающий источник наблюдается на солнечном лимбе, его координаты можно соотнести с высотой над фотосферой.

13.2.3 Разностные изображения

Ещё один способ выявления интересующих особенностей – анализировать изображение, соответствующее некоторому характерному моменту события (например, в максимальной фазе), совместно с изображением, полученным до или после данного события. Вычитая предвспышечное или послевспышечное изображение, можно выявить те места, которые становились яркими во время вспышки. Однако, если в анализируемом событии действовали импульсные источники с длительностью меньше интервала между вычитаемыми изображениями, они могут быть пропущены. Для избежания таких потерь требуется набор изображений, полученных с интервалом меньшим, чем наименьшая интересующая длительность импульсного источника. Однако при таком подходе приходится иметь дело с большими наборами изображений. Помимо *фиксированной*, или *базовой* разности, когда один и тот же кадр, полученный до или после события, вычитается из всех остальных, полезна также *текущая* (*бегущая*) разность, поскольку она показывает резкие изменения, происшедшие между двумя соседними кадрами.

```
FOR J=1, NFILES-1 DO TVSCL, (X[*,*,J] - X[*,*,J-1]) ; базовая разность

FOR J=1, NFILES-1 DO TVSCL, (X[*,*,J] - X[*,*,0])   ; текущая разность
```

Просмотр последовательности разностных изображений в режиме кино помогает обнаружить такие изменения, но, опять же, при этом требуется рассматривать (*N–1*) кадров фиксированной или текущей разности. Метод текущей разности эффективен, если выбран оптимальный интервал между изображениями, достаточно большой для того, чтобы изменения от кадра к кадру были заметными, и достаточно малый для того, чтобы не пропустить интересующие особенности импульсного характера. Однако необходимо отдавать себе отчёт в том, что такие разностные изображения не свободны от артефактов (особенно *текущая разность*): если на каком-либо кадре было повышение яркости в некоторой области, а затем яркость вернулась к первоначальному уровню, то на последующем кадре это будет проявляться как ложное потемнение методического происхождения.

В качестве альтернативы авторы статьи Hanaoka et al.[16] анализировали временные профили ряда точек куба данных. Однако в этом способе требуется анализировать множество временных профилей, не располагая очевидным способом выбора контрольных точек в пространстве.

13.3 Дисперсионный метод[17]

Чтобы найти активные вспышечные источники в серии изображений вспышки, можно использовать *дисперсионные карты*. Дисперсионная карта представляет общую динамику всего события на единственном изображении. Применение дисперсионного метода в радиоастрономии было впервые предложено как минимум более 10 лет назад[18], затем он был более подробно разработан Кроуфордом с соавторами[19]. Однако до сих пор этот метод не получил распространения.

Дисперсия некоторого параметра – общеизвестная характеристика его изменчивости. Дисперсионная карта получается вычислением дисперсии в каждой пространственной точке куба данных вдоль его временного измерения. Значение каждой точки на дисперсионной карте отражает изменчивость во времени этой точки пространства. Поэтому на дисперсионной карте проявляются переменные источники: источники с изменявшейся яркостью, движущиеся, флуктуирующие и т.д. – разумеется, в соответствии со статистической значимостью их изменчивости.

По соображениям вычислительной эффективности целесообразно рассчитывать дисперсионную карту $\sigma^2{}_{ij}$ для куба данных x_{ijk} как

$$\sigma^2{}_{ij} = \frac{1}{N}\sum_{k=1}^{N} x^2{}_{ijk} - \frac{1}{N^2}\left(\sum_{k=1}^{N} x_{ijk}\right)^2 .$$

Здесь $i = 1,2,\ldots L$ – номер строки изображения, $j = 1,2,\ldots M$ – номер столбца и $k = 1,2,\ldots N$ – номер «слоя» изображения в кубе данных размерностью $L \times M \times N$. Данное выражение эквивалентно определению дисперсии, но не содержит перекрестных членов. Это позволяет избежать повторного сканирования куба данных, уменьшая время вычис-

---

[16] Hanaoka Y., Kurokawa H., Enome S. et al. 1994, Publications of Astronomical Society of Japan, 46, 205. Fig. 11 (p. 213).

[17] Grechnev V.V.: A method to analyze imaging radio data on solar flares. Solar Physics, 2003, *in press*. Существо метода было разработано автором в Радио Обсерватории Нобеяма, где он находился в качестве приглашённого иностранного исследователя Национальной Астрономической Обсерватории Японии.

[18] Robertson J.G. 1991, Australian Journal of Physics, 44, 729.

[19] Crawford D.F., Robertson J.G., Davidson G. 1996, Monthly Notes of Royal Astronomic Society, 283, 336.

лений. При работе в IDL можно вычислить дисперсионную карту с помощью следующей функции:

```
function varmap, x

Sz = size(x)
return, sqrt(total(temporary(x*x), 3)/Sz(3) - total(x, 3)^2/Sz(3)^2)

end
```

Дисперсионную карту можно отобразить, например, в полутоновом представлении величины дисперсии.

13.3.1 Что показывают дисперсионные карты?

Вычислив среднее значение по спокойной ненулевой области на ДК, можно найти инструментальную чувствительность σ_0. Все значения ниже этого уровня можно занулить.

Изменчивость главных нетепловых источников вспышек весьма высока. Поэтому для выявления более слабых источников целесообразно использовать нелинейное представление дисперсионной карты. На первом шаге для выявления всех изменчивых областей при визуализации можно использовать изображение с выровненной гистограммой (с помощью функции **HIST_EQUA**L: это преобразование делает равномерным амплитудное распределение яркости в изображении, в результате чего контраст резко снижается, и становятся одновременно видны как очень яркие, так и очень слабые детали). Затем полезно использовать для отображения степенную функцию (с показателем < 1), представление с помощью которой не столь грубо, как первое преобразование. По этой причине мы не делаем различия между картами в дисперсии или среднеквадратичном отклонении.

Регулярные инструментальные эффекты (например, боковые лепестки диаграммы направленности) также проявляются на дисперсионной карте. Это дает возможность судить о том, может ли быть источник, видимый на дисперсионной карте, инструментальным эффектом, если он расположен в области таких особенностей инструментального происхождения. Если временной профиль слабого подозрительного источника, расположенного в такой области, воспроизводит вариации главного вспышечного источника, ответственного за боковые лепестки, это дает основание отвергнуть этот слабый источник.

13.3.2 Дальнейший анализ

Когда активные вспышечные источники определены, можно проанализировать их временные профили по отдельности. Для этой цели выбираются небольшие области, полностью охватывающие эти источники, и суммируются значения пикселов по этим областям. Сумма, вычисленная по одному изображению, дает одну точку на временном профиле. Временные профили для каждого источника можно вычислить с помощью функции **TOTAL_FLUX**.

Когда найдены интересующие особенности на временных профилях, опять строятся, на сей раз, детальные наборы изображений с необходимым временным интервалом, вплоть до временного разрешения радиогелиографа. Такие наборы могут состоять из нескольких сотен изображений. Затем снова анализируются дисперсионные карты, и процесс может повторяться итеративно до тех пор, пока будет выявлена динамика пространственных вспышечных структур. Итерации выполняются для выявления всех значимых источников и построения для каждого из них временного профиля с той детальностью, которая определяется требованиями физической задачи.

Чтобы выяснить магнитную связность выявленных областей, анализируются временные профили потока их излучения в поляризации (параметр Стокса V). На магнитную связность указывают схожие особенности участков временных профилей противоположно поляризованных источников. Однако такой анализ не всегда возможен. При этом также полезны разностные изображения, полученные следующим образом. Вначале изображения, полученные во время локального пика, усредняются. Затем компонуется другое изображение из кадров, полученных до и/или после этого пика. Наконец, последнее изображение вычитается из первого.

13.3.3 Ограничения применимости метода

Изображения, которые предполагается анализировать с помощью дисперсионных карт, должны быть хорошо откалиброваны по величине и пространственно совмещены: не должно быть дрожаний и скачков изображений друг относительно друга. Дисперсионная карта сразу улавливает такие проблемы куба данных. Результатом случайных дрожаний меньше размера источника является его кольцеобразная форма на дисперсионной карте с более «темной» серединой и более «яркой» окантовкой по краю. Если сдвиги изображений друг относительно друга больше, то по форме и размерам ярких деталей на дисперсионной карте можно оценить погрешности центровки. Результатом скачков между несколькими фиксированными положениями является возникновение дополнительных ложных источников.

Если рассматриваемый интервал столь продолжителен, что существенным становится вращение Солнца, то либо оно должно быть скомпенсировано, либо приняты иные меры. В противном случае спокойные (напр., циклотронные) источники существенно смещаются, а это приводит к их ложному проявлению как сильно изменчивых раздвоенных источников.

При использовании процедуры «чистки» возможно также возникновение весьма специфических ложно-импульсных источников, если уровень чистки близок к яркостной температуре радиоисточников, проявляющих существенно меньшую изменчивость по другим причинам (например, вследствие правильных колебаний с 3-минутным периодом).

14 Построение программ с графическим интерфейсом

14.1 Общие замечания

Примерно 20 лет назад ряд специалистов в области вычислительной техники пытался создать «квазиестественный язык» для того, чтобы сделать работу с компьютером доступной для пользователей, не очень сведущих в программировании. Около десяти лет спустя такой «язык» был фактически создан: это *пользовательский графический интерфейс* или, по-английски, *Graphics User Interface* (GUI). Графический интерфейс представляет наиболее эффективный способ манипуляции стандартизованными данными с помощью достаточно хорошо разработанных методик. Программы с графическим интерфейсом – «*приложения*» (англ. *Application*) – удобны также для просмотра значительного количества однотипных данных. IDL позволяет создавать и такие программы. Когда методика проверена, её можно реализовать в виде программы для автоматической обработки данных или программы с графическим интерфейсом. Последний вариант предпочтителен, если при работе с данными нужно активное участие исследователя. При разработке какой-либо методики работы с данными последовательно создаются фрагменты программы, которые затем просто включаются в головную программу с графическим интерфейсом. Такой путь, при котором все стадии разработки методики и манипуляции данными ведутся в рамках единой языковой базы, представляется оптимальным.

14.1.1 Преимущества и ограничения

В общем случае, программы с графическим интерфейсом более гибки по сравнению с программами без графического интерфейса в выборе конкретной стратегии из нескольких возможных ветвей алгоритма. Однако никакая программа с графическим интерфейсом не способна выполнить операцию, не предусмотренную в соответствующем программном файле. Всякая программа с графическим интерфейсом может сделать лишь то, что она может.[20]

С другой стороны, комбинации предварительных операций с использованием программных файлов с последующей импровизацией в командной строке предоставляют неограниченные возможности для реализации самых разнообразных способов вычислений, обработки и анализа данных.

14.2 GUI builder и «ручное» программирование

Семейство IDL 5-й версии позволяет автоматизировать создание программ с графическим интерфейсом в «visual» режиме с помощью автоматизированного построителя *GUI builder*, а также позволяет создавать такие программы «вручную».

GUI builder – это программа, имеющая, как и всякая иная программа, свои преимущества и недостатки. Главное достоинство GUI builder – это простота создания программ с графическим интерфейсом. Однако модифицировать содержимое такой программы не так просто. Иногда в программах, созданных с помощью автоматизированного построителя, возникают трудно локализуемые проблемы. Чтение и понимание такой программы сильно затруднено. Программные файлы содержат множество «паразитных» процедур, которые в действительности лишь передают управление другим процедурам и не делают ничего больше. Поэтому, если после компиляции такого программного файла мы

[20] Заметим, однако, что в IDL имеется возможность снабдить программу встроенным компилятором с помощью функции **EXECUTE**. Но его возможности не могут быть очень широкими, и этот способ реализуется не так-то просто.

введём в командной строке **HELP, /ROUTINES**, мы увидим множество этих «паразитных» процедур, засоряющих полезную информацию. Положившись на GUI builder, мы лишимся возможностей использования ряда возможностей и приёмов программирования, которые могли бы применить при «ручном» написании программы. Например, GUI builder в IDL версии 5.2 не позволяет управлять видимостью каких-либо элементов графического интерфейса, что, однако, можно сделать в программе, которую мы создадим сами. В целом, работает известная закономерность: автоматизация позволяет что-то упростить, однако качество и возможности продуктов «ручной работы» всё же оказываются выше.

С точки зрения пользователя, создание программы с помощью GUI builder довольно прозрачно, и мы относительно подробно обсудим создание программ с графическим интерфейсом с помощью только «ручного» программирования.

14.3 Способ создания программы с графическим интерфейсом

14.3.1 Элементы графического интерфейса («widgets»)

Сначала о терминологии. Элементы графического интерфейса называются «widgets». Словарь «Random House Webster's College Dictionary» так объясняет значение английского слова «widget»: «1. маленькое механическое устройство, как, например, кнопку или выключатель, особенно такое, название которого неизвестно или не удаётся вспомнить». Иными словами, слово «виджет» используется для замены словосочетания «элемент графического интерфейса». Есть несколько типов виджетов: база (*widget_base*) – пустое поле, на котором размещаются любые элементы графического интерфейса; кнопки (*widget_button*); линейка с движком (*widget_slider*); текстовое поле (*widget_text*); метка (*widget_label* – небольшое текстовое поле для выдачи краткой информации); графическое поле (*widget_draw*); многоэлементный список (*widget_list*) и т.д. «Родительская» база используется для размещения на ней других виджетов-«детей», которые принадлежат графическому оформлению программы. В общем случае программа может включать несколько независимых баз. Однако, когда «лидер» прекращает существование (когда происходит выход из программы), при этом обычно исчезают и все другие базы.

В IDL имеется ряд процедур для создания элементов графического интерфейса – виджетов – и управления ими. Имя функции, создающей виджет, в общем случае называется **widget_** с последующим указанием типа виджета: **widget_base**, **widget_button**, **widget_list** и т.д. Имеется также многофункциональная процедура для управления всеми типами виджетов **widget_control**, которая позволяет изменять размеры баз, текстовых и графических полей; разрешать и запрещать события; управлять чувствительностью и видимостью различных элементов графического интерфейса; передавать значения переменных и т.д.

Каждому виджету можно присвоить *пользовательскую величину*, которая может иметь любой тип и размерность. Она может быть и структурой. Простым, однако основным, способом является присвоение элементам графического интерфейса некоторых текстовых значений, а в процедуре обработки событий эти значения используются для управления действиями, которые должны выполняться при возникновения того или иного события.

14.3.2 Части программы с графическим интерфейсом

Программа с графическим интерфейсом должна состоять как минимум из двух частей:

- процедуры, создающей графическое оформление программы на экране (головная процедура или функция) ;

- процедуры, выполняющей обработку событий, генерируемых в процессе выполнения программы.

События генерируются элементами графического интерфейса в результате любого действия пользователя: нажатия или отпускания кнопок мыши, перемещения курсора в графическом окне; кроме того, возможна генерация события таймером и т.п. Процедура обработки событий *должна предшествовать* процедуре создания графического интерфейса: к тому моменту, когда графическое оформление программы создаётся и появляется на экране, все действия, которые должны выполняться в ответ на появление событий, должны быть уже определены. В типичном случае (однако не обязательно) имя процедуры, обрабатывающей события, совпадает с именем головной процедуры, за которым следует подчерк и слово «*event*». В качестве примера попытаемся написать простую процедуру с графическим интерфейсом, имеющую имя **gui_test.pro**. Тогда процедура обработки событий будет называться **gui_test_event.pro**. Количество аргументов головной процедуры ничем не лимитировано. Аргументов может и вообще не быть. Процедура обработки события всегда вызывается с *единственным аргументом*, который является структурой, зависящей от события, вследствие которого была вызвана процедура. Однако эта структура всегда имеет три поля: **ID**, **TOP** и **HANDLER**. Поле **ID** содержит идентификационный номер виджета, который выдал событие. Поле **TOP** – это идентификационный номер элемента графического интерфейса *верхнего уровня*, «родителя» всех прочих виджетов-«детей». Поле **HANDLER** содержит идентификатор элемента графического интерфейса, связанного с процедурой обработки событий.

14.3.3 Передача переменных

Обычна ситуация, когда как процедура, создающая графическое оформление, так и процедура, обрабатывающая события, должны иметь доступ к некоторым переменным. По существу, имеется два способа реализации обмена переменными. Первый способ – использовать общие блоки (**COMMON**), переменные в которых доступны всем программам, в которых эти блоки заданы. **COMMON**-блоки аналогичны используемым в FORTRANe, с тем исключением, что в IDL они всегда именованные. Имеется очевидное неудобство их использования: если запущена более чем одна реализация одной и той же программы, то значения переменных в **COMMON**-блоках в обеих программах будут мешать друг другу.

Другой способ – использование *пользовательских величин*, присваиваемых виджетам. Поскольку тип пользовательских величин не лимитирован, можно использовать их для транспортировки переменных между процедурами. Типичный пример использования этого способа:

```
widget_control, ev.id, get_uvalue = variable, /no_copy
...................
...................
widget_control, ev.id, set_uvalue = variable, /no_copy
```

Установка ключевого слова **NO_COPY** экономит используемую память. Однако для обмена большими массивами данных лучше всё же использовать **COMMON**-блоки.

IDL не позволяет расширять **COMMON**-блоки в пределах одного сеанса. По нашему мнению, более удобно переменные, включаемые в **COMMON**-блоки, собирать в структуры. Это позволяет избежать перезагрузки IDL в тех случаях, когда мы хотим включить дополнительные переменные в **COMMON**-блок. Кроме того, количество *позиционных аргументов* в **COMMON**-блоке часто бывает столь велико, что контролировать их все по положению в блоке становится просто невозможным, а использование структур для сбора в них ряда переменных намного более удобно.

14.3.4 Пример

Попытаемся написать простую программу с графическим интерфейсом.

```
; ************** Процедура обработки событий ************************

pro gui_test_event, ev

widget_control, ev.id, get_uvalue = UV
      ; Пользовательская величина передаётся в переменную UV

CASE UV OF
      ; Здесь мы описываем, какие действия должны быть выполнены при
      ; появлении события

'Done':     widget_control, ev.top, /destroy ; - По событию «Done»
                        ; разрушить родительский виджет

ELSE:
      ; Полезно включать пустой оператор 'ELSE' для избежания ошибок
      ; в процессе разработки и отладки программы

ENDCASE

end

; ************** Головная процедура ************************

pro gui_test

BASE = widget_base(tit = 'GUI test', /column)
      ; Это ГЛАВНАЯ база. Она не имеет «родителей». На ней можно
      ; расположить любые виджеты-«дети» в строку (ROW) или
      ; колонку (Column)

BUTTON = widget_button(BASE, value = 'Done', uvalue = 'Done')
      ; Это будет виджет - кнопка на «родительской» базе «BASE»,
      ; с надписью «Done», и мы присваиваем ей такую же
      ; пользовательскую величину «Done». Это не обязательно, но удобно

widget_control, base, /realize
      ; Теперь мы «реализуем» нашу программу с графическим интерфейсом
      ; на экране

xmanager, 'gui_test', base
      ; Процедура XMANAGER регистрирует наше приложение и управляет
      ; обработкой событий. Наша задача - описать действия, которые
      ; должны выполняться в ответ на появление любого события.
      ; Эти действия указываются в процедуре обработки событий
      ; gui_test_event

end
```

В этом простом примере мы не обменивались никакими величинами между процедурой создания графического интерфейса и процедурой обработки событий. Теперь будем развивать наше простое приложение дальше и введём обмен переменными, изобразим

графическое окно, а также попробуем вывести текущие координаты курсора и значения пикселов в массиве в специальное окно-метку.

```
; ************** Процедура обработки событий ************************

pro gui_test_event, ev

common gui_test, DATA, a, b, c, d
      ; Это – общий блок для обмена нашими переменными. Переменные
      ; a, b, c, d резервируем для дальнейшего возможного использования


SZ = size(DATA)
      ; SZ содержит измерения массива DATA

if Sz[0] lt 2 then return
      ; возврат (RETURN), если массив DATA ещё не определён. Здесь
      ; это необязательно

widget_control, ev.top, get_uval = ID, /no_copy
      ; Передача структуры, содержащей идентификаторы виджетов,
      ; в переменную ID

      if ev.ID eq ID.DRAW then begin
            ; Обработка событий от графического поля (виджет «DRAW»)

      widget_control, ID.LABEL, set_value = $
            string(ev.X > 0 < (Sz[1]-1), ev.Y > 0 < (Sz[2]-1), $
                  DATA[ev.X > 0 < (Sz[1]-1), ev.Y > 0 < (Sz[2]-1)], $
                        format = '(i3, ", ", i3, ", ", f5.1)')

      widget_control, ev.top, set_uval = ID, /no_copy
            ; Присваиваем виджету наивысшего уровня в качестве
            ; пользовательской величины структуру, содержащую
            ; идентификаторы виджетов

      return
            ; Готово. Спокойный возврат
      endif

widget_control, ev.id, get_uvalue = uv
            ; Обработка событий от прочих виджетов

CASE uv OF

'Done':     begin
      widget_control, ev.top, /destroy
      DATA = (a = (b = (c = (d = 0))))
            ; Освобождаем память
      return
            ; RETURN здесь необходим, иначе следующее действие
            ; «widget_control, ev.top, set_uval = ID» вызовет попытку
            ; обращения к виджету EV.TOP, который уже не существует,
            ; что приведёт к ошибке

            end
```

```
ELSE:


ENDCASE


widget_control, ev.top, set_uval = ID, /no_copy
      ; Присваиваем виджету наивысшего уровня в качестве
      ; пользовательской величины структуру, содержащую идентификаторы
      ; виджетов

end


; ************** Головная процедура ************************

pro gui_test

common gui_test, DATA, a, b, c, d

device, get_screen_size = scr, decomposed = 0
      ; Определяем размер (разрешение) экрана для того, чтобы создать
      ; автоматически масштабирующийся виджет, а также изменяем
      ; цветовой режим на High Color

DRAWSIZE = fix(scr[1]*0.6)

DATA = dist(DRAWSIZE)
      ; Имитация данных

ID = {DRAW:0L, LABEL:0L}
      ; Определяем структуру ID для того, чтобы хранить в ней
      ; идентификаторы виджетов, которые должны быть доступны
      ; их процедуры обработки событий.
      ; Нет нужды включать туда идентификаторы всех виджетов

BASE = widget_base(tit = 'GUI test', /column)

BUTTON = widget_button(BASE, value = 'Done', uvalue = 'Done')

ID.DRAW = widget_draw(BASE, xsize = DRAWSIZE, ysize = DRAWSIZE, $
      /motion_events)
      ; Создаём графическое поле и активизируем его для генерации
      ; событий при перемещении курсора внутри этого поля

ID.LABEL = widget_label(BASE, /dynamic_resize, /frame)
      ; Виджет-метка для вывода координат и значений пикселов,
      ; автоматически изменяющий свои размеры в зависимости от
      ; выводимого текста

widget_control, BASE, /realize, set_uval = ID, /no_copy
      ; Присваиваем структуру, содержащую идентификаторы виджетов,
      ; виджету наивысшего уровня в качестве пользовательской величины

loadct, 3
```

```
tvscl, DATA

xmanager, 'gui_test', base

end
```

14.3.5 Системы координат

IDL запоминает и хранит преобразования систем координат, соответствующие последней выдаче на графическое устройство. Если наше приложение с графическим интерфейсом имеет более одного графического окна, для каждого из которых действуют разные системы координат, то нам следует запоминать все эти преобразования координат и самим устанавливать их при переходе к другому окну. Любые возможные преобразования координат полностью описываются системными переменными **!X**, **!Y**, **!Z** и **!MAP**. Поэтому очевидный способ запоминания/восстановления координатных систем – сохранение этих системных переменных – например, в структуры **{x:!X, y:!Y, z:!Z, map:!MAP}** – после каждой выдачи графики, и передача соответствующих значений из этих структур в системные переменные, когда обрабатывается другое графическое окно. Например, можно присвоить виджетам графических полей эти структуры в качестве пользовательских величин.

С помощью этого способа мы успешно управляли несколькими графическими окнами в одном приложении, вплоть до шести, и это, конечно же, не предел.

После освоения этого простого способа написания программ с графическим интерфейсом Вы сможете разрабатывать свои намного более сложные и удобные приложения.

15 Библиотека SolarSoftware

Библиотека SolarSoftware содержит очень большое количество процедур общего назначения и ориентированных на задачи солнечной физики, доступных через FTP. В настоящее время эта библиотека стала гигантской: SolarSoftware содержит более 11000 одних только программных файлов, а также многие и многие файлы баз данных. Эта библиотека включает программное обеспечение на языке IDL для обработки и анализа данных различных орбитальных и наземных солнечных инструментов: телескопов на борту ULISSES, CGRO (BATSE), Yohkoh, SOHO, TRACE, RHESSI, GOES и других космических аппаратов; солнечных радиотелескопов Нобеяма (NoRH), Овенс Вэлли (OVSA) и многих других инструментов. Эти пакеты программ – SolarSoftware – доступны через Интернет[21]. Основу SolarSoftware исторически составили пакеты программ для работы с данными Yohkoh, BATSE и GOES. Позже добавились другие пакеты. Ряд процедур общего назначения также используется многими другими процедурами и функциями.

Имеются поисковые системы для этих процедур: в лаборатории Lockheed на Web-странице Yohkoh [22] и поисковая страница SOHO[23].

Мы приведём немногие примеры и некоторые рекомендации по разработке вашего собственного программного обеспечения, которые помогут свести к минимум возможные проблемы.

15.1 Некоторые примеры чтения и отображения данных с помощью SolarSoftware

15.1.1 FITS-файлы CGRO/BATSE типа «DISCSC» 64-миллисекундного разрешения[24]

Эти файлы относятся к типу FITS BINTABLE, и работа с ними несколько сложнее, чем с простейшими файлами FITS. В общем случае FITS BINTABLE позволяет записывать в одном файле несколько массивов различного типа и размерности. Его можно также рассматривать как удобный машинно-независимый формат для сохранения ряда структур.

```
; ******************* Чтение данных *********************

file = dialog_pickfile(filter = 'discsc*.fits')

head = headfits(file,ext=0)
e_struct=mrdfits(file,1,head1)
d_struct=mrdfits(file,2,head2)

ut = anytim(avg(d_struct.times,0))

basetime = tjd2ymd(fxpar(head2,'basetime'))

; ***************** Отображение данных ********************
```

[21] SolarSoftware is available at the Web sites http://sohowww.nascom.nasa.gov/solarsoft/ and http://www.lmsal.com/solarsoft/

[22] http://www.lmsal.com/SSW/idl/index.html

[23] http://orpheus.nascom.nasa.gov/~zarro/xdoc/

[24] R. Schwartz suggested this way.

```
!p.multi = [0,1,4]

for j=0,3 do utplot, ut, d_struct.rates(j,*), basetime

!p.multi = 0

      end
```

15.1.2 FITS-файлы радиогелиографа Нобеяма (NoRH)

Изображения и корреляционные кривые, полученные на Радиогелиографе Нобеяма, записываются в FITS-файлы. Можно считывать данные из них с помощью функции **READFITS** либо воспользоваться программами, разработанными сотрудниками солнечной группы обсерватории Нобеяма:

```
path = '…'        ; Вместо «…» следует указать полный путь каталога, в
                  ; котором размещаются файлы данных

files = findfile(path + 'ipa*')
files = files[sort(files)]          ; Сортировка файлов в возрастающем
                                    ; по времени порядке

norh_rd_img, files, index, data           ; Чтение куба данных

utplot, index, total(total(data,1),1)     ; Вывод ненормированного графика
                                          ; интегрального потока
```

Здесь **DATA** – трёхмерный куб данных, и мы можем просмотреть его в режиме кино или работать с ним с использованием любого из способов, рассмотренных выше. **INDEX** – это структура, содержащая дату, время, центр поля зрения, размер пиксела и многие другие полезные параметры.

Корреляционные кривые можно прочитать с помощью процедуры **NORH_RD_TCX**.

15.1.3 Чтение FITS-файлов изображений, полученных в крайнем ультрафиолете

Солнечные изображения этого типа получаются на телескопах крайнего ультрафиолетового диапазона на борту космических обсерваторий SOHO (EIT) и TRACE. Для изображений в крайнем ультрафиолете характерен очень широкий динамический диапазон. Поэтому, если мы попытаемся вывести их на экран с линейным масштабированием яркости, то, скорее всего, не увидим ничего, кроме самых ярких деталей либо отдельные точки ярких дефектов изображений. Чтобы сделать видимыми также более слабые детали, можно использовать логарифмическую шкалу яркости либо применить функцию **HIST_EQUAL**. Приведённый ниже пример позволяет прочитать файл EIT и вывести его содержимое на экран в линейном и нелинейном представлении.

```
path = '…'        ; Вместо «…» следует указать полный путь каталога, в
                  ; котором размещаются файлы данных

files = findfile(path + 'efz*')
```

```
x = readfits(files[0], header, /si)

device, get_screen_size = screen
winsize = screen[0]*0.5

window, 0, xsize = winsize, ysize = winsize
tvscl, congrid((x), !d.x_size, !d.y_size, /int)

window, 2, xsize = winsize, ysize = winsize
tvscl, congrid(hist_equal(x), !d.x_size, !d.y_size, /int)

end
```

Заметим опять, что преобразование с выравниванием гистограммы полезно и для визуализации изображений мягкого рентгеновского диапазона и многих других типов изображений для того, чтобы одновременно показать и очень яркие, и слабые детали.

15.2 Программы Д. Зарро[25]

Доминик Зарро (Dominic Zarro) разработал ряд программ IDL для анализа солнечных изображений с использованием объектно-ориентированных методов.

В простейшей форме объект IDL «карта» – это структура, которая содержит данные двумерного изображения и сопровождающую их информацию о координатах пикселов и пространственном масштабе. Последние параметры определяются как свойство карты и являются уникальными для каждого источника изображений. Произвольное изображение, определённое таким образом, можно обрабатывать или преобразовывать независимо от его происхождения (источника).

Начиная с версии 5, в IDL введена поддержка объектно-ориентированного программирования. Практически это означает, что структуры, которые ранее могли содержать лишь поля переменных данных, теперь могут содержать дополнительные поля, являющиеся в действительности функциями (или процедурами), оперирующими с собственно данными. Эти структурные функции называются методами. В контексте объектно-ориентированного программирования карты изображений с такими методами известны как объекты.

Программы Зарро позволяют создавать объекты карт для обработки солнечных изображений. Типичные задачи обработки включают коррекцию наклона изображения, масштабирование, сдвиги, компенсацию вращения Солнца и совмещение изображений.

Поскольку и изображение, и сопровождающая информация содержатся в одной и той же структуре, к изображению можно обратиться как к полю структуры. Этот способ работы с данными не столь прозрачен для исследователя, как простейшее представление различными переменными. Процедуры и переменные фактически становятся непонятными «чёрными ящиками». Однако этот путь имеет очевидное преимущество: поскольку изображения жёстко связаны со служебными данными, возможности ошибок резко уменьшаются.

---

[25] Эти программы доступны на WWW-странице http://orpheus.nascom.nasa.gov/~zarro/idl/maps.html. Приведённое здесь краткое описание также написано Д. Зарро.

15.3 Трудности и возможные рекомендации

Как мы уже отметили, в настоящее время SolarSoftware стало огромным. Конечно же, трудно освоить такое программное обеспечение и непросто ориентироваться в нём. У этой ситуации имеется несколько причин:

(a) существует множество солнечных инструментов;

(b) при использовании IDL часто оказывается намного проще разработать собственную процедуру, чем найти уже готовую;

(c) основы SolarSoftware были заложены, когда, в отличие от современной ситуации, ещё не было многих встроенных процедур IDL, а его возможности резко отличались от возможностей современного IDL;

(d) обмен опытом работы с IDL между исследователями всё ещё недостаточен.

Мы пока не знаем решения этой проблемы. Сейчас мы можем только дать несколько рекомендаций[26] по разработке собственного программного обеспечения так, чтобы свести такие проблемы к минимуму.

- Сводите к минимуму количество программных файлов, которые должны составить библиотеку. Если какая-то процедура имеет лишь вспомогательное значение для другой и не самодостаточна, её следует поместить в тот же программный файл, в котором содержится главная процедура.
- Процедуры общего назначения должны быть «элементарными» и должны решать единственную полезную задачу с простейшим возможным входом, и никаких других задач.
- Избегайте размещения одного и того же программного файла в разных подкаталогах файловой системы. Ситуация, когда один и тот же программный файл встречается несколько раз в различных подкаталогах (что обычно в SolarSoftware) не только неправильна, но и опасна. Изменение одной из них, но не всех сразу, приводит к несовместимости и непредсказуемым результатам.
- Имена программных файлов не должны содержать больших букв, иначе IDL при работе в системе UNIX не сможет найти такой программный файл.
- Имена процедур должны быть максимально информативными. Следует избегать неинформативных или неоднозначных имен. Во избежание путаницы между процедурами с идентичными именами, разработанными различными исследовательскими группами, можно снабжать имена процедур префиксами группы разработчиков, названия инструмента и т.п. (например, **norh_rd_img**, **ovsa_get_index** и т.д.)
- Имена аргументов процедур (функций) должны быть максимально информативными. Например, варианты

```
ROUTINE_NAME, IMAGE
```

или

```
ROUTINE_NAME, TWO_D_ARRAY
```

– лучше, чем

```
ROUTINE_NAME, X
```

---

[26] Эти рекомендации – результат дискуссий с проф. Д. Гэри из Университета Нью-Джерси (США).

Обратите внимание на то, что IDL выдаст Вам по запросу

```
HELP, /ROUTINES
```

– и Вы убедитесь в целесообразности этой рекомендации.

- Не применяйте русских слов для названий процедур и аргументов, и букв русского алфавита – для комментариев. Понятные Вам обозначения будут бессмыслицей для Ваших коллег, не знающих русского языка. В то же время каждый, кто знаком с основами программирования или работы с операционной системой, имеет представление о значении английских слов. Комментарии же на русском языке, написанные при работе в Microsoft Windows, превратятся в непонятный код при работе в UNIX. И наоборот.
- Избегайте изменения системных переменных без очень существенных оснований. Даже если Вы восстанавливаете их при выходе из программы, представьте себе заурядную ситуацию, когда в программе происходит останов по какой-либо ошибке.

---

16 Заключение

Подошёл к концу тот материал, который мы успели подготовить. Возможно, было бы полезным также привести ряд вопросов, которые автор получал в письмах, и ответы на них, однако систематизация этой обширной переписка потребовала бы немалого времени, что задержало бы выход этой книги. Возможно, в будущем автор возьмётся и за эту работу.

Мы полагаем, что изучение этой книги, сколь внимательным оно бы ни было, не может гарантировать читателю освоения на достаточно высоком уровне обработки и анализа данных в среде IDL. Эта книга – лишь пособие, она не содержит систематического изложения основ языка. Необходимо также изучение документации по языку IDL и, конечно же, постоянная практика. По своему опыту мы заметили, что IDL приучает к несколько иному стилю мышления, чем при использовании традиционных языков программирования. Бывает, что составление оптимальной программы напоминает решение задачи на сообразительность, и такие «головоломки», несомненно, способствуют развитию изобретательности, что окажет исследователю впоследствии добрую услугу при разработке новых методик.

Появление IDL, по существу, привело к возникновению нового направления на стыке информатики и математической физики – *интерактивной работы с данными*. Оно существенно отличается от традиционного программирования непрерывной работой с реальными данными и методами, а не их умозрительной моделью; основной целью – извлечением определённой информации из данных, а не «запрограммированием» поставленной кем-то задачи; постоянным эффективным контролем результатов каждого действия с неограниченными возможностями возврата и выбора других путей.

Мы постарались также показать существующие трудности и проблемы, возникающие при использовании IDL, и поделиться своими соображениями о том, как свести их к минимуму. Возможно, читателям удастся преодолеть их кардинальным образом.

В заключение автор выражает благодарность всем, кто так или иначе способствовал появлению этой книги.

Приложение.

17 Интерфейс MS Windows

1. Переключатель клавиатуры **RUS/LAT** (наиболее часто)**:**

   Windows 3.1(1) – левый **Shift** + правый **Shift**

   Windows 95 – левый **Alt** + левый **Shift**

2. Управление задачами

   ◊ **Ctrl** + **Esc:**

   Windows 3.1(1) – Вызов диспетчера заданий (Task Manager)

   Windows 95 – Вызов программ (Taskbar)

   ◊ **Alt** + **Tab** – Переключение заданий

   ◊ **Ctrl** + **Tab** – Переход по группам Диспетчера Программ (Program Manager)

   ◊ **Tab** – Переход между элементами

   ◊ **Shift** + **Tab** – Переход между элементами в обратном направлении

   ◊ **Alt** – Вход в МЕНЮ

   ◊ **Alt** + **буква** – Переход к данной позиции МЕНЮ

   ◊ **Alt** + **F4** – Закрыть приложение

3. Управление окнами

   ◊ **Alt** + **пробел** – Управление размерами и положением главного окна приложения

   ◊ **Alt** + **-** – Управление размерами и положением внутреннего окна

   ◊ **Левый Alt** + **Enter** – Переключение размеров окна DOS

   ◊ **Ctrl** + **F6** – Переход между внутренними окнами

4. Клавиши перемещения

   ◊ ←, ↑, →, ↓, PgUp, PgDn, Home, End – обычное назначение

   ◊ **Ctrl** + **Home** – Перемещение к началу файла

   ◊ **Ctrl** + **End** – Перемещение к концу файла

   Для IDL версий 3.5 и 3.6:

   ◊ **Ctrl** + PgUp – Перемещение к началу файла

   ◊ **Ctrl** + PgDn – Перемещение к концу файла

5. Клавиши редактирования

- ◊ **Shift** + **движение** (←, ↑, →, ↓, PgUp, PgDn, Home, End, Ctrl + Home, Ctrl + End) – Выделение
- ◊ **Ctrl** + **Insert** – Копирование выделенного фрагмента в буфер
- ◊ **Del** – Удаление символа справа или фрагмента (если он выделен)
- ◊ **BackSpace** – Удаление символа слева или фрагмента (если он выделен)
- ◊ **Shift** + **Insert** – Вставка выделенного фрагмента из буфера
- ◊ **Левый Alt** + **BackSpace** – Отмена последней операции (UnDo/ReDo)

  В WinWord – последовательная отмена операций

Пример с сортировкой словаря

Примеры с графикой (plot, surface, contour,tvscl)

PostScript

Полноцветные рисунки с комбинацией цветовых таблиц

Обрезанный синус

Бесформатные запись и чтение файла

Остановился на стр. 9