Н.Н.Мартынов, А.П.Иванов

MATLAB 5.X.
ВЫЧИСЛЕНИЯ, ВИЗУАЛИЗАЦИЯ, ПРОГРАММИРОВАНИЕ

Книга является компактным учебным пособием по работе с популярнейшим пакетом математических и инженерных вычислений MATLAB. Она является расширенной версией учебного курса, разработанного авторами и много лет читаемого на физическом факультете Московского государственного университета им. М. В. Ломоносова.

Изложение базируется на возможностях версий 5.x пакета MATLAB, ставшего в настоящее время стандартным средством поддержки изучения высшей математики, численного анализа, теории изображений и других учебных курсов во многих университетах мира.

Будучи очень компактным, данное учебное пособие не носит справочного характера и не дублирует широчайшую информацию, представленную во встроенной системе помощи и в электронных книгах, поставляемых вместе с пакетом MATLAB. В книге излагаются основные возможности вычислений и визуализации результатов, предоставляемые ядром системы MATLAB. Кроме того, рассматривается популярное расширение пакета MATLAB - Symbolic Math Toolbox, реализующее возможности символьных вычислений и преобразований.

Наибольшее внимание в книге уделяется вопросам создания законченных приложений на базе пакета MATLAB: использованию графического интерфейса пользователя, подключению существующих и написанию новых функций на языке C, взаимодействию внешних приложений с ядром системы MATLAB, применению математических библиотек системы MATLAB в самостоятельных Win32-приложениях, созданных компилятором Microsoft Visual C++. Именно эти вопросы чаще всего отсутствуют в печатных пособиях по системе MATLAB, а в электронных книгах изложены лишь фрагментарно.

Книга рекомендуется преподавателям и студентам университетов и технических вузов, программистам, инженерам и научным работникам, всем, кто интересуется применением компьютеров для решения задач математики, физики, химии и других наук, в том числе преподавателям и ученикам средних школ с углубленным изучением соответствующих дисциплин.

Оглавление

Часть 1. Вычисления и визуализация 3
Глава 1. Числовые массивы в системе MATLAB **3**
Рабочее пространство системы MATLAB и ее командное окно 3
Вещественные числа и тип данных double 9
Комплексные числа и комплексные функции 16
Формирование одномерных числовых массивов 19
Двумерные массивы чисел: матрицы и векторы 24
Многомерные числовые массивы 29
Вычисления с массивами 33
Множественная индексация массивов в системе MATLAB 39

Глава 2. Визуализация результатов вычислений **47**
Построение графиков функций 47
Оформление графиков и графических окон 52
Специальная графика системы MATLAB 60
Трехмерная графика 65
Дополнительные детали оформления трехмерных графиков 73
Растровые изображения и тип данных uint8 77
Глава 3. Массивы символов, структур, ячеек. Файловые операции **87**
Массивы символов и тип данных char 87
Встроенные функции для обработки строк 94
Массивы структур 98
Массивы ячеек 102
Чтение и запись произвольных бинарных файлов 109
Чтение и запись произвольных текстовых файлов 119
Глава 4. Краткий обзор встроенных средств решения типовых задач алгебры и анализа **125**
Решение систем линейных уравнений 125
Операции линейной алгебры над матрицами. Матричные функции 126
Разреженные матрицы 130
Вычисление спецфункций математической физики 131
Нахождение нулей функций 133
Поиск минимума функции 136
Вычисление определенных интегралов 139
Решение систем обыкновенных дифференциальных уравнений 143
Глава 5. Интерактивный режим работы и его автоматизация с помощью сценариев **148**
Сохранение результатов вычислений интерактивного сеанса работы 148
Операторы цикла. Векторизация как альтернатива циклам 153
Анимация и звук в системе MATLAB 157
Сценарии и M-файлы 162
Аналитические вычисления с помощью пакета расширения Symbolic Math Toolbox 166
Справочная подсистема пакета MATLAB 172
Часть 2. Программирование в среде системы MATLAB 176
Глава 6. Программирование функций на M-языке **176**
Синтаксис определения и вызова M-функций 176
Конструкции управления 181
Интерактивное взаимодействие M-функций с пользователем 185
Локальные, глобальные и статические переменные 190
Рекурсивные функции. Производительность M-функций 193
M-функции с переменным числом входных параметров и выходных значений 198
Контроль входных параметров и выходных значений M-функции 200
Практические советы по разработке и отладке M-функций 205

Глава 7. Примеры конкретных разработок М-функций **208**
функции, работающие со временем и датами 208
Обработка текстов 213
Функции для работы с файлами данных 219
Динамическое построение графика функции 222
Вращение трехмерных графиков 227
Глава 8. Программирование функций на языке С **230**
Интерфейс MEX-функций с системой MATLAB 230
Создание и компиляция DLL-проекта в среде Microsoft Visual C++ 234
Вызов функций MATLAB API 238
Отладка MEX-функций 243
Примеры конкретных разработок MEX-функций 247
Вызов функций и команд системы MATLAB из MEX-функций 256
Часть 3. Создание законченных приложений 260
Глава 9. Законченные приложения на базе графического интерфейса пользователя системы MATLAB **260**
Графические окна системы MATLAB и элементы управления 260
Создание основных элементов управления 263
Графический объект axes 270
Callback-функции 275
Применение утилиты guide для формирования пользовательского интерфейса 280
Динамическая перестройка элементов управления 282
Использование манипулятора мышь в графических окнах пакета MATLAB 290
Создание меню 293
Глава 10. Взаимодействие внешних приложений с системой MATLAB **299**
Взаимодействие приложений Windows с MATLAB Engine 299
Создание и компиляция EXE-проекта в среде Microsoft Visual C++ 311
С-библиотеки математических функций системы MATLAB 314
Изолированные от matlab.exe приложения Windows 318
Приложение **322**
Создание новых типов данных. Классы и объекты 322

Часть 1

Вычисления и визуализация

Глава 1

Числовые массивы в системе MATLAB

Рабочее пространство системы MATLAB и ее командное окно

После запуска программы MATLAB на дисплее компьютера появляется ее главное окно, содержащее меню, инструментальную линейку с кнопками и клиентскую часть окна со *знаком приглашения* >>. Это окно принято называть *командным окном* системы MATLAB.

Теперь можно вводить с клавиатуры числа, имена переменных и знаки операций, что в совокупности составляет некоторое выражение. Имена переменных должны начинаться с буквы и состоять из букв, цифр и знаков подчеркивания. MATLAB распознает в именах переменных до 31 символа (а остальные игнорирует) и различает регистр символов. Простейшими знаками операций являются всем хорошо известные знаки арифметических операций + и -. Знак = соответствует операции присваивания. Нажатие клавиши Enter заставляет систему MATLAB вычислить выражение и показать результат, как это изображено на рис. 1.1.

В командном окне показываются вводимые с клавиатуры числа, переменные, результаты вычислений. Обычно вычисления повторяются многократно: вводятся с клавиатуры новые числовые данные и новые символьные выражения. В результате в командном окне не хватает свободного места и автоматически производится так называемая *вертикальная протяжка* (по-английски – *scrolling*) – все строки сдвигаются на одну позицию вверх, так что самая верхняя строка покидает область видимости, а в самом низу окна появляется свободная строка для ввода новых данных. Очевидно, что эта строка содержит знак приглашения >>.

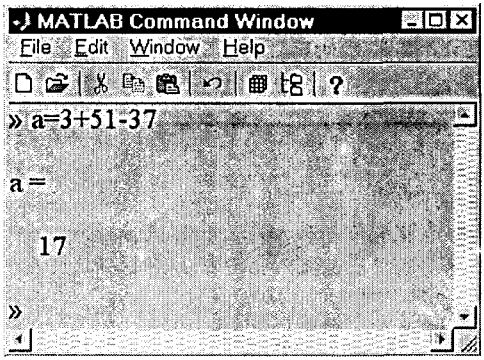


Рисунок 1.1

Та информация, что покинула видимую часть окна, никуда не исчезает. Ее всегда можно просмотреть снова, если осуществить вертикальную протяжку содержимого командного окна стандартным графическим средством управления – *полосой протяжки* (по-английски – scrollbar). Для этого нужно щелкнуть левой клавишей мыши на этой полосе или «протащить» с помощью мыши ползунок полосы протяжки в нужном направлении (вверх или вниз) (см. рис. 1.2).

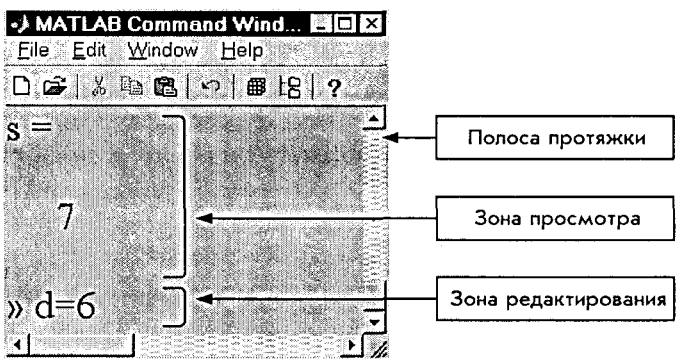


Рисунок 1.2

Можно также осуществлять протяжку содержимого командного окна системы MATLAB с помощью следующих клавиш клавиатуры: PageUp, PageDown, Ctrl+Home (одновременное нажатие клавиш Ctrl и Home) и Ctrl+End.

Клавиши «Стрелка вверх» и «Стрелка вниз», в любом текстовом редакторе осуществляющие перемещение курсора вверх-вниз и вертикальную протяжку содержимого окна, в системе MATLAB работают по-другому. Эти клавиши позволяют вернуть в строку ввода ранее введенные с клавиатуры команды и другую входную информацию. Вся эта информация запоминается в специальной области памяти, которую называют *стеком команд*, так как самая последняя входная информация при ее прокрутке клавишей «Стрелка вверх» появится первой. Затем появится предпоследняя команда и т. д. Клавиша «Стрелка вниз» осуществляет прокрутку команд в противоположном направлении.

В итоге можно сказать, что вся видимая информация в окне системы MATLAB располагается в двух принципиально разных зонах: *зоне просмотра* и *зоне редактирования*.

В зоне просмотра уже ничего нельзя исправить; хотя в нее и можно поместить курсор, однако реакцией на ввод с клавиатуры будет автоматическое перемещение курсора (то есть точки ввода) в строку ввода, расположенную в зоне редактирования. В зоне просмотра можно выделять (селектировать) с помощью мыши любую информацию и копировать ее в буфер обмена (Clipboard) операционной системы Windows, чтобы потом вставить ее либо в документ текстового редактора (например, Microsoft Word), либо в строку ввода.

Зона редактирования обычно занимает одну (последнюю) строку командного окна системы MATLAB, в которой находится знак приглашения >>. Ее мы и называем *строкой ввода*. Однако при необходимости эту «логическую строку» можно распространить на несколько физических строк командного окна. Для этого нельзя просто нажать клавишу Enter, так как при этом ввод информации будет закончен и MATLAB приступит к вычислениям и дальнейшему показу результата. Поэтому для продления ввода с показом вводимой информации на следующих физических строках требуется нажать Enter только после трех или более точек, что и показано на рис. 1.3.

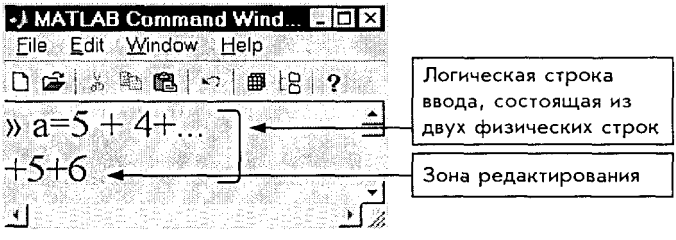


Рисунок 1.3

Однако и в этом случае зона редактирования распространяется только на самую последнюю строку (теперь она уже не содержит знака приглашения >>), а в предыдущих физических строках логической строки ввода изменить уже ничего нельзя. Кроме того, суммарная длина логической строки ввода ограничена 256 символами.

Все значения переменных, вычисленные в течение текущего сеанса работы, сохраняются в специально зарезервированной области памяти компьютера, называемой *рабочим пространством системы MATLAB* (по-английски – *Matlab Workspace*).

Командой

```
clc
```

можно стереть видимое содержимое командного окна системы MATLAB, однако это не затронет содержимого рабочего пространства. Действительно,

если после этого набрать имя ранее вычисленной переменной a, то после нажатия клавиши Enter мы снова увидим ее значение:

```
>>a
a =
  20
```

По мере разбухания размера рабочего пространства эффективность работы будет снижаться. Такое разбухание вполне вероятно, так как система MATLAB может работать с данными гигантских размеров. Поэтому, когда исчезает необходимость в хранении ряда переменных в текущем сеансе работы, их можно стереть из памяти компьютера командой

```
clear имя1 имя2 ...
```

удаляющей из рабочего пространства переменные с именами имя1 и имя2. Чтобы удалить сразу все переменные, нужно использовать команду

```
clear
```

Если вы хотите проверить, какие переменные остались в рабочем пространстве, то для этого нужно выполнить команду

```
who
```

которая выведет список всех переменных, входящих на данный момент в рабочее пространство системы MATLAB (см. рис. 1.4).

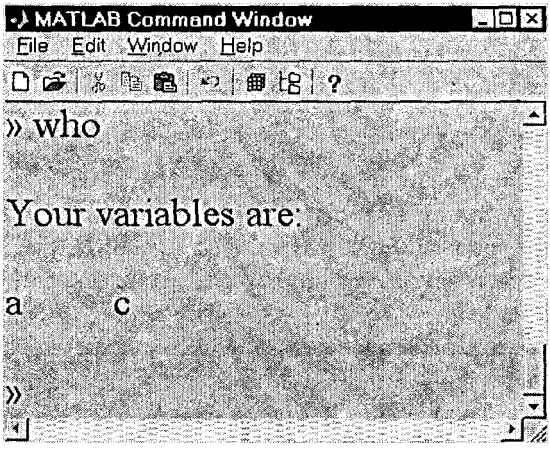


Рисунок 1.4

Для просмотра значения любой переменной из текущего рабочего пространства системы MATLAB достаточно набрать ее имя и нажать клавишу Enter.

После закрытия сеанса работы с системой MATLAB все ранее вычисленные переменные теряются. Чтобы сохранить в файле на диске компьютера содер-

жимое рабочего пространства системы MATLAB, нужно выполнить команду меню

```
File | Save Workspace As...
```

после чего появляется стандартное диалоговое окно операционной системы Windows для выбора каталога на диске и имени файла. Расширение имени файла должно быть mat, поэтому такие файлы принято называть *MAT-файлами*.

Вместо рассмотренной команды меню можно непосредственно в командном окне системы MATLAB набрать команду

```
save путь_к_файлу\имя_MAT-файла
```

и нажать клавишу Enter. Результат будет тот же самый.

В последующих сеансах работы для загрузки в память компьютера ранее сохраненного на диске рабочего пространства нужно выполнить команду меню

```
File | Load Workspace...
```

которая в диалоговом окне Load.mat file потребует указать нужный MAT-файл.

Выполнив эту команду несколько раз с разными файлами, мы можем соединить в текущем рабочем пространстве системы MATLAB содержимое нескольких предыдущих сеансов работы! Однако если имена переменных из разных сеансов совпадают, то в текущем рабочем пространстве будет представлена лишь переменная из последнего открытого MAT-файла.

Вместо рассмотренной команды меню можно набрать команду

```
load имя_MAT-файла
```

непосредственно в командном окне системы MATLAB. Можно также из записанного на диске MAT-файла считать в рабочее пространство значения отдельных переменных. Для этого нужно предыдущую команду дополнить именами переменных:

```
load имя_MAT-файла имя1, имя2, ...
```

В результате из MAT-файла будут считаны переменные с именами имя1, имя2 и т. д. Если MAT-файл указан без полного пути к нему, то он должен находиться в *текущем каталоге* системы MATLAB, который всегда можно узнать с помощью команды cd, а изменить его можно командой

```
cd путь_к_новому_каталогу
```

В каждом сеансе работы с системой MATLAB целесообразно в качестве текущего каталога задавать тот каталог, с файлами которого предстоит работать чаще всего.

Заканчивая краткий рассказ про командное окно системы MATLAB, отметим следующие особенности ее команд. Под командами пользователя мы понимаем предписания системе MATLAB выполнить некоторое действие, например показать текущий каталог. Одни команды системы MATLAB могут задаваться раз-

ными способами: с помощью меню главного (командного) окна, с помощью кнопок на полосе инструментов и с помощью ввода с клавиатуры ключевых (зарезервированных) слов с последующим нажатием клавиши Enter. Другие команды можно реализовать только с помощью ввода с клавиатуры соответствующих им ключевых слов (например, команда cd).

Часть команд требует дополнительной информации от пользователя; например, команда clear использует имена переменных, подлежащих удалению из текущего рабочего пространства. Эту дополнительную информацию обычно указывают в командной строке через пробел после ключевого слова. Однако есть и еще одна возможность – заключить дополнительные параметры в круглые скобки. Например, ранее рассмотренную команду удаления переменных из рабочего пространства можно записать и в другом виде:

```
clear( 'имя1', 'имя2' )
```

имеющим форму функционального вызова. Про эти две возможности говорят как о *дуальности* формы вызова команд системы MATLAB. Существуют случаи, когда функциональная форма вызова команд предпочтительнее. Здесь же отметим, что при функциональной форме вызова команд явно заданные имена надо заключать в *апострофы*.

Мы пока изучили не все команды системы MATLAB. В дальнейшем мы будем изучать их по мере необходимости в соответствии с излагаемым материалом. При этом будут изучены самые основные и часто применяемые на практике команды. По любой команде системы MATLAB можно получить быструю справку, выполнив команду

```
help имя_команды
```

Работая с командным окном системы MATLAB, то есть вводя команды, задавая числовые значения переменных и конструируя различные математические выражения, подлежащие вычислению, а также вызывая многочисленные встроенные в систему MATLAB математические функции, легко можно выполнить серьезные вычисления и визуализировать результаты. Такой режим работы мы назовем *интерактивным*. Это одновременно и простой, и продуктивный вариант работы с системой MATLAB.

Если встроенных, чрезвычайно обширных возможностей системы MATLAB все же окажется недостаточно для решения конкретной задачи, пользователь может самостоятельно запрограммировать необходимые для решения проблемы функции. Это можно выполнить как на внутреннем M-языке системы MATLAB, так и на языках Fortran, C и C++. Вопросы, касающиеся *программирования*, будут подробно рассмотрены в ч. 2 настоящего пособия. В первой же части пособия мы будем подробно знакомиться со всеми возможностями интерактивного режима.

Вещественные числа и тип данных double

Основным типом данных, с которым производятся вычисления в среде MATLAB, являются конечные десятичные дроби, приближающие с заданной точностью произвольные вещественные числа. Последние в общем случае представимы лишь в виде бесконечных десятичных дробей. Можно сказать, что MATLAB работает с вещественными числами приближенно.

Вещественные числа задаются в системе MATLAB мантиссой и показателем степени и записываются в следующем виде:

```
2.851038547e+12; -456.38456978; 0.0045692e0; 185e-1; 4.5; -123
```

где буквой e обозначается основание степени, равное 10.

У целых чисел отсутствуют дробные части, но они все равно представляются системой MATLAB на машинном уровне в той же форме, что и дробные числа. Этот основной тип данных называется double.

Под мантиссу и показатель степени (на машинном уровне используется двоичная система записи) отводится 8 байт памяти. В результате для десятичных чисел достигается точность порядка 15 значащих цифр. При этом максимальным по модулю представимым в системе MATLAB вещественным числом является

```
1.797693134862316e+308
```

а минимальным по модулю является следующее вещественное число:

```
2.225073858507202e-308
```

Для этих чисел даже зарезервированы имена realmax и realmin.

Чтобы не перегружать излишними подробностями свое командное окно, MATLAB по умолчанию использует формат short для вывода вещественных чисел, при котором показываются только четыре десятичные цифры после запятой (см. рис. 1.5).

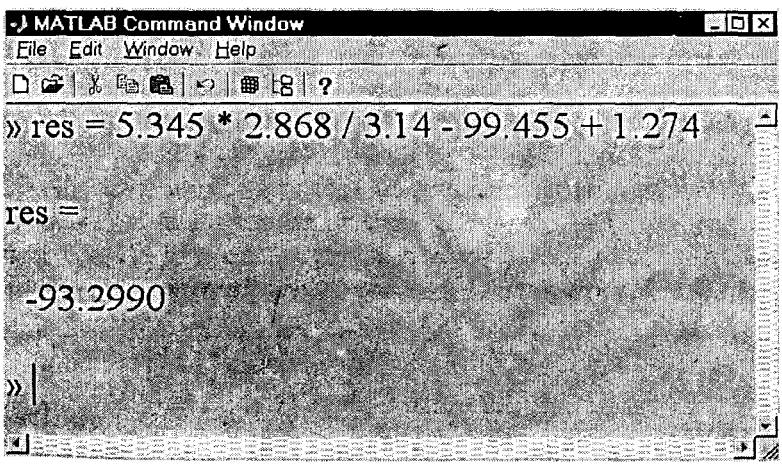


Рисунок 1.5

Если требуется полное представление вещественных чисел, то нужно ввести с клавиатуры команду

```
format long
```

после чего набрать имя переменной res, в которую выше был записан результат вычислений. Нажав клавишу Enter, получим более подробную информацию:

```
res =
   -93.29900636942675
```

Теперь все результаты вычислений будут показываться с такой высокой точностью в течение данного сеанса работы в среде системы MATLAB. Если требуется до прекращения текущего сеанса работы вернуться к старой точности визуального представления вещественных чисел в командном окне, нужно ввести и исполнить (нажав клавишу Enter) команду

```
format short
```

Другим интересным форматом является формат показа вещественных чисел в виде обыкновенных дробей, для чего вводится команда

```
format rat
```

После выполнения этой команды ранее вычисленная переменная res показывается в командном окне системы MATLAB в следующем виде:

```
res =
  -9050/97
```

Ну и, наконец, если операнды и результаты вычислений являются целыми, то, хотя они и представляются в памяти машины так же, как и дробные числа, визуально в командном окне MATLAB они показываются в виде целых чисел. Это иллюстрируется на рис. 1.6, на котором специальным именем ans система MATLAB обозначила результат вычисления выражения, поскольку он не был присвоен никакой переменной.

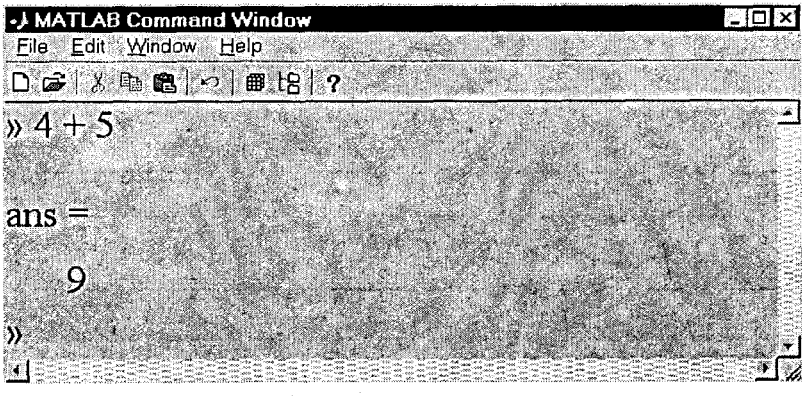


Рисунок 1.6

Итак, система MATLAB всегда хранит в переменной ans последнее из вычисленных и не сохраненных пользователем выражений.

Над вещественными числами (их машинное представление часто называют числами с плавающей запятой) и переменными типа double производятся арифметические операции сложения, вычитания, умножения и деления, для которых в системе MATLAB используются традиционные для любого языка программирования знаки +, -, * и /. Кроме того, есть еще операция возведения в степень, обозначаемая знаком ^. Результаты применения этой операции показаны ниже:

```
5 ^ 2
ans =
25
t = ans ^ ( 0.5 )
t =
  5
```

Приоритет в выполнении арифметических операций обычный: сначала (то есть самый высший приоритет) – возведение в степень, затем – умножение и деление и потом – сложение и вычитание. Операции одинакового приоритета выполняются в порядке слева направо, но круглые скобки могут изменить этот порядок.

Начав рассказывать о точных правилах использования операций и записи выражений с операндами типа double (вещественными числами), мы фактически стали описывать *внутренний язык программирования* системы MATLAB, который принято называть *М-языком*. По мере дальнейшего знакомства вы обнаружите, что этот язык весьма традиционен для языков высокого уровня (не требует знакомства с устройством памяти компьютера и его аппаратной конфигурацией) и по своей простоте приближается к широко известному языку программирования BASIC. В то же время он специально сконструирован для решения математических задач и в нем много специфических операций, позволяющих эффективно решать именно такие задачи.

Помимо арифметических операций над операндами типа double выполняются еще *операции отношения* и *логические операции*.

Операции отношения сравнивают между собой два операнда по величине. Эти операции записываются следующими знаками или комбинациями знаков:

<	<=	>	>=	==	~=
Меньше	Меньше или равно	Больше	Больше или равно	Равно	Не равно

В случае истинности операции отношения ее величина (то есть результат вычисления выражения) равна 1, а в случае ложности – 0. Вычисление выражений с операциями отношения показано на рис. 1.7.

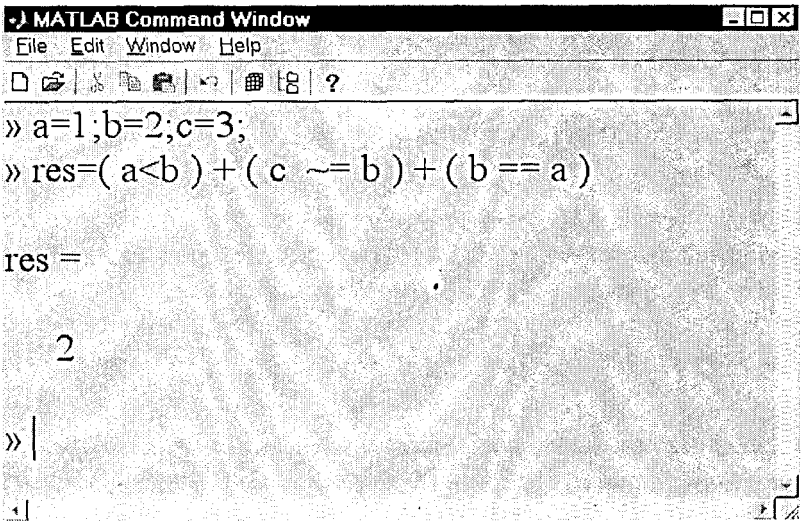


Рисунок 1.7

Здесь выражение a<b вырабатывает единицу в силу того, что величина переменной a действительно меньше величины переменной b («истина»). Выражение c ~= b является истинным, так как на самом деле c, равное 3, не равно b, которое равно 2. В итоге оно вырабатывает значение 1. Последнее выражение, b == a, не является истинным и вырабатывает 0. В результате переменная res, равная сумме значений этих трех выражений, оказывается равной 2.

Операции отношения имеют *более низкий приоритет*, чем арифметические операции, поэтому рассмотренная выше переменная res равна сумме значений трех операций отношения только потому, что мы заключили эти операции в круглые скобки. Очень важно всегда помнить об этом, так как отсутствие круглых скобок может привести к изменению результата. Например, если a = 1, b = 1, c = 3, то выражение

```
c + ( b == a )
```

равно 4, в то время как выражение без круглых скобок

```
c +   b == a
```

равно 0.

Теперь обратим внимание на важную роль *точки с запятой* в М-языке системы MATLAB. В нем точка с запятой может использоваться для разных целей. Когда мы вводим с клавиатуры некоторое выражение (оно расположено после знака приглашения >>) и нажимаем клавишу Enter, MATLAB производит

вычисление этого выражения и выводит результат в свое командное окно. Если мы не хотим тотчас же видеть результат вычислений (это характерно, например, для промежуточных результатов), то в конце введенного выражения следует поставить точку с запятой и только после этого нажать Enter. Таким образом, точка с запятой *подавляет вывод* результатов вычислений в командное окно системы MATLAB.

Кроме того, если мы хотим за один раз, то есть одним нажатием клавиши Enter, вычислить несколько разных выражений, а их значения присвоить разным переменным, то эти выражения следует отделить друг от друга точкой с запятой. Таким образом, точка с запятой работает и как *разделитель*.

Имеются и другие варианты применения точки с запятой в М-языке системы MATLAB, о которых мы расскажем позже.

Логические операции над вещественными числами обозначаются знаками, перечисленными в следующей таблице:

&	|	~
И	ИЛИ	НЕ

Первые две из этих операций являются двухоперандными (бинарными), а операция «НЕ» является унарной (одноопе рандной). Знак ~ ставится перед операндом, а знаки & и | ставятся между операндами.

Логические операции трактуют свои операнды как «истинные» (не равные нулю) или «ложные» (равные нулю). Если оба операнда операции «И» истинны (не равны нулю), то результат этой операции равен 1 («истина»); во всех остальных случаях операция «И» вырабатывает значение 0 («ложь»). Операция «ИЛИ» вырабатывает 0 («ложь») только в случае, когда являются ложными (равными нулю) оба операнда. Наконец, операция «НЕ» инвертирует «ложь» на «истину» и наоборот. То есть если ее операндом является ненулевое число, то эта операция вырабатывает 0, а если операнд нулевой, то тогда результатом применения операции «НЕ» будет 1.

Логические операции имеют *самый низкий приоритет*.

В одном и том же выражении можно использовать все перечисленные операции: арифметические, логические и операции сравнения. Последовательность выполнения операций определяется их расположением внутри выражения, их приоритетом и наличием круглых скобок.

В системе MATLAB присутствуют все основные элементарные функции для вычислений с вещественными числами: степенные, показательные, тригонометрические и обратные к ним. Любая функция характеризуется своим именем, списком входных аргументов (перечисляются через запятую и стоят внутри круглых скобок, следующих за именем функции) и вычисляемым (возвращаемым) значением.

Сначала рассмотрим логическую функцию xor, дополняющую ранее рассмотренный набор логических операций. Эта функция имеет два входных аргумента и вычисляет над ними операцию «исключающее ИЛИ», которая вырабатывает 1 («истина») только в случае, когда один из числовых аргументов истинен (не равен нулю), а другой ложен (равен нулю). Например,

```
a = 1; b = 0;
xor( a, b )
ans =
    1
```

а если оба аргумента истинны или оба ложны, то эта функция вырабатывает 0:

```
a=1; b=1;
xor( a, b )
ans =
    0
```

Помимо операции возведения в степень, реализуемой с помощью знака ^, есть еще функция извлечения квадратного корня sqrt, функция exp для возведения в степень числа e, функция pow2 для возведения в степень числа 2. Также присутствуют обратные к ним функции: log – натуральный логарифм, log10 – логарифм по основанию 10, log2 – логарифм по основанию 2.

В системе MATLAB можно получить справочную информацию по любой элементарной функции, выполнив команду

```
help имя_функции
```

Тригонометрические функции представлены весьма полно: sin, cos, tan (тангенс), cot (котангенс), asin (арксинус), acos (арккосинус), atan (арктангенс), acot (арккотангенс). Имеются также и менее употребительные функции типа секанса, косеканса, а также гиперболические функции.

Для примера вычислим выражение 2 * asin(1), включающее вычисление функции asin, и получим следующий результат:

```
ans =
    3.1416
```

соответствующий числу ¶. В системе MATLAB для числа ¶ есть специальное обозначение pi.

Упомянем еще функции, связанные с целочисленной арифметикой. Например, функции округления: round (округление до ближайшего целого), fix (усечение дробной части числа), floor (округление до меньшего целого), ceil (округление до большего целого).

Кроме того, есть еще функции mod (остаток от деления с учетом знака), rem (остаток в смысле модульной арифметики), sign (знак числа), factor (разложе-

ние числа на простые множители), isprime (истинно, если число простое), primes (формирование списка простых чисел), rat (приближение числа в виде рациональной дроби), lcm (наименьшее общее кратное), gcd (наибольший общий делитель).

Функции mod и rem дают одинаковый результат для положительных аргументов. В частности,

```
mod( 7, 2 ) == rem( 7, 2 ) == 1
```

но для операций с аргументами разных знаков они вырабатывают разные значения:

```
mod( -7, 2 ) =  1  ;  rem( -7, 2 ) = -1
```

В общем случае эти функции связаны с функциями округления следующим образом:

```
rem( x, y ) == x - y * fix( x / y )
mod( x, y ) == x - y * floor( x / y )
```

И наконец, есть функции, решающие стандартные задачи комбинаторики: функция perms вычисляет число перестановок, а функция nchoosek – число сочетаний. Например, число сочетаний из 10 по 3 легко находится вызовом функции nchoosek(10, 3) (см. рис. 1.8).

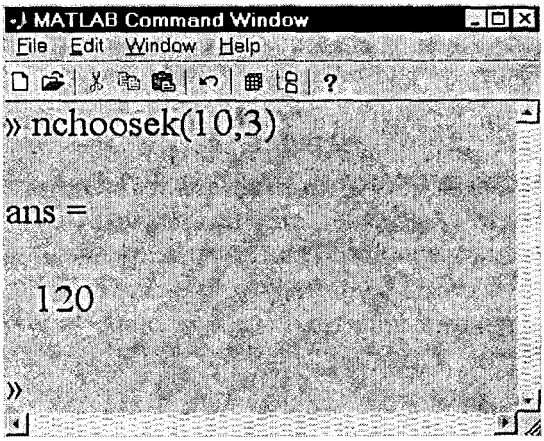


Рисунок 1.8

Многие из перечисленных функций имеют область определения, отличную от R (множество всех действительных чисел). Когда для функции задается недопустимое значение аргумента или совершается попытка выполнить недопустимую операцию, в командном окне появляется предупреждающее сообщение. Например, сообщение

```
Warning: Divide by zero.
```

появляется при попытке деления на нуль. А в качестве результата выводится

```
ans =
    Inf
```

где `Inf` символизирует бесконечность. Тот же результат получается при попытке вычислить логарифм от нуля.

Комплексные числа и комплексные функции

Система MATLAB осуществляет вычисления как с вещественными, так и с комплексными числами. При этом никакой специальной маркировки, то есть предварительного описания числовых переменных, не требуется. Это справедливо и для вещественных, и для комплексных переменных. Все они автоматически имеют тип `double`. Естественно, что для записи комплексного числа требуется в два раза больше памяти компьютера, чем для записи вещественного числа, поскольку по 8 байт памяти отводится как для действительной, так и для мнимой части комплексного числа.

Комплексные числа в системе MATLAB записываются следующим образом:

```
2 + 3i ; -6.789 + 0.834e-2 * i ; 4 - 2j
```

Отсюда видно, что для записи *мнимой единицы* зарезервированы (на выбор) буквы `i` или `j`.

Система MATLAB производит комплексные вычисления, когда явно задаются комплексные значения для переменных или аргументов функций. Однако возможны случаи, при которых система MATLAB сама выходит в область комплексных чисел без специального на то указания со стороны пользователя. Это происходит, когда для математической функции задается вещественный аргумент, не позволяющий получить вещественное значение функции. Например, при вычислении функции `sqrt( -1 )` получается следующий результат (см. рис. 1.9).

Ясно, что вычислить `sqrt( -1 )`, оставаясь в рамках только вещественных чисел, нельзя. Поэтому система MATLAB перешла к комплексным вычислениям и получила результат, равный `i`. При этом, как мы видим из рисунка, самой системой MATLAB для обозначения мнимой единицы используется именно буква `i` (а не `j`).

В случае, когда коэффициентом при мнимой единице является не число, а переменная, нельзя писать просто `x + iy`, а нужно обязательно использовать знак умножения, то есть `x + i * y`.

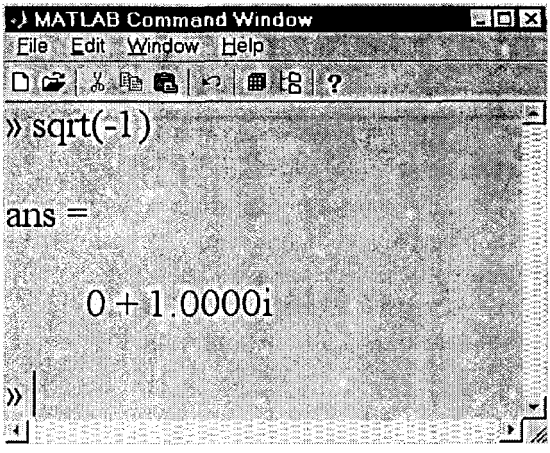


Рисунок 1.9

Почти все элементарные функции допускают вычисления с комплексными аргументами. Например:

```
res = sin( 2 + 3i ) * atan( 4i ) / ( 1 - 6i )
res=
   -1.8009 - 1.9190i
```

Научившись выполнять вычисления с комплексными функциями, мы для примера можем проверить знаменитую формулу Эйлера

```
exp( i * x ) = cos( x ) + i * sin( x )
```

Придавая вещественной переменной x различные значения, вычисляем выражения из правой и левой частей этой формулы. Например, при `x = 1` имеем как для левого, так и для правого выражения одно и то же значение: `0.5403 + + 0.8415i`. Для `x = 2` обе стороны этого равенства дают снова одинаковый результат `-0.4161 + 0.9093i`, и т. д.

Но в очевидных случаях о комплексных аргументах не может быть и речи. Действительно, какие могут быть комплексные аргументы у функции `mod(x,y)`, вычисляющей остаток от деления x на y? Поэтому появление сообщения об ошибке в следующем примере абсолютно предсказуемо (см. рис. 1.10).

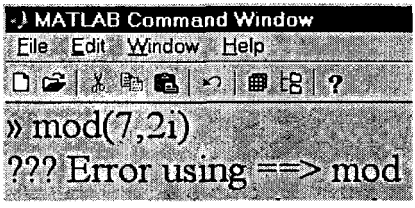


Рисунок 1.10

Специально для работы с комплексными числами предназначены следующие функции: abs (абсолютное значение комплексного числа), conj (комплексно сопряженное число), imag (мнимая часть комплексного числа), real (действительная часть комплексного числа), angle (аргумент комплексного числа), isreal («истина», если число действительное).

Комплексные числа можно складывать, вычитать, перемножать и делить точно так же, как и вещественные числа. Поэтому в отношении арифметических операций ничего нового для комплексных чисел (по сравнению с вещественными) сказать невозможно. То же самое относится и к операциям отношения «равно» и »не равно». Другое дело – остальные операции отношения и логические операции.

В математике операции сравнения комплексных чисел на «больше-меньше» не определены: нельзя сказать, что одно комплексное число, например, больше другого. Исходя из этого, можно было бы предположить, что такие операции сравнения для комплексных операндов запрещены. Однако разработчики системы MATLAB реализовали более «щадящий» вариант. Эти операции работают и с комплексными операндами, но их результат вырабатывается исходя только из действительных частей этих операндов. Например,

```
c = 2 + 3i; d = 2i;
c > d
ans =
   1
c = 5i;
c <= d
ans =
   1
c >= d
ans =
   1
```

Логические операции трактуют операнды как ложные, если они равны нулю. Если же у комплексного операнда не равна нулю хотя бы одна его часть (вещественная или мнимая), то такой комплексный операнд трактуется как истинный. Отсюда вытекают результаты следующих логических операций:

```
c = 2 + 3i; d = 2i;
~c
ans =
   0
c & d
ans =
   1
```

В итоге не требуется жесткого контроля со стороны пользователя за типом числовых операндов. В одном выражении разрешается смешивать вещественные и комплексные операнды. Все же если нужно в процессе вычислений формально определить, является ли переменная комплексной (речь идет о ее значении), то можно вызвать функцию

```
isreal( x )
```

возвращающую «истину» (то есть 1), если числовая переменная x не является комплексной, и »ложь» в противном случае.

В заключение еще раз подчеркнем, что никаких специальных соглашений об именах комплекснозначных переменных и комплексных функций не существует. Переменные не требуют никакого предварительного описания. Все вычисления перетекают из вещественной области в комплексную абсолютно автоматически. Это происходит при задании комплексных операндов (используются зарезервированные имена i или j для мнимой единицы) или при невозможности ограничиться лишь вычислениями с действительными числами (как в случае вычисления квадратного корня из минус единицы).

Формирование одномерных числовых массивов

В среде MATLAB можно производить вычисления с набором вещественных (или комплексных) чисел так же легко, как и с одиночными числами. Это является одним из самых заметных и важных преимуществ системы MATLAB над другими программными пакетами, ориентированными на вычисления и программирование.

Именованные наборы чисел в различных языках программирования традиционно называют *массивами*. Всему массиву присваивается одно имя, а доступ к отдельным *элементам массива* осуществляется по целочисленному *индексу*, то есть по номеру элемента в массиве.

В зависимости от количества индексов, с помощью которых осуществляется доступ к отдельным элементам, массивы разделяются на *одномерные* (единственный индекс), *двумерные* (два индекса) и массивы больших размерностей (три индекса и более). Последние принято называть *многомерными* массивами.

Сначала рассмотрим одномерные числовые массивы. Это линейные наборы чисел, в которых позиция каждого элемента задается единственным числом – его номером. Можно говорить о первом элементе массива, о втором и т. д.

Массивы в системе MATLAB не образуют никакого нового типа данных. Числовые массивы (вещественные или комплексные) являются массивами элементов типа double. Настало время сообщить, что в системе MATLAB даже переменные, принимающие единственное числовое значение, то есть являющиеся по существу скалярами, в своем внутреннем представлении являются массивами,

состоящими из единственного элемента. Помимо памяти, необходимой для хранения числовых элементов (по 8 байт на каждый в случае вещественных чисел и по 16 байт в случае комплексных чисел), MATLAB автоматически при создании массивов выделяет еще и память для управляющей информации. В этой области памяти хранится размерность массива, количество элементов по каждой размерности, тип элементов (вещественные или комплексные) и т. д. Очень важно, что при построении массивов система MATLAB не требует от пользователя сразу же сообщить всю информацию. Пользователь может вводить ее постепенно, а MATLAB реагирует на нее соответственно и может даже динамически перестраивать структуру массива.

Для создания одномерного массива можно использовать *операцию конкатенации*. Эта операция обозначается с помощью квадратных скобок `[]`. Например, следующее выражение, использующее операцию конкатенации,

```
a1 = [ 1 2 3 ]
```

формирует переменную с именем `a1`, являющуюся одномерным массивом из трех элементов (вещественных чисел). При использовании операции конкатенации объединяемые в одномерный массив элементы должны располагаться между открывающей и закрывающей квадратными скобками и отделяться друг от друга либо пробелом, либо запятой. Так что выражение

```
a1 = [ 1, 2, 3 ]
```

по своему результату абсолютно идентично предыдущему. Однако если массивы состоят из комплексных чисел или элементы задаются выражениями, то с точки зрения наглядности лучше использовать в качестве *разделителя элементов* запятую, как в следующем примере, в котором создается массив комплексных чисел:

```
d = [ 1 + 2i, 2 + 3i, 3 - 7i ];
```

Для доступа к индивидуальному элементу одномерного массива нужно применить *операцию индексации*, для чего после его имени указать в круглых скобках индекс (номер) элемента. В итоге третий элемент массива `a1` обозначается как `a1(3)`, первый элемент – как `a1(1)`, второй элемент – как `a1(2)`.

Если требуется изменить третий элемент сформированного выше операцией конкатенации массива `a1`, то можно применить операцию индексации и операцию присваивания:

```
a1(3) = 789
```

Далее, пусть, к примеру, второй элемент массива `a1` должен стать равным среднему арифметическому первого и третьего элементов. Для этого выполняем следующее действие:

```
a1(2) = ( a1(1) + a1(3) ) / 2
```

Количество элементов в одномерном массиве всегда можно узнать с помощью функции length:

```
length( a1 )
ans =
    3
```

Как мы только что убедились на рассмотренных примерах, операцию индексации можно применять как справа от знака операции присваивания, так и слева от него. Про эти случаи говорят, что осуществляется доступ к элементу массива «по чтению» или «по записи». При попытке чтения несуществующего элемента (например, четвертого элемента массива a1) в командном окне появится сообщение об ошибке (см. рис. 1.11).

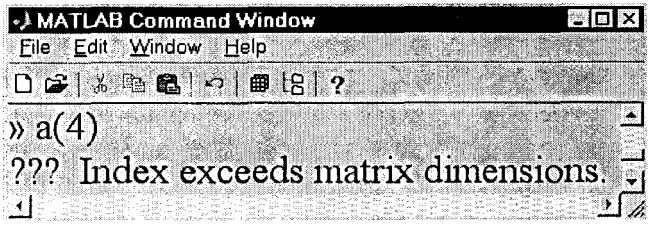


Рисунок 1.11

В этом сообщении утверждается, что индекс превысил размер массива.

В то же время запись несуществующего элемента вполне допустима – она означает добавление нового элемента к уже существующему массиву:

```
a1(4) = 7
```

Применяя после выполнения этой операции к массиву a1 функцию length, находим, что количество элементов в массиве возросло до четырех:

```
length( a1 )
ans =
    4
```

То же самое действие – «удлинение массива a1» – можно выполнить и с помощью операции конкатенации:

```
a1 = [ a1 7 ]
```

Здесь операндами операции конкатенации являются массив a1, состоящий из трех элементов, и добавляемый к нему четвертый элемент, равный 7. Можно подвергнуть конкатенации и несколько массивов. Например, следующий код:

```
a2 = [ a1 a1 3 a1 ]
```

порождает одномерный массив a2, состоящий из 13 элементов: его первые четыре элемента повторяют элементы массива a1, элементы с пятого по восьмой

делают то же самое, девятый элемент равен числу 3 и, наконец, последние четыре элемента опять совпадают с соответствующими элементами массива a1 (см. рис. 1.12).

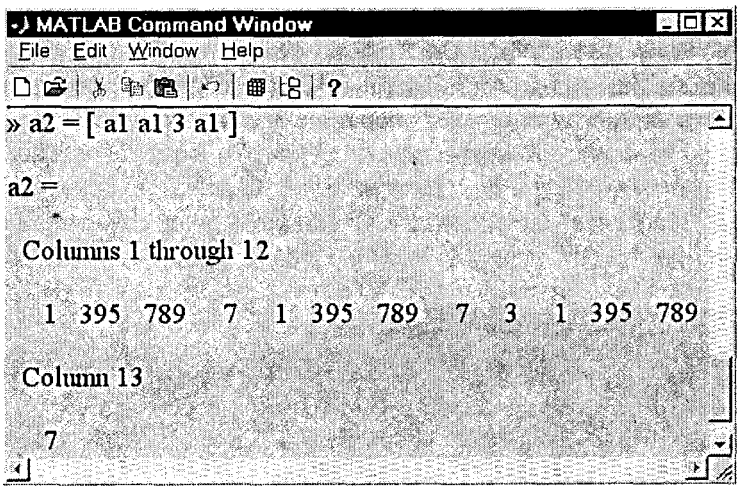


Рисунок 1.12

Теперь создадим одномерный массив a3 без применения операции конкатенации. Используем другой способ: будем прописывать каждый элемент создаваемого массива по отдельности:

```
a3(1) = 67
a3(2) = 7.8
a3(3) = 0.017
```

Такое постепенное создание массива из трех элементов возможно потому, что система MATLAB с каждым новым присваиванием автоматически перестраивает свою служебную информацию о массиве, а также область памяти, отводимую под данные (под элементы). После первого присваивания MATLAB считает, что массив a3 состоит из одного элемента. При втором присваивании выясняется, что есть еще и второй элемент. Тут уже система MATLAB вынуждена перестраивать всю структуру памяти, отведенную под данный массив. С каждым последующим присваиванием перестройку приходится повторять.

Ясно, что этот способ создания одномерного массива не является эффективным и проигрывает в быстродействии операции конкатенации. Проигрыш в быстродействии мало заметен в интерактивном режиме, когда пользователь вводит всю информацию с клавиатуры. Однако это становится критическим в программном режиме, когда MATLAB подряд исполняет многочисленные инструкции с массивами.

Существуют два простых способа приблизительно в 100 раз увеличить быстродействие в рассмотренной ситуации. Во-первых, можно *предварительно выделить* всю необходимую *память* под конечный размер массива. Это достигает-

ся вызовом функций ones или zeros, которые сразу создают массив нужного размера, заполненный единицами или нулями. После этого постепенное прописывание элементов нужными значениями не требует перестройки структуры памяти, отведенной под массив. К примеру, для массива a3 можно перед присваиваниями сделать следующий вызов функции ones:

```
a3 = ones( 1, 3 )
a3 =
   1 1 1
```

где вывод в командное окно результата вызова функции ones показывает, что сразу создается массив из трех элементов, равных единице. После этого можно осуществить показанные выше присваивания нужных значений элементам массива a3.

Во-вторых, можно осуществить присваивание значений элементам массива, начиная с последних по номеру элементов и заканчивая первым:

```
a3(3) = 0.017
a3(2) = 7.8
a3(1) = 67
```

Здесь при выполнении первого же присваивания система MATLAB выделяет память под три вещественных числа, присваивает указанное значение третьему элементу, а первому и второму по умолчанию присваивает нули.

Теперь снова вернемся к рассмотрению различных способов создания одномерных массивов. К настоящему моменту мы изучили три таких способа: операцию конкатенации, операцию индексации, вызов специальных функций (например, ones или zeros). Еще один способ основан на применении специальной операции, обозначаемой *двоеточием*. Эту операцию можно назвать *операцией формирования диапазона* числовых значений. Пусть требуется сформировать одномерный массив чисел в диапазоне от 3.7 до 8.947 с приращением 0.3. Легче всего решить эту задачу с помощью операции *двоеточие*:

```
diap1 = 3.7 : 0.3 : 8.947;
```

Последняя точка с запятой здесь использована для подавления немедленного вывода в командное окно системы MATLAB результатов операции, то есть всех элементов массива diap1. В случае большого числа элементов их показ в командном окне будет сопровождаться быстрой вертикальной протяжкой содержимого окна, а это замедляет работу и утомительно для глаз.

Операция формирования диапазона работает следующим образом. Сначала она включает в формируемый массив *левую границу диапазона* (это число, стоящее левее первого двоеточия). Затем она к этому числовому значению прибавляет *приращение*, которое указывается после первого двоеточия. Если сумма не превосходит *верхней границы диапазона* (число, стоящее после второго двоеточия), то она включается в качестве элемента в формируемый массив. Это

все повторяется до тех пор, пока очередное числовое значение не превысит верхнюю границу.

Несмотря на подробное объяснение работы этой операции, довольно трудно так, сразу в уме подсчитать количество попадающих в заданный диапазон и, соответственно, в массив diap1 элементов. Поэтому лучше вызвать функцию length:

```
length( diap1 )
ans =
   18
```

и выяснить, что в сформированный с помощью операции двоеточие массив diap1 попало 18 элементов.

Чаще всего эту операцию применяют для формирования диапазона целых числовых значений:

```
diap2 = 4 : 2 : 26;
```

Если приращение равно единице, то его можно для краткости опустить:

```
diap3 = 2:45;
```

В случае целых чисел количество попадающих в заданный диапазон элементов формируемого массива подсчитывается без всякого труда. Совершенно очевидно, что в массив diap3 попадает 44 элемента.

Двумерные массивы чисел: матрицы и векторы

Двумерные массивы можно трактовать как наборы чисел, упорядоченные в виде *прямоугольной таблицы*. Для доступа к индивидуальному элементу используется два индекса – номер строки и номер столбца (на пересечении которых и стоит выбранный элемент).

Двумерный массив характеризуется количеством строк и количеством столбцов. Сформируем операцией конкатенации двумерный массив a, состоящий из двух столбцов и трех строк (см. рис. 1.13).

Из этого рисунка хорошо видно, что в качестве *разделителя строк* в формируемом с помощью операции конкатенации двумерном массиве служит *точка с запятой*. Это еще одно предназначение точки с запятой в М-языке системы MATLAB.

Двумерные массивы в математике принято называть *матрицами*. Любая строка матрицы является одномерным массивом, и любой столбец матрицы также является одномерным массивом. Однако есть разница в упорядочении их элементов с точки зрения матриц: элементы первого одномерного массива упо-

рядочены вдоль строк матрицы (горизонтально), а элементы второго – вдоль столбцов (вертикально). Если явно учитывать в понятии одномерного массива эту разницу, то тогда массивы первого типа называют *вектор-строками*, а второго типа – *вектор-столбцами*. В этом случае также можно считать, что вектор-строки являются частным случаем матрицы с количеством строк, равным единице, а вектор-столбцы являются частным случаем матрицы с количеством столбцов, равным единице.

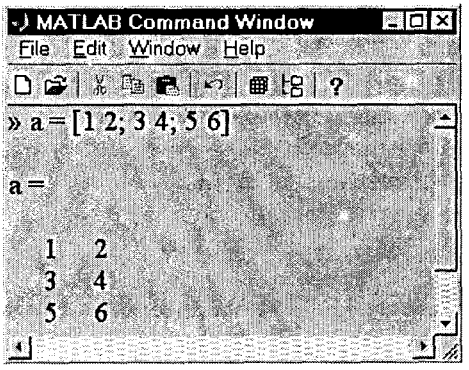


Рисунок 1.13

В системе MATLAB все одномерные массивы трактуются либо как вектор-строки, либо как вектор-столбцы. До сих пор мы вводили только вектор-строки, так как использовали в операциях конкатенации в качестве разделителей либо пробелы, либо запятые. Следующее выражение, использующее операцию конкатенации, задает уже вектор-столбец

```
b =[ 1; 2; 3]
```

состоящий из трех строк, так как точка с запятой в операции конкатенации означает переход на новую строку.

Для массива `b` функция `length(b)` возвращает число 3, так как действительно этот массив состоит из трех элементов. Функция `length` не различает вектор-строки и вектор-столбцы.

Если попросить систему MATLAB показать значение переменной `b`, то мы увидим следующую картину (см. рис. 1.14).

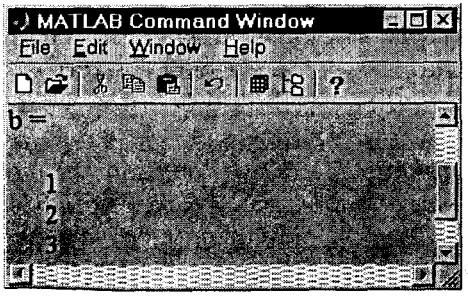


Рисунок 1.14

Таким образом, система MATLAB распознает «геометрию» этого одномерного массива и наглядно отображает его, располагая элементы массива b для показа в своем окне вертикально.

Полученную ранее матрицу a размером 3 x 2 (первым указывается число строк, вторым – число столбцов) можно сформировать также вертикальной конкатенацией вектор-строк:

```
a = [ [1 2]; [3 4]; [5 6] ]
```

или горизонтальной конкатенацией вектор-столбцов:

```
a = [ [1; 3; 5] , [2; 4; 6] ]
```

Вертикальную и горизонтальную конкатенации можно также осуществить с помощью функции cat. Для вертикальной конкатенации ее первый параметр равен 1:

```
a = cat( 1, [1 2], [3 4], [5 6] )
```

а для горизонтальной конкатенации он равен 2:

```
a = cat( 2, [1; 3; 5], [2; 4; 6] )
```

Чтобы узнать *размеры* двумерного массива и «геометрию» векторов (вектор-столбцы или вектор-строки), нужно использовать функцию size. Для рассмотренного выше двумерного массива a получается следующий результат:

```
size( a )
ans =
    3 2
```

где первым показывается число строк, а вторым – число столбцов.

Теперь применим эту функцию к одномерным массивам. Вот что из этого получается для сформированного выше вектор-столбца b, состоящего из трех строк и одного столбца:

```
size( b )
ans =
    3 1
```

Наконец, попробуем применить эту функцию к переменной, состоящей из единственного числового значения, то есть к скаляру:

```
var = 5;
size( var )
ans =
    1 1
```

Отсюда видно, что система MATLAB трактует даже скалярные по существу величины как двумерные массивы размером 1 x 1. Векторы рассматриваются как матрицы, размер которых по одному из направлений равен единице.

В системе MATLAB существует также *пустой массив*, то есть массив, не содержащий данных. Он обозначается квадратными скобками [] (между которыми нет операндов) и трактуется как матрица размером 0 x 0.

Поэтому для ранее созданных переменных a, b, var и пустого массива [] функция ndims возвратит число 2 (*размерность* массива).

Структуру созданных массивов можно также узнать с помощью команды whos, которая работает со всеми переменными из текущего рабочего пространства системы MATLAB (см. рис. 1.15).

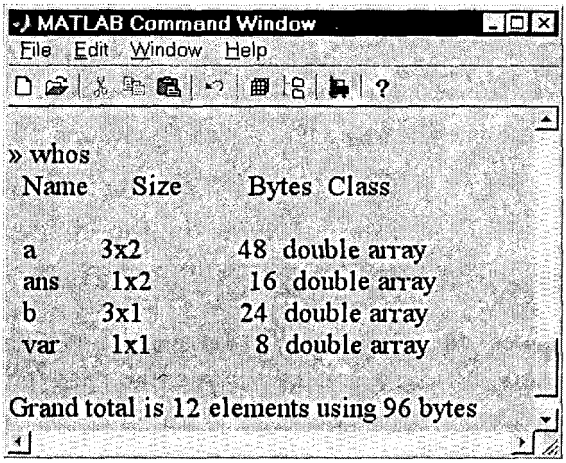


Рисунок 1.15

Итак, все, с чем работает MATLAB, является массивами различной размерности и размеров. Основным объектом встроенного в эту систему М-языка является массив. Размерность массива можно узнать функцией ndims, а размеры – функцией size. Тип массива определяется типом его элементов. Мы пока работаем с числовыми массивами типа double, элементами которых служат вещественные или комплексные числа (точнее, их приближенные машинные представления).

Продолжим рассмотрение способов создания двумерных числовых массивов (матриц). Как и рассмотренные ранее одномерные массивы (векторы), двумерные массивы можно создать с помощью операции индексации, прописывая по отдельности его элементы необходимыми числовыми значениями. Например, рассмотренный ранее массив a можно создать следующим образом:

```
a(1,1) = 1; a(1,2) = 2;
a(2,1) = 3; a(2,2) = 4;
a(3,1) = 5; a(3,2) = 6;
```

где для доступа («по чтению») к отдельным элементам используются круглые скобки (операция индексации), внутри которых через запятую перечисляются индексы. Первым указывается номер строки, вторым – номер столбца.

Как и в случае одномерных массивов, это решение является неэффективным, так как по мере присваиваний системе MATLAB приходится перестраивать структуру массива. Проблема легко преодолевается, если присваивание

```
a(3,2) = 6;
```

поместить первым. Кроме того, можно сразу создать двумерный масссив нужного размера функциями ones или zeros:

```
ones( 3, 2 ) или zeros( 3, 2 )
```

а затем осуществить присваивания отдельным элементам нужных значений (причем порядок присваиваний в этом случае уже не имеет значения). У этих функций первый параметр задает число строк, а второй – число столбцов.

И наконец, если после формирования массива X потребуется, не изменяя элементов массива, изменить его размеры, можно воспользоваться функцией

```
reshape( X, M, N )
```

где M и N – новые размеры массива X (M – число строк, N – число столбцов). Возникнет ошибочная ситуация, если количество элементов в массиве X не равно произведению M на N.

К примеру, если требуется ранее сформированную матрицу a размером 3 x 2 превратить в матрицу размером 2 x 3, то вызываем функцию reshape:

```
reshape( a, 2, 3 )
ans =
   1 5 4
   3 2 6
```

Объяснить работу этой функции можно, только исходя из способа, каким система MATLAB хранит элементы массивов в памяти компьютера. Она хранит их в *непрерывной области памяти упорядоченно по столбцам*: сначала располагаются элементы первого столбца, вслед за ними расположены элементы второго столбца и т. д. Помимо собственно данных (элементов массива) в памяти компьютера хранится также управляющая информация: тип массива (например, double), размерность и размеры массива и другая служебная информация. Этой информации достаточно для определения границ столбцов. Отсюда следует, что для переформирования матрицы функцией reshape достаточно изменить только служебную информацию и не трогать собственно данные.

Поменять местами строки матрицы с ее столбцами можно операцией *транспонирования*, которая обозначается знаком ' (апостроф). Например,

```
A = [ 1 1 1; 2 2 2; 3 3 3];
B = A'
```

```
B =
  1 2 3
  1 2 3
  1 2 3
```

Для *квадратных матриц* (число строк равно числу столбцов) эта операция имеет наглядную геометрическую интерпретацию: на своих местах остаются элементы главной диагонали квадратной матрицы, а остальные *отражаются симметрично* относительно этой диагонали.

Вектор-строки операцией транспонирования преобразовываются в вектор-столбцы, и наоборот:

```
v = [ 1; 2; 3];
u = v';
u =
  1 2 3
```

В рассмотренном примере вектор-столбец `v` операцией транспонирования был преобразован в вектор-строку `u`.

Многомерные числовые массивы

Многомерными называются массивы с размерностью больше двух. Для индексации элементов таких массивов требуется три или более индекса, указывающие на положение выбираемого элемента вдоль нескольких направлений упорядочения.

Дадим наглядную иллюстрацию многомерных массивов на примере массива трех измерений (размерность равна трем). Допустим, что ежедневно в течение месяца проводятся измерения одних и тех же величин, причем значения этих величин за один день сводятся в прямоугольную таблицу. Тогда в конце месяца имеются 30 двумерных таблиц. Как упорядочить всю эту совокупность опытных данных? Для этого можно расположить эти матрицы вдоль некоторого направления и пронумеровать их. Это будет уже третье направление упорядочения данных, так как вдоль первого направления (условно – вертикального) упорядочиваются строки, вдоль второго (горизонтального) – столбцы. В результате третий индекс для доступа к отдельному элементу данных будет равен номеру таблицы вдоль *направления дней месяца*. Получившаяся совокупность упорядоченных данных иллюстрируется рис. 1.16.

Если создать трехмерный массив `A1`, содержащий упорядоченные указанным образом данные, то к индивидуальному данному можно обращаться с помощью трех индексов. Элемент `A1(1,2,2)` находится в первой строке, втором столбце и во второй матрице. Из рисунка видно, что этот элемент равен 3.

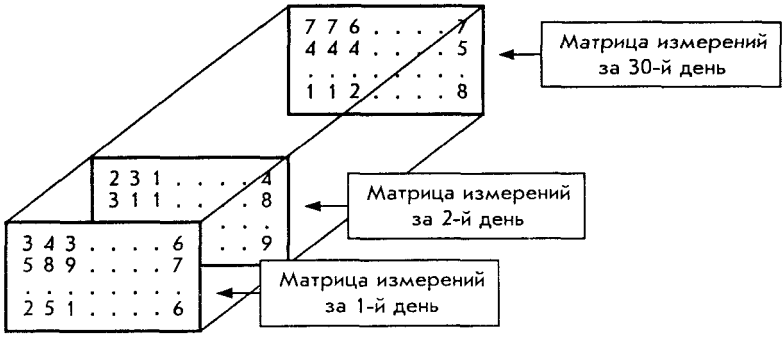


Рисунок 1.16

Теперь изучим способы создания таких массивов. Первый способ – это прямое прописывание всех его элементов:

```
A1(1,1,1) =3; A1(2,1,1)=5;...
A1(1,2,1) =4; A1(2,2,1)=8;...
...
A1(1,1,2) =2; A1(2,1,2)=3;...
A1(1,2,2) =3; A1(2,2,2)=1;...
...
A1(1,1,30)=7; A1(2,1,30)=4;...
A1(1,2,30)=7; A1(2,2,30)=4;...
...
```

Здесь для многократного увеличения эффективности целесообразно первым прописать самый правый нижний элемент 30-й матрицы, так как система MATLAB сразу получит информацию об истинном размере создаваемого трехмерного массива.

Другим способом увеличения эффективности (быстродействия), как и в ранее рассмотренных случаях векторов и матриц, является создание массива нужного размера вызовом функций `ones` или `zeros`, которые теперь должны получить три входных аргумента:

```
ones( M, N, L ) или zeros( M, N, L )
```

где `L` – это размер трехмерного массива вдоль третьего измерения. Созданные указанными функциями трехмерные массивы содержат в качестве элементов либо единицы, либо нули. Далее нужно осуществить прописывание индивидуальных элементов правильными значениями. Процесс будет эффективным, так как не потребуется перестройка структуры массива с каждым новым присваиванием.

Теперь предположим, что в трехмерном массиве `A2` собраны данные измерений за другой месяц года. Как собрать воедино данные трехмерных массивов `A1` и `A2`? Достаточно очевидным решением будет их группировка вдоль четвертого

измерения – *измерения вдоль месяцев*. Технически такая группировка является конкатенацией. Однако применить операцию конкатенации, которая обозначается квадратными скобками, невозможно, поскольку ее разделители в виде запятой (пробела) и точки с запятой допускают конкатенацию только вдоль двух первых измерений. Нам же сейчас нужно осуществить конкатенацию трехмерных массивов вдоль четвертого измерения. На помощь приходит второй из известных способов выполнения конкатенации – с помощью функции cat. Эта функция не имеет никаких ограничений и в данном конкретном примере вызывается в следующем виде:

```
A = cat( 4, A1, A2 )
```

где число 4 задает номер измерения, вдоль которого осуществляется конкатенация.

Ясно, что если потребуется группировка данных, полученных за разные годы, то можно осуществить конкатенацию вдоль пятого направления и получится пятимерный массив (массив размерности 5).

Проиллюстрируем все сказанное про многомерные массивы на следующем простом примере. Сначала создадим трехмерный массив a:

```
a = ones( 2, 3, 2 );
a( 1, 2, 2 ) = 7
a(:,:,1) =
1 1 1
1 1 1
a(:,:,2) =
1 7 1
1 1 1
```

состоящий из двух матриц размером 2 x 3, сгруппированных в третьем направлении (измерении). При показе входящих в этот трехмерный массив матриц MATLAB использует для обозначения первой из них выражение a(:,:,1), а для обозначения второй матрицы – выражение a(:,:,2). Такие обозначения не могут быть сейчас до конца поняты, так как мы еще ничего не рассказывали о применении операции двоеточие (операция задания диапазона значений) при индексации массивов. Мы подробно остановимся на этом вопросе ниже в данной главе пособия, а сейчас только скажем, что операция : задает весь возможный диапазон индексов для данного измерения.

Далее по той же схеме создадим трехмерный массив b:

```
b = ones( 2, 3, 2 );
b( 1, 1, 1 ) = 8
b(:,:,1) =
8 1 1
1 1 1
```

```
b(:,:,2) =
1 1 1
1 1 1
```

Наконец, сгруппируем эти два трехмерных массива вдоль четвертого измерения, а результат присвоим массиву c. Такая операция группировки возможна, поскольку трехмерные массивы a и b имеют одинаковые размеры. В противном случае возникла бы ошибка. Для выполнения группировки (конкатенации) воспользуемся функцией cat:

```
c = cat( 4, a, b );
```

Для полученного массива c вызов функции ndims(c) вернет число 4, что означает, что c является массивом размерности 4. Размер массива c вдоль каждой из его четырех размерностей можно узнать вызовом функции size. Эта функция возвращает вектор-строку из четырех элементов. Первый из этих элементов равен размеру массива вдоль первого измерения, второй элемент дает размер вдоль второго измерения и т. д.:

```
v = size( c )
v =
  2 3 2 2
```

Содержимое четырехмерного массива c система MATLAB показывает следующим образом:

```
' c
c(:,:,1,1) =
    1 1 1
    1 1 1
c(:,:,2,1) =
    1 7 1
    1 1 1
c(:,:,1,2) =
    8 1 1
    1 1 1
c(:,:,2,2) =
    1 1 1
    1 1 1
```

Если все столбцы, показанные здесь системой MATLAB, сложить (слева направо и затем сверху вниз) вдоль одного столбца, то мы получим реальное расположение элементов этого четырехмерного массива в памяти компьютера. Об этой особенности хранения элементов массивов в памяти компьютера мы ранее рассказывали в связи с двумерными массивами (матрицами). Однако это верно для массивов системы MATLAB любой размерности.

Сформированный многомерный массив можно переформировать (без изменения элементов), то есть изменить его размерность или размеры, с помощью функций reshape, squeeze, shiftdim, permute и ipermute. Справочную информацию по работе каждой из этих функций легко получить, выполнив команду help и указав имя функции. Например:

```
' help squeeze
 SQUEEZE Remove singleton dimensions.
B = SQUEEZE(A) returns an array B with the same elements as
a but with all the singleton dimensions removed. a singleton is
a dimension such that size(A,dim)==1. 2-D arrays are unaffected
by squeeze so that row vectors remain rows.
For example, squeeze(rand(2,1,3)) is 2-by-3.
```

Здесь сказано, что функция squeeze устраняет (удаляет) те измерения, вдоль которых размер равен единице (то есть вдоль этих направлений нечего группировать). Этого, однако, не делается для вектор-строк и вектор-столбцов.

Вычисления с массивами

В традиционных языках программирования вычисления с массивами осуществляются поэлементно в том смысле, что нужно запрограммировать каждую отдельную операцию над отдельным элементом массива. В М-языке системы MATLAB допускаются мощные групповые операции над всем массивом сразу. Именно групповые операции системы MATLAB позволяют чрезвычайно компактно задавать выражения, при вычислении которых реально выполняется гигантский объем работы.

Начнем рассмотрение с арифметических операций. Над массивами одинаковых размеров допускаются операции сложения и вычитания, обозначаемые стандартными знаками + и -. Если a и B – массивы любой размерности, но одинаковых размеров, то допустимы следующие выражения:

```
C = a + B; D = a - B;
```

где элементы массивов C и D равны сумме или разности соответствующих элементов массивов a и B. Таким образом, эти операции выполняются поэлементно и порождают массивы тех же размеров, что и исходные операнды. Например:

```
A = [1 1 1; 2 2 2; 3 3 3]; B = [0 0 0; 7 7 7; 1 2 3];
A + B
ans =
   1 1 1
   9 9 9
   4 5 6
```

Если используются операнды разных размеров, выдается сообщение об ошибке (см. рис. 1.17).

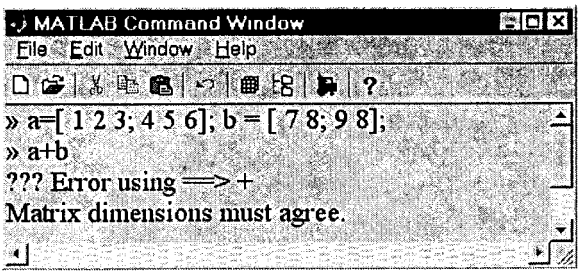


Рисунок 1.17

за исключением случая, когда один из операндов является скаляром:

```
A + 5
ans =
   6 6 6
   7 7 7
   8 8 8
```

В таких случаях скаляр предварительно расширяется до массива размером с матричный операнд. Например, из скаляра 5 сначала генерируется матрица [5 5 5; 5 5 5; 5 5 5], которая и складывается далее поэлементно с матрицей A.

Для поэлементного перемножения и поэлементного деления массивов одинаковых размеров применяются операции, обозначаемые комбинациями двух символов: .* и ./. Использование комбинаций символов объясняется тем, что символами * и / обозначены специальные операции линейной алгебры над векторами и матрицами.

Кроме операции ./, называемой *операцией правого поэлементного деления*, есть еще *операция левого поэлементного деления* .\. Объясним разницу между этими операциями. Выражение a./B приводит к матрице с элементами A(k,m)/B(k,m), а выражение a.\B приводит к матрице с элементами B(k,m)/A(k,m).

Знак * закреплен за перемножением матриц и векторов в *смысле линейной алгебры*. Читателям, не знакомым с линейной алгеброй, можно пропустить изучение этой операции. Для остальных же напомним, что эта операция выполнима только тогда, когда число столбцов в левом операнде равно числу строк в правом операнде:

```
A = [ 1 2 3; 4 5 6; 7 8 9]; x = [ 1, 2, 3]';
B = a * x;
B =
   14
   32
   50
```

Здесь мы для разнообразия задали вектор-столбец x не с помощью операции вертикальной конкатенации [1;2;3], а использовали операцию транспонирования и получили из вектор-строки [1,2,3] нужный нам вектор-столбец.

Традиционный для операции деления знак / (а также знак \) закреплен в системе MATLAB за решением довольно сложной задачи линейной алгебры – *нахождением корней систем линейных уравнений*! Например, если требуется решить систему линейных уравнений

```
Ay = b
```

где A – заданная квадратная матрица размера N x N, b – заданный вектор-столбец длины N, то для нахождения неизвестного вектор-столбца y (неизвестны его элементы) достаточно вычислить выражение a\b. Приведем пример.

```
A = [1,-2,3,-1;2,3,-4,4;3,1,-2,-2;1,-3,7,6];
b = [6;-7;9;-7];
y = a \ b
y =
   2.0000
  -1.0000
   0
  -2.0000
```

Операцию, обозначаемую знаком /, рассмотрим позже в специальной главе, посвященной решению задач линейной алгебры.

Типичные задачи *аналитической геометрии в пространстве*, связанные с нахождением длин векторов и углов между ними, с вычислением скалярного и векторного произведений, легко решаются разнообразными средствами системы MATLAB. Например, для нахождения *векторного произведения* предназначена специальная функция cross:

```
u = [ 1 2 3]; v = [ 3 2 1];
cross( u, v )
ans =
   -4  8 -4
```

Скалярное произведение векторов вычисляется с помощью функции общего назначения sum, вычисляющей сумму всех элементов векторов (для матриц эта функция вычисляет суммы для всех столбцов). Скалярное произведение, как известно, равно сумме произведений соответствующих координат (элементов) векторов. Таким образом, выражение

```
sum( u .* v )
ans =
   10
```

действительно вычисляет скалярное произведение двух пространственных векторов (имеющих по три координаты) u и v, которое равно 10.

Длина вектора вычисляется с помощью скалярного произведения и функции извлечения квадратного корня:

```
len1 = sqrt( sum( u .* u ) );
```

Угол между векторами легко вычисляется на основе определения скалярного произведения, гласящего, что оно равно произведению длин векторов на косинус угла между ними. Отсюда находим выражение для вычисления угла между ранее заданными векторами u и v:

```
len1 = sqrt(sum(u .* u)); len2 = sqrt(sum(v .* v));
phi = acos( sum(u .* v) / ( len1 * len2 ) )
phi = 0.7752
```

Ранее рассмотренные для скаляров *операции отношения* и *логические операции* выполняются в случае массивов *поэлементно*. Оба операнда должны быть одинаковых размеров, при этом операция возвращает результат такого же размера. В случае, когда один из операндов скаляр, производится его *предварительное расширение*, смысл которого уже был пояснен на примере арифметических операций.

Для иллюстрации выполним над матрицами

```
A = [1 1 1; 2 2 2; 3 3 3]; B = [0 0 0; 7 7 7; 1 2 3];
```

операцию *меньше или равно*. Результат этой операции показан на рис. 1.18:

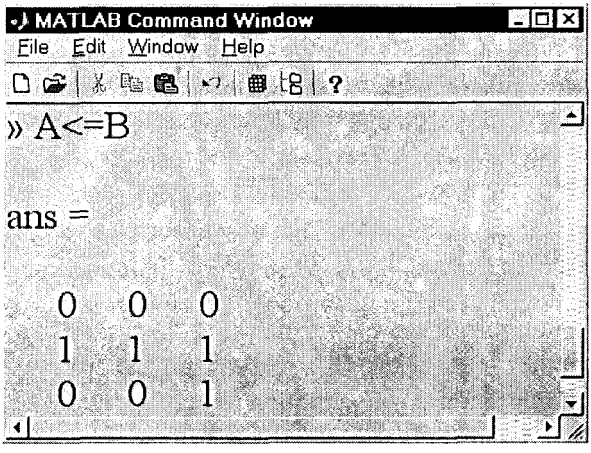


Рисунок 1.18

где каждый нуль означает «ложь» для данной позиции внутри матриц, а единица означает «истину». Полученная матрица показывает (своими единичными элементами), в каких позициях элементы матрицы A на самом деле меньше или равны соответствующим элементам матрицы B.

Работу *логических операций* над массивами проиллюстрируем на примере операции «НЕ». Пусть задан вектор

```
v = [ 1 9 9 9 0 ];
```

Для этого вектора результат операции «НЕ», то есть ~v, равен

```
~v
ans =
   0 0 0 0 1
```

Выше при изучении вычислений с вещественными скалярами мы рассматривали работу логической функции xor («исключающее ИЛИ»). Эта функция работает и с массивами одинаковых размеров, поэлементно реализуя операцию «исключающее ИЛИ». Напоминаем, что каждый элемент трактуется как истинный, если он не равен нулю, и как ложный в случае его равенства нулю. Покажем результат работы этой функции над ранее заданными матрицами A и B:

```
xor( A, B )
ans =
   1 1 1
   0 0 0
   0 0 0
```

Другими логическими функциями (помимо функции xor) являются функции all и any. Функция all в случае векторов возвращает 1 («истина»), если все элементы вектора не равны нулю (истинны), и возвращает 0, когда хотя-бы один элемент вектора ненулевой. Функция any действует противоположным образом.

В случае матриц обе эти функции работают с их столбцами, возвращая для каждого столбца результат по описанной выше схеме. Например,

```
all( a )
ans =
   1 1 1
all( B )
ans =
   0 0 0
```

Существует множество разноплановых функций, специально предназначенных для работы с массивами. Таковыми являются ранее рассмотренные функции ones, zeros, sum, cat, size, ndims, reshape и многие другие. Часть из этих функций сообщает служебную информацию о массивах, другая группа функций обеспечивает контролируемое изменение их структуры, третья группа предназначена для генерации массивов с заданными свойствами и т. д.

Среди функций, генерирующих матрицы с заданными свойствами, упомянем здесь функцию eye, производящую единичные квадратные матрицы, а также широко применяемую на практике функцию rand, генерирующую массив со

случайными элементами, равномерно распределенными на интервале от 0 до 1. Например, выражение

```
A = rand( 3 )
```

порождает массив случайных чисел размером 3 x 3 с элементами, равномерно распределенными на интервале от 0 до 1 (см. рис. 1.19).

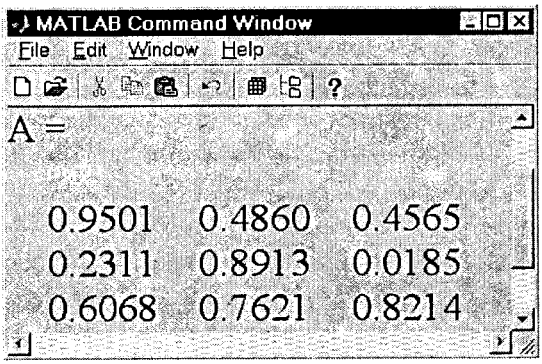


Рисунок 1.19

Если вызвать эту функцию с двумя аргументами, например

```
R = rand( 2, 3 )
```

то получится матрица R случайных элементов размером 2 x 3. При вызове функции rand с тремя и более скалярными аргументами производятся многомерные массивы случайных чисел.

Среди функций, производящих простейшие вычисления над массивами, помимо рассмотренной выше функции sum, упомянем еще функцию prod, которая во всем аналогична функции sum, только вычисляет она не сумму элементов, а их произведение. К примеру, для определенной выше матрицы B она возвращает следующий результат:

```
prod( B )
ans =
   0 0 0
```

Функции max и min ищут соответственно максимальный и минимальный элементы массивов. Для векторов они возвращают единственное числовое значение, а для матриц они порождают набор экстремальных элементов, вычисленных для каждого столбца. Например,

```
max( B )
ans =
   7 7 7
```

Функция sort сортирует в возрастающем порядке элементы одномерных массивов, а для матриц она производит такую сортировку для каждого столбца отдельно.

Наконец, рассмотрим уникальную возможность М-языка системы MATLAB производить групповые вычисления над массивами, используя обычные математические функции, которые в традиционных языках программирования работают только со скалярными аргументами. В результате с помощью крайне компактных записей, удобных для ввода с клавиатуры в интерактивном режиме работы с командным окном системы MATLAB, удается произвести большой объем вычислений. Например, всего два коротких выражения

```
x = 0 : 0.01 : pi/2;  y = sin( x );
```

вычисляют значения функции sin сразу в 158 точках, формируя два вектора x и y со 158 элементами каждый. Это уже весьма большой объем информации о функции, достаточный для построения ее подробного графика. В частности, уже в следующей главе мы расскажем о том, как с помощью вызова всего одной функции системы MATLAB, на вход которой нужно подать полученные сейчас векторы x и y, можно построить график изучаемой функции в правильно подобранном масштабе, с отметками на осях координат и т. д.

Множественная индексация массивов в системе MATLAB

Мы только что рассмотрели уникальную возможность системы MATLAB осуществлять множественные вычисления обычными математическими функциями. Теперь мы рассмотрим другую уникальную особенность М-языка системы MATLAB – множественную индексацию массивов.

Начнем рассмотрение этого вопроса на примере векторов (одномерных массивов). Сформируем вектор v

```
v1 = [ 9 8 7 6 5 4 3 2 1 ];
```

состоящий из девяти элементов. Пусть нам надо из этого вектора осуществить выборку нескольких элементов для формирования другого вектора v2. Для определенности будем выбирать второй, шестой и восьмой элементы. Тогда, если действовать в духе традиционных языков программирования, можно предложить следующее решение:

```
v2 = [ v1(2), v1(6), v1(8) ]
```

Здесь мы трижды осуществляем индексацию исходного массива v1, каждый раз со своим индексом, и полученные результаты группируем (при помощи операции конкатенации) в результирующий массив v2.

Синтаксис М-языка системы MATLAB позволяет выполнить эту работу короче, без видимого применения операции конкатенации результатов. Вот это решение, основанное на *множественной индексации* массивов в системе MATLAB:

```
v2 = v1( [ 2 6 8 ] )
```

Здесь вместо скалярного индекса подставляется вектор, содержащий целый набор индексов (индексами являются его элементы). В результате получается тот же результат, что и при нескольких индивидуальных индексациях. Множественная индексация обеспечивает не только более компактные записи, но и большее быстродействие вычислений.

С помощью множественной индексации можно не только извлечь подмножество элементов исходного массива, но и *продублировать* некоторые из них. Например, выражение

```
v2 = v1( [ 2 2 6 8 8 8 ] )
v2 =
  8 8 4 2 2 2
```

не только извлекает из исходного вектора v1 второй, шестой и восьмой элементы, но и повторяет их по нескольку раз. В частности, второй элемент вектора v1 взят в двух экземплярах, а восьмой элемент – в трех.

С помощью операции множественной индексации легко *переставлять местами* элементы сформированного массива:

```
v2 = v2( [ 1 3 2 4 5 6 ] )
v2 =
  8 4 8 2 2 2
```

Здесь мы переставили местами второй и третий элементы массива v2. Если из массива требуется извлечь *подряд расположенные элементы*, то очень удобно использовать *операцию формирования диапазона*. Например, если из вектора v1 требуется извлечь подвектор v2, состоящий из элементов с пятого по восьмой, то эта задача решается с помощью следующего выражения, использующего множественную индексацию и операцию формирования диапазона:

```
v2 = v1( 5:8 );
v2 =
  5 4 3 2
```

Выражение

```
v1( : )
```

где у операции формирования диапазона не указаны ни левый, ни правый операнды, подразумевает взятие всего диапазона возможных для исходного вектора v1 индексов, причем в их естественном порядке возрастания. Это означает, что в естественном порядке извлекаются все элементы исходного вектора. Здесь можно подумать, что результат будет в точности равен исходному вектору.

Однако это не так. Дело в том, что исходный вектор v1 является именно вектор-строкой, а результат будет вектор-столбцом. Здесь в самый раз вспомнить, что система MATLAB хранит данные (элементы) массивов упорядоченно по столбцам. Это и определяет результат представленной выше операции:

```
v1( : )
ans =
   9
   8
   7
   6
   5
   4
   3
   2
   1
```

Теперь рассмотрим множественную индексацию на примере матриц (двумерных массивов). Эта операция позволяет извлечь из матрицы некоторую подматрицу ее элементов:

```
A = [ 1 2; 3 4; 5 6 ];
B = A( [ 2 3 ], 1:2 )
B =
  3 4
  5 6
```

Если при индексации матрицы A фиксировать единственный первый индекс (номер строки) и задать весь возможный диапазон для второго индекса (номера всех существующих столбцов), то можно извлечь из матрицы конкретную строку. Например, выражение

```
A( 2, : )
ans =
   3 4
```

извлекает из матрицы A ее вторую строку.

Выражение A(:,:) полностью дублирует исходную матрицу, а выражение A(:) означает взятие всех ее элементов в том порядке, в котором они хранятся в памяти компьютера (то есть в виде одного длинного столбца):

```
A( : )
ans =
   1
   3
   5
```

```
   2
   4
   6
```

Итак, элементы матрицы хранятся линейно: сначала идут элементы первого столбца, затем – второго. Выражение `A(4)` означает взятие четвертого по счету элемента из так организованного для хранения матрицы единого столбца:

```
A( 4 )
ans =
   2
```

Можно заменить в исходной матрице некоторую строку на другую:

```
A( 2, : ) = [ 8 9 ];
```

или удалить из матрицы целую строку, если использовать в операции присваивания пустой массив:

```
A( 2, : ) = [];
```

Рассмотренные операции очень удобны для формирования трехмерных массивов. Например, предварительно сформировав 30 матриц B1, B2, ..., B30 одинакового размера, следующими групповыми присваиваниями создаем трехмерный массив A:

```
A(:,:,1) = B1; A(:,:,2) = B2; ...A(:,:,30) = B30;
```

где для повышения эффективности нужно последнее из указанных присваиваний осуществить первым.

Применяя операцию множественной индексации, можно легко переставлять местами строки и столбцы матриц. Например, у матрицы

```
F = [ 1 2 3; 4 5 6; 7 8 9 ]
```

с помощью выражения

```
F = F( :, [ 3 1 2 ] )
```

последний столбец ставят на первое место, второй – на последнее, а первый столбец исходной матрицы становится вторым (см. рис. 1.20).

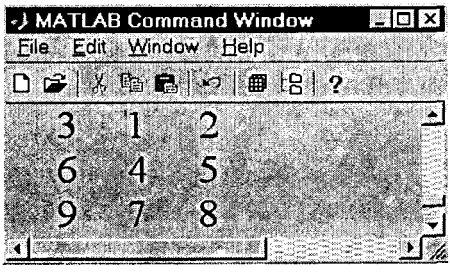


Рисунок 1.20

Часть строк можно рассмотренным способом переставлять местами, часть строк можно дублировать, а некоторые строки исходной матрицы можно вообще не помещать в результирующую матрицу. Вот пример, позволяющий из только что полученной матрицы F размером 3 x 3 сформировать матрицу E размером 4 x 5 (см. рис. 1.21).

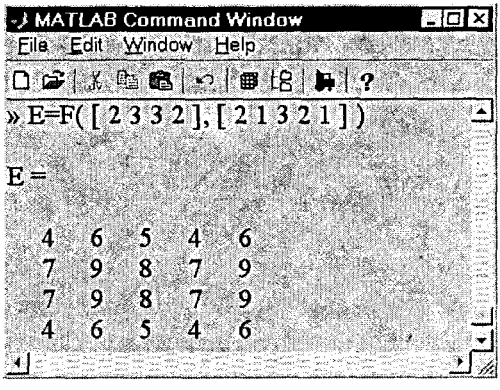


Рисунок 1.21

Отсюда наглядно видно, что размер результата определяется количеством индексов, использованном при множественном индексировании. Это позволяет, размножая вектор-строки или вектор-столбцы, получать матрицы. Для примера получим матрицу из вектор-столбца

```
v = [ 1 ; 2 ];
```

Будем осуществлять индексацию вектор-столбца v двумя индексами, так как его можно рассматривать как матрицу, содержащую только один столбец. Естественно, второй индекс может быть только единицей. Но в соответствии с техникой множественного индексирования эту индексирующую единицу можно указать несколько раз:

```
A = v( :, [ 1 1 1 ] )
A =
   1 1 1
   2 2 2
```

и мы получили матрицу 2 x 3 размножением исходного вектор-столбца вдоль горизонтального направления (вдоль строк).

Совсем *другим вариантом множественной индексации*, по сравнению с рассмотренной техникой, является *индексирование матрицей*. Пусть задана матрица

```
I = [ 1 1; 2 2; 2 2; 1 1 ];
```

у которой значения всех элементов являются допустимыми индексами сформированного выше вектор-столбца v. Тогда выражение

```
A = v( I );
```

осуществляющее технику индексирования матрицей, порождает в качестве результата матрицу, размер которой совпадает с размером индексирующей матрицы, а элементы поставляются вектором v посредством индексирования индивидуальными элементами матрицы. Отсюда понятен получающийся результат:

```
A =
  1 1
  2 2
  2 2
  1 1
```

Этот результат не изменится, если вместо вектор-столбца v применить вектор-строку [1,2], состоящую из тех же элементов.

Рассмотренная техника индексирования матрицей применяется для очень эффективного решения следующей задачи. Дана некоторая матрица A размером m x n. Нужно размножить эту матрицу в вертикальном направлении M раз и в горизонтальном направлении N раз. Вот решение этой задачи, основанное на технике множественного индексирования. Пусть, для определенности,

```
A = [ 1 3 8; 5 7 4 ]; M = 2; N = 3;
```

заданы исходная матрица 2 x 3 и число повторений в вертикальном и горизонтальном направлениях. Тогда следующие несколько выражений решают поставленную задачу (см. рис. 1.22):

```
v1 = (1:2)'; v2 = (1:3)';
I1 = v1( :, ones(1,M) ); I2 = v2( :, ones(1,N) );
B = A( I1, I2 );
```

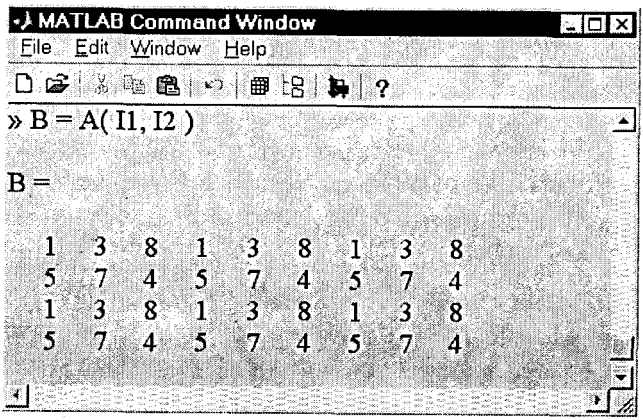


Рисунок 1.22

Здесь нужно только пояснить одну деталь. При индексировании исходной матрицы A двумя индексирующими матрицами I1 и I2 последние рассматрива-

ются как единый столбец их элементов. В результате первая матрица I1 поставляет для организации строк результирующей матрицы B четыре своих элемента, так что матрица B состоит из четырех строк. Так как вторая матрица I2 состоит всего из девяти элементов, то столько же и будет столбцов в результирующей матрице B. Конкретные значения элементов матрицы B получаются в результате подстановки индивидуальных элементов матриц I1 и I2 и последующего извлечения элементов матрицы A.

Рассмотренное решение задачи о размножении матрицы в горизонтальном и вертикальном направлениях достаточно сложно для восприятия человеком, однако эффективность такого решения в сотни раз выше, чем простое и наглядное для человека прописывание отдельных элементов.

Для группового воздействия на элементы массивов (помимо множественной индексации) можно применить функцию find, которая в качестве аргумента принимает условие, а возвращает набор индексов элементов, удовлетворяющих этому условию. Зададим для примера вектор

```
v = [ 1 0 3 6 5 1 ];
```

Тогда выражение

```
ind = find( v > 1 )
ind =
  3 4 5
```

формирует набор (одномерный массив) ind индексов тех элементов исходного вектора v, которые по величине больше единицы. С помощью операции множественного индексирования все эти элементы можно сделать равными, например, девяти:

```
v( ind ) = 9
v =
  1 0 9 9 9 1
```

Получение с помощью функции find набора индексов и операцию множественного индексирования можно совместить в пределах единственного выражения:

```
v( find( v > 1 ) ) = 9;
```

Для матриц функция find возвращает несколько векторов индексов, каждый из которых получается обработкой отдельного столбца матрицы. Например, для матрицы

```
A = [ 1 0 3; 0 4 5; 6 7 8 ];
```

вызов функции find

```
[ u1, u2 ] = find( a )
```

формирует вектор-столбцы u1 и u2, первый из которых содержит номера строк отличных от нуля элементов матрицы A, а второй – номера столбцов этих элементов (см. рис. 1.23).

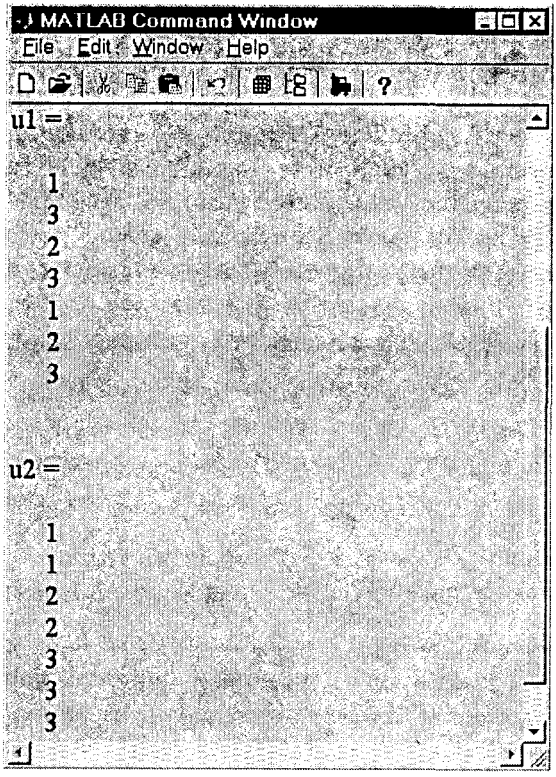


Рисунок 1.23

Во время работы функции find столбцы матрицы обрабатываются в порядке слева-направо, то есть сначала обрабатывается первый столбец, затем – второй и так далее. Внутри столбца элементы просматриваются сверху-вниз.

Итак, рассмотренные возможности М-языка системы MATLAB позволяют с помощью компактных выражений выполнять большой объем вычислений. В результате таких вычислений получаются большие объемы числовых данных, для успешной интерпретации которых крайне важна их наглядная визуализация. Этому вопросу посвящена вторая глава настоящего пособия.

Глава 2

Визуализация результатов вычислений

Построение графиков функций

В результате вычислений в системе MATLAB обычно получается большой массив данных, который трудно анализировать без наглядной визуализации. Поэтому система визуализации, встроенная в MATLAB, придает этому пакету особую практическую ценность.

Графические возможности системы MATLAB являются мощными и разнообразными. В первую очередь целесообразно изучить наиболее простые в использовании возможности. Их часто называют *высокоуровневой графикой*. Это название отражает тот приятный факт, что пользователю нет никакой необходимости вникать во все тонкие и глубоко спрятанные детали работы с графикой.

Например, нет ничего проще, чем построить график функции одной вещественной переменной. Следующие выражения:

```
x = 0 : 0.01 : 2;
y = sin( x );
```

вычисляют массив y значений функции `sin` для заданного набора аргументов. После этого вызовом единственной функции

```
plot( x , y )
```

удается построить вполне качественно выглядящий график функции (см. рис. 2.1).

MATLAB показывает графические объекты в специальных *графических окнах*, имеющих в заголовке слово *Figure* (фигура, изображение).

При построении графиков функций сразу проявляется тот факт, что очень большую часть работы MATLAB берет на себя. Мы в командной строке ввели лишь одну функцию, а система сама создала графическое окно, построила оси координат, вычислила диапазоны изменения переменных x и y, проставила на осях метки и соответствующие им числовые значения, провела через опорные точки график функции некоторым, выбранным по умолчанию, цветом, в заголовке графического окна надписала номер графика в текущем сеансе работы.



Рисунок 2.1

Не убирая с экрана дисплея первое графическое окно, вводим с клавиатуры выражения

```
x = 0 : 0.01 : 2; z = cos( x );
plot( x , z )
```

и получаем новый график функции в том же самом графическом окне (при этом старые оси координат и график пропадают – этого можно также добиться командой `clf`, командой `cla` удаляют только график с приведением осей координат к их стандартным диапазонам от 0 до 1) (см. рис. 2.2).

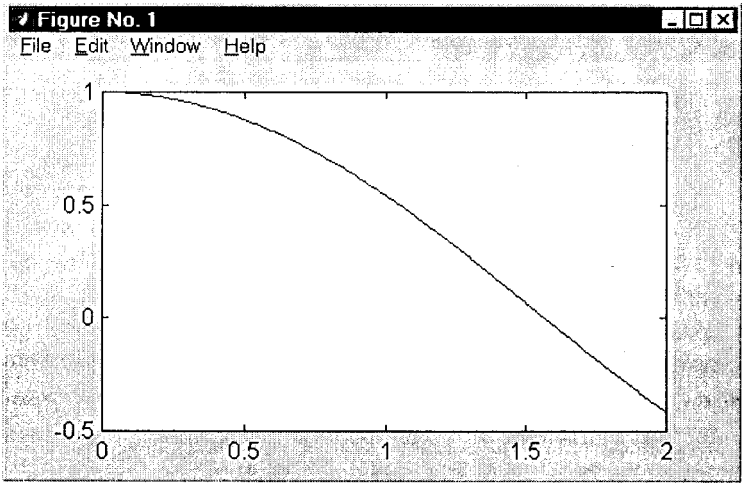


Рисунок 2.2

Если нужно второй график провести «поверх первого графика», то перед вторичным вызовом графической функции `plot` нужно выполнить команду

```
hold on
```

которая предназначена для удержания текущего графического окна. В результате будет получено следующее изображение (см. рис. 2.3).

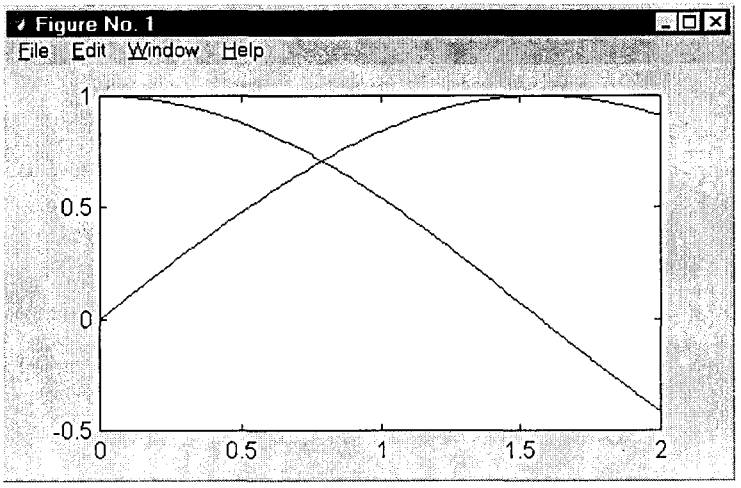


Рисунок 2.3

Того же самого можно добиться, потребовав от функции `plot` построить сразу несколько графиков в рамках одних и тех же осей координат:

```
x = 0 : 0.01 : 2;
y = sin( x ); z = cos( x );
plot( x , y , x , z )
```

У такого способа есть еще одно (кроме экономии на команде `hold on`) преимущество: разные графики автоматически строятся разным цветом.

К недостаткам указанных способов построения нескольких графиков в пределах одних и тех же осей координат относится использование одного и того же диапазона изменения координат, что при несопоставимых значениях двух функций приведет к плохому изображению графика одной из них.

Если все же нужно одновременно визуализировать несколько графиков так, чтобы они не мешали друг другу, то это можно сделать двумя способами. Первым решением является построение их в разных графических окнах. Например, построив графики функций `sin` и `cos` в пределах одного графического окна (показано выше), вычисляем массив значений `w` для функции `exp`:

```
w = exp( x );
```

После этого выполняем команды

```
figure; plot( x , w )
```

которые строят график функции exp в новом графическом окне, так как команда figure создает это новое графическое окно и заставляет все последующие за ней функции построения графиков выводить их туда (см. рис. 2.4). В результате в первом графическом окне (Figure No.1) по вертикальной оси переменные изменяются в диапазоне от -0.5 до 1, а во втором графическом окне (Figure No.2) – от 0 до 8.

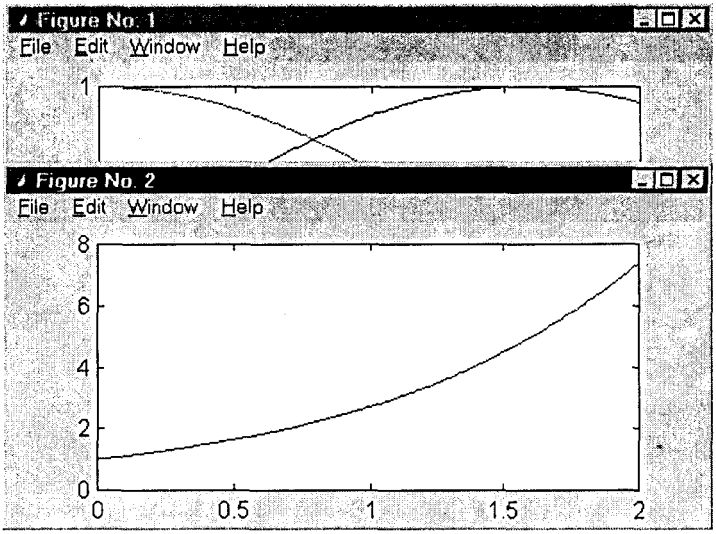


Рисунок 2.4

Вторым решением показа нескольких графиков без конфликта диапазонов осей координат является использование функции subplot. Эта функция позволяет разбить область вывода графической информации на несколько подобластей, в каждую из которых можно вывести графики различных функций.

Например, для ранее выполненных вычислений с функциями sin, cos и exp строим графики первых двух функций в первой подобласти, а график третьей функции – во второй подобласти одного и того же графического окна:

```
subplot(1,2,1); plot(x,y,x,z)
subplot(1,2,2); plot(x,w)
```

в результате чего получаем графическое окно следующего вида (см. рис. 2.5).

Диапазоны изменения переменных на осях координат этих подобластей независимы друг от друга.

Функция subplot принимает три числовых аргумента, первый из которых равен числу рядов подобластей, второй равен числу колонок подобластей, а третий аргумент – номеру подобласти (номер отсчитывается вдоль рядов с переходом на новый ряд по исчерпании).

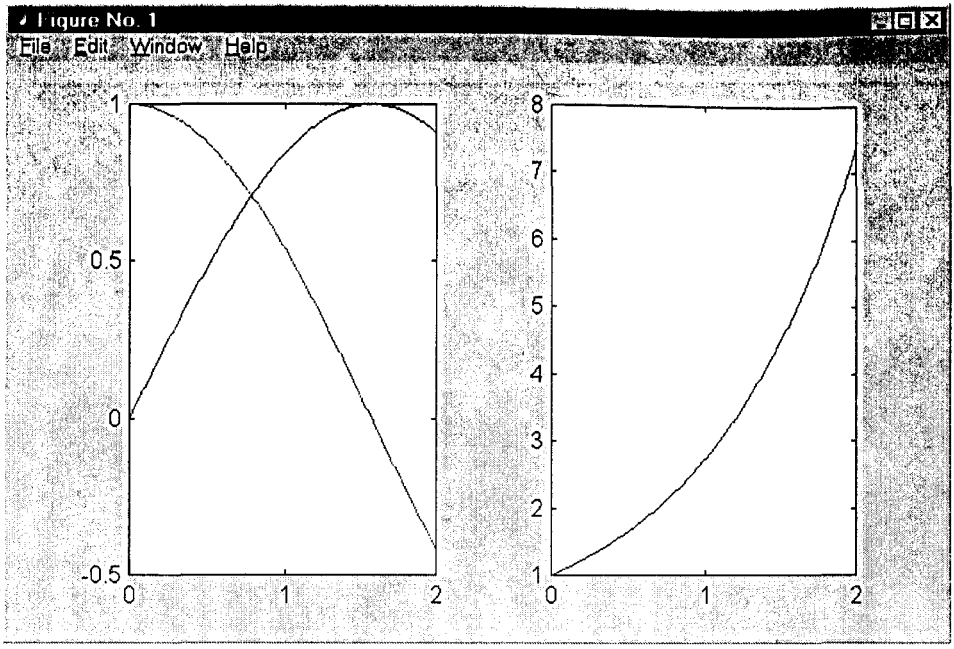


Рисунок 2.5

Если для одиночного графика диапазоны изменения переменных вдоль одной или обеих осей координат слишком велики, то можно воспользоваться функциями построения графиков в *логарифмических масштабах*. Для этого предназначены функции `semilogx`, `semilogy` и `loglog`. Подробную информацию по использованию этих функций всегда можно получить при помощи команды

```
help имя_функции
```

выполняемой в командном окне системы MATLAB.

Итак, уже рассмотренные примеры показывают, как подсистема высокоуровневой графики системы MATLAB легко справляется с различными случаями построения графиков, не требуя слишком большой работы от пользователя. Еще одним таким примером является построение графиков в *полярных координатах*.

Например, если нужно построить график функции `r = sin(3 φ)` в полярных координатах, то следующие несколько команд

```
phi = 0 : 0.01 : 2 * pi; r = sin( 3 * phi );
polar( phi , r )
```

состоящие из вычисления выражений и вызова графической функции `polar`, специально предназначенной для построения графиков в полярных координатах, решают эту задачу (см. рис. 2.6).

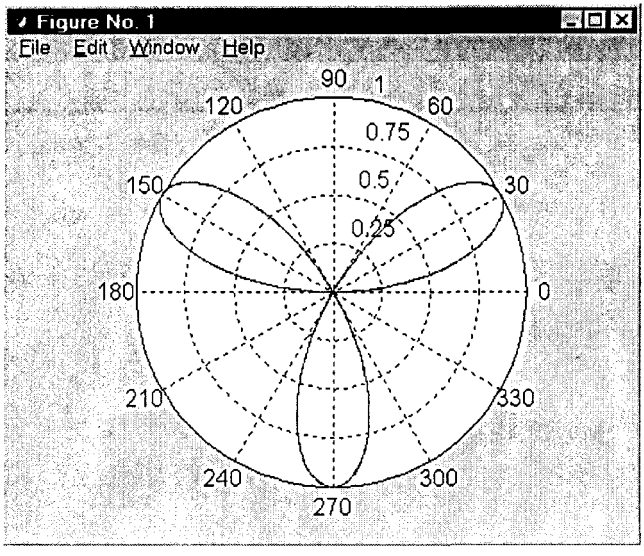


Рисунок 2.6

Оформление графиков и графических окон

Функции построения графиков, рассмотренные нами выше, осуществляли вполне приемлемое автоматическое оформление графиков. Мы сейчас рассмотрим дополнительные возможности, связанные с управлением внешним видом графиков – задание цвета и стиля линий, а также размещение различных надписей в пределах графического окна.

Например, команды

```
x = 0 : 0.1 : 3; y = sin( x );
plot( x, y, 'r-', x, y, 'ko' )
```

позволяют придать графику вид красной сплошной линии, на которой в дискретных вычисляемых точках проставляются черные окружности. Здесь функция `plot` дважды строит график одной и той же функции, но в двух разных стилях. Первый из этих стилей отмечен как `'r-'`, что означает проведение линии красным цветом (буква `r`), а штрих означает проведение сплошной линии. Второй стиль, помеченный как `'ko'`, означает проведение черным цветом (буква `k`) окружностей (буква `o`) на месте вычисляемых точек (см. рис. 2.7).

В общем случае функция

```
plot( x1, y1, s1, x2, y2, s2, … )
```

позволяет объединить в одном графическом окне несколько графиков функций `y1(x1)`, `y2(x2)`,... проведя их со *стилями* `s1`, `s2`,... и т. д.

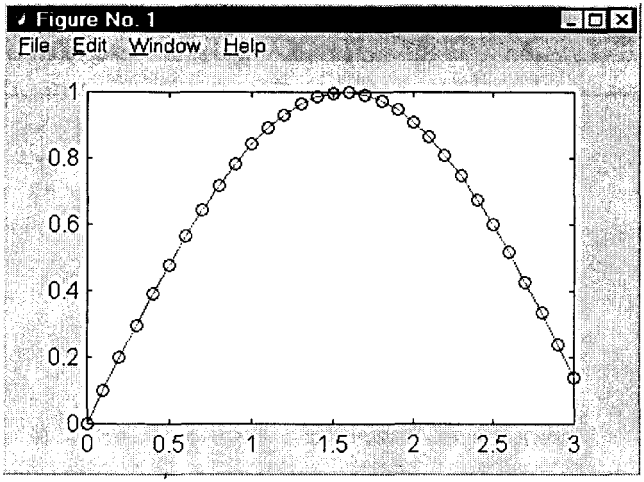


Рисунок 2.7

При помощи функции вида

```
plot( x1, y1, s1, x1, y1, s2 )
```

мы можем провести линию графика функции `y1(x1)` одним цветом, а точки на нем (вычисляемые точки) – другим цветом, что и было продемонстрировано выше на примере функции `sin(x)`.

Стили `s1`, `s2`,... задаются в виде набора трех *символьных маркеров*, заключенных в одиночные кавычки (апострофы). Один из этих маркеров задает тип линии:

Маркер	-	--	:	-.
Тип линии	Непрерывная	Штриховая	Пунктирная	Штрихпунктирная

Другой маркер задает цвет:

Маркер	Цвет линии	Маркер	Цвет линии
c	Голубой	g	Зеленый
m	Фиолетовый	b	Синий
y	Желтый	w	Белый
r	Красный	k	Черный

Последний маркер задает тип проставляемых «точек»:

Маркер	.	+	*	o	x
Тип точки	Точка	Плюс	Звездочка	Кружок	Крестик

Можно указывать не все три маркера. Тогда используются маркеры, установленные по умолчанию. Порядок, в котором указываются маркеры, не является существенным, то есть 'r+-' и '-+r' приводят к одинаковому результату.

Если в строке стиля поставить маркер типа точки, но не проставить маркер на тип линии, то тогда отображаются только вычисляемые точки, а непрерывной линией они не соединяются.

Наиболее мощным способом оформления графиков функций (и выполнения других графических работ) является *дескрипторный метод*, относящийся к так называемой низкоуровневой графике системы MATLAB. Дескрипторная графика имеет дело с базовыми графическими объектами, каждый из которых может быть индивидуально настроен посредством задания всех его свойств. Детальное знакомство со свойствами графических объектов системы MATLAB невозможно осуществить минимальными усилиями, так как количество свойств отдельных графических объектов доходит до ста. Однако во многих случаях применение дескрипторной графики не вызывает никаких затруднений. Приведем сейчас иллюстрирующие примеры.

Выше мы оформляли график функции `sin` с помощью непрерывной красной линии и черных кружков. Теперь попробуем ограничиться лишь непрерывной линией, но очень толстой. Как это можно сделать? Вот простое решение на базе дескрипторной графики:

```
x = 0 : 0.1 : 3; y = sin( x );
hPlot = plot( x, y );
set( hPlot, 'LineWidth', 7 );
```

Функция `plot` через опорные (вычисленные) точки с координатами `x`, `y` проводит отрезки прямых линий. Прямые линии в системе MATLAB представляют собой графические объекты типа `line`. Эти объекты имеют огромное число свойств и характеристик, которые можно менять. Доступ к этим объектам осуществляется по их *описателям* (*дескрипторам*; по-английски – *handles*).

Описатель графического объекта типа `line`, использованного для построения нашего графика, возвращается функцией `plot`. Мы его запоминаем в переменной `hPlot`. Затем этот описатель предлагается функции `set` для опознания конкретного графического объекта. Именно для такого опознанного объекта функция `set` изменяет характеристики, указанные в других ее аргументах. В нашем примере мы указали свойство `'LineWidth'` (толщина линии) и его новое значение `7` (по умолчанию – `0.5`). В результате получается следующая картина (см. рис. 2.8).

Текущее значение любого параметра (атрибута, характеристики) графического объекта можно узнать с помощью функции `get`. Например, если после получения показанного на рисунке графика ввести и исполнить команду

```
width = get( hPlot, 'LineWidth' )
```

то для переменной `width` будет получено значение `7`.

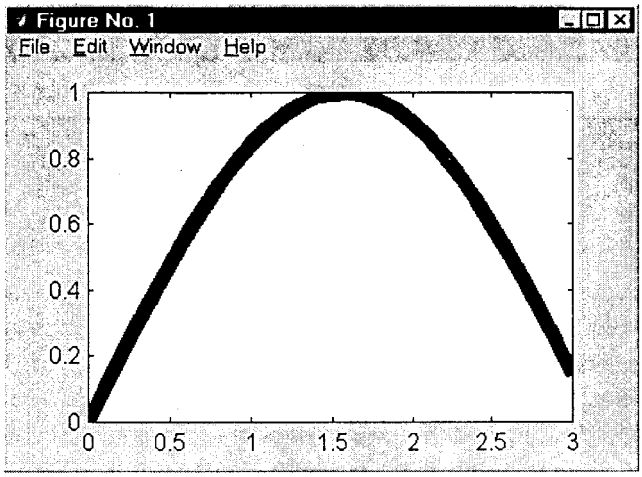


Рисунок 2.8

Чтобы ознакомиться со списком всех свойств графического объекта, нужно вызвать функцию `get`, указав ей описатель объекта в виде единственного параметра. Например, для описателя `hPlot` объекта типа `line` находим весь список его свойств:

```
get( hPlot )
  Color = [0 0 1]
  EraseMode = normal
  LineStyle = -
  LineWidth = [7]
  Marker = none
  MarkerSize = [6]
  MarkerEdgeColor = auto
  MarkerFaceColor = none
  XData = [(1 by 31) double array]
  YData = [(1 by 31) double array]
  ZData = []
  ButtonDownFcn =
  Children = []
  Clipping = on
  CreateFcn =
  DeleteFcn =
  BusyAction = queue
  HandleVisibility = on
  HitTest = on
  Interruptible = on
  Parent = [2.00049]
```

```
Selected = off
SelectionHighlight = on
Tag =
Type = line
UIContextMenu = []
UserData = []
Visible = on
```

Среди всех этих многочисленных свойств встречаются интуитивно понятные. В частности, мы видим значение толщины линии (`LineWidth`), равное 7; свойство `Color` отвечает за цвет линии: он равен `[ 0 0 1 ]` (RGB-кодировка, то есть Красный Зеленый Синий), что соответствует синему цвету. Для успешного применения иных свойств требуется их подробное и кропотливое изучение, без которого, однако, вполне можно обойтись, так как наиболее важным свойствам система MATLAB присваивает по умолчанию вполне в среднем приемлемые значения. Они указываются в списке свойств справа от знака равно. Некоторые свойства не задействованы и являются резервом, который применяют в специальных случаях.

Теперь от оформления непосредственно линий перейдем к оформлению осей координат, к надписям на осях и т. д. Система MATLAB устанавливает пределы на горизонтальной оси равными тем значениям, что указаны пользователем для независимой переменной. Для зависимой переменной по вертикальной оси MATLAB самостоятельно вычисляет диапазон изменения значений функции. В результате график функции оказывается как бы вписанным в прямоугольник.

Если мы хотим отказаться от этой особенности масштабирования при построении графиков в системе MATLAB, то мы должны явным образом навязать свои пределы изменения переменных по осям координат. Это делается с помощью функции

```
axis( [ xmin, xmax, ymin, ymax ] )
```

причем команду на выполнение этой функции можно вводить с клавиатуры сколько угодно раз уже после построения графика функции, чтобы, глядя на получающиеся визуальные изображения, добиться наилучшего восприятия. Такое масштабирование позволяет получить подробные изображения тех частей графика, которые вызывают наибольший интерес в конкретном исследовании. Например, для ранее полученного графика функции `sin` можно сузить пределы по осям координат

```
axis( [ 1.5, 2.5, 0.5, 2 ] )
```

чтобы получше разглядеть вершину синусоиды (см. рис. 2.9).

Чаще всего этот прием увеличения масштаба изображения применяют при графическом решении уравнений с тем, чтобы получить более высокую точность приближения к корню.

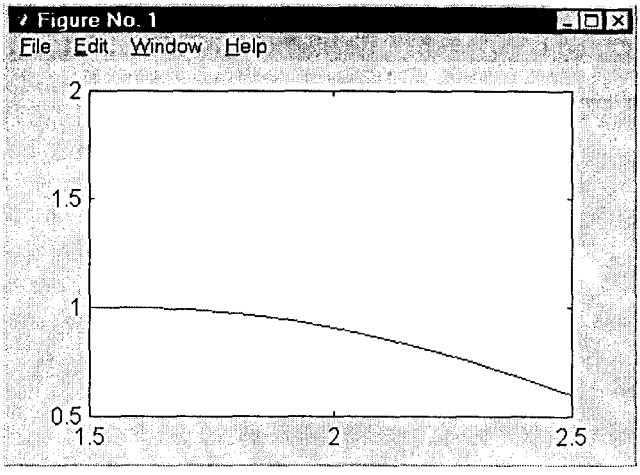


Рисунок 2.9

Теперь изменим количество числовых меток на осях. Их может показаться недостаточно (на горизонтальной оси последнего рисунка их всего три – для значений 1.5, 2 и 2.5).

Изменить отметки на осях координат можно с помощью функции set, обрабатывающей графический объект axes. Это объект, который содержит оси координат и белый прямоугольник, внутри которого проводится график функции. Для получения описателя такого объекта применяют функцию gca. Эту функцию вызывают без параметров. В итоге фрагмент кода

```
hAxes = gca;
set( hAxes, 'xtick', [ 1.5, 1.75, 2.0, 2.25, 2.5 ] )
```

выполняющийся после построения графика, устанавливает новые метки (в количестве пяти штук) на горизонтальной оси координат.

Для проставления различных надписей на полученном рисунке применяют функции xlabel, ylabel, title и text. Функция xlabel предназначена для проставления названия горизонтальной оси, функция ylabel – то же для вертикальной оси (причем эти надписи ориентированы вдоль осей координат).

Если требуется разместить надпись в произвольном месте рисунка, применяем функцию text:

```
text( x, y, 'some text')
```

Общий заголовок для графика проставляется функцией title. Кроме того, используя команду

```
grid on
```

можно нанести измерительную сетку на всю область построения графика. Применяя все эти средства:

```
title( 'Function sin(x) graph' );
```

```
xlabel( 'x coordinate' ); ylabel( 'sin(x)' );
text( 2.1, 0.9, '\leftarrowsin(x)' ); grid on;
```

придаем графику функции следующий вид (см. рис. 2.10).

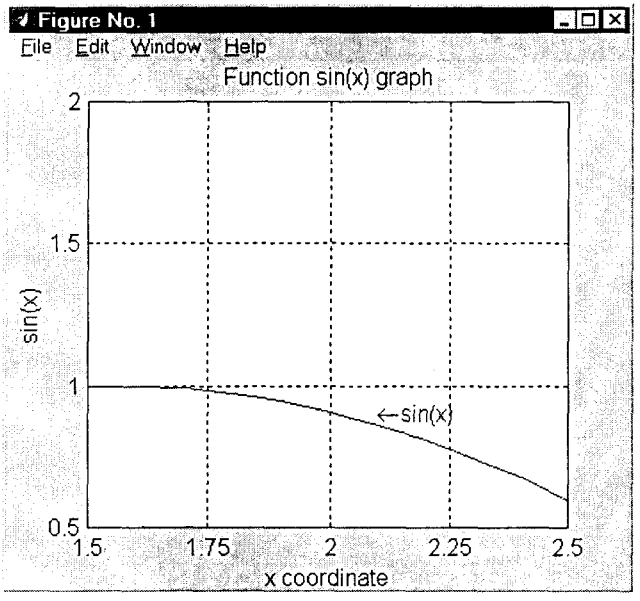


Рисунок 2.10

Надпись функцией `text` помещается начиная от точки с координатами, указанными первыми двумя аргументами. По умолчанию координаты задаются в тех же единицах измерения, что и координаты, указанные на горизонтальной и вертикальной осях. Специальные *управляющие символы* вводятся внутри текста после символа \ (обратная косая черта). В примере мы ввели таким образом специальный символ «стрелка влево». Обозначения для специальных символов совпадают с таковыми в системе подготовки научных текстов TeX. В частности, для вывода части текста в следующую строку применяется управляющий символ `newline`.

Чтобы воздействовать на шрифт, которым изображается надпись, нужно запомнить описатель (дескриптор), возвращаемый этой функцией:

```
hText = text( 2.1, 0.9, '\leftarrowsin(x)' );
```

Это описатель графического объекта типа `text`. Кстати, если нужно узнать, какой тип графического объекта представляет некоторый описатель, то для этого нужно вызвать функцию `get`, запросив у нее свойство `'Type'`:

```
type = get( hText, 'Type' )
type =
    text
```

Располагая описателем объекта типа text, можно изменять ряд его атрибутов. Например, фрагмент кода на М-языке

```
set( hText, 'Color', [ 0 1 0 ], 'FontSize', [18] );
```

изменит внешний вид ранее выведенной надписи – теперь используется более крупный шрифт, а цвет надписи стал зеленым.

Завершая рассказ о способах оформления графиков функций, изменим цвет фона, на котором эти графики рисуются. По умолчанию этот цвет белый. Сделаем его слегка зеленоватым, присвоив цветовому свойству объекта типа axes значение [0.5,0.8,0.5]. Но сначала надо получить описатель этого объекта:

```
hAxes = gca;
```

Функция gca предназначена для поиска описателя *текущего* объекта axes. Так как у нас имеется единственный объект axes, то проблемы выбора нет. Если бы было несколько областей, в которых рисуются графики функций, то есть несколько объектов типа axes, то в этом случае сначала нужно щелкнуть мышью на том из них, который должен стать текущим, и только после этого вводить представленный выше код.

Сменить цвет фона легко:

```
set( hAxes, 'Color', [ 0.5,0.8,0.5] );
```

Аналогично меняем цвет фона всего графического окна:

```
FigureColor = [ 0.8,0.5,0.5];  hFigure = gcf;
set( hFigure, 'Color', FigureColor );
```

где с помощью функции gcf мы получаем описатель объекта типа figure, представляющего графическое окно, а далее меняем цвет его фона на красноватый. В результате вместо рисунка, показанного нами ранее для иллюстрации использования жирных линий при показе графика функции sin, мы получаем то же самое графическое окно, в котором цвета фона всего окна и области вывода графика уже иные. При черно-белой печати это видно лишь приблизительно, однако вот полученное при этом изображение (см. рис. 2.11):

Здесь мы еще вывели произвольный (шутливый) текст, расположив его в двух строках:

```
text( 1, 0.3,'Very good\newlinetext' );
```

а также обвели этот текст прямоугольной рамкой:

```
line([0.8,0.8,2,2,0.8],[0.2,0.45,0.45,0.2,0.2]);
```

где в качестве первого аргумента функции line (конструктор графического объекта типа line) использована вектор-строка первых координат углов прямоугольника, а второй аргумент есть вектор-строка вторых координат этих точек.

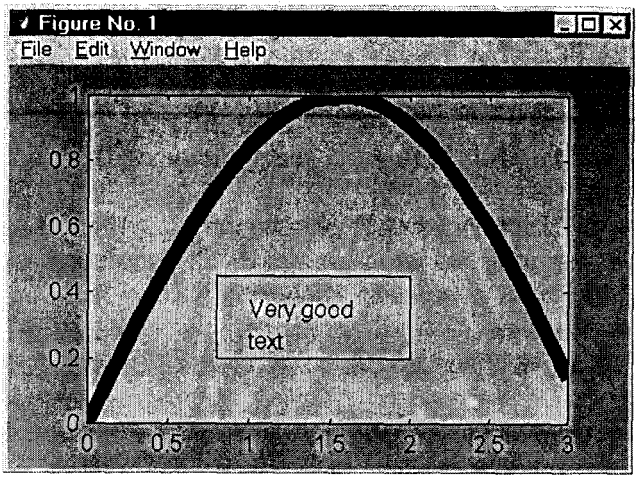


Рисунок 2.11

Разнообразные графические средства системы MATLAB (как высокоуровневые функции, так и низкоуровневые графические объекты) позволяют получить наглядные и хорошо документированные графики функций.

Специальная графика системы MATLAB

Специальная графика системы MATLAB направлена в первую очередь не на выявление функциональных зависимостей между переменными, а на визуализацию информации, накопленной в виде больших массивов числовых данных.

Для визуализации данных, накопленных в одномерном массиве, хорошо подходит специальная графическая функция системы MATLAB – функция `bar`. Будучи примененной к вектору x,

```
x = [ 4 1 8 3 7 4]; bar( x );
```

она порождает наглядную *столбцовую диаграмму* (см. рис. 2.12).

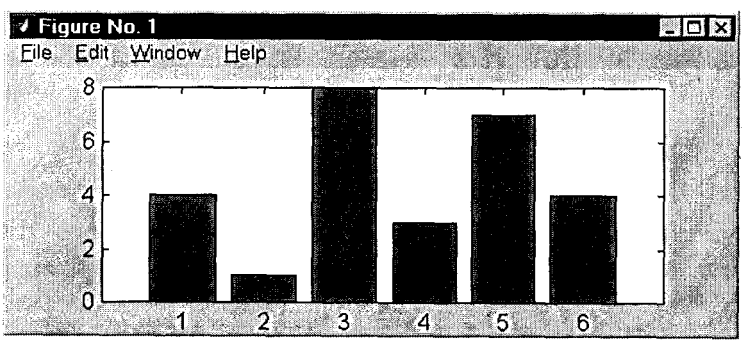


Рисунок 2.12

В столбцовых диаграммах каждому элементу массива соответствует один столбец. Этот принцип сохраняется и при отображении функцией bar содержимого матриц, но для наглядности столбцы, относящиеся к одной строке, группируются. В результате для матрицы

```
A = [ 5 2 4; 1 3 5; 2 1 4 ];
```

функция bar(a) порождает изображение, в котором разные элементы из одной и той же строки изображаются разными цветами. Номер строки проставлен вдоль горизонтальной оси под соответствующей группой столбцов (см. рис. 2.13).

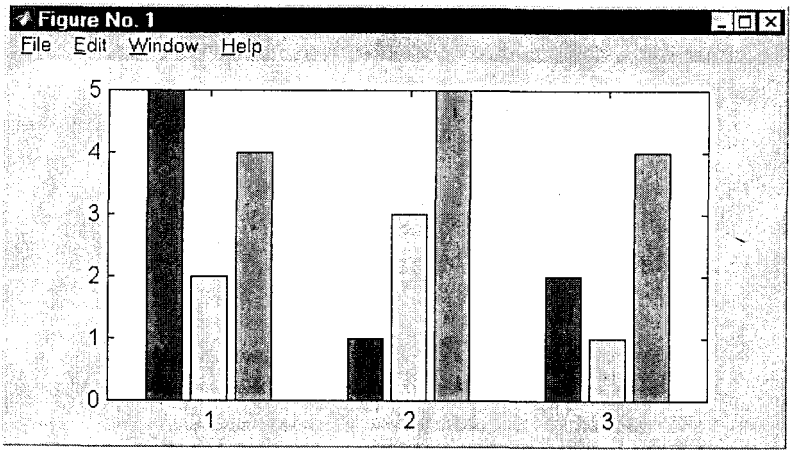


Рисунок 2.13

Наглядность показа элементов матрицы в виде вертикальных столбцов только усилится, если перейти к трехмерным изображениям. Для этого достаточно применить функцию bar3. Например, вызов функции bar3(a) порождает такую картину (см. рис. 2.14):

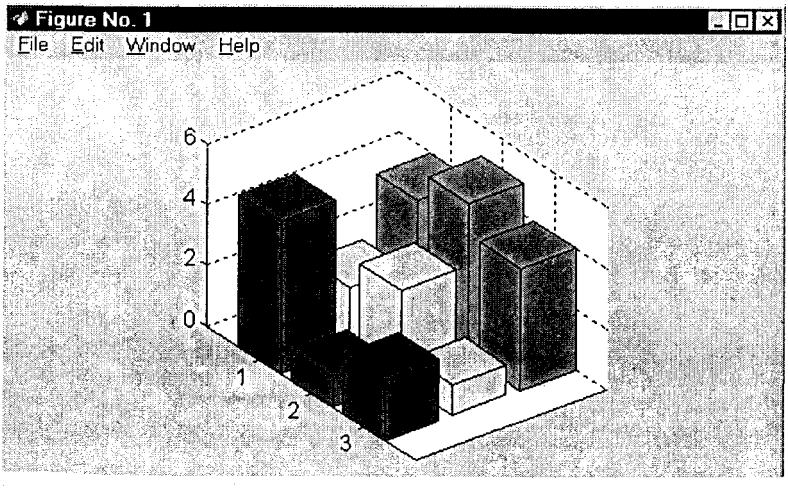


Рисунок 2.14

Здесь напротив цифры 1 расположен набор столбцов, относящихся к первой строке матрицы A, напротив цифры 2 – то же для второй строки этой матрицы, напротив 3 – третьей строки.

Помимо функций для построения столбцовых диаграмм в системе MATLAB имеются также функции для построения *круговых диаграмм*. Круговые диаграммы показывают в наглядной форме, какой процент от суммы всех элементов массива составляет его конкретный элемент. Например, для вектора

```
x = [ 3, 7, 1, 2];
```

круговая диаграмма строится функцией `pie( x )` (см. рис. 2.15).

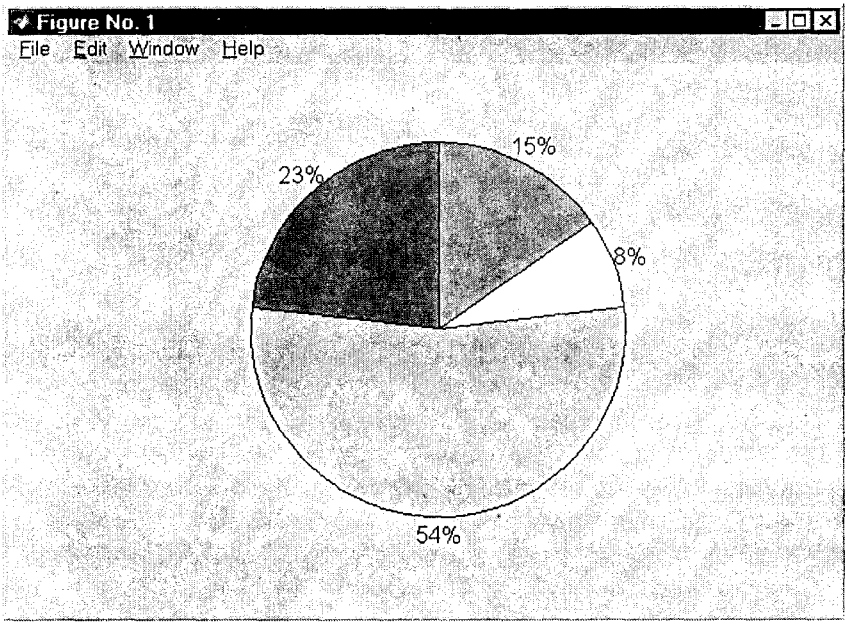


Рисунок 2.15

Отсюда видно, что первый элемент вектора составляет 23% от суммы всех элементов, второй элемент составляет 54% и т. д. против часовой стрелки. Чтобы этот рисунок стал более наглядным, его целесообразно снабдить надписями, говорящими о том, каким элементам вектора соответствуют конкретные куски этого «пирога». Сделать это, однако, не так легко без помощи дескрипторной графики. Вот решение этой задачи:

```
h = pie( x ); hT = findobj(h,'Type','text');
Pos=get(hT(1),'Position'); Str=get(hT(1),'String');
set(hT(1),'String',[ '1-',Str],'Position',Pos+[-0.1,0,0])
```

в котором мы добиваемся изменения надписи напротив куска «пирога», соответствующего первому элементу вектора. Здесь мы применили функцию `findobj`, которая возвращает массив описателей четырех текстовых объектов. Далее мы у первого текстового объекта запрашиваем его позицию и его текстовое содержание, модифицируем их соответствующим образом и с помощью функции `set` устанавливаем новые значения этих свойств. Задача решена.

Кстати, заодно здесь сообщим, что *для удаления* уже проставленной записи нужно функцией `set` установить свойство `'String'` равным *пустому массиву*.

Теперь рассмотрим еще один метод визуализации числовой информации. Когда в массиве сосредоточено большое количество данных, то прежде, чем визуализировать эти данные, их удобно сгруппировать в рамках подынтервалов значений и выводить суммарное количество элементов, попадающих в тот или иной подынтервал. Такой подход к визуализации называется построением *гистограмм* данных.

В системе MATLAB для построения гистограмм применяется функция `hist`:

```
x=rand( 1, 10000 );
hist( x );
```

Здесь с помощью функции `rand` генерируется вектор из `10 000` случайных элементов, равномерно распределенных на интервале от `0` до `1`. Функция `hist` строит гистограмму результатов, разбивая интервал от `0` до `1` на `10` подынтервалов (см. рис. 2.16).

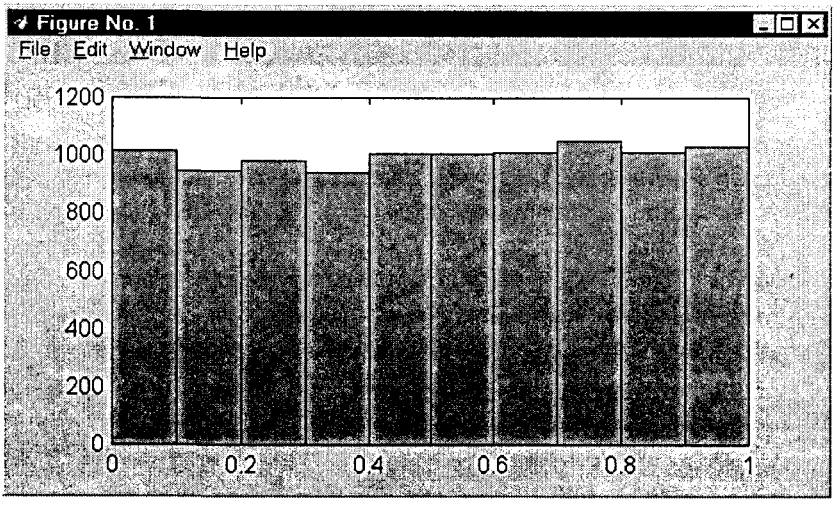


Рисунок 2.16

Для этого функция `hist` подсчитывает количество случайных чисел, попавших в тот или иной подынтервал, и строит столбцы соответствующей длины для

каждого из подынтервалов. Если требуется другое число подынтервалов, то его следует указать в виде второго параметра у функции `hist`.

Помимо изображения элементов массивов в виде столбцов или «кусков пирога» система MATLAB предоставляет еще способ визуализации, когда каждый элемент массива изображается отрезком вертикальной линии с маленьким кружочком на конце. Такие диаграммы строит функция с именем `stem`, что переводится с английского как стебель или ножка, черенок.

Наиболее интересным применением такого рода диаграмм является случай, когда такие «черенки» позиционируются вдоль горизонтальной оси не напротив номеров элементов массива (вызов функции `stem(x)`), а напротив значений другого одномерного массива. Если значения одного массива вычисляются как функции значений другого массива или экспериментально измеряются при таких значениях, то этот сорт диаграмм хорошо соответствует отображению дискретных зависимостей между числовыми данными. Например, для нескольких вычисленных значений функции `asin` (арксинус)

```
x = -1 : 0.1 : 1;
y = asin( x );
```

следующая «черенковая диаграмма», для построения которой вызывается функция

```
stem( x, y )
```

наглядно отображает массив вычисленных значений (см. рис. 2.17).

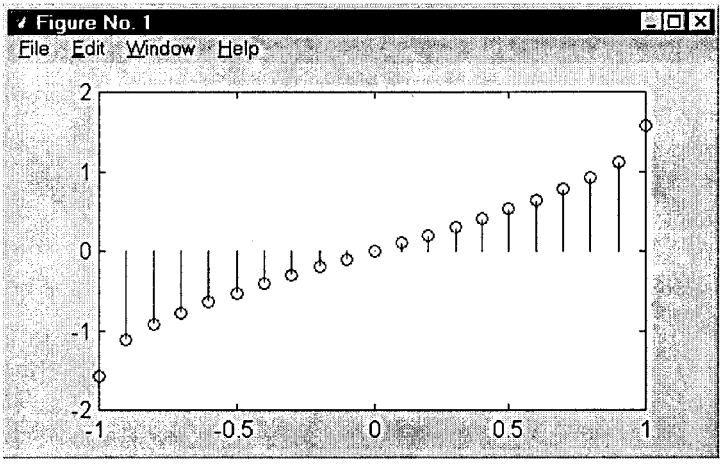


Рисунок 2.17

Для наблюдения за движением объектов по некоторой траектории нужно применить средство *динамической визуализации* системы MATLAB, реализуемое функцией `comet` (комета). Пусть, к примеру, речь идет о наблюдении за равномерным движением по окружности в течение времени `t` от `0` до `20`:

```
t = 0 : 0.1 : 20;
```

```
x = cos( t ); y = sin( t );
```

Движение материальной точки по окружности моделируется наглядно в виде движения абстрактной «кометы» с хвостом, цвет которого отличается от цвета всей траектории. Для этого осуществляется следующий вызов функции comet:

```
comet( x, y )
```

при котором наглядно изображается движение по круглой траектории. Если же нужно наблюдать за динамическим изменением координат x и y со временем t, то нужно применить два других вызова функции comet:

```
comet( t, x ); comet( t, y )
```

каждый из которых разворачивает во времени процесс изменения координат.

Трехмерная графика

Возможности отображения трехмерных графических объектов в системе MATLAB весьма обширны. В частности, имеется возможность решать разнообразные задачи *трехмерного моделирования* объектов реального мира, опираясь на графические объекты типа patch. Это исключительно обширный раздел современной компьютерной графики, излагать который в двух словах бессмысленно, а для более-менее подробного изложения требуется отдельная книга. Поэтому в данном пособии этот вопрос затрагиваться не будет.

Мы сосредоточимся на изображении пространственных линий и на построении графиков функций двух вещественных переменных, которые представляют собой поверхности в пространстве. Начнем с линий.

Каждая точка в пространстве характеризуется тремя координатами. Набор точек, принадлежащих некоторой линии в пространстве, нужно задать в виде трех векторов, первый из которых содержит первые координаты этих точек, второй вектор – вторые их координаты, ну а третий вектор – третьи координаты. После чего эти три вектора можно подать на вход функции plot3, которая и осуществит проектирование соответствующей трехмерной линии на плоскость и построит результирующее изображение. При этом все будет сделано автоматически в лучших традициях высокоуровневой графики системы MATLAB: будут просчитаны пределы изменения переменных по осям координат, нанесены графические и числовые метки, подобран цвет линии и фона и т. д.

Например, следующий фрагмент кода

```
t = 0 : pi/50 : 10*pi ;
x = sin( t );
y = cos( t );
plot3( x , y , t );
grid on
```

где применена известная по плоским графикам команда

```
grid on
```

для проставления сетки координатных значений в области построения графика (также можно использовать ранее изученные команды и функции по дополнительному оформлению графиков), позволяет построить винтовую линию, изображение которой показано на рис. 2.18.

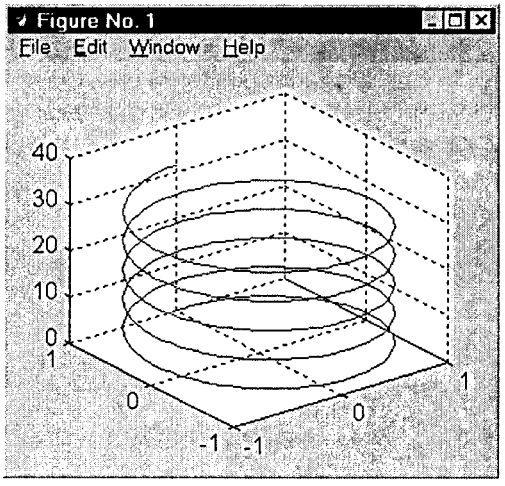


Рисунок 2.18

Эту же функцию `plot3` можно применить и для изображения поверхностей в пространстве, если, конечно, провести не одну линию, а много; конкретно надо провести семейство линий пересечения поверхности с набором параллельных друг другу плоскостей, которые к тому же параллельны одной из координатных плоскостей. Всю эту работу и выполняет функция `plot3`, когда ей на вход подаются не три одномерных массива, как было рассмотрено выше, а три матрицы одинакового размера. Рассмотрим все это подробнее.

Графики функций двух переменных представляют собой куски поверхностей, нависающие над областями определения функций. Отсюда ясно, что изображение графиков функций двух переменных требует реализации «трехмерной графики» на плоском экране дисплея компьютера.

Высокоуровневая графическая подсистема MATLAB автоматически реализует трехмерную графику без специальных усилий со стороны пользователя. Пусть в точке с координатами `x1`, `y1` вычислено значение функции `z = f(x,y)` и оно равно `z1`. В некоторой другой точке (то есть при другом значении аргументов) `x2`, `y2` вычисляют значение функции `z2`. Продолжая этот процесс, получают массив (набор) точек (`x1,y1,z1`), (`x2,y2,z2`), ..., (`xN,yN,zN`) в количестве `N` штук, расположенных в трехмерном пространстве. Специальные функции системы MATLAB проводят через эти точки гладкие поверхности и отображают их проекции на плоский дисплей компьютера.

Чаще всего точки аргументов расположены в области определения функции регулярно в виде прямоугольной сетки (матрицы точек). Такая сетка точек порождает две числовые матрицы одной и той же структуры: первая матрица содержит значения первых координат этих точек (x-координат), а вторая матрица содержит значения вторых координат (y-координат). Обозначим первую матрицу как X, а вторую – как Y. Есть еще и третья матрица – матрица значений функции `z = f(x,y)` при этих аргументах. Эту матрицу обозначим буквой Z.

Как мы и обещали, рассмотрение вопроса о построении графиков функций двух переменных в системе MATLAB начнем с функции `plot3`, которая является простейшей в семействе предназначенных для этой цели функций. Ее вызов для построения таких графиков осуществляется в виде

```
plot3( X , Y , Z )
```

где X, Y и Z – матрицы одинаковых размеров, смысл которых мы только что объяснили.

В системе MATLAB имеется специальная функция для получения двумерных массивов X и Y по одномерным массивам x, y (см. рис. 2.19).

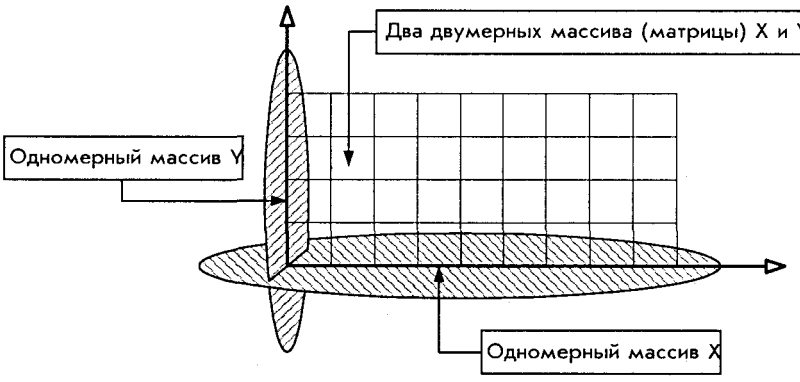


Рисунок 2.19

Пусть по оси x задан диапазон значений в виде вектора

```
u = -2 : 0.1 : 2
```

а по оси y этот диапазон есть

```
v = -1 : 0.1 : 1
```

Для получения матриц X и Y, представляющих первые и вторые координаты получающейся прямоугольной сетки точек, используют специальную функцию системы MATLAB

```
[ X , Y ] = meshgrid( u, v )
```

Как мы видим, эта функция получает на входе два одномерных массива (вектора), представляющие массивы точек на осях координат, и возвращает сразу

два искомых двумерных массива. На прямоугольной сетке точек вычисляем значения функции, например функции exp:

```
Z = exp( -X.^2 - Y.^2 )
```

Наконец, применяя описанную выше функцию plot3, получаем следующее изображение трехмерного графика этой функции (см. рис. 2.20):

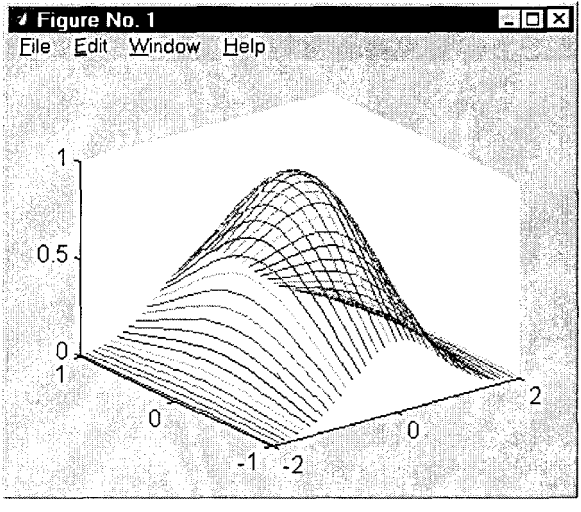


Рисунок 2.20

Из рисунка видно, что функция plot3 строит график в виде набора линий в пространстве (создается 41 графический объект типа line), каждая из которых является сечением трехмерной поверхности плоскостями, параллельными плоскости yOz. По-другому можно сказать, что каждая линия получается из отрезков прямых, соединяющих набор точек, координаты которых берутся из одинаковых столбцов матриц X, Y и Z. То есть, первая линия соответствует первым столбцам матриц X, Y, Z; вторая линия – вторым столбцам этих матриц и т. д.

Помимо этой простейшей функции система MATLAB располагает еще рядом функций, позволяющих добиваться большей реалистичности в изображении трехмерных графиков. Это функции mesh, surf и surfl. Они порождают графические объекты типа surface.

Функция mesh соединяет друг с другом все соседние точки поверхности графика отрезками прямых и показывает в графическом окне системы MATLAB плоскую проекцию такого объемного *каркасно-ребристого* (по-английски – *wireframe mesh*) тела. Каркасно-ребристое тело состоит из четырехугольных граней белого цвета, а ребра граней окрашиваются в разные цвета. По умолчанию более высоким точкам графика соответствуют красные цвета, а более низким (меньшие значения третьей координаты) – темно-синие. Промежуточные области окрашиваются в светло-синие, зеленые и желтые цвета. В результате,

поскольку разные области поверхности графика (конкретно – ребра каркасного тела) окрашиваются в разные цвета, применение функции `mesh` порождает весьма наглядное изображение трехмерного графика.

К примеру, вместо ранее показанного при помощи функции `plot3` графика функции

```
exp( -X.^2 - Y.^2 )
```

состоящего из 41 пространственной линии, можно вызовом функции

```
hS1 = mesh( X, Y, Z );
```

получить вот такое изображение каркасно-ребристого тела (см. рис. 2.21):

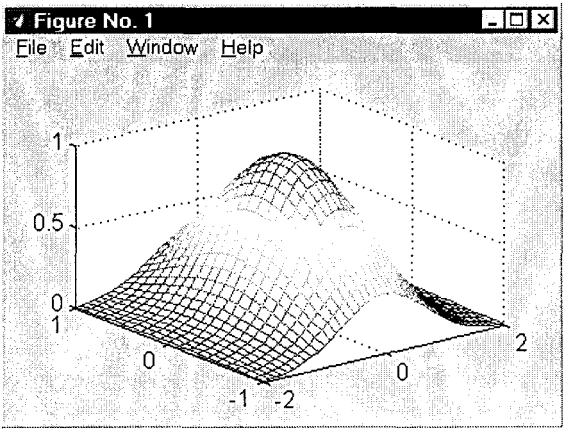


Рисунок 2.21

Как мы уже говорили, для лучшего восприятия «объемности изображения» разные ребра автоматически окрашиваются в разные цвета. Кроме того, в отличие от функции `plot3` осуществляется *удаление невидимых линий*. Если вы считаете, что изображенное ребристое тело является прозрачным и не должно скрывать задних ребер, то можно ввести команду

```
hidden off
```

после чего такие линии появятся на изображении. В одних случаях это позволяет улучшить изображение, в других – нет. Всегда можно вернуться к имеющему место по умолчанию режиму сокрытия при помощи команды

```
hidden on
```

Кроме того, можно потребовать изображения отдельных ребер переменным цветом, чтобы цвет изменялся более плавно при переходе снизу вверх (по умолчанию цвет всех точек отдельного ребра одинаковый). Для этого нужно дополнительно ввести команду

```
shading interp
```

которая изменит одно из свойств графического объекта типа surface, создаваемого функцией mesh. Описатель такого объекта возвращается этой функцией, и выше мы запомнили его в переменной hS1. Убедиться в том, что создается графический объект именно этого типа, можно следующим образом:

```
get( hS1, 'Type' )
ans =
  surface
```

Вместо применения команды shading interp можно напрямую воздействовать на свойство EdgeColor построенного объекта типа surface:

```
set( hS1, 'FaceColor', 'interp' )
```

и эффект будет тем же самым. Разница лишь в том, что применение высокоуровневой команды позволяет обойтись без знания мелких деталей низкоуровневого графического объекта. Однако знание таких деталей открывает больше возможностей. В любом случае ознакомиться со списком всех свойств графического объекта типа surface можно следующим образом:

```
get( hS1 )
   CData = [(21 by 41) double array]
   ...
   EdgeColor = interp
   ...
   Visible = on
```

где мы из реального большого списка привели лишь первое, последнее и нужное нам сейчас свойства.

Более «плотного» изображения поверхности можно добиться за счет раскраски разными цветами не ребер, а граней каркасно-ребристого тела. Для этого вместо функции mesh нужно применить функцию surf:

```
surf( X, Y, Z )
```

В результате получается следующее изображение, представляющее плотную (непрозрачную) сетчатую поверхность, причем отдельные ячейки (грани) этой сетчатой поверхности (плоские четырехугольники) автоматически окрашиваются в разные цвета (см. рис. 2.22). Как и в случае ранее рассмотренной функции mesh, здесь также по умолчанию более высокие точки графика окрашиваются в красный цвет, а более низкие – в темно-синий. При этом все ребра изображаются черным цветом. Командой shading interp или соответствующими приемами низкоуровневой графики (были рассмотрены выше) можно добиться плавного перехода цветов граней, а прорисовку ребер убрать вообще (свойства объекта surface при этом следующие: EdgeColor = none; FaceColor = interp).

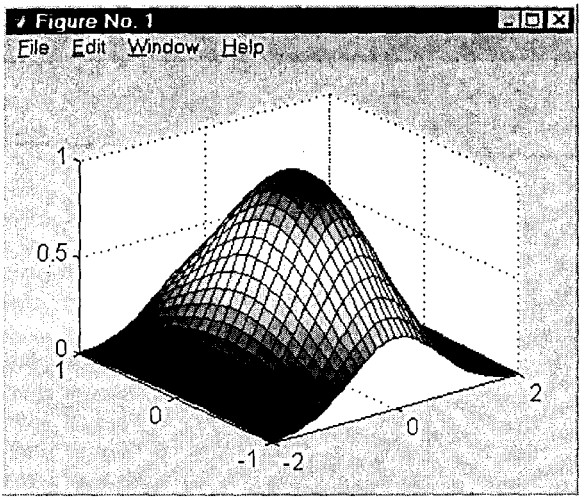


Рисунок 2.22

С помощью функции `surf` получаются хотя и искусственно раскрашенные, но весьма наглядные изображения. Если же мы хотим добиться более естественных и объективных способов окрашивания поверхностей, то следует использовать функцию `surfl`.

Функция `surfl` в отличие от функции `surf` не применяет искусственных приемов закраски поверхности трехмерных графиков. Упрощенно говоря (детальные разъяснения требуют отдельной книги), эта функция сразу же использует понятие освещения поверхности графика. По умолчанию она использует встроенную засветку графика со стороны некоторой геометрической точки пространства. В результате точки поверхности графика, обращенные в сторону источника света (условно более яркие), изображаются красным цветом, а точки, находящиеся «в тени», – темно-синим.

Если далее трактовать поверхность графика как *материальную поверхность* с определенными физическими свойствами по отражению света, то нужно задать эти свойства явно. Так как разные материалы по-разному отражают падающие лучи, то можно подобрать некоторый материал, чтобы получить наилучшее (с точки зрения пользователя) изображение. В частности, можно применить функцию

```
colormap( copper )
```

с помощью которой для изображения графика выбирается набор цветов (по-английски – *colormap*), который характерен для света, отражающегося от медной поверхности (медь по-английски – *copper*). После этого применение функции

```
surfl( X, Y, Z )
```

приводит к получению реалистически выглядящего и очень наглядного графика (см. рис. 2.23).

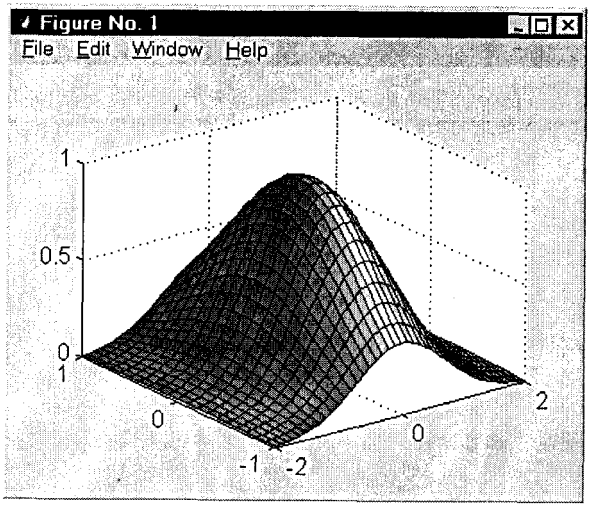


Рисунок 2.23

Можно с такого графика убрать черные линии, изображающие ребра, а также добиться еще более плавного перехода освещения поверхности, если выполнить команду

```
shading interp
```

означающую, что теперь цвет (освещенность) будет меняться даже внутри отдельных граней (ячеек). В итоге будет получаться совсем уж реальное изображение некоторого трехмерного материального объекта (см. рис. 2.24).

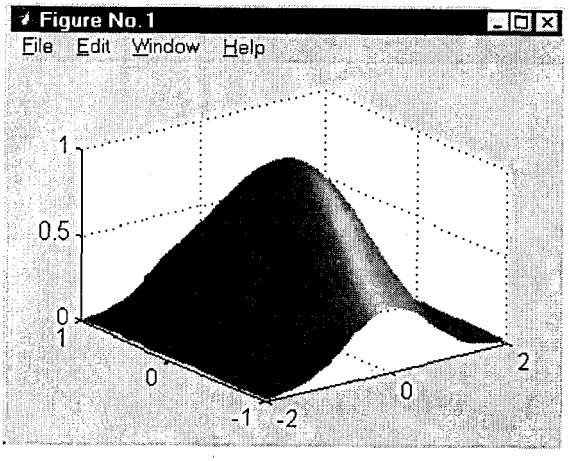


Рисунок 2.24

Тем не менее система MATLAB на этом не останавливается: существует возможность еще точнее проявить отдельные участки поверхности графика, включив *дополнительные источники освещения*, которые реализуются в виде графических объектов типа `light`. Свет от таких источников смешивается

с рассеянным фоновым освещением (AmbientLight), цвет и сила которого встроены в виде характеристик в объекты axes (свойство AmbientLightColor) и surface (свойство AmbientStrength).

В следующем примере:

```
hS = surfl( X, Y, Z );
set(hS,'FaceLighting','phong','FaceColor','interp');
set(hS,'AmbientStrength',0.5);
light('Position',[1 0 0],'Style','infinite');
```

задаются свойства поверхности (свойства графического объекта типа surface), которые влияют на результирующее изображение при использовании дополнительного источника света. После чего создается такой источник в позиции, определяемой его свойством Position, испускающий *параллельные лучи* (Style=infinite, то есть *бесконечно удаленный источник*).

Можно также задать цвет дополнительного источника подсветки поверхности графика (свойство Color объекта light). Чтобы сделать источник более сильным, нужно продублировать его несколько раз, создавая все новые источники в том же месте и с теми же свойствами. При этом описатели каждого вновь создаваемого источника света нужно запоминать в соответствующих переменных. Тогда впоследствии можно с помощью функции set изменять их свойства и суммарную силу света (для «выключения» конкретного источника нужно задать его свойство Color равным black), добиваясь желаемого внешнего вида графика. В этом заключается огромное преимущество работы с системой MATLAB в интерактивном режиме. Можно не спеша оценить достигнутый результат и, если что-то не нравится, выполнить отдельные части работы (а не всю работу) заново.

Дополнительные детали оформления трехмерных графиков

Многие приемы оформления трехмерных графиков совпадают с теми, что были рассмотрены при изучении плоских графиков функций одного переменного. В частности, для масштабирования удобно использовать функцию axis, которая в трехмерном случае принимает уже три пары скалярных аргументов:

```
axis( [ xmin, xmax, ymin, ymax, zmin, zmax ] )
```

По-прежнему можно использовать функции text, xlabel, ylabel, zlabel, title, а также можно наносить отметки на осях координат с помощью функции set. Можно также с помощью функции subplot разместить в одном графическом окне несколько трехмерных графиков.

К новым методам дополнительного оформления трехмерных графиков можно отнести возможность вызывать функцию `mesh` с суффиксами `z` и `c` (`meshz` и `meshc`), а функцию `surf` – с суффиксом `c` (`surfc`). Использование суффикса `z` приводит к построению графика с *пьедесталом*. Например, фрагмент кода

```
[X,Y] = meshgrid( -2 : 0.1 : 2 );
Z = X .* exp( -X.^2 - Y.^2 ); meshz( X, Y, Z )
```

строит следующий график (см. рис 2.25):

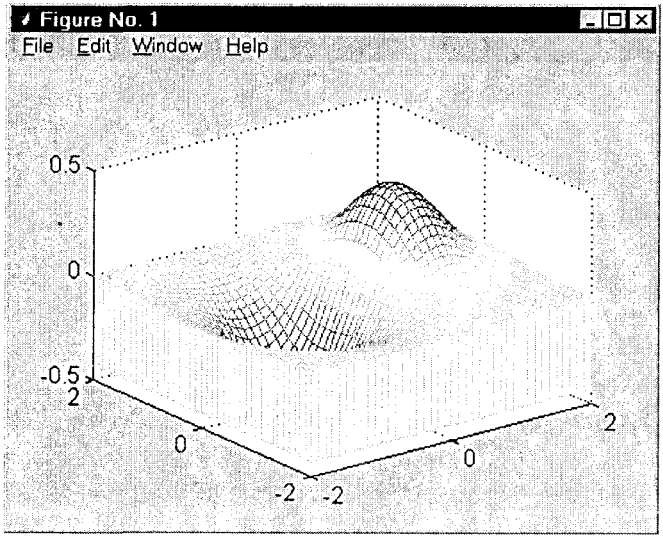


Рисунок 2.25

Функции с суффиксом `c` помимо собственно трехмерного графика строят еще и так называемые *линии уровня*. Например, фрагмент

```
[X,Y,Z] = peaks(30);   surfc(X,Y,Z);
colormap( hsv );       axis([-3 3 -3 3 -10 5]);
```

приводит к следующему изображению (см. рис. 2.26):

Функция с именем `peaks` (является некоторой масштабированной комбинацией стандартных гауссовых функций) часто употребляется в примерах к системе MATLAB для наглядной иллюстрации графических функций.

Наконец, для трехмерных графиков существует возможность изменять свойства условной *камеры*, фиксирующей сцену с графиком: можно менять положение этой камеры, расстояние от сцены, а также свойства ее *объектива*. Это мощный способ влиять на детали изображения. Изменять свойства такой камеры можно с помощью дескрипторного метода. Как мы знаем, графическим объектом, соответствующим поверхности трехмерного графика функции, является объект типа `surface`. Дескриптор (описатель) такого объекта возвращается функциями `mesh`, `surf` и `surfl`. Объект `surface` всегда принадлежит объекту

axes. Получить описатель объекта axes можно, вызвав функцию gca, которая и возвращает описатель текущего объекта axes (если имеется несколько таких объектов, то перед вызовом функции нужно щелкнуть мышью на требуемом объекте axes). Получив описатель, можно функцией set изменить свойства объекта axes, в частности свойства, связанные с камерой, фиксирующей сцену с расположенным на ней трехмерным графиком. Приведем список свойств объекта axes, имеющих отношение к этому вопросу:

```
get( hA )
...
  CameraPosition
  CameraPositionMode
  CameraTarget
  CameraTargetMode
  CameraUpVector
  CameraUpVectorMode
  CameraViewAngle
  CameraViewAngleMode
...
```

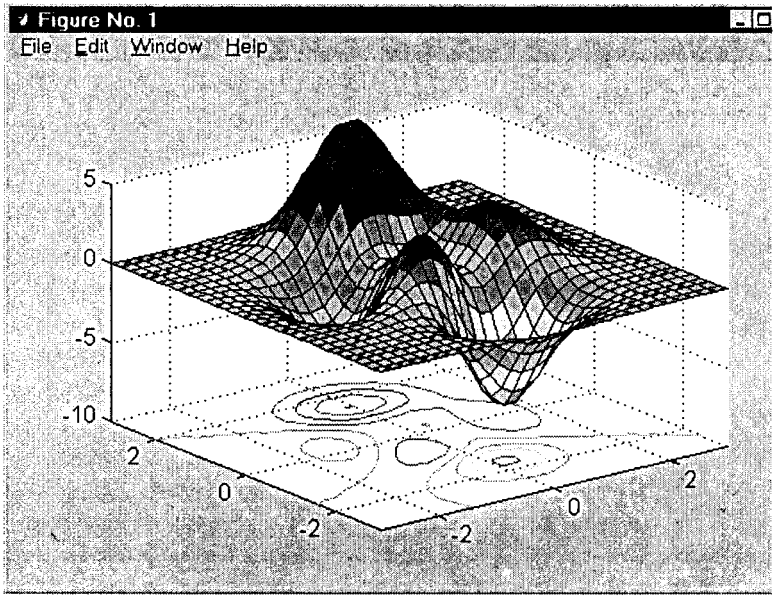


Рисунок 2.26

Более простым по сравнению с изменением свойств условной камеры наблюдения является метод изменения *точки обзора* (*viewpoint*). В частности, для точки обзора нет понятия расстояния до сцены и, тем более, свойств объектива. Для нее можно менять только углы, задающие ориентацию этой точки в пространст-

ве: *угол азимута* (часто обозначают как `az`) и *угол возвышения* (часто обозначают `el`). Изменение первого угла означает вращение плоскости `xOy` вокруг оси `Oz` против часовой стрелки. Угол возвышения есть угол между направлением на точку обзора и плоскостью `xOy`.

Когда выполняются высокоуровневые графические функции `mesh`, `surf` или `surfl`, то по умолчанию устанавливаются значения `az = -37.5°`, `el = 30°`. Эти значения в любой момент времени можно изменить специальной функцией

```
view( [ az , el ] )
```

где названия аргументов говорят сами за себя. В частности, если после построения показанного выше графика с пьедесталом для функции

```
X .* exp( -X.^2 - Y.^2 )
```

выполнить команду

```
view( [ -15 , 20 ] )
```

то график изменит свой вид, поскольку мы уменьшили угол возвышения с `30°` до `20°` градусов, а угол азимута изменили с «умолчательного значения» в `-37.5` градуса на значение, равное `-15` градусам. В итоге мы теперь глядим на график больше сбоку, а не сверху, и преимущественно вдоль одной из независимых координат (см. рис. 2.27).

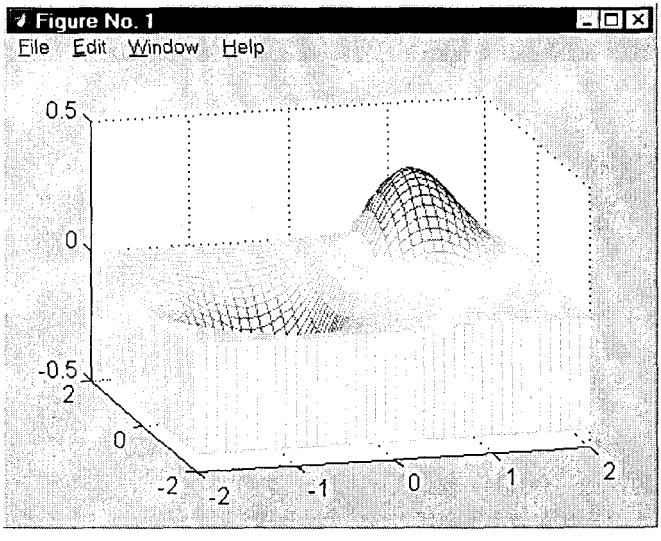


Рисунок 2.27

Подводя итоги, заметим, что система MATLAB обладает огромными, трудно поддающимися обзору возможностями по оформлению графиков функций. Все рассмотренные до сих пор возможности основаны на так называемой *векторной графике*, когда изображаемый объект задан своими координатами (числовыми

данными, накопленными в массивах), которые сама система MATLAB в момент отображения переводит в значения (цвет) пикселов дисплея. Векторная графика прекрасно поддается масштабированию.

Однако существуют графические объекты реального мира, заданные в *растровой форме*, например сосканированные для ввода в компьютер фотографии. Система MATLAB обладает также развитыми средствами работы с растровой графикой. Ей посвящен следующий подраздел данной главы.

Растровые изображения и тип данных uint8

Любое изображение на дисплее компьютера (в том числе и графики функций, полученные в результате работы векторной графической подсистемы пакета MATLAB) представляет собой массив пикселов, каждый из которых характеризуется своим цветом.

Чтобы *сохранить* полученные графики функций как *растровые изображения в файлах на диске компьютера*, нужно выполнить следующую команду:

```
print -options FileName
```

где имя файла выбирается произвольно, но если не указать полный путь к файлу, то запись произойдет только в текущий каталог системы MATLAB.

Параметр `options` определяет *формат графического файла*. Например, `dbitmap` задает стандартный растровый формат операционной системы Windows; значение `dmeta` создает так называемый *метафайл* (в строгом смысле это не формат растровой графики, так как в файл записываются команды, а не пикселы); значение `dill` соответствует графическому формату известного пакета иллюстративной графики *Adobe Illustrator*. В последнем случае сохраненный файл можно будет открывать в этом пакете и подвергать его содержимое дальнейшей обработке средствами пакета Illustrator.

Сохраненные в файлах на диске компьютера изображения, полученные ранее средствами пакета MATLAB, можно вставить в текст документа редактора Microsoft Word в том случае, если вместе с этим текстовым редактором были инсталлированы конвертеры файлов данного графического формата.

Еще проще передать изображение из системы MATLAB в текстовый редактор Microsoft Word *посредством буфера обмена операционной системы Windows*. Чтобы поместить изображение из графического окна системы MATLAB в буфер обмена, нужно выполнить команды

```
print -dbitmap  или  print -dmeta
```

которые вводятся с клавиатуры в командном окне, или выполнить *команду меню* `Edit|CopyFigure`. В документ редактора Microsoft Word изображение вставля-

ется командой его главного меню Edit|Paste. Далее вместе со всем документом это изображение можно будет распечатать на принтере.

Очень важное замечание: перед сохранением изображения в файле или в буфере обмена Windows осуществите в рамках пакета MATLAB необходимое *масштабирование*, то есть добейтесь необходимого физического размера картинки. После сохранения изображения в форматах растровой графики дальнейшее масштабирование осуществляется с неприемлемыми искажениями.

Итак, мы уже научились сохранять растровые изображения, соответствующие графическим окнам системы MATLAB, в буфере обмена Windows и в файлах некоторых графических форматов. Существует еще третий способ «запечатлеть на память» картинку из графического окна. Для этого достаточно вызвать функцию capture, которая позволит сохранить информацию о растровой картинке из графического окна системы MATLAB в двух числовых массивах:

```
[ X, map ] = capture( 1 );
```

Входным параметром для функции capture является номер графического окна. Выходными значениями для функции capture являются матрица X, соответствующая матрице пикселов изображения (в ней столько же рядов, сколько пикселовых строк в изображении, и столько же столбцов, – сколько реальных пикселовых столбцов), и матрица цветов map (три столбца в формате RGB), использованная для построения изображения. При этом каждый элемент матрицы X равен номеру одной из строк матрицы map. Каждый элемент матрицы цветов map представляет собой положительное число от 0 до 1 – оно характеризует интенсивность соответствующей составляющей цвета (красной, зеленой или синей). Итак, этих двух матриц достаточно для запоминания информации о цветах всех пикселов в растровом изображении.

Прежде чем продолжить детальное изучение этих матриц, кратко перечислим основные варианты их практического использования. Во-первых, располагая этими матрицами, можно записать изображение в JPEG-файлы, удобные для использования в среде Internet. Это делается вызовом функции imwrite:

```
imwrite( X, map, 'MyOwnName.jpg' );
```

Здесь создается сжатый графический файл MyOwnName.jpg, который удобен для передачи по сети Internet и предназначен для просмотра в среде браузеров типа *Microsoft Internet Explorer*.

Во-вторых, располагая матрицами X и map, можно либо сразу же восстановить исходное изображение в рамках графического окна системы MATLAB, применив следующий код:

```
colormap( map ); image( X );
```

либо, предварительно обработав исходное изображение (произведя вычисления над элементами полученных функцией capture матриц), вывести в графическое окно системы MATLAB уже новое изображение.

Очень важно понимать, что функция image не воссоздает всю исходную информацию, характерную для векторной графики. Она лишь формирует массив пикселов с правильными цветами, и больше ничего. Она создает *объект растровой графики* по имени image. Этот объект можно масштабировать, изменяя обычным образом размеры графического окна, но картинка при этом будет искажаться. Это отличительная черта растровой графики, и с этим ничего нельзя поделать. Такова природа вещей.

Теперь продолжим подробное изучение строения матриц, возвращаемых функцией capture и являющихся основой растровой графики системы MATLAB.

Еще раз напомним, что произвольное изображение на экране компьютера представляет собой массив пикселов, каждый из которых характеризуется своим цветом. Цвет пиксела определяется тремя составляющими: красным, зеленым и синим (Red, Green, Blue – RGB). Каждая составляющая цвета, как мы уже говорили выше, кодируется вещественным числом от 0 до 1. В результате на каждый пиксел расходуется по 8 x 3 = 24 байта памяти компьютера. Это очень расточительный по отношению к памяти компьютера способ хранения информации о растровых изображениях. А неэкономный расход памяти, в свою очередь, сильно понижает производительность (быстродействие).

В то же время для задания величины одной составляющей цвета пиксела достаточно 1 байта памяти (8 бит), где можно записать целые числа от 0 до 255 (всего 256 значений). Каждому пикселу экрана в таком случае будут соответствовать три целых числа в диапазоне от 0 до 255, которые займут в памяти компьютера всего лишь 3 байта памяти. Это очень значительная экономия памяти компьютера, поэтому в системе MATLAB для таких целых чисел специально создан соответствующий тип данных, обозначаемый как uint8. Под такой тип данных отводится в памяти всего 1 байт вместо 8 байт для обычных вещественных (дробных) чисел типа double.

По умолчанию любой переменной в системе MATLAB ставится в соответствие тип double независимо от числовых значений, которые вы присваиваете переменным. Например, в результате строки кода

```
iVar1 = 128;
```

создается переменная с именем iVar1 и типом double, которой присваивается значение 128. Для хранения такого значения достаточно 1 байта памяти, однако для переменной iVar1 типа double отводится 8 байт памяти (естественно, мы здесь говорим только о собственно данных, так как помимо данных для любой переменной MATLAB отводит еще и память под вспомогательную информацию, необходимую для управлением структурой массивов, а любой объект в системе MATLAB в своем внутреннем представлении является массивом). Налицо явный перерасход памяти компьютера.

Чтобы избежать такого перерасхода памяти, переменную нужно *явно объявлять* как целую, используя *модификатор* `uint8`:

```
iVar2 = uint8( 128 );
```

Так созданная переменная `iVar2` считается целой переменной (а не вещественной), и под нее отводится 1 байт памяти. Такие переменные в системе MATLAB специально *предназначены для хранения целых значений* от `0` до `255` (с целью экономии памяти) и *не предназначены для вычислений*! По-крайней мере в версии MATLAB 5.2 это еще так. В результате для фрагмента

```
iVar2 = iVar2 + 1;
```

получаем сообщение об ошибке

```
??? Function '+' not defined for variables of class 'uint8'.
```

дословно означающее, что операция «сложение» для переменных типа `uint8` не определена.

Чтобы узнать, какой тип имеет та или иная переменная из рабочего пространства системы MATLAB, нужно ввести и выполнить команду `whos`, в результате чего в командном окне появится сообщение, из которого видно, что `iVar1` является массивом размера `1 x 1` (то есть фактически скаляром) типа `double` и занимает в памяти 8 байт, а `iVar2` имеет тип `uint8` и занимает в памяти только 1 байт (в 8 раз меньше). При этом обе переменные имеют одинаковые значения (см. рис. 2.28).

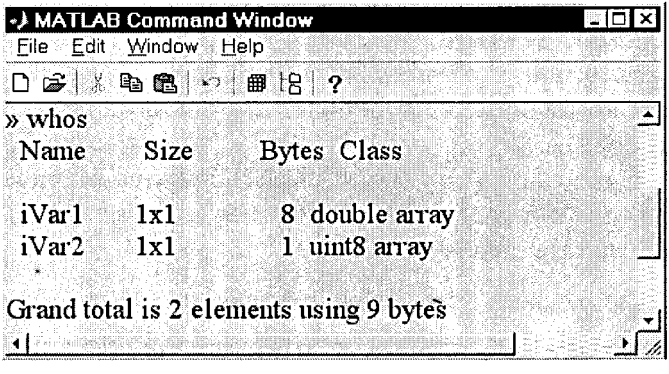


Рисунок 2.28

Воспользуемся, наконец, типом данных `uint8` для экономии памяти компьютера в задаче хранения растрового изображения. Мы уже знаем, что следующий вызов функции `capture`:

```
[X,map] = capture(1)
```

создает матрицу `X` размером `m x n`, где `m` – число рядов пикселов в изображении, а `n` – число столбцов пикселов в нем. Обычно это очень большие числа, так что

количество элементов матрицы X обычно равно нескольким десяткам тысяч и более. Именно эту матрицу система MATLAB позволяет конвертировать в тип uint8:

```
X = uint8( X - 1 );
```

Теперь каждый элемент матрицы X занимает только 1 байт памяти, что приводит к восьмикратной экономии памяти компьютера. Небольшой платой за такую экономию является следующая путающая пользователя деталь: элементы матрицы X по-прежнему имеют целочисленные значения, но каждое значение теперь на единицу меньше своего прежнего значения (которое было при типе double матрицы X). Таким образом, если в случае типа double у матрицы X каждый ее элемент равен номеру некоторой строки матрицы цветов map, то в случае типа uint8 элементы матрицы X являются смещениями этих же строк от начала матрицы цветов map и тем самым они должны быть на единицу меньше своих предыдущих значений. Действительно, для первой строки матрицы map ее смещение от начала равно нулю, вторая строка имеет смещение 1 и т. д.

Еще раз напомним, что тип данных uint8 предназначен для хранения информации, а не для вычислений. Если требуется произвести над изображением какие-то численные преобразования, то перед этим матрицу X типа unit8 надо конвертировать в матрицу типа double

```
X = double( X ) + 1;
```

произвести вычисления, а затем снова можно для экономии памяти превратить результат в тип uint8. Конвертация производится в обе стороны с учетом соответствующего уменьшения или увеличения элементов матрицы X на единицу. Функция image принимает в качестве аргумента оба этих типа данных и работает с ними адекватно.

Набор цветов (в количестве m штук), называемый *палитрой* (colormap), можно оформить в виде матрицы размером m x 3 типа double. Эту матрицу система MATLAB позволяет хранить (и далее использовать) только в этом виде. Матрицу цветов можно получить функцией capture, а можно и сформировать в результате вычислений или прямых присваиваний. Например, матрица map1

```
map1(1,1) = 0.12;  map1(1,2) = 0.123; map1(1,3) = 0.987;
map1(2,1) = 0.456; map1(2,2) = 0.7;   map1(2,3) = 0.22;
map1(3,1) = 0.88;  map1(3,2) = 0.19;  map1(3,3) = 0.611;
map1(4,1) = 0.255; map1(4,2) = 0.298; map1(4,3) = 0.128;
map1(5,1) = 0.01;  map1(5,2) = 0.78;  map1(5,3) = 0.60;
```

задает набор из пяти цветов. Каждая строка соответствует одному цвету. Элементы строки (слева направо) задают красную, зеленую и синию составляющие цвета.

Далее сформируем матрицу k x L типа uint8, каждый элемент которой будет равен одному из номеров (*минус единица*) строк таблицы цветов map1. Такой матрицы вместе с матрицей цветов будет достаточно, чтобы показать на экране компьютера массив пикселов, то есть произвольное растровое изображение.

Например, матрица X1

```
X1 = uint8( [ 1 4 1 3 2; 4 0 2 1 3 ] )
```

задает массив типа uint8 размером 2 x 5 пикселов. Этот массив занимает в памяти 10 байт, а не 80 байт, как было бы в случае массива типа double. Согласно матрице X1 первый пиксел в первом ряду имеет цвет, задаваемый второй строкой матрицы map1, второй пиксел в этом же ряду соответствует пятой строке матрицы map1 и т. д.

Итак, для примера мы вручную сформировали матрицу цветов map1 типа double, состоящую из 5 строк (задает пять цветов), и матрицу X1 размером 2 x 5, формирующую растровое изображение из 10 пикселов (2 ряда по 5 пиксел в каждом). Чтобы заставить систему MATLAB отобразить в одном из своих графических окон сформированную нами картину пикселов, вызываем функции image и colormap:

```
image( X1 ); colormap( map1 );
```

В результате создается графический объект системы MATLAB типа image, которому в графическом окне соответствует следующее растровое изображение (см. рис. 2.29).

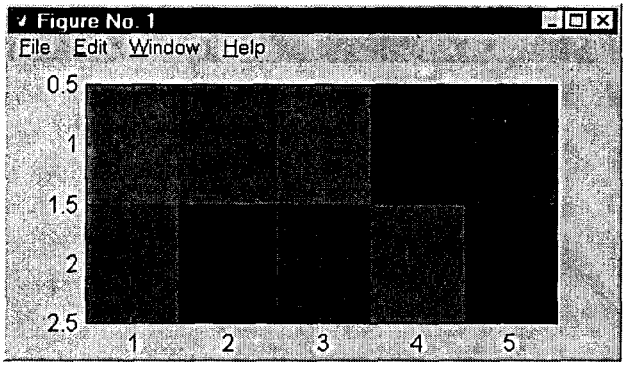


Рисунок 2.29

Поскольку мы не управляем размером графического окна системы MATLAB, то оно появляется на экране с некоторым размером, заданным по умолчанию. Так как наше изображение состоит из двух рядов по 5 пикселов в каждом, а это очень мелкое изображение (физический размер пиксела экрана примерно равен 0,2 мм), то MATLAB по умолчанию масштабирует его (увеличивает), чтобы можно было разглядеть это изображение. Если требуется отменить такое масштабирование, то следует явно указать нужные размеры:

```
[ m , n ] = size( X1 );
figure( 'Units', 'pixels', 'Position', [100 100 n m] );
image( X1 ); colormap( map1 );
set(gca,'Position',[0 0 1 1]);
```

Здесь размеры m и n изображения X1 навязываются в качестве физического размера картинки в графическом окне системы MATLAB. Для слишком маленьких картинок при этом ничего хорошего не получится, однако для изображений графиков функций это как раз самый подходящий вариант, иначе сразу же будут сильные искажения исходного изображения. Последняя инструкция (оператор) из представленного фрагмента кода заставляет объект axes заполнить собой все внутреннее пространство графического окна. Это нужно делать в обязательном порядке для изображений, ранее сохраненных в массивах данных командой capture, так как эта команда сохраняет информацию о всех пикселах внутренней (клиентской) области графического окна системы MATLAB.

Чтобы отобразить в графическом окне системы MATLAB уже готовые картинки, записанные в файлах, нужно прочесть содержимое этих файлов функцией imread. Ранее мы записывали изображения в файлы с помощью функции imwrite. Теперь их можно прочесть:

```
[ X2, map2 ] = imread( 'myfile1.jpg' )
```

и показать в графическом окне. Заметим только, что файл должен быть в текущем каталоге системы MATLAB, иначе его нужно указать вместе с полным путем к нему.

Рассмотренные данные для объекта image состоят из двух матриц, одна из которых построчно задает цвета, а вторая своими элементами указывает входы в таблицу (матрицу) цветов. Этот вариант строения объекта типа image называется более точно как indexed image (*индексированное изображение*). Есть и другой тип объекта image – так называемый truecolor image (картинки с очень большим количеством цветов – до 16 миллионов). Этот второй тип объектов image устроен по-другому.

Для объектов truecolor image таблица цветов не требуется, так как массивы данных таких объектов непосредственно определяют цвета. Эти массивы имеют размер m x n x 3, то есть являются массивами размерности 3. Величины m и n определяют размер картинки на экране (m x n пикселов), а вдоль третьего направления располагаются RGB-составляющие цвета каждого пиксела.

Трехмерные массивы данных для объектов truecolor image могут иметь тип double или тип uint8. В первом случае каждый элемент такого массива является вещественным числом от 0 до 1 и занимает в памяти компьютера 8 байт, а во втором случае каждый элемент является целым числом типа uint8 со значением от 0 до 255 и занимает в памяти компьютера 1 байт. Последний случай

предпочтительнее. Зададим, к примеру, следующий трехмерный массив типа uint8 для изображения truecolor:

```
xTrue(1,1,1)=uint8(127); xTrue(1,1,2)=uint8(127);
xTrue(1,1,3)=uint8(127); xTrue(1,2,1)=uint8(19);
xTrue(1,2,2)=uint8(12);  xTrue(1,2,3)=uint8(255);
xTrue(1,3,1)=uint8(245); xTrue(1,3,2)=uint8(127);
xTrue(1,3,3)=uint8(1);   xTrue(2,1,1)=uint8(6);
xTrue(2,1,2)=uint8(203); xTrue(2,1,3)=uint8(128);
xTrue(2,2,1)=uint8(100); xTrue(2,2,2)=uint8(1);
xTrue(2,2,3)=uint8(80);  xTrue(2,3,1)=uint8(60);
xTrue(2,3,2)=uint8(249); xTrue(2,3,3)=uint8(5);
```

Массив xTrue создает изображение 2 x 3 пикселов с помощью вызова одной функции image(xTrue) (см. рис. 2.30).

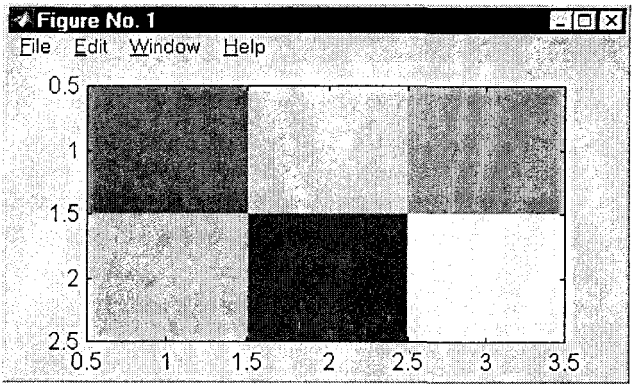


Рисунок 2.30

Если изображение находится в файле и вы заранее не знаете, какой оно имеет тип (индексное, то есть с палитрой цветов, или truecolor), то все равно его следует читать обычным образом:

```
[ X, map ] = imread( 'name.xxx' )
```

В случае truecolor изображений массив X получит размер m x n x 3, а матрица палитры map будет *пустой*:

```
size( map ) =
           0  0
```

В дальнейшем функция image автоматически по размерности и размеру массива X распознает тип изображения и действует корректно в обоих случаях, а функция colormap при пустом входном массиве map не делает ничего, так что оба этих случая могут быть обработаны одинаково.

Однако если бы было заранее известно, что в файле содержится изображение типа `truecolor`, то его можно было бы прочитать более коротким кодом:

```
X = imread( 'name.xxx' )
```

а для показа этого изображения в графическом окне системы MATLAB было бы достаточно вызова одной лишь функции `image(X)`.

Чтобы заранее узнать тип изображения в файле, нужно вызвать функцию

```
imfinfo( 'name.xxx' )
```

Вот пример, когда на компьютере с 24-битовым графическим режимом работы видеоадаптера изображение из графического окна системы MATLAB сохраняется функцией `capture` в массивах `X` и `map`. Затем эту информацию функцией `imwrite` записывают в JPEG-файл. В результате для этого файла функция `imfinfo` извлекает следующую справочную информацию (см. рис. 2.31):

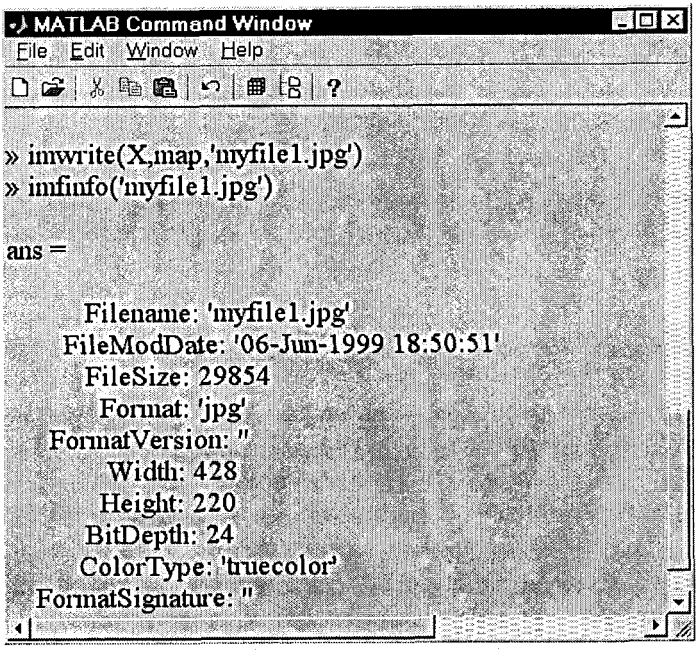


Рисунок 2.31

Отсюда видно, что изображение имеет тип `truecolor`. Это означает, что можно полностью обойтись без матрицы цветов. Действительно, в результате чтения такого файла

```
[X,map]=imread('myfile1.jpg');
```

получается пустая матрица цветов `map`:

```
map =
   []
```

Кроме того, из полученной функцией `imfinfo` информации виден размер содержащегося в файле растрового изображения, равный `428 x 220` пикселов. Указан также размер (FileSize) сжатого файла типа JPEG, в котором и размещается `truecolor`-изображение указанного размера. Этот размер равен `29 854` байтам. Если бы не было сжатия, предусмотренного специальным алгоритмом по спецификации JPEG, то тогда потребовалось бы для хранения файла `3 * 428 * 220 = 282 480` байт. За счет сжатия размер файла снижен примерно в 10 раз.

Глава 3

Массивы символов, структур, ячеек. Файловые операции

Массивы символов и тип данных char

До сих пор мы преимущественно имели дело с массивами вещественных или комплексных чисел. Про такие массивы говорят, что они имеют тип `double`. Это основной тип данных системы MATLAB, предназначенный для вычислений. В то же время при рассмотрении растровой графики мы столкнулись с типом данных *короткое целое*, обозначаемое ключевым словом `uint8`. Этот тип данных предназначен для компактного хранения больших массивов целых чисел, что очень характерно для графических задач. Однако производить вычисления с типом данных `uint8` нельзя (по крайней мере в версиях системы MATLAB до 5.2 включительно). Если все же нужно произвести вычисления, то сначала тип данных `uint8` приводят явно к типу `double`, производят вычисления и возвращаются к типу `uint8` для дальнейшего хранения.

Во всех языках программирования, и MATLAB здесь не исключение, большую роль играют обработка и хранение текстовых данных (то есть текстов на естественных языках – английском, русском и т. д.). Для этой цели в системе MATLAB предусмотрен специальный *символьный* тип данных `char` (сокращение от английского слова *character* – символ, знак, буква, литера).

Каждому возможному символу (букве алфавита или специальному символу) в соответствии со стандартными таблицами кодировок ставится в соответствие целое числовое значение, для хранения которого в памяти машины всегда достаточно 2 байт памяти. Именно 2 байта и отводятся под каждый элемент символьного массива системы MATLAB.

Задать элемент символьного массива можно двумя способами. Во-первых, его можно задать целым числовым кодом, к которому применяется модификатор `char`:

```
c1(1) = char( 97 );
```

Здесь создан символьный массив `c1` размером `1 x 1`, единственный элемент которого согласно всем стандартизованным на сегодняшний день кодовым таблицам (так называемый код ASCII) соответствует английской букве a. В этом легко убедиться с помощью следующих команд системы MATLAB (см. рис. 3.1):

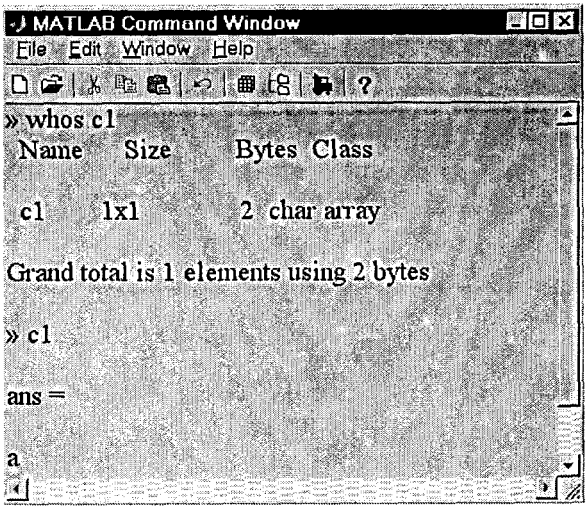


Рисунок 3.1

Во-вторых, того же результата можно добиться применением *апострофов*:

```
c1(1) = 'a';
```

Второй способ можно счесть более удобным, так как не требуется помнить код английской буквы a. Чтобы узнать код того или иного символа, нужно вычислить, например, следующее выражение (для чего достаточно набрать его в командном окне и нажать клавишу Enter):

```
double( 'b' )
ans =
   98
```

Первые 128 символов во всех таблицах кодировок совпадают (это и есть стандартная кодировка ASCII). Коды от 0 до 31 соответствуют так называемым управляющим символам, не имеющим визуального представления. Символы для кодов от 32 до 127 можно отобразить в командном окне системы MATLAB с помощью следующих выражений (см. рис. 3.2):

Символы для числовых кодов, больших 127, не определены столь же однозначно и зависят от конкретных вариантов кодировок. Для кодировки западноевропейских языков (и соответствующих им шрифтов) эти коды соответствуют диакритическим и другим специальным символам. *Русскому языку* в операционной системе Windows соответствует *кодировка* 1251. В ней присутствуют символы кириллицы. Если в командном окне системы MATLAB в качестве базового шрифта выбран кириллический шрифт, то тогда легко просмотреть соответствующие символы, например:

```
char(192:255)
ans =
АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯабвгдежзийклмнопрстуфхцчшщъыьэюя
```

Русским буквам Е и е соответствуют коды 168 и 184.

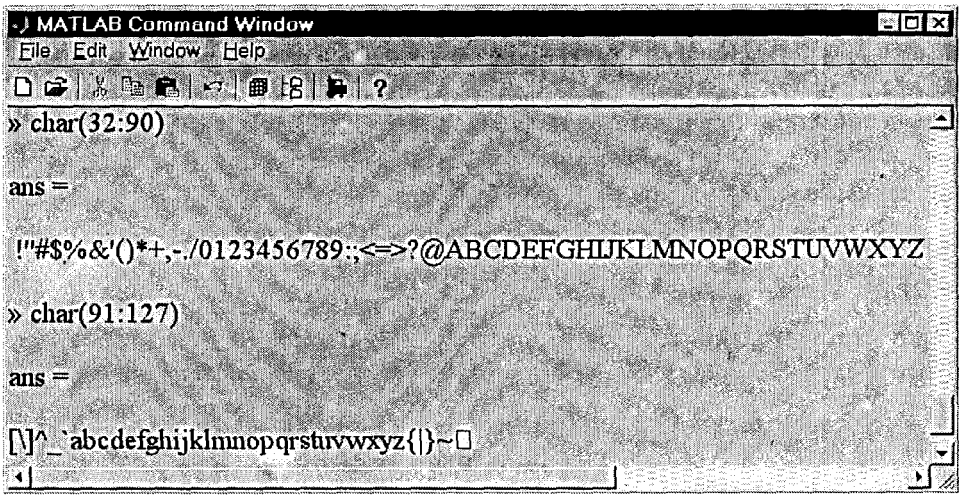


Рисунок 3.2

Символьные массивы из нескольких символов создаются и обычными для всех массивов системы MATLAB операциями конкатенации и индексации, и специальной символьной операцией конструирования массива, когда все символы массива записываются подряд и ограничиваются с двух сторон апострофами. Ниже приведены различные варианты создания символьных одномерных массивов:

```
c2=['a','b','c']; c2(4)='d'; c3 = 'Hello, World!';
```

В результате таких присваиваний создаются переменные c2 и c3 (это символьные массивы – в системе MATLAB все является массивами) типа `char` (см. рис. 3.3).

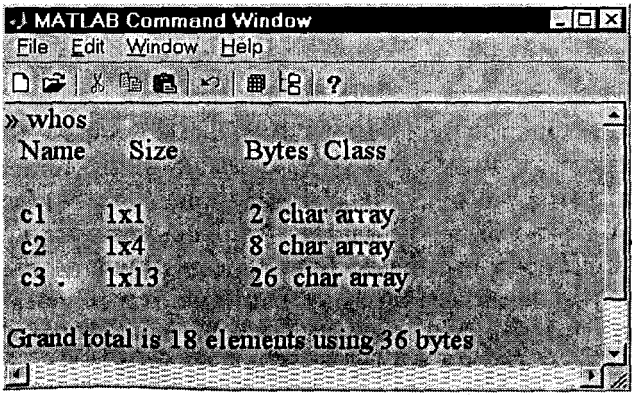


Рисунок 3.3

Так как под каждый символ отводится 2 байта памяти, то переменная c2 занимает в памяти компьютера 8 байт, поскольку этот массив имеет тип char и всего содержит четыре элемента. Массив c3 содержит 13 символьных элементов (включая не слишком заметный, но важный символ – пробел) и занимает в памяти 26 байт.

Модификатор char можно применять сразу к нескольким целым числовым значениям или к числовым матрицам и векторам, состоящим из нескольких целочисленных элементов. Например, код

```
x = [97,98,99,100]; c2 = char(x)
```

порождает то же самое символьное значение ('abcd') для переменной c2, которое выше было получено другим способом. В то же время код

```
c2 = char( 97, 98, 99, 100 )
```

приводит к символьному вектор-столбцу:

```
c2 =
a
b
c
d
```

А вот пример применения модификатора char к матрице из целочисленных элементов:

```
X = [ 97,98,99,100; 101,102,103,104 ]; s = char( X );
s =
abcd
efgh
```

Про переменную s можно сказать, что это матрица типа char размера 2 x 4, или символьная матрица.

Хорошей аналогией фрагмента строки текста естественного языка являются одномерные символьные массивы размера 1 x n, где n – количество входящих в эти массивы символов (включая пробелы, знаки пунктуации и т. д.). Как мы знаем, одномерные массивы размера 1 x n (1 строка и n столбцов) в системе MATLAB принято называть вектор-строками. Итак, символьные вектор-строки системы MATLAB соответствуют строкам текста на естественном языке, поэтому такие переменные называют *текстовыми строками* или просто *строками*.

Про символьные матрицы можно было бы сказать, что они соответствуют текстовым страницам, то есть наборам текстовых строк на естественном языке, если бы не одно «но». В естественном языке строки текста имеют в общем случае разную длину, а все строки матриц системы MATLAB обязаны состоять из одинакового количества элементов. Так что аналогия здесь неполная.

При создании матриц типа char, содержащих несколько текстовых строк, приходится выравнивать длины этих строк, добавляя им в конец ряд пробелов. При создании таких матриц с помощью модификатора char выравнивание длин пробелами осуществляется автоматически:

```
s1 = 'a'; s2 = 'ab'; s3 = 'abc';
X = char( s1, s2, s3 );
X =
a
ab
abc
```

Здесь матрица X типа char имеет размер 3 x 3. Каждая ее строка состоит из трех элементов, так как при создании этой матрицы с помощью модификатора char в конец текстовым строкам s1 и s2 были добавлены два пробела и один пробел соответственно. В этом легко убедиться, выполнив следующий фрагмент:

```
double( X( 1, 2 ) )
ans =
  32
```

показывающий, что второй элемент первой строки есть пробел (ASCII-код пробела равен 32). Точно так же убеждаемся, что третьи элементы первой и второй строк матрицы X являются пробелами.

При создании «текстовых матриц» операцией вертикальной конкатенации нужно не забывать добавлять выравнивающие длины строк пробелы, что неудобно и вообще является неэлегантным решением. Полным и четким решением этой проблемы в М-языке системы MATLAB является создание *массива ячеек*, о чем ниже будет подробно рассказано в подразделе, посвященном массивам этого типа.

В отличие от типа данных uint8 над типом данных char допустимы все операции, что обычно применяются к типу double. При этом фактически вычисления осуществляются над кодами символов, а результат вычислений имеет тип double. Например, пусть заданы три массива типа char:

```
s1 = 'Hello,'; s2 = ' World'; a = 'a';
```

Здесь переменная с именем a имеет своим значением символ английского алфавита a, ASCII-код которого есть 97. Тогда следующие арифметические и логические операции, примененные к этой переменной, дают результаты:

```
res1 = a + a; res2 = a*a; res3 = res1 & res2;
res1 =
  194
res2 =
```

```
  9409
res3 =
   1
```

основанные на фактических вычислениях с кодом ASCII этой буквы. Можно даже (если есть такое экзотическое желание) вычислить синус от символа:

```
res4 = sin( a );
res4 =
    0.3796
```

Переменные res1, res2, res3 и res4 имеют тип данных double, несмотря на то что исходными для вычислений были данные типа char. Поэтому выражение

```
s = s1 + s2;
s =
 104  188  219  222  219  144
```

допустимо только потому, что s1 и s2 состоят из одинакового числа элементов. Его вычисление приводит к переменной s типа double, являющейся числовым массивом из шести элементов типа double, а вовсе не объединенный текст Hello, World, который можно было бы ожидать по неосторожности. Ясно, что объединенный текст получается операцией горизонтальной конкатенации:

```
s = [ s1, s2 ];
s =
Hello, World
```

Вместо операций горизонтальной и вертикальной конкатенации для символьных переменных можно применять специализированные функции strcat и strvcat. Последняя из этих функций осуществляет вертикальную конкатенацию с автоматическим добавлением недостающих пробелов для выравнивания длин строк, а первая почти эквивалентна операции горизонтальной конкатенации, но она удаляет все концевые пробелы в объединяемых строках – имейте это в виду!

Конкатенации можно подвергать не только отдельные строки (то есть вектор-строки в самом общем смысле массивов системы MATLAB), но и их наборы, то есть символьные матрицы. В последнем случае у них должно быть одинаковое количество строк, иначе возникнет ошибочная ситуация. Рассмотрим для примера следующий фрагмент:

```
A=['a','b'; 'c','d']; B=['e','f'; 'g','h'];
C=[A,B]
C =
abef
cdgh
```

Итак, переменные типа double и char преобразовываются друг в друга при помощи явных модификаторов double и char, а также в процессе вычислений и присваиваний. В любой момент можно выяснить тип конкретной переменной с помощью функций isa или ischar. Функция ischar принимает в качестве своего единственного аргумента имя переменной и возвращает «истину» (единицу), если переменная имеет тип char, и возвращает «ложь» (нуль) в противном случае. Функция isa дополнительно имеет второй аргумент, идентифицирующий тип данных. Например,

```
isa( s, 'char' )
```

отвечает на вопрос, является ли переменная s переменной типа char. В остальном поведение этой функции аналогично поведению функции ischar.

Наконец, рассмотрим особый случай, когда с помощью символов изображаются числа:

```
s1 = '12.51';
```

Здесь символьная строка s1 изображает дробное число 12.51. Как получить само это число по его *символьному изображению*? Ясно, что модификатор double здесь ни при чем:

```
double( s1 )
ans =
  49   50   46   53   49
```

так как он порождает числовой массив ASCII-кодов, входящих в строку s1 символов. Здесь на помощь приходит специальная функция str2num:

```
str2num( s1 )
ans =
  12.5100
```

которая создает число по его символьному (строковому) представлению. Обратную задачу решает функция num2str, которая порождает символьное представление числа:

```
x = 12.51478; s = num2str( x );
s =
12.5148
```

с округлением до четырех цифр после запятой. Количество цифр после запятой можно увеличить, указав функции num2str в качестве второго аргумента общее количество символов в изображении числа. Небольшой вариацией является функция int2str, которая сначала округляет числовой аргумент до целого значения, которое затем и превращает в символьную строку:

```
s = int2str( x )
```

```
s =
13
```

Функцию int2str часто используют для *«приклеивания» номера* к постоянной части имени:

```
KernelName = 'SomeName'; Number = 567;
name = [ KernelName, int2str(Number) ];
```

где после конкатенации текстовая строка name примет значение 'SomeName567'.

Встроенные функции для обработки строк

Выше мы узнали, что система MATLAB допускает обычные математические вычисления с переменными типа char. Однако большой актуальности в этом нет, поскольку смысл типа данных char состоит все-таки в другом. Этот тип данных специально предназначен для обработки фрагментов текста на естественном языке.

Система MATLAB располагает полным набором функций для «классической» обработки текстов. К таким функциям относятся функции blanks, deblank, findstr, isletter, isspace, lower, repmat, strcmp, strcmpi, strmatch, strncmp, strncmpi, strrep, strtok, upper.

Функция repmat является не специализированной строковой функцией, а функцией общего назначения, позволяющей размножить произвольную матрицу в горизонтальном и вертикальном направлениях заданное число раз. В частности, для строк эта функция размножает текстовые отрезки. Например, эта функция позволяет создать строку из заданного числа символов:

```
s = 'a'; n = 1; m = 11; str = repmat( 'a', n, m );
str =
aaaaaaaaaaa
```

Если требуется создать строку из заданного числа пробелов, то проще применить функцию blanks. Например, вызов этой функции

```
str = blanks( 27 );
```

создает строку из 27 пробелов. В свою очередь, удалением пробелов занимается функция deblank, которая удаляет из строки все концевые пробелы.

С помощью функций isletter и isspace можно проанализировать входящие в строку символы. Функция isspace принимает в качестве аргумента строку, а возвращает числовой массив из нулей и единиц, причем единицы стоят на тех местах, на которых в исходной строке стоят так называемые *пробельные* символы. Так называются *пробел*, а также управляющие символы (их

ASCII-коды меньше 32 и они не имеют визуальных образов) *«табуляция»*, *«перевод строки»*, *«возврат каретки»* и т. п.

Функция `isletter` также возвращает числовой массив из нулей и единиц, но здесь единицы стоят на тех местах, на которых в исходной строке стоят буквы. Например,

```
str = 'abc123 #$% 567qwerty'; x = isletter( str );
x =
1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
```

Функция `isletter` легко определила, что в 20-символьной строке букв всего девять, причем стоят они на первых трех и на последних шести позициях (именно на этих позициях в числовом массиве `x` стоят единицы).

Теперь выясним, какова ситуация с буквами кириллицы. В справочной системе пакета MATLAB сказано, что поддерживается только кодировка Windows Latin-1, соответствующая западноевропейским языкам. Если же не верить написанному, а поэкспериментировать, то можно, например, получить следующий результат:

```
str = 'qwertуйцукен'; x = isletter( str );
x =
1 1 1 1 1 1 1 1 1 1 1 1
```

означающий, что русские буквы все же распознаются. Однако не надо торопиться. Вот вам еще один экспериментальный пример:

```
str = 'qwertyeE'; x = isletter( str );
x =
1 1 1 1 1 1 0 0
```

и мы видим, что русские буквы `е` и `Е` все же не распознаются. Приходится поверить справочной документации к пакету MATLAB. Таким образом, если присутствуют буквы кириллицы, то их позиционирование нужно будет выявлять самостоятельно. Встроенная функция `isletter` системы MATLAB не поможет нам в этом в полном объеме (выпадут буквы `е` и `Е`), и придется писать свою собственную функцию. Такая функция будет нами позже написана во 2-й части пособия в гл. 6, посвященной программированию так называемых М-функций.

Функции `lower` и `upper` очень просты. Первая из них заменяет все прописные символы (символы верхнего регистра) на символы нижнего регистра. Например,

```
str = 'Hello Matlab'; str1 = lower( str );
str1 =
hello matlab
```

Функция же `upper` работает противоположным образом: все символы нижнего регистра (строчные) она заменяет на символы верхнего регистра.

Функции findstr, strmatch и strtok находят или выделяют в строках подстроки, однако детали их работы существенно отличаются. Начнем с функции findstr.

Например, в следующем фрагменте функцией findstr ищется массив позиций вхождения слова 'Hello' в текст, содержащийся в переменной vStr:

```
innerStr = 'Hello';
vStr='Hello is the word. Hello is opposite to bye.';
positions = findstr( vStr, innerStr );
```

и в результате массив positions принимает следующее значение:

```
positions =
   1 20
```

Таким образом, функция findstr обнаружила два вхождения переменной innerStr в текст Vstr. Первое вхождение имеет место начиная с самого первого символа; второе вхождение имеет место на 20-м символе (включая пробелы, разумеется).

Если функция findstr не находит вхождений вообще, то она возвращает пустой массив, который надо проверять функцией isempty. Эта функция возвращает «истину» (единицу), если массив действительно пустой (не содержит ни одного элемента), и возвращает «ложь» в противном случае. Еще следует подчеркнуть, что функция findstr различает регистр символов.

Функция strmatch осуществляет поиск подстрок в символьных матрицах, содержащих несколько текстовых строк сразу. Эта функция ищет и возвращает номера строк матрицы, в которых содержимое начинается с заданного набора символов. Например, в следующем фрагменте:

```
A=['qwert'; 'asdfg'; 'zxcvb']; x=strmatch('as',A);
x =
  2
```

функция strmatch находит, что вторая строка исходной матрицы A начинается с 'as'. Если требуется выявить не начальное (частичное) совпадение, а точное совпадение строки матрицы с заданной подстрокой, то тогда в качестве третьего аргумента функции strmatch нужно передать строку 'exact':

```
A=['qwert';'asdfg';'zxcvb']; x=strmatch('as',A,'exact');
x =
  []
```

Так как в данном примере нет такого полного совпадения, то функция strmatch возвращает пустой числовой массив (обозначается как []).

Функция strtok позволяет выявлять в исходной строке произвольные *лексемы*, например слова. Она возвращает начальную часть строки, расположенную

до первого пробельного символа (смотри выше). Пробельные символы являются *ограничителями лексем* по умолчанию. Например, следующий вызов функции strtok выделяет из исходной строки первое слово, ограниченное пробелом:

```
str = 'Very good'; word1 = strtok( str );
word1 =
Very
```

Можно задать свои собственные ограничители лексем, перечислив их подряд в строке, передаваемой функции strtok в качестве второго параметра:

```
str = 'Yes,No'; a=strtok( str, ',:;!?');
a =
Yes
```

Если требуется сразу получить как лексему, так и остаток строки, то следует использовать вызов функции strtok *в варианте с двумя возвращаемыми значениями* (мы и раньше встречали случаи, когда функции системы MATLAB возвращали не одно, а несколько значений – например, функция meshgrid, о которой рассказывалось во 2-й главе в подразделе, посвященном трехмерной графике):

```
[ token, remainder ] = strtok( str, delimeter )
```

Например, в следующем фрагменте:

```
str = 'Yes,No'; [word, rem] = strtok(str,',:;!?');
word =
Yes
rem =
,No
```

в переменной word запоминается первое слово в обрабатываемой строке, а в переменной rem сохраняется для дальнейшей обработки остающаяся часть исходной строки.

Функция strrep позволяет найти в исходной строке заданный фрагмент текста и заменить его на другой фрагмент текста. Общий формат этой функции таков:

```
str = strrep( str1, str2, str3 )
```

Здесь функция strrep ищет в строке str1 подстроку str2 и, если находит, то заменяет ее на str3. Результат возвращается в строке str. Исходная строка str1 остается неизменной. В следующем фрагменте:

```
str = strrep('good bad best', 'bad', 'better' );
str =
good better best
```

функция strrep в исходной строке 'good bad best' заменяет слово 'bad' на слово 'better'.

И наконец, рассмотрим функции strcmp, strncmp, strcmpi и strncmpi. Все эти функции сравнивают значения двух строк между собой. Функции с буквой i на конце при сравнении строк нечувствительны к регистру символов, а функции с буквой n в середине их имен сравнивают не целиком строки, а только их начальные отрезки из заданного числа символов. Поэтому достаточно рассмотреть работу только функции strcmp. Эта функция принимает в качестве аргументов две строки и возвращает «истину» (единицу) в случае идентичности строк и «ложь» (нуль) в противном случае. Функция strcmp чувствительна к регистру символов и учитывает (а не игнорирует) как ведущие, так и концевые пробелы (не говоря уже о внутренних).

Из представленного объяснения результат работы функции

```
res = strcmp( 'Mexico', 'mexico' );
res =
    0
```

абсолютно очевиден, так как строки отличаются регистром отдельных символов.

Массивы структур

Бывает желательно под одним именем объединить числовые и текстовые данные (например, результаты физических экспериментов, данные о переписи населения и т. д.). Для этой цели в системе MATLAB предусмотрен специальный тип данных – struct (*структура*).

Переменные типа struct имеют в своем составе несколько именованных полей. Создадим переменную MyStruct1, состоящую из двух полей: одного числового поля с именем data и одного текстового поля с именем name:

```
MyStruct1.name = '1st result';
MyStruct1.data = [ 1, 2.5; -7.8, 2.3 ];
```

Имя поля отделяется от имени переменной точкой. В данном фрагменте кода создается массив MyStruct1 типа struct размером 1 x 1. Добавим в только что созданный массив MyStruct1 второй элемент:

```
MyStruct1( 2 ).name = '2nd res-t';
MyStruct1( 2 ).data = [ -5.7, -2.5; 7.1, 8.4 ];
```

Получился массив размера 1 x 2. В этом легко убедиться, если набрать в командном окне системы MATLAB имя переменной MyStruct1 и нажать клавишу Enter. В результате система покажет не содержимое этого массива, а его структуру (см. рис. 3.4).

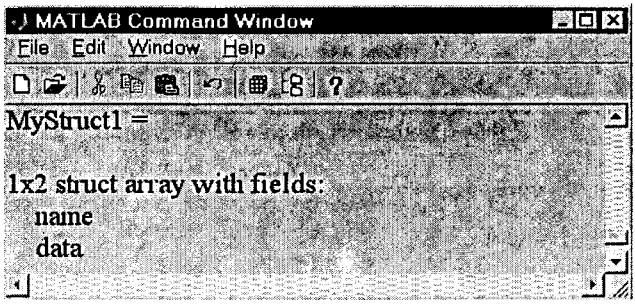


Рисунок 3.4

Можно и далее добавлять элементы к этому одномерному массиву. Достаточно очевидно, что все элементы массива типа `struct` имеют одинаковое количество полей с фиксированными именами. Если явно не задать значение какого-либо поля, то ему автоматически будет присвоен пустой массив `[]`.

Набор полей массива структур можно изменять *динамически*. Например, уже после того, как мы создали массив `MyStruct1` типа `struct` размера `1 x 2` с двумя указанными выше полями (`name` и `data`), можно выполнить присваивание

```
MyStruct1( 1 ).field = 'Third field';
```

после которого массив структур уже будет обладать тремя полями – `name`, `data` и `field`. Так как у второго элемента массива `MyStruct1` поле `field` явно не задано, то оно равно пустому массиву:

```
MyStruct1( 2 ).field
ans =
    []
```

Если два массива структур имеют одинаковый набор полей, то допускается групповое присваивание вида

```
MyStruct1( 3 ) = AnotherStruct( 2 );
```

когда значения всех полей второго элемента массива `AnotherStruct` копируются в поля третьего элемента массива `MyStruct1`.

Если работа ведется в интерактивном режиме, когда все данные вводятся с клавиатуры, рассмотренный процесс последовательного задания полей и элементов массива структур вполне оправдан. Однако в программном режиме он недостаточно хорош с точки зрения производительности. Вместо него лучше использовать функцию с именем `struct`:

```
MyStruct2 = struct( 'field1',[ 1 2 3],'field2','Hello');
MyStruct2(2)=struct('field1',[ 7 8 9],'field2','World');
```

Покажем теперь (хотя это и очевидно), как подобраться к единственному числовому значению в поле `data` первого элемента массива `MyStruct1`. Для этого

надо применить две операции индексации и одну операцию доступа к полю структуры. Например, для ранее созданного нами массива структур MyStruct1 выражение

```
MyStruct1( 1 ).data( 1 , 2 )
```

имеет значение 2.5, так как в поле data расположены числовые матрицы 2 x 2 и соответствующий элемент числовой матрицы действительно равен 2.5.

Теперь рассмотрим более сложный вопрос об одновременном доступе к содержимому одноименных полей всех элементов некоторого массива структур. Пусть, к примеру, требуется скопировать и сохранить в переменной v объединенное содержимое полей data всех элементов созданного выше массива структур MyStruct1. Вот компактное решение этой задачи, предлагаемое системой MATLAB:

```
v = [ MyStruct1.data ]
ans =
   1.0000   2.5000  -5.7000  -2.5000

  -7.8000   2.3000   7.1000   8.4000
```

Здесь применение прямоугольных скобок обязательно, так как в М-языке выражение

```
MyStruct1.data
```

эквивалентно следующему *списку величин, разделенных запятыми*:

```
MyStruct1(1).data, MyStruct1(2).data
```

Если заключить такой список в прямоугольные скобки, то получается синтаксически корректное выражение для горизонтальной конкатенации двух числовых матриц 2 x 2. В результате такой операции переменная v становится числовой матрицей 2 x 4.

Для удаления некоторого поля из всех элементов созданного массива структур применяют функцию rmfield:

```
MyStruct2=rmfield(MyStruct2,'field2');
```

После этого в массиве структур MyStruct2, созданном нами ранее (см. выше), остается только поле field1.

В качестве содержимого некоторого поля структуры может выступать другая структура, то есть структуры могут быть *вложенными*. В качестве имени вложенной структуры выступает имя поля объемлющей структуры:

```
Outer.number = 1;
Outer.inner.name='name1';
Outer.inner.data=[ 5 3; 7 8];
```

Здесь Outer – имя внешней (объемлющей) структуры, а inner – имя внутренней структуры, являющейся полем структуры Outer. Для чтения числового данного, стоящего во второй строке первого столбца в поле data, нужно применить выражение с двумя точками, где первая точка означает доступ к полю inner, являющемуся структурой, а вторая точка осуществляет доступ к полю data этой внутренней структуры:

```
Outer.inner.data( 2, 1 )
ans =
    7
```

О том, как устроены структуры Outer и inner, система MATLAB сообщает так, как показано на рис. 3.5.

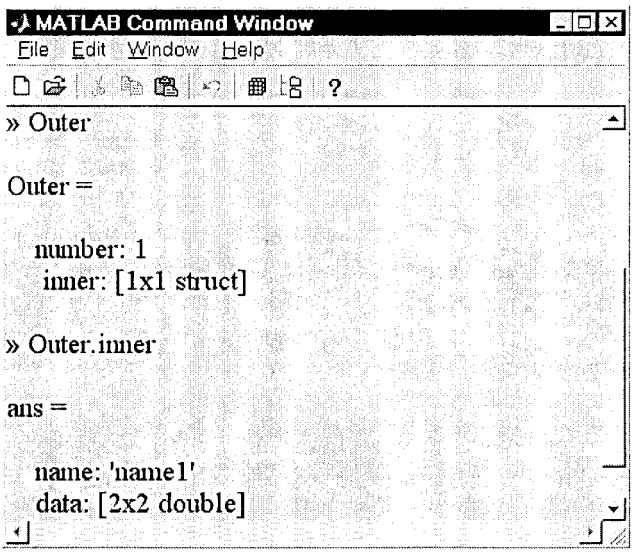


Рисунок 3.5

Массивы структур, как мы видели выше, могут динамически изменять состав своих полей. В любой момент, однако, можно с помощью специальных функций проверить, из каких полей состоит структура. Вот список этих функций:

`isstruct( St )` – возвращает истину, если аргумент структура.

`isfield( St, 'name' )` – возвращает истину, если имеется такое поле.

`fieldnames( St )` – возвращает массив строк с именами всех полей.

Структуры MATLAB можно назвать *агрегированным* типом данных. Другим агрегированным типом данных в системе MATLAB являются так называемые ячейки (cells). Только что упомянутая функция fieldnames возвращает массив

(набор) строк именно в виде массива ячеек, так что самое время приступить к изучению этого типа данных.

Массивы ячеек

Массив ячеек (*cell array*) содержит в качестве своих элементов массивы разных типов. Можно сказать, что массив ячеек является массивом массивов. По сути он является универсальным контейнером – его ячейки (элементы) могут содержать любые типы и структуры данных, с которыми работает MATLAB, – числовые массивы любой размерности, массивы типа `uint8`, строки, массивы структур и, наконец, вложенные массивы ячеек. В свою очередь, массив ячеек может быть полем структуры.

Методы создания массивов ячеек похожи на методы создания структур. Как и в случае структур, массивы ячеек могут быть созданы либо путем последовательного присваивания значений отдельным элементам массива, либо целиком при помощи специальной функции `cell`.

Подчеркнем с самого начала, что важно различать ячейку (элемент массива ячеек) и ее содержимое. Каждая ячейка является массивом системы MATLAB, то есть помимо собственно данных содержит в отведенной под нее памяти компьютера еще и служебную информацию, о которой мы уже много говорили выше. Образно говоря, ячейка – это ее полезное содержимое (собственно данные) плюс некоторая управляющая оболочка вокруг этого содержимого, позволяющая хранить в ячейке произвольные типы данных любого размера.

В системе MATLAB выражение любого типа можно превратить в ячейку, заключив его в *фигурные скобки*. Таким образом, фигурные скобки являются *конструктором ячеек*. Для сравнения напомним, что конструктором числовых массивов служат квадратные скобки (операция конкатенации), а конструктором символьных массивов – апострофы.

Отдельные ячейки массива ячеек могут содержать данные разных типов. Для примера построим массив ячеек `2 x 2`, элементами которого являются ячейки, содержащие соответственно текстовую строку (одномерный массив типа `char`), числовую матрицу `3 x 3`, структуру `MyStruct` и числовую вектор-строку `1 x 3`.

Сначала создадим структуру `MyStruct`:

```
MyStruct = struct('field1',[ 1 2 3],'field2','Hello');
```

после чего и сформируем искомый массив ячеек поэлементными присваиваниями:

```
MyCellArray( 1, 1 ) = { 'Bonjour!' };
MyCellArray( 1, 2 ) = { [ 1 2 3; 4 5 6; 7 8 9 ] };
MyCellArray( 2, 1 ) = { MyStruct };
MyCellArray( 2, 2 ) = { [ 9 7 5 ] };
```

Построенный массив MyCellArray является массивом ячеек, так как элементам этого массива были присвоены именно ячейки, каждая из которых формировалась с помощью фигурных скобок. Содержимое соответствующих ячеек определяется выражениями внутри фигурных скобок. Поэтому ячейка MyCellArray(1,1) содержит текстовую строку 'Bonjour!', ячейка MyCellArray(1,2) содержит числовую матрицу [1 2 3; 4 5 6; 7 8 9], ячейка MyCellArray(2,1) содержит структуру MyStruct и, наконец, ячейка MyCellArray(2,2) содержит числовую вектор-строку [9 7 5].

Теперь спросим систему MATLAB о типе массива MyCellArray, для чего введем и исполним (нажатием клавиши Enter) команду

```
whos MyCellArray
```

на что получим ответ, представленный на рис. 3.6.

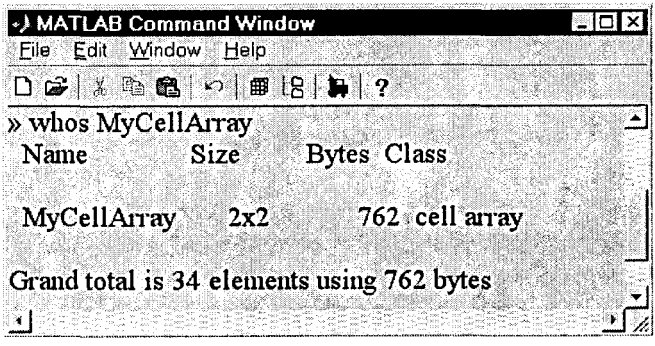


Рисунок 3.6

Построенный нами массив ячеек MyCellArray содержит разнородные данные, о чем нам и сообщает система MATLAB при вводе имени этого массива и нажатии клавиши Enter (см. рис. 3.7).

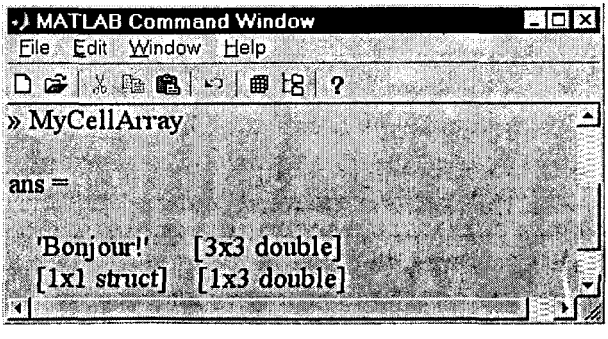


Рисунок 3.7

При этом показывается содержимое не всех ячеек этого массива. Более подробную информацию можно получить, вызвав функцию celldisp:

```
celldisp( MyCellArray )

MyCellArray{1,1} =
Bonjour!

MyCellArray{2,1} =
    field1 : [ 1 2 3 ]
    field2 : 'Hello'

MyCellArray{1,2} =
    1  2  3
    4  5  6
    7  8  9

MyCellArray{2,2} =
    9  7  5
```

Отсюда хорошо видно, что для того, чтобы подобраться к содержимому ячеек, нужно *индексировать массив ячеек при помощи фигурных скобок*. При обычной индексации круглыми скобками мы из массива ячеек извлекаем отдельную ячейку, которая сама является массивом. Напоминаем еще раз о том, что следует различать ячейку и ее содержимое (см. выше).

Массивы ячеек полностью решают типовую задачу хранения нескольких строковых данных под одним именем. Раньше мы уже формировали матрицы типа char, каждая строка которых обязана была иметь одну и ту же длину. Это очевидным образом ограничивает применение такого решения. В случае массива ячеек такого ограничения нет:

```
cellNames{ 1 } = 'function1';
cellNames{ 2 } = 'func2';
```

Здесь мы продемонстрировали *применение фигурных скобок в роли индексирующих элементов*, так что использовать фигурные скобки в правых частях операций присваивания не нужно (там теперь присутствуют значения, а не ячейки).

В результате под одним именем cellNames хранятся две текстовые строки, доступ к каждой из которых осуществляется по индексу в соответствии с синтаксисом массива ячеек. Вот код, который извлекает эти строки из массива ячеек cellNames и запоминает их в отдельных переменных str1 и str2:

```
str1 = cellNames{ 1 }; str2 = cellNames{ 2 };
```

Показанное выше поэлементное создание массива ячеек неэффективно с точки зрения производительности. Это не создает проблем в медленном интерак-

тивном режиме работы, но в программном режиме этот процесс лучше предварить вызовом функции cell:

```
MyCellArray = cell( 2, 2 );
```

которая сразу создаст массив ячеек требуемой размерности и размеров, причем каждая ячейка будет пустой. *Пустые ячейки* обозначаются как {[]}. Затем можно осуществлять ранее рассмотренные поэлементные присваивания, так как теперь они не требуют перестройки структуры массива с каждым новым присваиванием.

Содержимым пустой ячейки является пустой числовой массив, который, как мы знаем, обозначается []. Чтобы удалить некоторый диапазон ячеек из массива ячеек, нужно этому диапазону присвоить значение пустого массива []:

```
MyCellArray( 2, : ) = [];
```

Теперь массив ячеек MyCellArray имеет размер 1 x 2, так как мы только что удалили всю вторую строку этого массива ячеек.

Другим способом повышения эффективности (быстродействия) при создании массивов ячеек является возможность использования единственной пары фигурных скобок для создания сразу всех элементов массива ячеек. Например, вместо постепенного поэлементного присваивания

```
a(1,1)={1}; a(1,2)={'asd'}; a(2,1)={[1 2]}; a(2,2)={3};
```

можно создать весь массив ячеек за один раз:

```
a = { 1, 'asd'; [1 2], 3};
```

Такое решение является более производительным и использует к тому же более компактное выражение. Здесь точка с запятой используется для разделения рядов ячеек, а ячейки в пределах одного ряда отделяются друг от друга запятой или пробелом.

У массивов ячеек в силу их чрезвычайной гибкости существует множество типовых применений. О некоторых из них будет рассказано позже, в ч. 2 настоящего пособия в главе, посвященной программированию М-функций. А сейчас рассмотрим только то, что лежит на поверхности.

Самым простым и самым очевидным применением массивов ячеек является хранение набора текстовых строк. Массив ячеек, содержащий в качестве своих элементов текстовые строки, часто называют просто *массивом строк* (не путать с символьным массивом). В предыдущем подразделе была представлена функция fieldnames для работы с массивами структур, которая возвращает массив строк имен полей структуры. Здесь под массивом строк подразумевается именно массив ячеек. У нас выше была создана структура MyStruct с полями field1 и field2. Для этой структуры мы сейчас и вызовем функцию fieldnames:

```
res = fieldnames(MyStruct);
res =
  'field1'
  'field2'
```

Если бы переменная res была символьной матрицей (матрицей типа char), то ее строки отображались бы без апострофов, что мы и видели ранее в примерах из подраздела, посвященного массивам символов. Так что уже по этому признаку видно, что res является именно массивом ячеек (конкретно – массивом строк). Чтобы не оставалось никаких сомнений в этом, выполним команду

```
whos res
```

и получим от системы MATLAB ответ, показанный на рис. 3.8.

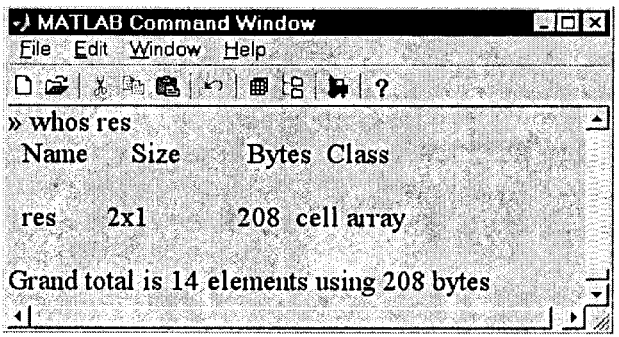


Рисунок 3.8

Многие из изученных нами ранее функций обработки строк могут принимать в качестве своих параметров массивы строк (массивы ячеек, содержащих строки). Сразу же вспоминается функция strmatch, которая работает именно с набором текстовых строк, а не с одной строкой. Применение массива ячеек для хранения набора строк удобнее, так как не требуется искуственного выравнивания длин строк концевыми пробелами. Вот пример применения функции strmatch в этом случае:

```
A = {'qw'; 'asfg'; 'zxcgvb'}; x = strmatch('as',A);
x =
  2
```

В качестве еще одного примера можно рассмотреть вариант работы функции strcmp с массивами ячеек, содержащими текстовые строки. Вот соответствующий пример:

```
STR1={'a11a';'b22b';'c33c'}; STR2={'a';'b22b';'c'};
strcmp( STR1, STR2 )
ans =
   0
   1
   0
```

в котором функция strcmp осуществляет поячеечное сравнение двух массивов строк (массивов ячеек).

Помимо использования массивов ячеек в качестве массивов строк у них есть еще одно достаточно полезное применение. Можно заменить список величин, разделенных запятыми, компактным индексным выражением с массивом ячеек. Для примера рассмотрим функцию plot, которая изучалась нами в главе, посвященной построению графиков функций. Этой функции в качестве параметров обычно передаются числовой массив значений независимой переменной, числовой массив значений зависимой переменной и текстовая строка, управляющая внешним видом графика. Для компактности все эти переменные (массивы) можно упаковать в массив ячеек, например:

```
F( 1 ) = { 0 : 0.1 : pi };
F( 2 ) = { sin( F{1} ) };
F( 3 ) = { 'bo:' };
```

Теперь вся информация, необходимая для построения графика функции, сосредоточена в единственной переменной F. Построение графика функции можно осуществить с помощью чрезвычайно компактного выражения

```
plot( F{ 1 : 3 } )
```

так как операция взятия *содержимого* диапазона ячеек (индексация именно фигурными скобками) с точки зрения системы MATLAB порождает список величин, разделенных запятыми (не путайте представленное индексирование с индексированием круглыми скобками, которое порождает просто подмассив ячеек). Итак, для массива ячеек F выражение F{1:3} эквивалентно следующему *списку величин, разделенных запятыми*:

```
F{1}, F{2}, F{3}
```

Очевидно, что использовать такие выражения можно только там, где по синтаксису М-языка допустимы списки величин, разделенных запятыми. Это допустимо, например, при индексации многомерных массивов, при передаче функциям их аргументов, в операциях горизонтальной конкатенации и в некоторых других случаях.

Передачу функции plot ее аргументов мы уже видели. А вот использование индексированных диапазоном массивов ячеек при индексации других массивов актуально лишь тогда, когда индексы массивов задаются громоздкими выражениями. В этом случае всю громоздкость можно перенести в определение массива ячеек, а конечную операцию индексирования записать компактно.

Ну и под самый конец немного разрядим сложное и подробное изложение синтаксиса массива ячеек двумя приятными фактами. Первый заключается в том, что имеется красивая графическая функция системы MATLAB, наглядно отображающая в графическом окне содержимое массивов ячеек. Это функция cellplot, которой в качестве первого аргумента передается имя массива ячеек. Тогда, к примеру, вызов этой функции

```
cellplot( MyCellArray, 'legend' )
```

приведет к появлению графического окна с «цветной информацией» о массиве ячеек `MyCellArray`, созданном нами ранее. На рис. 3.9 приведено черно-белое изображение этого окна (смеем уверить, что в цвете оно намного симпатичнее и информативнее):

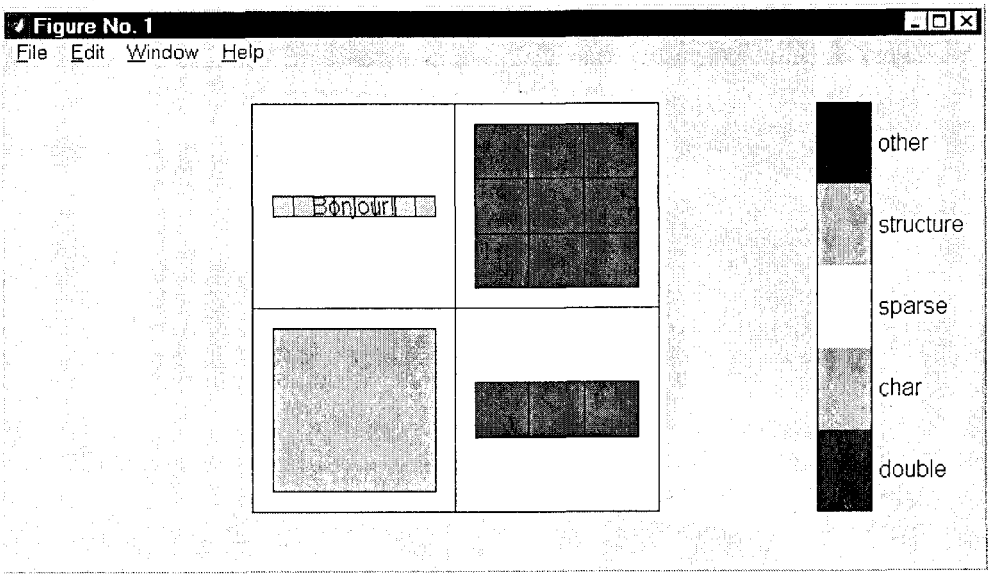


Рисунок 3.9

Расположенный в правой части этого окна вертикальный столбец задает соответствие между цветом отображения ячеек и их типом. В цветном варианте наглядно видно, что зеленый цвет соответствует структурам, и поэтому прямоугольник в левой нижней части имеет зеленый цвет.

Второй приятный факт состоит в том, что к настоящему моменту нами изучены почти все типы данных системы MATLAB. Остался лишь тип данных `sparse`, представляющий специальный метод хранения элементов *числовой разреженной матрицы* (количество ненулевых элементов матрицы много меньше общего числа элементов). Этот тип данных возникает и широко применяется, например, при решении типовых задач дифференциальных уравнений. Мы подробно рассмотрим этот узкоспециализированный тип данных в следующей главе, посвященной встроенным средствам системы MATLAB для решения типовых задач алгебры, анализа и статистики. А пока лишь продемонстрируем, как преобразовать, например, обычную числовую матрицу в разреженную:

```
A = [0 0 3; 7 0 0; 0 0 0]; spA = sparse( a );
spA =
   (2,1)   7
   (1,3)   3
```

Отсюда видно, что для матриц типа sparse в памяти компьютера хранятся не все элементы, а только ненулевые вместе с их индексами. В итоге матрица spA занимает в памяти компьютера 40 байт (можно узнать командой whos spA), а исходная матрица A типа double занимает в памяти 72 байта, так что экономия памяти компьютера налицо.

Оставив более подробное изучение типа данных sparse до следующей главы, нарисуем наглядную диаграмму, демонстрирующую набор всех типов данных системы MATLAB и их связь между собой (см. рис. 3.10).

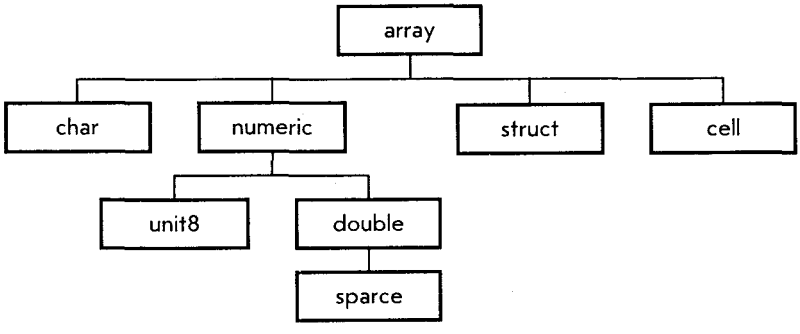


Рисунок 3.10

Здесь array – собирательный термин для всех массивов системы MATLAB, а термин numeric символизирует все числовые типы. Нельзя создать переменных типа array или numeric. Можно сказать, что это обобщенные (виртуальные) типы.

Чтение и запись произвольных бинарных файлов

Ранее нами были рассмотрены так называемые MAT-файлы, в которых сохраняются переменные из рабочей области системы MATLAB. Эти файлы имеют неопубликованный формат, но легко могут быть прочитаны командой load. Кроме того, система MATLAB предоставляет программисту специальный набор функций, которые можно вызывать из программ на языке C и которые предназначены для считывания информации из MAT-файлов.

Но как быть, если данные нужно ввести не из MAT-файла и не с клавиатуры, а считать из уже сформированного файла, причем запись этого файла могла производиться самыми разными способами и программными средствами. Или нужно сохранить результаты вычислений в файлах заранее оговоренного формата, предназначенных для дальнейшей работы в составе иных программных средств.

Для этих целей пакет MATLAB располагает полным набором специальных функций для работы с файлами произвольных форматов и типов. Под типами мы здесь имеем в виду разделение файлов на *бинарные файлы* и *текстовые файлы*.

Бинарные файла, как известно, предназначены для хранения произвольных данных в виде потока байтов, никак преднамеренно не интерпретируемых. Самые характерные операции с такими файлами – это операции записи или считывания заданного количества байтов информации. В противоположность этому содержимое текстовых файлов трактуется как набор всех символов из некоторой кодировки, имеющих визуальное представление (это можно назвать обобщенным текстом), а также заданный набор так называемых управляющих символов, таких, как «*возврат каретки*», «*перевод строки*», «*конец файла*». В полном согласии со своим названием текстовые файлы идеально подходят для хранения текстов, а бинарные файлы больше подходят для плотного (без излишнего расхода памяти) и скрытного (нельзя прочитать их содержимое простейшими текстовыми редакторами типа Notepad) хранения числовых данных и машинных кодов компьютерных программ.

Независимо от типа файла перед началом работы его нужно открыть специальной функцией `fopen`:

```
fid = fopen( 'имя_файла', 'флаг' )
```

где имя файла может предваряться полным путем к нему (иначе файл должен располагаться в текущем каталоге системы MATLAB). Второй параметр этой функции – так называемый *флаг открытия файла* говорит о способе дальнейшей работы с файлом:

'r'	Только для чтения
'w'	Только для записи (предыдущее содержимое теряется, а несуществующий файл создается)
'r+'	Чтение и запись одновременно
'a'	Добавление в конец файла

К указанным в текстовых строках буквам следует добавлять букву `'b'` для открытия файла в бинарном режиме и букву `'t'` для открытия файла в текстовом режиме. Например, флаг `'r+t'` означает окрытие текстового файла для чтения и записи, флаг `'wb'` означает открытие бинарного файла для записи.

Функция `fopen` возвращает числовой идентификатор открытого файла, который надо использовать в качестве параметра для функций чтения и записи в этот файл. Если операция открытия файла не удалась (это возможно как по причине отсутствия файла, так и по причине неправильного указания пути к нему на диске), то функция `fopen` возвращает `-1`. *Всегда следует проверять возврат функции* `fopen`.

Если файл больше не требуется, его следует как можно скорее (чтобы зря не напрягать компьютер работой с ненужными ресурсами) закрыть функцией `fclose`:

```
fclose( fid )
```

Покончив с изучением вопроса об открытии файлов, перейдем к вопросу о чтении и записи в них полезной информации. Сейчас сосредоточимся на бинарных файлах, а текстовые отложим до следующего подраздела.

Чтение и запись информации в бинарные файлы осуществляются функциями `fread` и `fwrite`. Функция `fwrite`, предназначенная для записи информации в бинарные файлы, имеет следующие аргументы:

```
fwrite( fid, A, 'precision' )
```

где `fid` – файловый идентификатор, возвращаемый функцией `fopen`; `A` – числовой вектор или матрица, чьи элементы подлежат записи в файл; строка `'precision'` говорит о размере памяти, отводимой под вещественные числа. Мы знаем, что в системе MATLAB под вещественные числа отводят 8 байт или 64 бита. Указанием функции `fwrite` о таком размере памяти служит текстовая строка `'float64'`.

Рассмотрим практический пример, иллюстрирующий работу функции `fwrite`. В следующем фрагменте создаются вектор-столбец `A` размером `3 x 1` и матрица `B` размером `2 x 3`, которые затем записываются в файл с именем `'dataTest.gqw'` (столь вычурное расширение имени файла выбрано специально, чтобы оно случайно не совпало с расширениями файлов известных Windows-приложений):

```
a = [ 1; 2; 3 ]; B = [ 4 5 6; 7 8 9 ];
fid1 = fopen( 'dataTest.gqw', 'wb' );
fwrite( fid1, a, 'float64' );
fwrite( fid1, B, 'float64' );
fclose( fid1 );
```

Фактически в файл `'dataTest.gqw'` записаны подряд девять вещественных чисел (то есть 72 байта) и больше ничего. Это нам может подтвердить Windows Explorer (см. рис. 3.11).

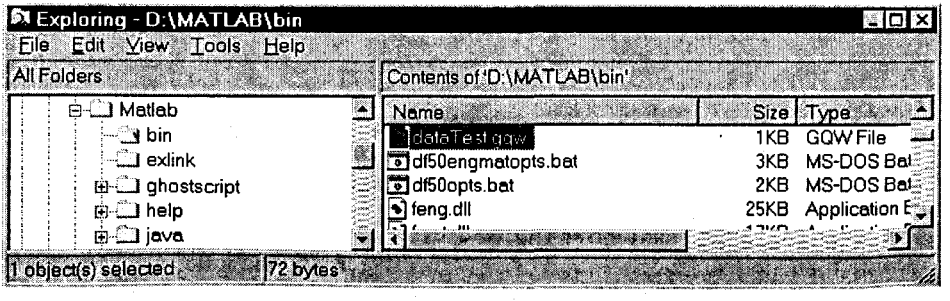


Рисунок 3.11

Истинный размер файла указывается в *строке состояния* (*statusbar*) главного окна программы Windows Explorer, где мы и наблюдаем число 72. Заодно мы здесь видим, что файл был записан в подкаталог `bin` каталога `Matlab`, поскольку именно этот каталог «по умолчанию» является текущим (надо не забыть потом стереть этот учебный файл, чтобы он не «засорял» рабочий каталог пакета MATLAB).

Нами получен замечательный результат. Теперь записанные в файл данные (девять вещественных чисел с двойной точностью) могут быть прочитаны не только системой MATLAB, но и многими другими программами. Например, на рис. 3.12 показано окно текстового редактора Notepad после прочтения им нашего файла:

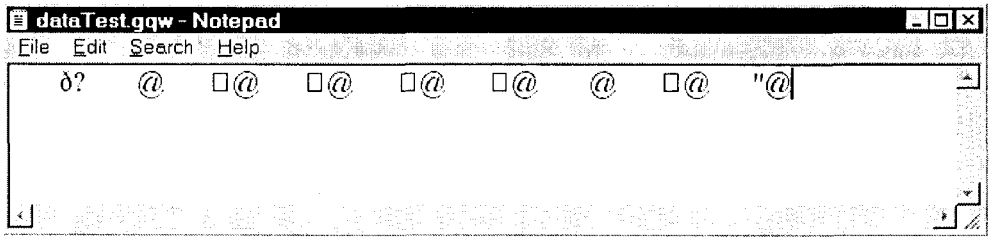


Рисунок 3.12

Это очень типичное изображение. Его можно назвать абракадаброй. Так всегда бывает, когда бинарные файлы просматриваются редакторами, ориентированными на текстовые файлы. Это мы и имели в виду выше, когда говорили, что сохранение информации в бинарных файлах носит достаточно скрытный характер.

Однако даже в редакторе Notepad, считая перемещения текстового курсора от начала строки к ее концу, можно определить, что всего в файл `'dataTest.gqw'` записано 72 байта информации. Абракадабра получается потому, что текстовый редактор пытается интерпретировать каждый байт как символ в соответствии с некоторой стандартной кодировкой. А составители этого файла (это мы сами и есть) и не предполагали такой «остроумной» трактовки его содержимого. В результате значения многих байтов в этом потоке не соответствуют никаким изображаемым символам. Попадание же отдельных байтов в существующие символы абсолютно случайно. В результате для показанных символов еще можно по таблицам кодировки определить истинное значение соответствующих числовых байтов, но для пустых пропусков этого сделать нельзя.

Настоящее «копание» в содержимом бинарных файлов можно осуществить с помощью редакторов бинарных файлов, которые вовсе не так распространены, как текстовый редактор Notepad. Эту роль (как и многие другие роли) может хорошо выполнить графическая среда компилятора языка C++ фирмы Microsoft. Ее принято называть `Microsoft Developer Studio`. На рис. 3.13 показано ее окно после открытия в бинарном режиме файла `'dataTest.gqw'`.

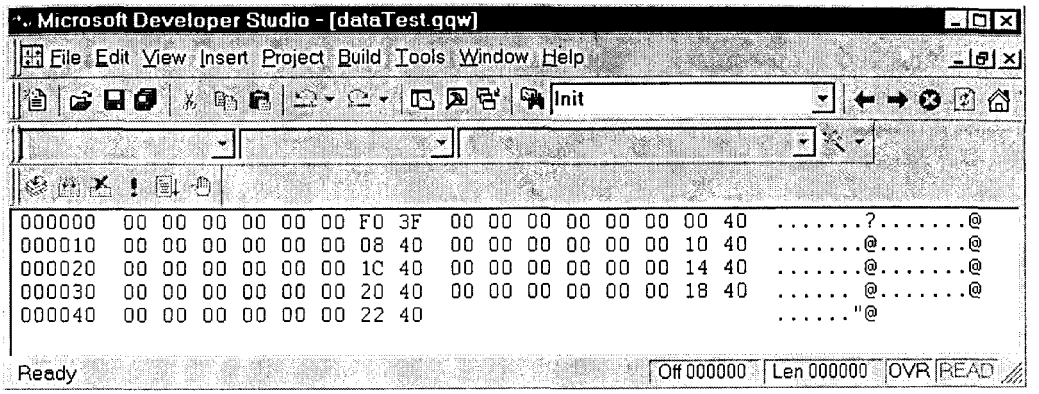


Рисунок 3.13

Здесь в правой части окна показана уже нам знакомая попытка символьной интерпретации байтов, но левее представлены сами числовые значения байтов в шестнадцатеричной форме, когда содержимое каждого байта записывается двумя шестнадцатеричными цифрами. Если очень захотеть и еще точно знать формат представления чисел с плавающей запятой двойной точности (мы пока знаем, что всего под такие числа отводится 8 байт, но мы не обсуждали точное распределение этих байтов под мантиссу и степень двойки, ведь числа в машинном представлении всегда двоичные), то можно точно представить записанные в файл десятичные дроби, но это явно непростая задача.

Теперь, когда мы прочувствовали все «прелести» двоичных файлов, настала пора сообщить, что прочитать из них числовые значения в программные переменные на языке C не представляет буквально никаких затруднений. Таким образом, записав числовые данные в бинарный файл из пакета MATLAB, мы легко можем прочитать их в другом языковом окружении. Еще легче прочитать их непосредственно в рамках системы MATLAB, если воспользоваться предназначенной для этого функцией `fread`. Вот три возможных формата вызова этой функции:

```
A = fread( fid )
A = fread( fid, numberVector )
A = fread( fid, numberVector, 'precision' )
```

Здесь строка `'precision'` имеет тот же самый смысл, что и рассмотренный выше для функции `fwrite`. Числовой вектор `numberVector` при чтении информации из файла задает как количество подлежащих прочтению числовых элементов с размером, определяемым третьим аргументом, так и организацию этих элементов в выходном числовом массиве `A`. Две другие формы вызова функции `fread` отвечают той или иной форме умолчательной практики (довольно опасной, так что лучше обойтись без нее).

Теперь можно прочитать из файла сохраненные там значения вектора и матрицы, что и иллюстрирует следующий фрагмент кода:

```
fid1 = fopen( 'dataTest.gqw', 'rb' );
[ a , count ] = fread( fid1, [1,3], 'float64' );
[ B , count ] = fread( fid1, [2,3], 'float64' );
fclose( fid1 );
```

Легко убедиться, что прочитанные значения совпадают с ранее записанными. Это произошло потому, что мы точно знаем, что и в каком порядке было записано в файл (ранее мы изучили, что в памяти компьютера массивы хранятся по столбцам, но они точно в таком же порядке записываются в файлы и читаются из них). Именно это имеют в виду, когда говорят о знании *формата файла*. Если формат бинарного файла неизвестен, то правильно интерпретировать его содержимое невозможно (если, конечно, исключить возможность угадывания на основе неполной информации).

Из представленного фрагмента видно также, что можно вызывать функцию `fread` и с двумя возвращаемыми значениями. При этом второе возвращаемое значение, то есть `count`, равно числу реально прочитанных вещественных чисел. При чтении вектора `a` это число будет равно 3, а при чтении матрицы `B` – будет равно 6:

```
B =
   4   5   6
   7   8   9

count =
   6
```

Теперь зададимся вопросом о том, что нужно сделать, чтобы сразу после открытия сформированного выше бинарного файла `'dataTest.gqw'` прочитать элементы матрицы `B` размером `2 x 3`, не читая предварительно вектор `a` размером `1 x 3`. Ответ состоит в том, что нужно воспользоваться функцией `fseek` для *позиционирования файлового указателя*.

После открытия файла его указатель расположен перед самым первым байтом, так что он показывает на тот байт, который будет прочитан первой же операцией чтения из этого файла. После прочтения одного вещественного числа файловый указатель продвинется на 8 байт вперед и расположится перед девятым байтом. Таким образом, по мере чтения чисел из файла его указатель продвигается вперед и всегда располагается перед первым еще не прочитанным байтом. Текущее положение файлового указателя можно узнать, вызвав функцию `ftell`:

```
CurPos = ftell( fid );
```

Для изменения текущего положения файлового указателя как раз и служит функция fseek:

```
fseek( fid, number, 'flag' )
```

перемещающая этот указатель на number байт (вперед или назад в зависимости от знака целого числа number) от его текущего положения, если 'flag' представлен строкой 'cof', или от начала файла, когда параметр 'flag' есть 'bof', и, наконец, в случае значения флага 'eof' перемещение файлового указателя отсчитывается от конца файла.

В результате следующий фрагмент позволяет прочитать из файла 'dataTest.gqw' только матричную составляющую этого файла:

```
fid1 = fopen( 'dataTest.gqw', 'rb' );
fseek( fid1, 24, 'bof' );
[ B , count ] = fread( fid1, [2,3], 'float64' );
fclose( fid1 );
```

поскольку мы с помощью функции fseek пропускаем от начала файла первые 24 = 3 * 8 байт (три вещественных числа в формате пакета MATLAB).

Теперь обсудим возможность ошибочной ситуации, когда осуществляется попытка чтения файла после того, как все его содержимое уже прочитано. Ясно, что перед такой попыткой чтения файловый указатель расположен вслед за последним байтом файла. Эту ситуацию можно выявить с помощью функции feof, принимающей в качестве своего единственного аргумента файловый идентификатор. Если конец файла уже достигнут, то функция возвращает логическую единицу («истину», то есть просто 1), а в противном случае эта функция возвратит нуль («ложь»).

Если не осуществлять проверку на достижение конца файла и продолжать читать файл, то получается следующий результат:

```
fread(fid1,8,'float64')
ans =
   Empty matrix: 8-by-0
```

говорящий о том, что результатом такого чтения файла может быть лишь пустой (не содержащий элементов) массив. Отсюда сразу становится ясно, что в данном файле больше читать нечего. Если все же возникают какие-то сомнения, то тогда следует вызвать функцию ferror, говорящую о том же самом, но уже открытым текстом (см. рис. 3.14).

Для повторного прочтения файла можно вызвать функцию

```
frewind( fid )
```

возвращающую файловый указатель на начало файла.

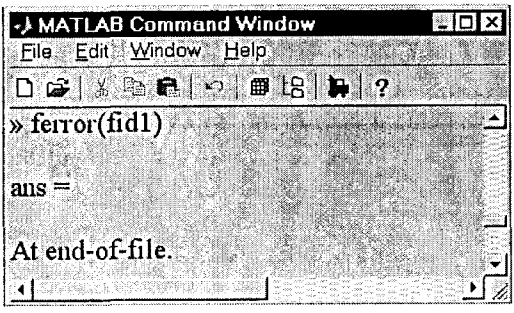


Рисунок 3.14

Подводя итог, можно сказать, что при попытке чтения за концом файла не происходит ничего страшного, если работа ведется в интерактивном режиме в командном окне системы MATLAB, так как можно легко продиагностировать эту ситуацию и изменить предпринимаемые действия. Однако эта ситуация опасна в программном режиме, о чем далее мы еще будем говорить в подразделе, посвященном программированию М-функций.

Опуская некоторые малосущественные нюансы, можно сказать, что функция `fwrite` предназначена в основном для записи в бинарные файлы числовых значений элементов массивов типа `double` системы MATLAB. Поэтому третий параметр этой функции почти всегда один и тот же, то есть является строкой `'float64'` (использование иных форматных точностных флагов, с одной стороны, не очень актуально, а с другой – это и не простой вопрос).

В то же время функция `fread` обязана обеспечить возможность чтения данных, созданных в других программных и языковых окружениях. В этой функции использование точностных флагов, отличных от строки `'float64'`, весьма актуально и очень широко применяется на практике.

Вот простейший пример на эту тему. В языке C (или C++) вещественные числа с плавающей запятой могут быть представлены как с двойной точностью (это единственный вариант пакета MATLAB), так и с одинарной. В последнем случае каждая переменная такого типа занимает в памяти компьютера 4 байта. Тогда при чтении из файла таких значений функцией `fread` точностной флаг должен быть строкой `'float32'`.

Легко проверить все сказанное. Воспользуемся интегрированной средой Developer Studio компилятора Microsoft Visual C++ 5.0/6.0 и создадим простейший *консольный проект* (чтобы не связываться со сложными, но посторонними вопросами типа графического интерфейса пользователя), в рамках которого напишем единственный файл, содержащий следующий текст программы на языке C++:

```
#include <stdio.h>
int main( void )
{
  FILE* fid = NULL;
```

```
  char* pF = 'd:\\Matlab\\bin\\cTest.xxx';
  float a=1.1, b = 2.1, c = 3.1;
  fid = fopen( pF, 'wb' );
  fwrite( &a, sizeof(float), 1, fid );
  fwrite( &b, sizeof(float), 1, fid );
  fwrite( &c, sizeof(float), 1, fid );
  fclose( fid );
  return 0;
}
```

Откомпилируем этот проект, запустим на выполнение получившийся программный модуль, который и запишет в бинарный файл `'cTest.xxx'` три вещественных числа типа `float` (размером по 4 байта), равные 1.1, 2.1 и 3.1 соответственно.

После этого в командном окне системы MATLAB выполним следующий фрагмент кода на М-языке:

```
fid1 = fopen( 'cTest.xxx', 'rb' );
[ a , count ] = fread( fid1, [1,3], 'float32' );
fclose( fid1 );
```

создающий числовой массив a типа `double` размером `1 x 3` (см. рис. 3.15). Очень важно отметить, что при чтении бинарных файлов функцией `fread` всегда создаются массивы (векторы или матрицы) именно типа `double` независимо от форматного точностного флага, с которым вызвана эта функция.

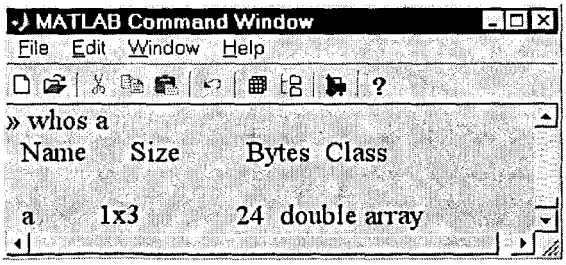


Рисунок 3.15

Проверим числовые значения элементов массива a, прочитанные из бинарного файла `'cTest.xxx'` показанным выше фрагментом кода:

```
a =
  1.1000  2.1000  3.1000
```

Отсюда видно, что прочитаны правильные значения. Если бы мы читали файл `'cTest.xxx'` с флагом `'float64'`, то мы не только бы получили неправильные числовые значения для элементов массива, но и вышли бы за границу файла.

Другим очень характерным примером использования в функции fread форматного точностного флага, отличного от 'float64', является случай формирования растрового изображения во внешних по отношению к системе MATLAB программах. Например, в следующей программе на языке C++

```
#include <stdio.h>
int main( void )
{
  FILE* fid = NULL;
  char* pF = 'd:\\Matlab\\bin\\cTest.xxx';
  unsigned char r = 200;
  unsigned char g = 100;
  unsigned char b = 50;
  fid = fopen( pF, 'wb' );
  int i;
  for(i=0;i<100*100;i++)fwrite( &r, 1, 1, fid );//RED
  for(i=0;i<100*100;i++)fwrite( &g, 1, 1, fid );//GREEN
  for(i=0;i<100*100;i++)fwrite( &b, 1, 1, fid );//BLUE
  fclose( fid );
  return 0;
}
```

создается простейшее (абсолютно однородное, красноватого цвета) изображение TrueColor размером 100 x 100 пикселов. А показанный ниже фрагмент кода на М-языке системы MATLAB, использующий флаг 'uchar' в функции fread (по одному байту на элемент массива):

```
fid1 = fopen( 'cTest.xxx', 'rb' );
[X1,count] = fread( fid1, [100,100], 'uchar' );
[X2,count] = fread( fid1, [100,100], 'uchar' );
[X3,count] = fread( fid1, [100,100], 'uchar' );
Img(:,:,1)=X1; Img(:,:,2)=X2; Img(:,:,3)=X3;
image(uint8(Img))
fclose( fid1 )
```

спокойно читает его и показывает в своем графическом окне. Помимо флага 'uchar' можно использовать эквивалентный ему флаг 'uint8', который также обеспечивает чтение по одному байту на элемент.

Так как нельзя функцией fread сразу прочесть содержимое бинарного файла в трехмерный массив, то мы сначала читаем отдельные *цветовые плоскости* (матрицы 100 x 100) в числовые матрицы X1, X2 и X3, а затем строим из них трехмерный массив Img типа double. Непосредственно перед показом растрового изображения функцией image мы конвертируем этот массив в тип uint8, о котором много говорилось в гл. 2, посвященной графической подсистеме пакета MATLAB.

Чтение и запись произвольных текстовых файлов

Для работы со строковыми данными идеально подходят *текстовые файлы*. Они хорошо приспособлены для записи больших фрагментов текста, поскольку этим файлам органически присуще «понимание» деления текста на строки (в самом что ни на есть книжном смысле этого слова), а бинарные файлы этого «не понимают», так как работают только с потоком байтов.

Однако текстовые файлы хороши и для записи чисел, если последние предварительно перевести в текстовое представление. Это бывает очень удобно, так как числовую информацию в текстовом представлении удобно читать текстовыми редакторами и готовить *смешанную тексто-числовую информацию* к «бумажной» публикации (печати). Рассмотрим по очереди все перечисленные возможности.

Чтобы работать с файлами в текстовом режиме, их нужно открыть с дополнительным флагом `'t'`, который добавляется к традиционным флагам `'r'` или `'w'`. Читать данные из текстовых файлов можно функциями `fgetl`, `fgets` и `fscanf`, а писать – функцией `fprintf`.

Начнем с чтения текстовых файлов. Для этого в текстовом редакторе Notepad подготовим файл `'MyString.txt'`, содержимое которого показано на рис. 3.16.

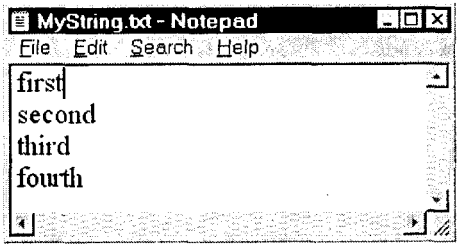


Рисунок 3.16

Теперь прочтем этот текст файловыми средствами пакета MATLAB. Для выполнения этой задачи хорошо подходят функции `fgetl` и `fgets`, каждая из которых читает из текстового файла одну строку текста (имеется в виду геометрическая строка текста, что на предыдущем рисунке соответствует, например, строке `first`, строке `second` и т. д.). Например,

```
fid1 = fopen( 'MyString.txt', 'rt' );
str1 = fgetl( fid1 );
str2 = fgetl( fid1 );
str3 = fgetl( fid1 );
str4 = fgetl( fid1 );
fclose( fid1 );
```

после чего проверяем содержимое текстовых массивов (вектор-строк) str1, str2, str3 и str4:

```
str1 =
first
str2 =
second
str3 =
third
str4 =
fourth
```

и убеждаемся, что чтение произошло корректно.

Если бы мы читали строки функцией fgets, то к концу каждой строковой переменной добавился бы символ *«перевод строки»*, имеющий ASCII-код 10. В результате длина переменной str1, например, стала бы равна шести (проверяется вызовом функции length).

На вопрос о том, какой из этих двух функций читать текстовые строки лучше, однозначно ответить трудно. Это всегда зависит от конкретных обстоятельств. В любом случае добавить к концу текстовой строки любой управляющий символ не составляет никакого труда. Вот пример добавления символа *«перевод строки»*:

```
str1 = [ str1, char( 10 ) ];
```

Функцией fscanf удобно читать текстовые файлы, в которых содержатся текстовые представления чисел. Функцией fscanf можно прочитать эти числа непосредственно в числовые переменные системы MATLAB. Работу этой функции рассмотрим после изучения вопроса о записи информации в текстовые файлы.

Для записи информации в текстовые файлы нужно применить функцию fprintf. Вот пример записи в файл как текстовой, так и числовой информации:

```
s1 = 'Number example'; n1 = 3.41; n2 = 7.18;
s2 = '----------------------';
fid = fopen( 'MyString5.txt', 'wt' );
fprintf(fid,'%s\n%s\n%4.2f  %4.2f\n%s',s1,s2,n1,n2,s2);
fclose( fid );
```

Одним вызовом функции fprintf удается записать по нескольку раз две строковые (символьные) переменные s1 и s2, а также записать в текстовый файл символьные представления числа n1 и числа n2. Управляет этим процессом текстовая строка, являющаяся вторым параметром функции fprintf. Назовем эту строку *управляющей строкой* функции fprintf. Эта строка помимо обычных символов, подлежащих выводу в файл, содержит еще *спецификаторы формата* и *управляющие символы*.

Спецификаторы формата и управляющие символы сами по себе в файл не выводятся. Они управляют процессом вывода в файл значений переменных, указанных в списке аргументов функции `fprintf` по порядку вслед за управляющей строкой. Управляющие символы (на самом деле это набор из двух символов, первым из которых является \ – обратная косая черта) требуют осуществить *табуляцию, переход на новую строку* или *переход на новую страницу*. Мы выше использовали единственный управляющий символ – `\n`, означающий переход на новую строку. Вместо символа `\n` реально в файл записывается некоторый числовой код, который в случае работы с файлом как текстовым всегда однозначно интерпретируется как требование перехода на новую физическую строку отображающего устройства.

Теперь рассмотрим спецификаторы формата. Функция `fprintf` просматривает свою управляющую строку слева направо и выводит в файл все встречающиеся в ней обычные символы за исключением спецификаторов формата – набора символов, начинающихся с `%`. Спецификаторы не являются данными, подлежащими выводу в файл. Они лишь говорят функции `fprintf` о том, в какой форме (отсюда слово *формат*) нужно значение очередной переменной, указанной в списке аргументов по порядку вслед за управляющей строкой, вывести в файл.

Количество и порядок спецификаторов в управляющей строке должны точно соответствовать списку переменных, подлежащих выводу и указанных в качестве параметров функции `fprintf` вслед за управляющей строкой.

Например, спецификатор `%s` означает, что очередная переменная, подлежащая выводу в файл, является строкой и надо реально выводить в файл числовые коды ее символов в соответствии с некоторой стандартной кодировкой. Другим очень распространенным спецификатором является набор символов `%f`, означающий, что выводу в текстовый файл подлежит вещественное число, которое предварительно нужно преобразовать в текстовую строку, содержащую символьный образ этого числа. Дополнительные числовые параметры и разделяющая их точка, стоящие в спецификаторе между символами `%` и `f`, означают полное количество позиций в символьном представлении числа и число позиций, отводимых для записи дробной части. Например, `%4.2f` означает, что вещественное число записывается набором символов в четырех позициях, из которых два отводится для записи дробной части числа.

С другими спецификаторами в любой момент можно ознакомиться, выполнив в командном окне системы MATLAB команду

```
help fprintf
```

Итак, рассмотренный выше фрагмент кода записывает в текстовый файл `'MyString5.txt'` как текст на естественном языке (английском), так и текстовое представление вещественных чисел `3.41` и `7.18`. После этого содержимое текстового файла `'MyString5.txt'` можно просмотреть любыми

текстовыми редакторами, в том числе простейшими редакторами типа Notepad (см. рис. 3.17).

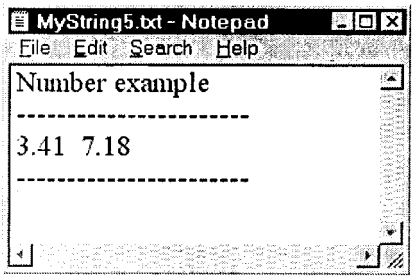


Рисунок 3.17

Это свойство текстовых файлов разительно отличает их от бинарных файлов, выуживание информации из которых готовыми стандартными средствами весьма затруднительно.

А теперь, наконец, рассмотрим функцию `fscanf`, предназначенную для чтения форматированных данных из текстовых файлов. Тренироваться будем как раз на только что записанном нами файле `'MyString5.txt'`. При этом текстовые строки будем извлекать более удобной для этого функцией `fgets`. Функцией же `fscanf` из третьей текстовой строки выделим два вещественных числа:

```
fid = fopen( 'MyString5.txt', 'rt' );
s1 = fgets( fid ); s2 = fgets( fid );
n = fscanf(fid,'%f'); s3 = fgets( fid );
fclose( fid );
```

Легко удостовериться, что в символьный массив `s1` прочитана строка `'Number example'`, а символьные массивы `s2` и `s3` содержат строку дефисов. Все это, правда, достаточно очевидно. А вот значение числовой переменной `n` может показаться весьма неожиданным (см. рис. 3.18).

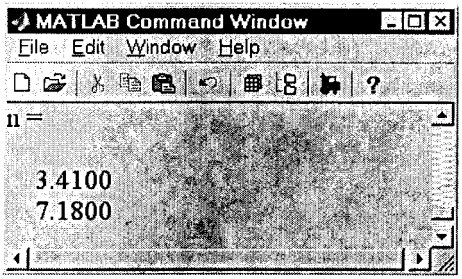


Рисунок 3.18

Переменная `n` оказалась числовым вектор-столбцом `2 x 1`. Это получилось потому, что в строке, которую читает функция `fscanf`, расположено текстовое представление двух вещественных чисел. В управляющей же строке функции

fscanf указан лишь один форматный спецификатор (он требует преобразования строки в вещественное число), но функция fscanf применяет его столько раз, сколько требуется для исчерпания числовой интерпретации прочитанной строки. В результате в переменную n читаются два числа, и поэтому она становится массивом 2 x 1.

Если бы нужно было прочитать числа в две разные переменные, то вместо представленного выше единственного вызова функции fscanf нужно подставить следующие три строки кода:

```
n1 = fscanf( fid, '%f', 1 );
n2 = fscanf( fid, '%f', 1 );
fgets( fid );
```

Теперь числовые переменные n1 и n2 имеют как раз те самые значения, которые мы выше записывали в файл 'MyString5.txt'. Для этого в функции fscanf был указан третий параметр, означающий количество чисел, читаемых за один вызов этой функции. Побочным следствием такого подхода является то обстоятельство, что функция fscanf не читает текстовую строку до самого конца. В итоге требуется вспомогательный вызов функции fgets для окончательного «исчерпания» строки, из которой извлекаются числа. Именно после такого исчерпания следующий вызов функции fgets:

```
s3 = fgets( fid );
```

успешно прочтет самую последнюю строку текстового файла 'MyString5.txt', состоящую из дефисов.

Из предыдущих примеров видно такое важное свойство файловых функций системы MATLAB, как *массовое* чтение (запись) данных за один вызов функции. Чтобы явно подчеркнуть это замечательное свойство файловых функций рассмотрим наглядный пример на эту тему. Требуется вычислить небольшую таблицу значений некоторой математической функции и записать эту таблицу в текстовый файл. В результате такую таблицу можно будет просматривать любым текстовым редактором, вставлять в качестве фрагмента в документ редактора Microsoft Word и выполнять другие стандартные манипуляции.

Для определенности выберем в качестве такой математической функции функцию «косинус» и построим таблицу ее значений для 11 значений аргумента:

```
x = 0:0.1:1; a =[ x; cos(x) ];
fid = fopen( 'MyCosinus1.txt', 'wt' );
fprintf( fid, '%5.3f  %5.3f\n', a );
fclose( fid );
```

Здесь матрица a состоит из двух строк и 11 столбцов. В каждом столбце присутствует (в первой строке) некоторое значение аргумента и (во второй строке) соответствующее ему значение функции. Далее единственным вызовом функции fprintf удается записать в текстовый файл 'MyCosinus1.txt' все значения

аргументов и все значения функции (это пример высочайшей компактности записи). При этом функция `fprintf` извлекает для вывода в файл элементы матрицы `a` по столбцам: сначала записываются в файл два элемента (сверху вниз) первого столбца, затем – два элемента второго столбца и т. д. вплоть до исчерпания всех столбцов. Кроме того, функция `fprintf` использует управляющий символ «перевод строки» после записи каждых двух чисел. Это обеспечивает горизонтальное расположение в результирующем текстовом файле аргумента и соответствующего ему значения функции. В этом можно убедиться, открыв файл `'MyCosinus1.txt'` в текстовом редакторе Notepad (см. рис. 3.19).

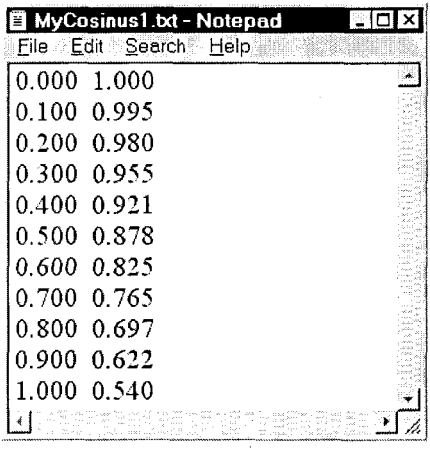


Рисунок 3.19

Восстановить в рабочем пространстве системы MATLAB ранее сформированную матрицу `A`, прочитав ее элементы из текстового файла `'MyCosinus1.txt'`, также не составляет большого труда. При этом один вызов функции `fscanf` обеспечивает массовое чтение чисел из текстового файла:

```
fid = fopen( 'MyCosinus1.txt', 'rt' );
A = fscanf( fid, '%f', [2 11]);
fclose( fid );
```

Здесь третий параметр функции `fscanf` подсказывает интерпретатору М-языка системы MATLAB о структуре матрицы `A` (2 строки и 11 столбцов). При этом восстановление элементов матрицы `A` производится, естественно, приближенное, так как при записи в файл производилось округление до трех десятичных цифр после запятой (был применен спецификатор формата `%5.3f`).

Завершая рассказ о файловых операциях, все-таки отметим, что полное и мастерское владение всеми рассмотренными файловыми функциями системы MATLAB требует солидного опыта в программировании (об этом можно догадаться даже по приведенным примерам). Мы не рассказали обо всех нюансах файловых операций, но восполнить этот пробел вполне реально интенсивной практикой.

Глава 4

Краткий обзор встроенных средств решения типовых задач алгебры и анализа

Решение систем линейных уравнений

Как мы знаем, в системе MATLAB для решения систем линейных уравнений предусмотрены знаки операций – это знаки / и \.

Со вторым из этих знаков мы уже знакомились раньше в гл. 1, когда решали систему линейных уравнений вида

```
Ay = b
```

где A – заданная квадратная матрица N x N, а b – заданный вектор-столбец длины N. Для нахождения неизвестного вектор-столбца y достаточно применить операцию \ и вычислить выражение A\b. Вот соответствующий пример:

```
A = [1,-2,3,-1;2,3,-4,4;3,1,-2,-2;1,-3,7,6];
b = [6;-7;9;-7];
y = A \ b
y =
   2.0000
  -1.0000
   0
  -2.0000
```

В общем случае операция \ называется *левым делением матриц* и, будучи примененная к матрицам A и B в виде A\B, примерно эквивалентна вычислению выражения

```
inv(A)*B
```

но при этом вычисляется по-другому. Здесь под inv(A) понимается вычисление матрицы, обратной к матрице A.

Операцию / называют *правым делением матриц*. Выражение A/B примерно соответствует вычислению выражения B*inv(A). Значит, эта операция позволяет решать системы линейных уравнений вида

```
YA = B
```

так как решением именно этого уравнения служит выражение B*inv(A).

Операции линейной алгебры над матрицами. Матричные функции

С численным решением линейных уравнений существует известное число трудноразрешимых проблем. В частности, говорят, что *матрица системы плохо обусловлена*, если численное решение очень чувствительно к небольшим изменениям исходных данных.

Следующие встроенные функции системы MATLAB позволяют оценить числа обусловленности матриц.

Функция `cond(A)` вычисляет *число обусловленности* как отношение самого большого сингулярного значения `A` к наименьшему. Если это число близко к единице, то матрица хорошо обусловлена.

Функция `condeig(A)` вычисляет вектор чисел обусловленности для собственных значений `A`.

Функция `rcond(A)` вычисляет обратную величину обусловленности матрицы `A`, основываясь на методах широко известного пакета линейной алгебры LINPACK. Возвращаемые значения, близкие к единице, говорят о хорошей обусловленности, а близкие к нулю – о плохой.

Проблема выбора одного из перечисленных методов достаточно сложна, так как сама проблема обусловленности систем линейных уравнений не является простой и однозначной. Разные методы, примененные к одной и той же матрице, могут дать различающиеся результаты. В любом случае могут выручить отличные знания по линейной алгебре, но это уже проблема вовсе не пакета MATLAB.

Полный набор функций линейной алгебры, реализованный в системе MATLAB, впечатляет. По этой проблематике пакет MATLAB является чемпионом. Однако сама эта проблема является математически чрезвычайно сложной и не входит в активный математический багаж большинства инженеров и физиков. Эту проблематику целесообразно подробно обсудить в специальной математической литературе.

Здесь же мы кратко перечислим основные функции линейной алгебры, реализованные в рамках пакета MATLAB.

Функция `det` вычисляет определитель квадратной матрицы. Для вычисления ранга матрицы используется функция `rank`. Функция `norm` вычисляет норму матрицы – скаляр, дающий оценку величины элементов матрицы. Для определения ортонормированных базисов матриц применяется функция `orth`. Функции `rref` и `rrefmovie` используются для приведения матриц к треугольной форме.

Функция `chol` осуществляет *разложение Холецкого*. Выражение

```
t = chol(A)
```

возвращает для положительно определенной матрицы `A` верхнюю треугольную матрицу `t`, такую, что

```
t'*t = A
```

Функции lu и qr выполняют так называемые *lu-* и *qr-разложения матриц*. Например, функция lu вызывается следующим образом:

```
[ L, U ] = lu( A )
```

и возвращает верхнюю треугольную матрицу U и нижнюю треугольную матрицу L, причем имеет место равенство

```
A = L * U
```

Например, если

```
A = [ 1,2,3; 4,3,2; 2,1,5];
```

то

```
[ L, U ] = lu( A );
L =
  0.2500   1.0000   0
  1.0000   0        0
  0.5000  -0.4000   1.0000
U =
  4.0000   3.0000   2.0000
  0        1.2500   2.5000
  0        0        5.0000
```

Теперь осуществляем проверку, перемножая L и U:

```
L * U
ans =
  1  2  3
  4  3  2
  2  1  5
```

и получаем в точности исходную матрицу A.

Функция qr вызывается следующим образом:

```
[ Q, R ] = qr( A )
```

и возвращает верхнюю треугольную матрицу R того же размера, что и A, и унитарную матрицу Q, причем имеет место равенство

```
A = Q * R
```

Например, если

```
A = rand(3,2);
```

то мы получаем

```
[ Q, R ] = qr( A );
```

```
Q =
  -0.8256   0.4039  -0.3940
  -0.2008  -0.8629  -0.4637
  -0.5273  -0.3037   0.7936
R =
  -1.1508  -0.9821
   0       -0.8043
   0        0
```

матрицы Q и R с описанными выше свойствами.

Функции, вычисляющие *собственные значения* и *сингулярные числа матриц*, обязаны входить в любой достаточно полный набор функций линейной алгебры. В системе MATLAB вычислением собственных значений занимается функция eig.

Формат вызова этой функции весьма многообразен. Например, вызов

```
d = eig( A )
```

просто возвращает вектор d собственных значений матрицы A. А формат вызова

```
[V,D]=eig( A )
```

помимо матрицы собственных значений D возвращает еще и матрицу собственных векторов V, причем справедливо равенство

```
A * V = V * D
```

Матрицу D называют канонической формой матрицы A, и состоит эта диагональная матрица (помимо нулей) из элементов, стоящих на главной диагонали и равных собственным значениям матрицы A. Матрица же V состоит из столбцов, каждый из которых есть один из собственных векторов матрицы A.

Для вычисления сингулярных чисел матриц предназначена функция svd. В варианте вызова

```
s = svd( A )
```

она просто возвращает вектор s, элементы которого являются искомыми сингулярными числами. При другом возможном вызове:

```
[U,S,V]=svd( A )
```

она возвращает помимо диагональной матрицы S с сингулярными числами на диагонали еще две унитарные матрицы U и V, такие, что имеет место равенство

```
A = U * S * V'
```

Вот пример этого вызова функции svd:

```
A = [1,2,3;4,5,6;7,8,9];
[U,S,V]=svd( A );
```

```
S =
  16.8481   0        0
  0         1.0684   0
  0         0        0.0000

U =
  0.2148    0.8872  -0.4082
  0.5206    0.2496   0.8165
  0.8263   -0.3879  -0.4082

V =
  0.4797   -0.7767   0.4082
  0.5724   -0.0757  -0.8165
  0.6651    0.6253   0.4082
```

Имеются также функции `cdf2rdf`, `hess`, `qz`, `rsf2csf` и `schur`, осуществляющие приведение матриц к специальным формам Хессенберга и Шура.

Теперь кратко рассмотрим *матричные функции* `expm`, `logm` и `sqrtm`. Эти функции имеют своим аргументом матрицу и возвращают матрицу. Они не имеют ничего общего с применением обычных функций `exp`, `log`, `sqrt` к матричному аргументу, так как в последнем случае вычисления производятся просто поэлементно. Настоящие же матричные функции, которые имеют суффикс `m`, осуществляют вычисления с целыми матрицами по правилам линейной алгебры, отталкиваясь от разложений математических функций в степенные ряды.

Например, если

```
A = [ 1,1,0; 0,0,2; 0,0,-1 ];
```

то тогда

```
expm( A )
ans =
  2.7183   1.7183   1.0862
  0        1.0000   1.2642
  0        0        0.3679
```

в то время как

```
exp( A )
ans =
  2.7183   2.7183   1.0000
  1.0000   1.0000   7.3891
  1.0000   1.0000   0.3679
```

и мы видим, что результаты разные, хотя диагональные элементы и совпадают.

Разреженные матрицы

При решении многих прикладных задач, особенно при решении граничных задач в уравнениях с частными производными, возникают матрицы большого размера, у которых большинство элементов равны нулю. Если хранить такие матрицы обычным образом, то будет явный перерасход памяти. Кроме того, быстродействие вычислений с так организованными матрицами при их больших размерах будет мало.

В системе MATLAB эта проблема преодолевается введением специальных структур данных для таких матриц, то есть это еще один тип массивов системы MATLAB. Этот тип называется *разреженными матрицами*.

Если некоторая заданная матрица A имеет большинство нулевых элементов, то ее будет целесообразно перевести в другую форму внутреннего представления, то есть преобразовать ее в разреженную матрицу с помощью функции sparse:

```
S = sparse( A )
```

Здесь матрица S уже является разреженной матрицей системы MATLAB, так что она хранится в памяти компьютера более эффективно, чем исходная матрица A, и с ней возможны более быстрые вычисления.

Обратное преобразование разреженной матрицы в обычную производится функцией full:

```
A = full( S );
```

Приведем иллюстрирующий пример. Пусть дана обычная матрица A:

```
A =[1,0,0;1,0,0;0,1,0];
S = sparse( A );
S =
   (1,1)  1
   (2,1)  1
   (3,2)  1
```

Отсюда хорошо видно, что в памяти компьютера для разреженных функций хранятся только ненулевые элементы вместе с их индексами. Ясно, что этой информации достаточно, чтобы иметь возможность восстановить исходную матрицу целиком.

Матрица S уже не имеет тип double, как это имеет место для исходной матрицы A. Она теперь имеет тип sparse и занимает меньший объем памяти, в чем всегда можно легко убедиться, вызвав функцию whos из командной строки системы MATLAB (см. рис. 4.1).

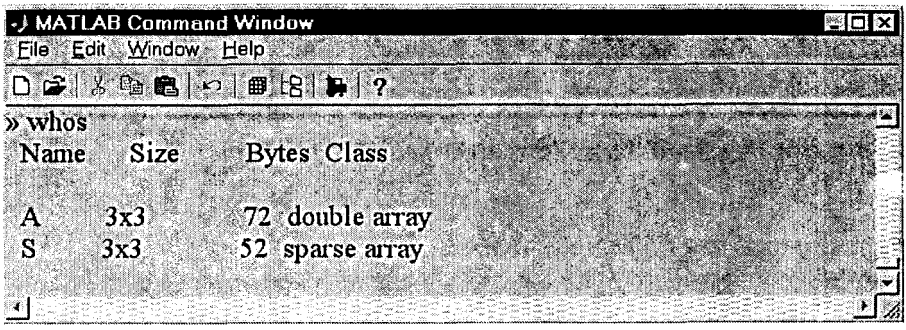


Рисунок 4.1

Для разреженных матриц в рамках системы MATLAB предназначено большое количество особых функций. Мы не будем подробно рассматривать здесь этот специальный вопрос. Любую справочную информацию по этому поводу можно всегда посмотреть в справочном файле

```
\help\pdf_doc\matlab\Using_ml.pdf
```

в разделе, озаглавленном «Sparse Matrices».

Вычисление спецфункций математической физики

Помимо всех элементарных функций система MATLAB предоставляет возможность вычисления многих стандартных специальных функций математической физики. Эти функции встречаются в различных задачах математической физики и либо являются решениями некоторых дифференциальных уравнений, либо с самого начала задаются в виде рядов Тейлора (иногда эти ряды конечные, как в случае известных ортогональных многочленов).

Самыми известными специальными функциями математической и теоретической физики являются *функции Бесселя*. Бывают функции Бесселя первого, второго и третьего рода.

Для вычисления функций Бесселя первого рода в системе MATLAB служит функция

```
besselj( n, X )
```

где `n` – порядок функции; `X` – массив аргументов этой функции (напоминаем, что в системе MATLAB допустимы групповые вычисления, когда вместо единственного скалярного аргумента на вход функции подается целый массив аргументов).

Для вычисления функций Бесселя второго рода в системе MATLAB служит функция

```
bessely( n, X )
```

а для вычисления функций Бесселя третьего рода – функция

```
besselh( n, X )
```

которую также называют *функцией Ханкеля*.

Поведение этих функций лучше всего проиллюстрировать графически:

```
X  = 0:0.01:15;
y0 = besselj(0,X);
y1 = besselj(1,X);
y2 = besselj(2,X);
plot(x,y0,x,y1,x,y2)
```

Этот фрагмент позволяет получить графики функций Бесселя первого рода для трех значений порядка этих функций (см. рис. 4.2).

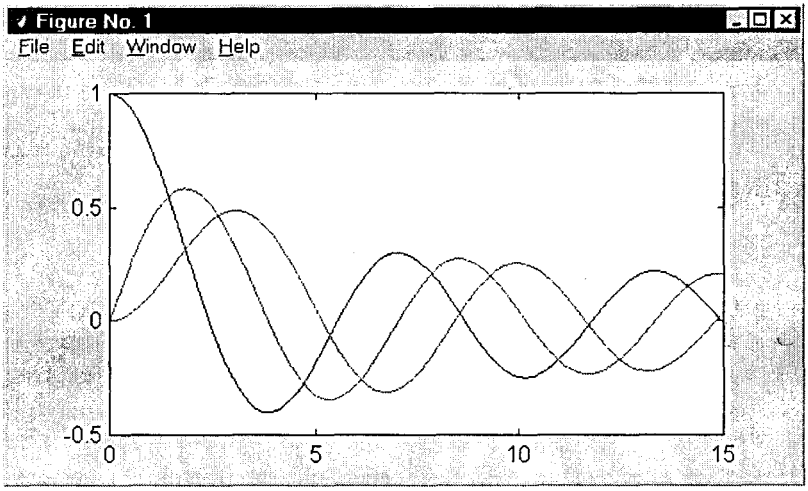


Рисунок 4.2

Точно так же можно получить графики для функций Бесселя второго и третьего рода.

Помимо функций Бесселя система MATLAB включает функции, позволяющие вычислить числовые значения функций Эйри, бета-функций, гамма-функций, функций ошибки (`erf(x)`), интегральных показательных функций. Также можно вычислять эллиптические интегралы и эллиптические функции Якоби. Также есть функция `legendre` для вычисления ортогональных многочленов Лежандра. Форматы вызова этих функций легко узнать из справочного файла

```
\help\pdf_doc\matlab\Ref\Refbook.pdf
```

В целом система MATLAB располагает достаточно полным (хотя и неисчерпывающим) набором встроенных функций, позволяющих получать численные значения основных специальных функций математической физики.

Нахождение нулей функций

На практике часто приходится сталкиваться с задачей о *нахождении корней уравнений*. Любое уравнение можно записать в виде *равенства некоторой функции нулю*, так что это и есть задача о нахождении нулей функций.

Решение указанной задачи осуществляет функция `fzero`. В качестве первого аргумента ей передается имя функции, задающей исходное уравнение. Вторым аргументом служит *начальное приближение к корню*:

```
fzero( name, x0 )
```

Возвращаемым значением функции `fzero` является нуль функции `name` в окрестности точки `x0`.

Для примера рассмотрим задачу о нахождении нулей функции `cos(x)` на отрезке от `0` до `pi`. В качестве начального приближения примем `x0=1`. Вызываем функцию `fzero` с указанным начальным приближением и получаем следующий результат:

```
x = fzero( 'cos', 1 )
x =
  1.5708
```

Легко видеть, что мы в качестве нуля функции `cos(x)` получили значение, близкое к точному значению корня, равному `pi/2`.

Если требуется найти корень функции, отличной от стандартной (встроенной в систему MATLAB) и тем самым не имеющей в рамках системы MATLAB фиксированного имени, то нужно приписать некоторое имя выражению, вычисляющему функцию. Этот вопрос относится к программированию так называемых М-функций, которые мы будем подробно и систематически изучать в гл. 5 и 6 из ч. 2 данного пособия. Сейчас же ограничимся простейшими сведениями.

Пусть, например, требуется найти корни уравнения `cos(x) = x`, что эквивалентно нахождению нулей функции, вычисляемой по формуле `y = cos(x) - x`, не имеющей в рамках системы MATLAB фиксированного имени. В этом случае нужно в любом простейшем текстовом редакторе (типа встроенного в операционную систему Windows редактора Notepad) набрать две строки следующего кода:

```
function y = MyFunction1( x )
y = cos(x) - x;
```

и запомнить их в файле MyFunction1.m, который нужно разместить в текущем каталоге системы MATLAB (узнать его можно командой cd). После этого можно воспользоваться функцией fzero:

```
x = fzero('MyFunction1',pi/2)
x =
  0.7391
```

Проверим, насколько близко к нулю значение функции MyFunction1 в точке 0.7391 (см. рис. 4.3).

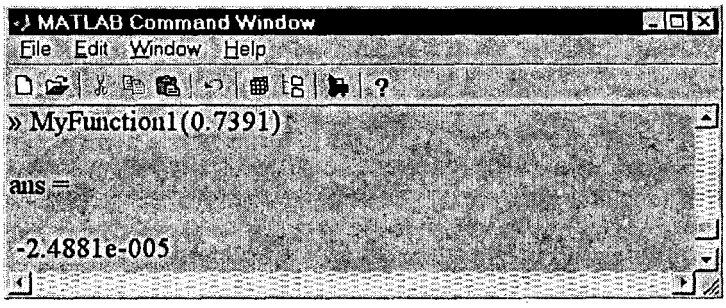


Рисунок 4.3

Если найдено абсолютно точное значение корня, то значение функции в этой точке равно нулю. Таким образом, величина функции в приближенно найденном нуле косвенно характеризует погрешность результата. Чтобы управлять погрешностью, нужно осуществлять вызов функции fzero с тремя аргументами:

```
fzero( name, x0, tol )
```

где параметр tol задает требуемую величину погрешности (ошибки). Повторим предыдущие вычисления, потребовав большей точности расчетов (то есть меньшей погрешности):

```
format long
x = fzero('MyFunction1',pi/2, 1e-8)
x =
 0.73908513263090
MyFunction1(x)
ans =
9.778322596076805e-010
```

откуда видно, что действительно достигнута большая точность нахождения нуля функции.

Еще раз подчеркнем, что функция fzero находит нули только вещественнозначных функций одной вещественной переменной. Однако часто бывает необходимо найти комплексные корни вещественнозначных функций, особенно в случае многочленов.

Для этой цели в системе MATLAB существует специальная функция roots, которой в качестве аргумента передается массив коэффициентов многочлена. Например, для многочлена x ^ 4 - 3x ^ 3 + 3x ^ 2 - 3x + 2, имеющего два вещественных (1 и 2) и два комплексных корня (i и -i), нужно сначала сформировать массив его коэффициентов:

```
Coef = [ 1, -3, 3, -3, 2 ]
```

после чего вызвать функцию roots:

```
r = roots( Coef )
r =
2.00000000000000
0.00000000000000 + 1.00000000000000i
0.00000000000000 - 1.00000000000000i
1.00000000000000
```

В задаче о нахождении нулей функции сложным моментом является нахождение начального приближения к нулю функции, а также априорная оценка их количества. Поэтому важно параллельно с применением функций типа roots или fzero визуализировать поведение искомых функций на том или ином отрезке значений аргумента. Максимальное содействие в этом может оказать функция fplot(name,[x0,x1]), строящая график функции с именем name на отрезке от x0 до x1.

Например, для ранее рассмотренной функции MyFunction1 с помощью вызова функции

```
fplot( 'MyFunction1', [0,pi/2] )
```

убеждаемся, что на отрезке от 0 до pi/2 действительно существует единственный нуль этой функции (см. рис. 4.4).

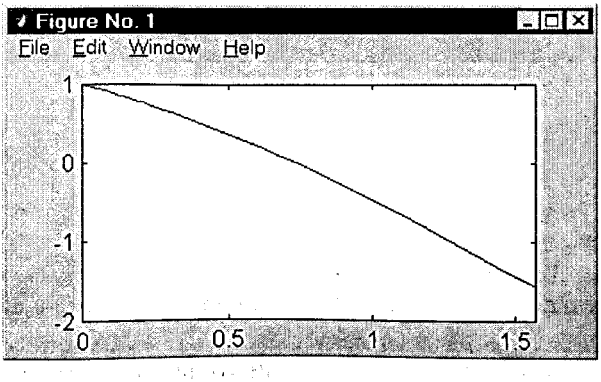


Рисунок 4.4

Функция fplot требует от пользователя еще меньше подготовительной работы, чем даже изученная нами ранее функция plot (см. гл. 2).

Поиск минимума функции

В системе MATLAB имеются специальные функции для поиска минимумов заданных функций. При этом возможен поиск минимума как для функции одной вещественной переменной, так и для функций многих переменных.

Для функций одной переменной их минимумы разыскивает функция `fmin`:

```
fmin( name, x0, x1 )
```

Здесь `name` представляет имя функции, у которой находятся минимумы, а `x0` и `x1` задают отрезок поиска. Иллюстрировать работу этой функции будем на примере функции `hump` (переводится как «горб»), специально поставляющейся с системой MATLAB в демонстрационных целях. Эта функция задается формулой

```
y = 1./((x - 0.3).^2 + 0.01)+1./((x - 0.9).^2 + 0.04) - 6;
```

а ее график легко получается с помощью вызова функции

```
fplot( 'humps', [0,3] )
```

Из рисунка 4.5 видно, что локальный минимум этой функции существует на отрезке от `0.5` до `1.0`. Попробуем найти точку минимума следующим вызовом функции `fmin`:

```
x = fmin( 'humps', 0.5, 1.0 )
x =
  0.63701067459059
```

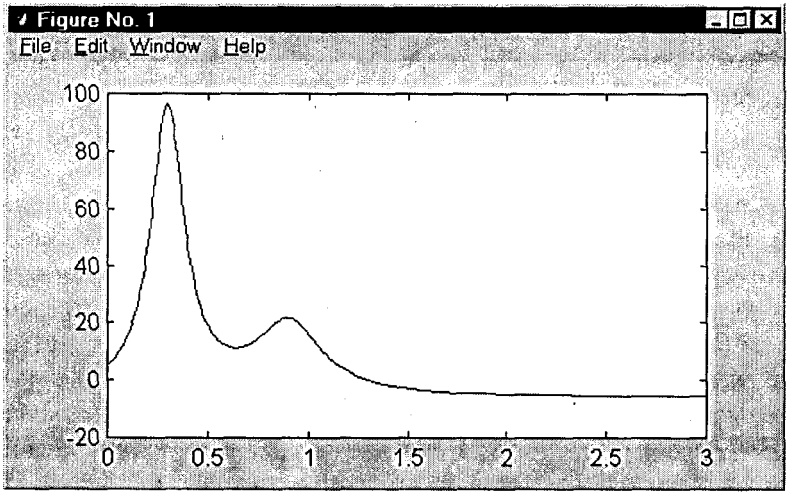


Рисунок 4.5

Для поиска минимума функции нескольких переменных применяется функция fmins:

```
xmin = fmins( name, x0 )
```

Здесь name является именем функции нескольких переменных, для которой ищется минимум, а x0 – это вектор ее аргументов, с которого начинается поиск.

Для иллюстративного примера создадим простую функцию двух переменных

```
function y = MyFunc2( x )
y = x(1)^2 + x(2)^2;
```

имеющую минимумом точку (0,0). Этот текст надо записать в файл MyFunc2.m в текущий каталог системы MATLAB. После этого можно вызвать функцию fmins:

```
xmin = fmins( 'MyFunc2', [1,1] );
```

которая приближенно находит вектор xmin координат точки минимума:

```
xmin(1)
ans =
-2.102352926236483e-005
xmin(2)
ans =
2.548456493279544e-005
```

Обе найденные координаты близки к своим точным значениям, равным нулю.

Для функций нескольких переменных еще важнее, чем для ранее рассмотренных функций одной вещественной переменной, постараться априорно оценить количество и приблизительное нахождение локальных минимумов. Тут могут существенно помочь трехмерные графики, которые мы в гл. 2 строили для функций двух вещественных переменных.

Например, для функции peaks в подразделе «Дополнительные детали оформления трехмерных графиков» из гл. 2 функцией surfc был построен очень наглядный трехмерный график, на котором также были нанесены линии уровня. Из представленной графической картинки хорошо видно, что имеются три локальных минимума. Первый располагается вблизи точки с координатами [0,0], а двое других – вблизи точек [0,-1.5] и [-1.5,0] соответственно. Можно воспользоваться этой приближенной информацией и найти уточненные значения локальных минимумов этой функции. Однако самой функцией peaks здесь воспользоваться не удастся, так как она возвращает несколько значений, не согласующихся с функцией fmins. Поэтому мы сами создадим функцию MyPeaks:

```
function z = MyPeaks( X )
x = X( 1 ); y = X( 2 );
```

```
z = 3*(1 - x).^2.*exp(-(x.^2) - (y+1).^2) - …
    10*(x/5 - x.^3 - y.^5).* exp(-x.^2 - y.^2) - …
    1/3*exp(-(x+1).^2 - y.^2);
```

где многоточие означает перенос записи выражения на другую строку. Здесь для вычисления значения z функции MyPeaks по двум значениям аргументов x и y используется та же самая формула, что и в функции peaks. Далее представленный текст функции запишем в файл MyPeaks.m, после чего вызовем функцию поиска минимума fmins:

```
xmin1 = fmins( 'MyPeaks', [0,0] );
xmin1(1)
ans =
   0.2964
xmin1(2)
ans =
   0.3202
xmin2 = fmins( 'MyPeaks', [0,-1.5] );
xmin2(1)
ans =
   0.2283
xmin2(2)
ans =
   -1.6255
xmin3 = fmins( 'MyPeaks', [-1.5,0] );
xmin3(1)
ans =
   -1.3474
xmin3(2)
ans =
   0.2045
```

Если бы мы захотели узнать, какое количество итераций пришлось осуществить для нахождения уточненного результата, то функцию fmins нужно было бы вызывать в формате с двумя возвращаемыми значениями:

```
[ xmin, Options ] = fmins( name, x0 )
```

где возвращаемый вектор Options содержит в качестве отдельных своих элементов самые разнообразные сведения о процессе нахождения минимума. В частности, элемент Options(10) содержит количество совершенных итераций. Если бы мы в только что рассмотренном примере использовали этот формат вызова функции fmins, то нам было бы известно число итераций:

```
[xmin3,Opts] = fmins( 'MyPeaks', [-1.5,0] );
```

```
xmin3(1)
ans =
   -1.3474
xmin3(2)
ans =
    0.2045
Opts(10)
ans =
    88
```

Обычно чем дальше от истинного минимума располагается начальная точка для итераций, тем больше требуется их число:

```
[xmin3,Opts] = fmins( 'MyPeaks', [-2,0] );
xmin3(1)
ans =
   -1.3474
xmin3(2)
ans =
    0.2045
Opts(10)
ans =
    106
```

Как только мы задали худшее начальное приближение, так тут же получили возросшее число необходимых итераций для достижения одной и той же точности.

Вычисление определенных интегралов

Классической задачей численного анализа является задача о вычислении определенных интегралов. Из всех методов вычисления определенных интегралов самым простым, но в то же время довольно успешно применяемым является *метод трапеций*. Система MATLAB не могла обойти вниманием этот численный метод.

Для вычисления интегралов методом трапеций в ней предусмотрена функция `trapz`:

```
Integ = trapz( x, y );
```

Одномерный массив `x` (вектор) содержит дискретные значения аргументов подынтегральной функции. Значения подынтегральной функции в этих точках сосредоточены в одномерном массиве `y`. Чаще всего для интегрирования выби-

рают равномерную сетку, то есть значения элементов массива x отстоят друг от друга на одну и ту же величину – *шаг интегрирования*. Точность вычисления интеграла зависит от величины шага интегрирования: чем меньше этот шаг, тем больше точность.

Вычислим простой интеграл

$$\int_0^{\pi} \cos(x)dx$$

методом трапеций с разной величиной шага интегрирования. Сначала зададим шаг интегрирования равным pi/10:

```
dx = pi/10; x = 0:dx:pi; y=cos(x); I1 = trapz(x,y);
I1 =
  5.5511e-017
```

и получим поразительно точный результат (абсолютно точный равен нулю), который бесполезно пытаться улучшать, так как уже достигнута предельная точность, обусловленная конечной точностью хранения вещественных чисел на компьютере (примерно 16 десятичных знаков после запятой).

Такая высокая точность является скорее исключением, чем правилом. Обычно же для достижения высокой точности требуется выполнять интегрирование с очень малыми шагами, а контроль достигнутой точности осуществлять путем сравнения последовательных результатов. Например, при вычислении методом трапеций интеграла

$$\int_0^{5} \sin(x) * exp(-x)dx$$

с шагом интегрирования dx = 1

```
dx = 1; x = 0:dx:5; y = sin(x) .* exp(-x);
I2 = trapz(x,y);
```

получается следующий результат:

```
I2 =
  0.4226
```

Если провести повторное интегрирование с шагом dx = 0.1, то результат будет

```
I2 =
  0.5014
```

Уменьшая далее шаги интегрирования последовательно в 10 раз, находим, что при dx = 0.01 интеграл равен 0.5023, то есть уже здесь видна тенденция к стабилизации результата. При dx = 0.001 снова получаем

```
I2 =
  0.5023
```

так что в пределах четырех десятичных цифр после запятой результат получен окончательно. Напомним, что если мы хотим наблюдать дальнейшие десятичные цифры, то нужно предварительно ввести и исполнить команду

```
format long
```

Метод трапеций является очень универсальным методом и хорошо подходит для интегрирования не слишком гладких функций. Если же функция под знаком интеграла является гладкой (существуют и непрерывны несколько первых производных), то лучше применять методы интегрирования более высоких порядков точности. При одном и том же шаге интегрирования методы более высоких порядков точности достигают более точных результатов.

В системе MATLAB методы интегрирования более высоких порядков точности реализуются функциями quad (метод Симпсона) и quad8 (метод Ньютона – Котеса 8-го порядка точности). Оба этих метода являются к тому же *адаптивными*. Последнее означает, что пользователю нет необходимости контролировать достигнутую точность результата путем сравнения последовательных значений, соответствующих разным шагам интегрирования. Все это указанные адаптивные функции выполняют самостоятельно.

У функции quad8 более высокий порядок точности по сравнению с функцией quad, что очень хорошо для гладких функций, так как обеспечивается более высокая точность результата при большем шаге интегрирования (меньшем объеме вычислений). Однако функция quad может иметь не меньшее, а даже большее быстродействие для не слишком гладких функций (разрывны или велики по абсолютной величине вторая или третья производные). В любом случае обе эти функции по умолчанию обеспечивают одинаковую относительную точность результата, равную 0.001.

Как и многие другие функции системы MATLAB, функции quad и quad8 могут принимать различное количество параметров. Минимальный формат вызова этих функций включает в себя три параметра: имя подынтегральной функции, нижний предел интегрирования и верхний предел интегрирования. Если применяется четвертый параметр, то он является требуемой относительной точностью результата вычислений. Кстати, если обе эти адаптивные функции не могут обеспечить получение необходимой точности (расходящийся или близкий к этому интеграл), то они возвращают символическую бесконечность Inf.

Выше в подразделе «Поиск минимума функции» мы приводили график функции humps, для которой там разыскивались локальные минимумы. Вычислим теперь определенный интеграл от этой функции в пределах от нуля до трех, воспользовавшись для этого обеими адаптивными функциями:

```
[ I, cnt ] = quad( 'humps', 0, 3 );
I =
  23.9681
cnt =
  225
```

```
[ I8 cnt8 ] = quad8( 'humps', 0, 3 );
I8 =
  23.9681
cnt8 =
  113
```

Второе из возвращаемых значений для этих функций означает количество точек, в которых пришлось вычислять подынтегральную функцию. Таким образом, этот параметр характеризует трудоемкость метода. Сравнивая полученные показатели, приходим к выводу, что при интегрировании такой гладкой функции, как humps, преимущество имеет метод quad8, так как для достижения минимальной точности в 0.001 (а по ходу дела может достигаться и более высокая точность) ему требуется меньшее количество вычислений с подынтегральной функцией (113 против 225).

Из высшей математики известно, что к определенным интегралам могут быть сведены многие другие типы интегралов, например криволинейные интегралы. Таким образом, с помощью функций quad, quad8 (или trapz) можно вычислить и эти интегралы.

Рассмотрим пример на криволинейные интегралы первого рода. Пусть требуется вычислить массу M винтовой линии C:

```
x = sin(t); y = 2cos(t); z =3t; 0 <= t <= 2;
```

с постоянной линейной плотностью, равной 5. Задача решается с помощью *криволинейного интеграла первого рода*:

$$M = \int_C 5ds,$$

который сводится к вычислению следующего обыкновенного определенного интеграла:

$$M = 5\int_0^2 \sqrt{(x')^2 + (y')^2 + (z')^2}dt = 5\int_0^2 \sqrt{\cos^2 t + 4\sin^2 t + 9}dt.$$

Для вычисления подынтегральной функции создадим следующий текст:

```
function z = MyFunc321( t )
z = sqrt( cos(t).^2 + 4*sin(t).^2 + 9 );
```

который запишем в файл MyFunc321.m, после чего вызываем функцию quad:

```
M = 5 * quad( 'MyFunc321', 0, 2 );
M =
  34.2903
```

Двойные интегралы сводятся к вычислению повторных определенных интегралов, один из которых является внутренним, а другой внешним. Внутренний интеграл является подынтегральной функцией для внешнего интеграла. Можно

было бы написать некоторую цепочку вычислений (программу), в которой многократные вычисления подынтегральной функции сводились бы к многократным вызовам функции quad. Однако нет необходимости делать это самостоятельно, так как в системе MATLAB для этого имеется специальная функция dblquad.

В результате в случае необходимости вычисления двойного интеграла, такого, как, например, интеграл

$$\int_0^1 \int_1^2 (x \sin(y) + y \sin(x)) dxdy,$$

достаточно оформить подынтегральную функцию в следующем виде:

```
function z = Fof2Var( x, y )
z = x.*sin(y) + y.*sin(x);
```

(записав этот текст в файл Fof2Var.m) и вызвать функцию dblquad:

```
J = dblquad( 'Fof2Var', 0, 1, 1, 2 );
J =
  1.1678
```

в результате чего и получим искомое значение двойного интеграла (это чисто учебный пример, так как данный интеграл аналитическими преобразованиями сводится к произведению однократных определенных интегралов).

Решение систем обыкновенных дифференциальных уравнений

В предыдущем подразделе мы познакомились на примере задачи о вычислении определенных интегралов с функциями системы MATLAB, которые можно было бы назвать *интеллектуальными решателями* конкретной математической проблемы. Там такими «решателями» были адаптивные функции quad и quad8 (а также dblquad), которые самостоятельно вели процесс вычислений, на ходу определяя нужные размеры шагов интегрирования.

Для решения систем обыкновенных дифференциальных уравнений в системе MATLAB также имеются необходимые «решатели». Это функции ode23, ode45, ode113, ode15s, ode23s, ode23t и ode23tb.

Функции с суффиксом s предназначены для решения так называемых *систем жестких дифференциальных уравнений*. Их мы рассмотрим чуть позже, а для всех остальных систем дифференциальных уравнений наиболее употребительной является функция ode45, реализующая алгоритм Рунге – Кутта 4 – 5-го порядка (разные порядки точности используются для контроля шага интегрирования).

Прежде чем применить на практике функцию ode45, опишем ее аргументы, связанные со структурой системы обыкновенных дифференциальных уравнений. Последняя имеет вид

```
y1' = F1( x, y1, y2, …, yn );
y2' = F2( x, y1, y2, …, yn );
…
yn' = Fn( x, y1, y2, …, yn);
```

то есть состоит из n уравнений, разрешенных относительно первых производных функций y1, y2, …, yn (это все функции от x). Введем вектор-столбцы Y и F, состоящие из y1, y2, …, yn и F1, F2, …, Fn соответственно. Тогда система дифференциальных уравнений примет следующий векторный вид:

```
Y' = F( x, Y );
```

Чтобы применить «решатель» ode45, нужно оформить в виде собственной функции правую часть системы уравнений F(x, Y).

Пусть, к примеру, требуется решить следующую систему уравнений:

```
y1' =  y2  + K * x * x;
y2' = -y1;
```

с начальными условиями y1(0) = 0, y2(0) = 1. Здесь K – коэффициент нелинейности задачи. При K = 0 задача становится чисто линейной и описывает гармонические колебания. Если постепенно увеличивать значение коэффициента K и находить соответствующие решения, то можно будет наглядно наблюдать постепенное проявление нелинейного характера колебаний.

Для данного примера неизвестная вектор-функция Y состоит из двух элементов:

```
Y = [ y1, y2 ]
```

Так как функции y1 и y2 вычисляются во многих точках в процессе нахождения решения, то реально y1 и y2 являются вектор-столбцами. Вектор F правых частей системы уравнений для K = 0.01 вычисляем с помощью собственной функции MyDifEq1:

```
function F = MyDifEq1( x, y)
F = [ 0.01 * x * x + y(2); -y(1) ];
```

текст которой записываем в файл MyDifEq1.m. Эта функция вызывается каждый раз, когда требуется вычислить правые части уравнений в конкретной точке x, так что здесь x является скаляром, а вектор y состоит всего из двух элементов.

Другая функция – MyDifEq0:

```
function F = MyDifEq0( x, y )
F = [ y(2); -y(1) ];
```

описывает правые части дифференциальных уравнений при нулевом значении коэффициента K, то есть она соответствует чисто линейному случаю.

Теперь можно вызывать функцию ode45:

```
[ X, Y ] = ode45( 'MyDifEq0',[0,20],[0,1] );
```

находящую решение нашей системы дифференциальных уравнений с начальными условиями [0,1] на отрезке [0,20]. Поскольку здесь использована функция MyDifEq0 для правых частей уравнений, то мы нашли решение для линейного случая гармонических колебаний. Это решение (как y1 = Y(:,1), так и y2 = Y(:,2)) с помощью команд

```
plot( X, Y(:,1) )
hold on
plot( X, Y(:,2) )
```

отображается на следующем графике (см. рис. 4.6).

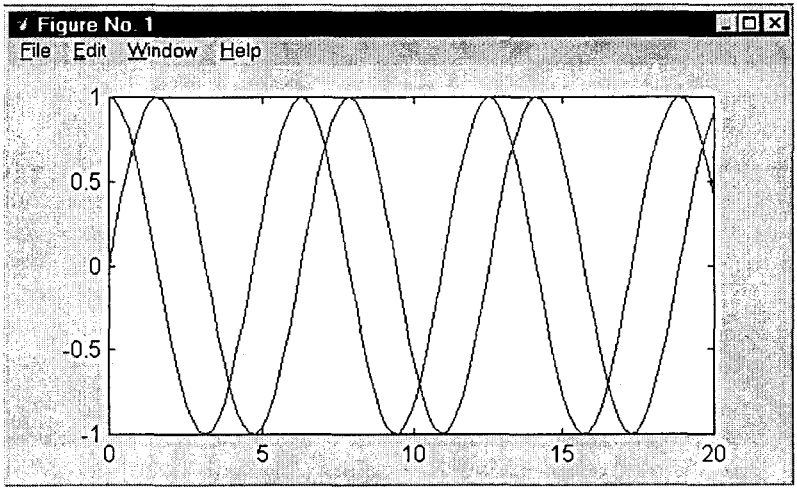


Рисунок 4.6

Затем вызовем «решатель» ode45 уже с функцией MyDifEq1, соответствующей нелинейному случаю, когда K = 0.01:

```
[ X, Y ] = ode45( 'MyDifEq1',[0,20],[0,1] );
```

и отобразим на графике соответствующие ему решения (см. рис. 4.7).

Разница в решениях, соответствующих линейному и нелинейному случаям, видна невооруженным взглядом и не нуждается в дополнительных комментариях. Отметим лишь чрезвычайное удобство и высокую скорость, с которыми в системе MATLAB можно решать дифференциальные уравнения и визуализировать получающиеся результаты.

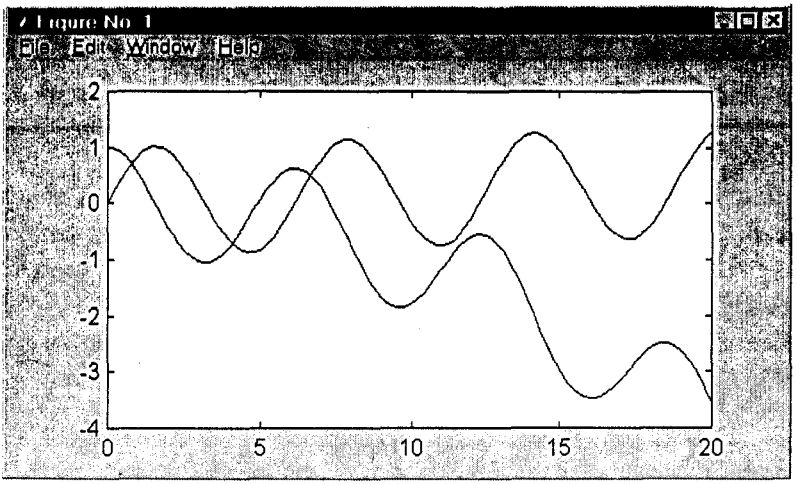


Рисунок 4.7

Теперь перейдем к так называемым *жестким системам дифференциальных уравнений*. Не вдаваясь в точные определения (оставим их для специальных математических пособий), отметим лишь, что это такие системы дифференциальных уравнений, решения которых на разных отрезках изменения независимых переменных ведут себя абсолютно по-разному: в одних местах наблюдается чрезвычайно быстрое изменение зависимых переменных, в то время как в других местах имеет место их сверхмедленная эволюция. Это слишком сложный характер поведения для обычных алгоритмов численного решения дифференциальных уравнений. Поэтому для надежного решения жестких систем уравнений применяют специальные методы. Как мы упомянали выше, в системе MATLAB для этих целей предусмотрены «решатели» с суффиксом s (stiff – жесткий) в их именах.

В электронных книгах, поставляемых с системой MATLAB, приводится пример знаменитых нелинейных дифференциальных уравнений Ван-дер-Поля, которые описывают *нелинейные релаксационные колебания* в различных электронных устройствах. Это очень хороший учебный пример системы жестких дифференциальных уравнений, рассматриваемый в многочисленных математических книгах, так что мы тоже не станем нарушать сложившуюся традицию и воспользуемся этим практическим примером.

Уравнение Ван-дер-Поля имеет следующий вид:

```
y1' =  y2;
y2' = -y1 + K * (1 - y1 * y1) * y2;
```

с начальными условиями `y1(0) = 2, y2(0) = 0`. Здесь `K = 1000` – большой коэффициент нелинейности задачи. Составим функцию `MyVanDerPol`, которая описывает правые части представленных дифференциальных уравнений:

```
function F = MyVanDerPol( x, y )
F = [ y(2); -y(1) + 1000*( 1 - y(1)^2 ) * y(2) ];
```

Для решения дифференциальных уравнений Ван-дер-Поля на отрезке изменения независимой переменной `[0,3000]` используем «решатель» `ode15s`:

```
[X, Y] = ode15s( 'MyVanDerPol', [0,3000], [2,0] );
```

после чего визуализируем решение, а именно первый столбец матрицы **Y**:

```
plot( X, Y(:,1) );
```

Вот как выглядит это решение на рис. 4.8:

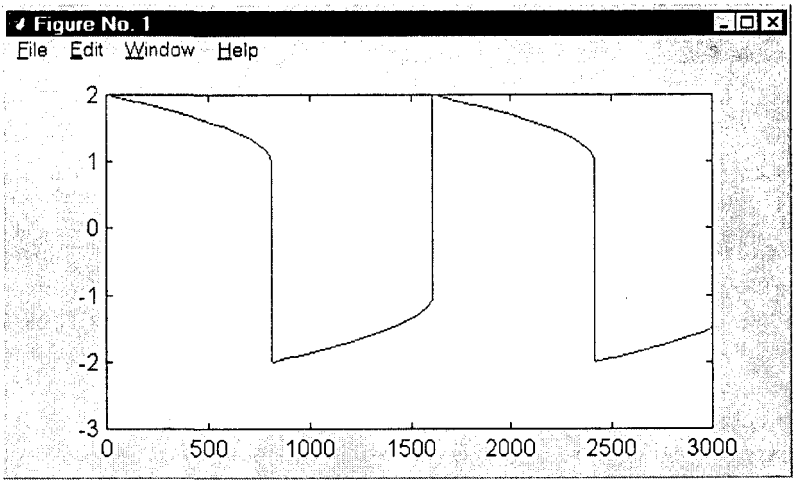


Рисунок 4.8

Релаксационные колебания, описываемые уравнениями Ван-дер-Поля, имеют ярко выраженные фазы относительно медленных эволюций и фазы исключительно быстрых и резких изменений. Однако «решатель» `ode15s` системы MATLAB прекрасно справляется с такой сложной задачей.

Глава 5

Интерактивный режим работы и его автоматизация с помощью сценариев

Сохранение результатов вычислений интерактивного сеанса работы

Все предыдущие главы данного пособия освещали те или иные приемы и методы работы с системой MATLAB в интерактивном режиме. Интерактивный режим – это пользовательский режим ввода с клавиатуры команд и выражений, в результате выполнения которых получаются необходимые числовые результаты, которые можно легко и быстро визуализировать встроенными графическими средствами пакета MATLAB.

Вводимые с клавиатуры выражения подчиняются синтаксису М-языка пакета MATLAB. В соответствии с этим синтаксисом допустимые выражения включают в себя имена переменных, знаки стандартных операций, вызовы предопределенных функций, изначально имеющихся в составе пакета MATLAB. Этих функций имеется громадное количество, причем с их помощью решаются самые разные задачи – от простейших до невероятно сложных, покрывающих целые проблемные области современной математики.

Но в то же время, как бы ни была велика совокупная мощность поставляемых с пакетом MATLAB предопределенных (готовых к применению) функций, всегда может возникнуть практическая ситуация, когда этих функций окажется недостаточно. В этом случае придется разрабатывать собственные функции. В среде пакета MATLAB разрабатывать функции можно на М-языке и на языках C/C++ (есть еще вариант с языком Фортран, но он в данном пособии не рассматривается). Реальная разработка функций на языках C/C++ осуществляется, конечно же, с помощью специальных компиляторов с этих языков. В данном пособии будет рассмотрен компилятор Microsoft Visual C++ 5.0/6.0. В свою очередь, пакет MATLAB предоставляет интерфейс взаимодействия с такими функциями, чтобы их можно было вызывать как из командной строки системы MATLAB, так и из М-функций (функций на М-языке).

Программированию в среде MATLAB посвящена вся 2-я часть данного пособия. А сейчас изучим оставшиеся до сих пор нерассмотренными возможности интерактивного режима работы с системой MATLAB. Таковых еще довольно

много: применение операторов цикла, мультимедийные возможности пакета MATLAB, взаимодействие со сторонними приложениями по традиционным для платформы Windows технологиям, а также вопросы автоматизации интерактивного режима работы с помощью сценариев.

Начнем же с вопроса о сохранении результатов вычислений и визуализации, полученных и накопленных в рабочем пространстве системы MATLAB за один сеанс работы с этой системой.

Напомним, что командой `save` значения всех переменных, хранящихся в рабочей области системы MATLAB (это некоторая область памяти компьютера), сохраняются на диске для долговременного хранения. Обратной командой `load` они могут быть впоследствии (в других сеансах работы с пакетом MATLAB) загружены во вновь созданное рабочее пространство. О конфликте имен, который при этом может возникнуть, мы уже говорили в гл. 1. Теперь же сосредоточим свое внимание на том, что же именно хранится в получающемся при записи командой `save` бинарном MAT-файле.

Пусть, к примеру, в рамках одного сеанса интерактивной работы с пакетом MATLAB были созданы три переменные: `X`, `string1` и `MyCell2`:

```
X = [ 1 2; 3 4]; string1 = 'Hello';
MyCell2 = { [ 1 2 3] };
```

Эти переменные (каждая из них является массивом) имеют соответственно типы `double`, `char` и `cell`. Если выполнить команду

```
save test1.mat
```

то в бинарный файл с именем `test1.mat` (имя нужно указывать вместе с полным путем к нему для записи в каталог, отличный от текущего каталога системы MATLAB) будет записана *вся информация* об этих переменных: их имена, тип переменных, размерность, размер и все элементы этих массивов (то есть, собственно, данные).

Файл `test1.mat` является бинарным, то есть его содержимое невозможно разумно прочитать с помощью текстовых редакторов, да и его просмотр с помощью сторонних редакторов бинарных файлов даст мало пользы без знания формата таких файлов, а он как раз и неизвестен. Однако имеются специальные функции, поставляемые вместе с пакетом MATLAB, предназначенные для чтения информации из MAT-файлов. Эти функции можно вызывать из программ на языках C/C++. Мы во 2-й части данного пособия обязательно вернемся к этому вопросу.

В самом начале нового сеанса работы с пакетом MATLAB рабочее пространство является пустым, так что команда `who` не выдает никакой информации. Если в этот момент выполнить команду загрузки в рабочее пространство содержимого нашего MAT-файла:

```
load test1.mat
```

то в результате переменные X, string1 и MyCell2 восстановятся в полном объеме (см. рис. 5.1).

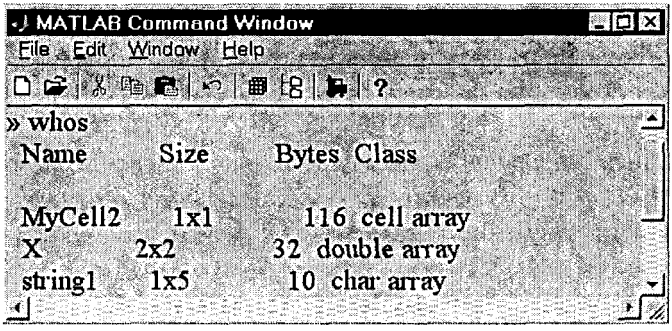


Рисунок 5.1

Легко также убедиться, что сохранились и значения этих переменных:

```
X =
  1 2
  3 4
string1 =
Hello
MyCell2{1} =
  1 2 3
```

Запись переменных в MAT-файл командой save отличается от записи числовой информации файловыми функциями fwrite, fprintf (рассмотренными в гл. 3) в первую очередь тем, что эта команда «знает» все типы данных системы MATLAB, включая комплексные числа, структуры, ячейки, многомерные массивы и т. д. Файловые же функции работают только с вещественными числами и почти «не знают» типов данных системы MATLAB, кроме, разве что, вещественных числовых и символьных матриц.

Попытка заменить команды save и load на работу с изученными в гл. 3 низкоуровневыми файловыми функциями приведет к громадному объему работы, когда потребуется из комплексных чисел извлекать по отдельности вещественную и мнимую части, из структур и ячеек извлекать содержимое (отдельных ячеек или полей) до тех пор, пока не получится некоторый набор вещественных матриц. Для записи последних функции fwrite и fprintf как раз и предназначены. Но после самого процесса записи нужно будет хранить информацию о формате записанного файла, то есть нужно помнить тип файла (бинарный или текстовый), последовательность, в какой записывались в него многочисленные переменные и некоторые другие подробности, составляющие в своей совокупности то, что принято называть частным форматом файла.

Из этого короткого сравнения хорошо видно, что команды save и load являются сильным средством, автоматизирующим работу по сохранению перемен-

ных рабочей области, накопленных за сеанс интерактивной работы с пакетом MATLAB.

Помимо переменных рабочей области может потребоваться сохранить текстовое содержимое всего командного окна системы MATLAB, где показываются все вводимые команды и выражения, а также результаты вычислений. Это легко можно сделать сразу двумя способами.

Первый способ предполагает выделение содержимого этого окна командой меню `Edit|Select All` (или мышью), после чего исполняется команда `Copy`, так что текстовое содержимое командного окна системы MATLAB копируется в буфер обмена операционной системы Windows, откуда оно может быть вставлено в документ любого текстового редактора командой `Paste`. На рис. 5.2 показано окно простого текстового редактора Notepad с содержимым командного окна системы MATLAB.

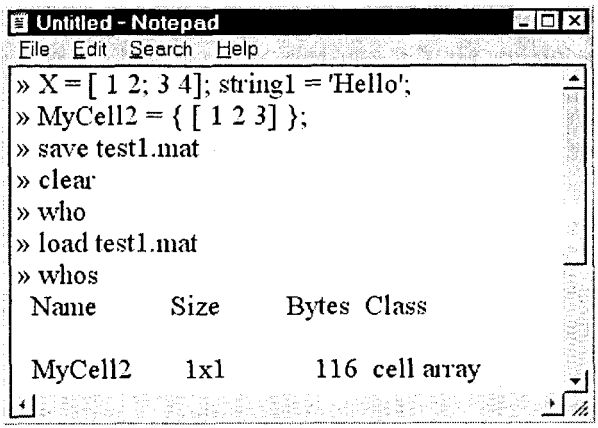


Рисунок 5.2

Второй способ (может быть, даже менее удобный) заключается в том, что вводится и исполняется специальная команда, после которой запись в текстовый файл содержимого командного окна осуществляется самой системой MATLAB автоматически. Вот пример такой команды:

```
diary MyMonitorFile1.txt
```

После выполнения этой команды весь вывод в командное окно дублируется в текстовый файл `MyMonitorFile1.txt`. Процесс копирования прекращается командой `diary off`. Содержимое такого файла не отличается от содержимого текстового файла, в который мы выше копировали информацию из буфера обмена.

Часто требуется сохранить содержимое командного окна системы MATLAB в виде «твердой копии» на бумаге, то есть распечатать его на принтере. Эту задачу также можно выполнить многими способами. В самом командном окне есть команда меню `File|Print...`, позволяющая быстро решить эту задачу. Бо-

лее качественных результатов можно добиться, если скопировать содержимое командного окна через буфер обмена в текстовый редактор Microsoft Word, отформатировать его там качественным образом (шрифты, абзацы, выравнивание, цвет и т. д.), после чего распечатать на принтере средствами самого текстового редактора.

Копирование содержимого графических окон системы MATLAB через буфер обмена в редактор Microsoft Word также является наилучшим способом поместить графические иллюстрации в текстовые документы и осуществить печать средствами текстового редактора. Как известно, копирование из графического окна в буфер обмена осуществляется с помощью меню графического окна командой `Edit|Copy Figure`. Однако и в самом графическом окне есть средства для печати графики на принтере – команда меню `File|Print...` Важно помнить, что перед печатью из графического окна и перед копированием в буфер обмена нужно осуществить масштабирование картинки (изменением размеров самого графического окна), иначе потом всякое изменение размера изображения будет сопровождаться ухудшением его качества.

Запись содержимого графических окон системы MATLAB в специализированные графические файлы ранее подробно обсуждалась в гл. 2, где также подчеркивалось, что перевод фактически векторного графического изображения в растровое функцией `capture` нарушает возможность дальнейшего масштабирования изображения без потери качества.

Однако существует возможность сохранить всю информацию о графических объектах системы MATLAB, созданных за текущий сеанс работы с этой системой, в MAT-файле обычной командой `save`. Предварительно нужно сохранить в переменных рабочей области все характеристики (числовые и текстовые) всех открытых графических объектов, что потребует введения для этой цели многих десятков переменных. Это, может быть, и не слишком сложная с принципиальной точки зрения задача, но выполнить ее интерактивно, то есть вводя все команды и выражения последовательно с клавиатуры, практически невозможно. Для этой цели существует специальное средство, позволяющее автоматизировать интерактивный режим работы с системой MATLAB. Этим средством являются *сценарии* – специальные текстовые файлы, в которые предварительно (один раз) записываются все необходимые команды. Затем эти команды выполняются в *автоматическом режиме* без непосредственного участия пользователя, от которого требуется лишь ввести в командной строке имя файла. Сценарии будут подробно рассмотрены ниже в специально для этого предназначенном подразделе.

Операторы цикла. Векторизация как альтернатива циклам

Нам до сих пор удавалось обходиться без специальных *конструкций управления*, позволяющих осуществлять *циклический повтор кода* на М-языке. Это объясняется уникальными особенностями М-языка системы MATLAB – в нем присутствуют многочисленные компактные операции, направленные на массовую обработку массивов произвольной размерности и размеров.

В любых традиционных языках программирования упомянутые выше конструкции управления принято называть *операторами цикла*. Есть операторы цикла и в М-языке системы MATLAB.

Операторы цикла задаются с помощью специальных *зарезервированных ключевых слов*. Их широко применяют в текстах программ на М-языке, с чем мы еще столкнемся во 2-й части настоящего пособия, посвященной различным аспектам программирования в среде MATLAB. Сейчас же сообщим, что их можно использовать и непосредственно в интерактивном режиме, когда все элементы оператора цикла постепенно вводятся с клавиатуры. При этом система MATLAB надежно распознает ситуацию и выполняет цикл только после ввода завершающего ключевого слова `end`.

В зависимости от способа определения *условия останова* (окончания циклических повторений) различают два вида операторов цикла в М-языке системы MATLAB. Первый из них использует ключевые слова `while` («пока») и `end`. Он имеет вид

```
while выражение
...
end
```

Здесь повтор участка кода, обозначенного многоточием, продолжается все время, пока выражение «истинно» (не равно нулю). В случае массивов истинность наступает, когда все элементы массива истинны.

Рассмотрим для примера следующий фрагмент, который вычисляет сумму отрезка ряда:

```
S = 0; k = 1; u = 1;
while u > 1e-8
S = S + u;
k = k + 1; u = 1/k^2;
end
```

Здесь условием останова служит требование к слагаемым быть больше некоторого заранее определенного числа: как только очередное слагаемое станет меньше этого числа, выражение, стоящее после ключевого слова `while`, станет ложным и суммирование прекратится.

Приведенный выше фрагмент кода, в котором оператор цикла занимает целых четыре физических строки, вводится с клавиатуры последовательно строка за строкой, причем клавиша Enter нажимается каждый раз после ввода очередной строки. На рис. 5.3 показано командное окно системы MATLAB после ввода первой строки и нажатия клавиши Enter.

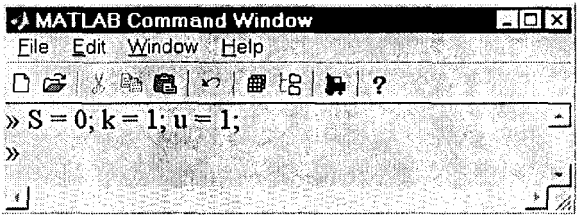


Рисунок 5.3

Из этого рисунка прекрасно видно, что после ввода первой строки произошло выполнение всех инструкций из нее, то есть были выполнены три присваивания, после чего система MATLAB перешла в режим ожидания ввода новых команд. Об этом говорит знак >>, стоящий в начале новой строки ввода. А теперь посмотрим на командное окно после ввода второй строки, в которой расположен *заголовок оператора цикла* (см. рис. 5.4).

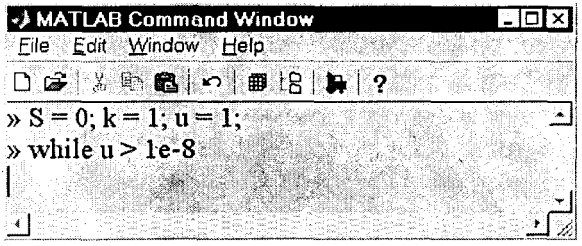


Рисунок 5.4

После ввода заголовка оператора цикла и нажатия клавиши Enter *текстовый курсор* переместился на следующую физическую строку, но знак >> не появился. Это и означает, что система MATLAB распознала ситуацию и ожидает, что сейчас с клавиатуры будут последовательно вводиться инструкции, составляющие *тело цикла*. Эти инструкции могут занимать несколько физических строк командного окна. Текстовый курсор перемещается на следующую строку после нажатия клавиши Enter, и только после ввода ключевого слова end система MATLAB приступает к циклически повторяющемуся выполнению инструкций из тела цикла. При этом *курсор мыши* принимает форму *песочных часов*. По завершении этого процесса (его длительность зависит от сложности и суммарного объема вычислений) на следующей за циклом строке появляется знак >>, что мы и видим на рис. 5.5.

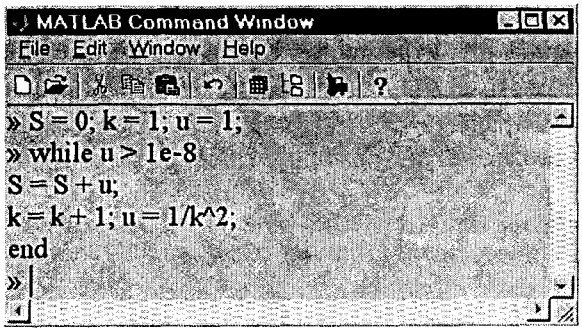


Рисунок 5.5

Итак, система MATLAB уверенно распознает две фазы: *фазу ввода оператора цикла* и *фазу его выполнения*. Однако можно и совместить обе эти фазы, введя весь цикл в одну строку ввода (как мы знаем еще из 1-й главы строка ввода может быть продолжена с помощью многоточия на последующие физические строки командного окна, но в сумме должна содержать не более 256 символов).

В этом случае заголовок цикла нужно отделить от тела цикла запятой (или точкой с запятой, что безразлично). Специальных усилий для отделения последней инструкции тела цикла от ключевого слова `end` часто не требуется, так как инструкции тела цикла почти всегда заканчиваются точкой с запятой по соображениям подавления вывода промежуточных результатов вычислений в командное окно. Если этого не сделать, то получится неприятное для глаз мелькание содержимого командного окна и грандиозное увеличение времени выполнения цикла.

Вот пример записи предыдущего оператора цикла в форме ввода в одну *логическую строку ввода* (см. рис. 5.6).

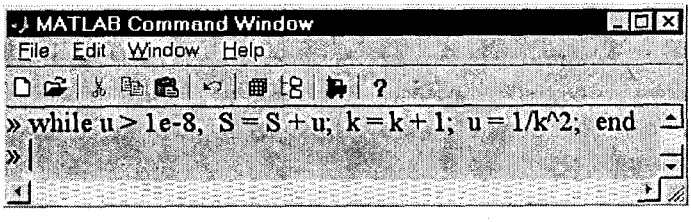


Рисунок 5.6

Второй вариант оператора цикла в М-языке системы MATLAB использует ключевые слова `for` («для») и `end`. Он имеет вид:

```
for varName = выражение
...
end
```

где `varName` – произвольно выбираемое программистом имя так называемой *переменной цикла*. В отличие от первого варианта оператора цикла здесь легко прогнозировать количество итераций (повторов), так как обозначенное многото-

чием тело цикла выполняется для всех возможных значений переменной varName. Набор возможных значений для переменной цикла поставляет выражение, которое стоит после ключевого слова for. Чаще всего это выражение представлено с помощью изученной ранее в гл. 1 операции *диапазон значений*. В следующем фрагменте кода осуществляется сложение 57 членов ряда:

```
S = 0;
for k = 1 : 1 : 57
S = S + 1/k^2;
end
```

При каждом новом проходе переменная цикла k увеличивается на единицу. Как легко заметить, здесь осуществляется суммирование того же ряда, что и в примере, посвященном оператору цикла while...end. В предыдущем примере условием останова было требование к величине очередного слагаемого, а теперь этим условием является исчерпание всех возможных значений переменной цикла. В итоге можно сделать вывод, что использование того или иного варианта оператора цикла диктуется особенностями конкретной математической задачи.

Вместо операции задания диапазона можно явно указать весь набор возможных значений в виде вектор-строки, например:

```
for m = [ 2, 5, 7, 8, 11, 23 ]
```

что приведет к шести итерациям. При первой итерации переменная цикла m будет равна 2, при второй – 5 и т. д. до исчерпания всех возможных значений.

Достаточно необычным может показаться использование матриц в управляющем выражении:

```
A = [ 1 2; 3 4];
for k = A
```

Такой цикл будет повторяться ровно столько раз, сколько столбцов в матрице A, то есть два раза для данного конкретного случая. В каждой итерации переменная цикла принимает значение очередного столбца матрицы, то есть является вектор-столбцом. Например, фрагмент

```
S = 0; a = [ 1 2; 3 4];
for k = A
S = S + sqrt( k(1)^2 + k(2)^2 );
end
```

вычисляет сумму «длин» столбцов матрицы A.

Оба вида операторов цикла можно досрочно прервать, если написать ключевое слово break внутри тела цикла. Это позволяет размещать фактическое условие останова цикла внутри тела цикла, а не в его заголовке. В этом случае в заголовке цикла можно писать выражения вида while 1 (всегда истинное выражение).

В самом начале данного подраздела, посвященного операторам цикла, мы сказали, что в системе MATLAB очень часто можно отказаться от использования этих операторов, применив вместо них компактные операции М-языка, направленные на массовую обработку массивов. Сейчас усилим это высказывание: везде, где это возможно, *вместо операторов цикла лучше применять эквивалентные по результатам операции с массивами*, так как последние исполняются в системе MATLAB намного быстрее.

Например, вместо фрагмента кода с оператором цикла

```
k = 0;
for x = 0 : 0.1 :100
 k = k + 1; y( k )=cos( x );
end
```

лучше использовать операции с массивами:

```
x = 0 : 0.1 : 100; y = cos( x );
```

так как они исполняются быстрее и записываются короче.

Замену операторов цикла эквивалентными им по результату массовыми операциями М-языка принято называть *векторизацией* кода.

Анимация и звук в системе MATLAB

В гл. 2 мы изучали графические возможности системы MATLAB по изображению статических (неизменных во времени) графиков функций, как двумерных, так и трехмерных. Также были рассмотрены вопросы, связанные с показом произвольных растровых изображений. А теперь мы рассмотрим возможности пакета MATLAB по показу движущихся (меняющихся во времени) изображений, что важно для визуализации динамических данных и для всестороннего изучения 3D-объектов.

При этом никаких сложных специализированных программ не потребуется. Задача полностью решается встроенными средствами системы MATLAB, то есть легко выполнима в интерактивном сеансе (или нескольких сеансах) работы.

Существуют два основных способа показа анимационных изображений. Первый способ заключается в том, что мы в цикле выводим меняющиеся картинки в графический объект `axes`, свойства которого (через свойства его дочерних объектов) устанавливаются специальным образом для достижения приемлемого качества *перерисовки* содержимого графического окна системы MATLAB. Второй способ состоит в предварительной подготовке отдельных *кадров*, которые запасаются для дальнейшего показа в некотором числовом массиве. Сам же последовательный покадровый показ (так и хочется сказать «кинофильм») осуществляется специальной функцией `movie` (переводится как «кино»).

Начнем изложение с первого способа. Для примера в графическом окне системы MATLAB продемонстрируем равномерное движение по окружности материального тела в виде бусинки. Сначала выполним подготовительный фрагмент кода:

```
x  = -1 : 0.01 : 1;
y1 = sqrt(1-x.*x);
y2 = -y1;
plot(x,y1,'k',x,y2,'k');
axis square;
hold on
```

где рисуется черным цветом круговая траектория, по которой и будет двигаться бусинка. Команда `hold on` нужна для того, чтобы весь последующий графический вывод осуществлялся в то же самое графическое окно.

Теперь проставляем бусинку в ее начальном положении:

```
x = 1; y = 0;
h = plot(x, y, '.r');
set(h,'EraseMode','xor','MarkerSize',18);
```

«Бусинка» представляет собой тоже график функции, являющийся прямой линией, то есть графическим объектом типа `line`, состоящим из единственной точки красного цвета. Далее мы динамически будем изменять координаты точки, относящейся к графическому объекту типа `line` с описателем `h`. Система MATLAB будет при этом стирать предыдущие изображения этого графического объекта и рисовать его новые изображения. Чтобы такая перерисовка была более «гладкой» (хотя все равно она не получается идеальной), нужно для объекта типа `line` установить свойство `'EraseMode'` (способ стирания) равным значению `'xor'` (производится операция «исключающее ИЛИ» над объектом и подложкой). Иначе перерисовка будет просто ужасной – слишком медленной при интенсивном «моргании» всего графического окна.

Наконец осуществляем циклическое продвижение бусинки и ее перерисовку:

```
t=0; dt=0.01; a=0.5;
while 1, t=t+dt; x=cos(a*t); y=sin(a*t); …
set(h,'XData',x,'YData',y); end
```

Для остановки «бесконечного» цикла (условие для циклического повтора всегда истинно) нужно воспользоваться комбинациями клавиш Ctrl+C или Ctrl+Break.

С помощью представленного выше кода достигается относительно приемлемое изображение движения бусинки, в первую очередь из-за того, что объем перерисовки бусинки по сравнению со всей площадью объекта `axes` невелик. Если бы требовался большой объем перерисовки, то представленный способ не мог бы достичь приемлемой скорости.

В таких случаях используют принципиально другой способ, при котором все промежуточные кадры рассчитываются и запоминаются заранее, после чего «проигрываются» специальной «кинематографической» функцией movie. С помощью этого способа изобразим «броуновское дрожание» 200 точек в рамках заданного прямоугольника, причем хаотичность движения точек («атомов») будет обеспечиваться функциями rand и randn, порождающими соответственно равномерно распределенные и нормально распределенные случайные числа.

Подготовим графическое окно для показа 100 кадров (Frames) «кинофильма» про броуновское движение 200 атомов, среднюю скорость которых характеризуем параметром v:

```
nA = 200; nFrames = 100; v = 0.03;
x = rand(nA,1) - 0.5;
y = rand(nA,1) - 0.5;
h = plot(x,y,'.');
set(h,'MarkerSize',10);
axis([-1 1 -1 1]); axis square;
Buf = moviein( nFrames );
```

В последней строке с помощью функции moviein создается гигантский буфер Buf в памяти компьютера под все кадры кинофильма:

```
size( Buf )
ans =
   24920  100
```

Здесь под каждый из 100 кадров (100 столбцов массива Buf типа double) отводится 24 920 позиций (строк массива Buf). Последнее число зависит от текущего размера графического окна и является переменным; так, размер этого окна в любой момент можно изменить с помощью мыши. В данном конкретном случае массив Buf состоит из 2 492 000 элементов и занимает в памяти компьютера 19 936 000 байт (около 20 Мбайт).

Теперь рассчитываем каждый кадр и «захватываем» его в буфер Buf (по одному кадру на каждый столбец массива Buf) специальной функцией getframe:

```
for k=1:nFrames, x=x+v*randn(nA,1);y=y+v*randn(nA,1); …
set(h,'XData',x,'YData',y); Buf(:,k) = getframe; end
```

Окончательно прокручиваем «снятый ролик» с помощью функции movie 5 раз подряд:

```
movie( Buf, 5 );
```

Как только отпадает необходимость в хранении кинофильма, сразу же освобождаем память в рабочем пространстве системы MATLAB, занятую под буфер Buf:

```
clear Buf
```

Это нужно сделать как можно быстрее, поскольку обычно видеобуферы имеют гигантские размеры и зря расходуют драгоценные ресурсы компьютера.

Отдельные кадры подготовленного кинофильма можно скопировать функцией `frame2im` в два числовых массива: в массив пикселов и таблицу цветов (эти массивы были подробно рассмотрены в гл. 2). Следующий фрагмент запоминает k-й кадр в массиве пикселов `X` и таблице цветов `map`:

```
[X, map] = frame2im( Buf( : , k ) );
```

В гл. 2 описано, как можно массивы `X` и `map` записать в файлы различных форматов для длительного хранения на диске.

В то же время является актуальной и обратная задача. Из файла информация читается в массив пикселов `X1` и таблицу цветов `map`, после чего функцией `im2frame` они монтируются как кадр фильма системы MATLAB:

```
Buf = moviein( nFrames );
Buf( : ,1) = im2frame( X1, map);
```

Здесь, однако, следует отметить два важных обстоятельства. Во-первых, функция `im2frame` работает только с индексированными изображениями (см. гл. 2) и не работает с изображениями типа `truecolor` (что, конечно, вызывает сожаление). Если у вас имеется именно изображение `truecolor`, то его нужно будет самостоятельно преобразовать в индексированное изображение. Во-вторых, зная размер изображения в пикселах, следует предварительно (перед вызовом функции `moviein`) установить размер графического окна системы MATLAB равным этому размеру (как это сделать, рассказано в гл. 2).

Затем читается содержимое другого графического файла и оно монтируется уже в следующий кадр фильма `Buf`:

```
Buf( : ,2) = im2frame( X2, map);
```

и так далее вплоть до последнего кадра с номером `nFrames`, после чего вызывается функция `movie`:

```
movie( Buf )
```

В формате вызова этой функции с единственным аргументом прокрутка видеоролика осуществляется один раз, так что нет специальной необходимости вызывать функцию `movie` с двумя аргументами, то есть `movie(Buf,1)`.

Мультимедийные способности пакета MATLAB базируются как на анимации, рассмотренной выше, так и на возможности *воспроизводить звук* (если на компьютере присутствует соответствующее оборудование) стандартного для платформы Windows формата `WAVE`. Соответствующие файлы имеют, как известно, расширение `wav`. Система MATLAB позволяет читать и записывать файлы этого формата. Чтение осуществляется с помощью функции `wavread`, а запись – с помощью функции `wavwrite`. Наконец, функция `sound` (а также почти аналогичная ей функция `soundsc`) осуществляет реальное воспроизведение зву-

ка, получая в качестве аргумента вещественный вектор, содержащий последовательность измерений громкости звука. Такая последовательность чисел характеризуется *частотой дискретизации* (для формата CD Audio – 44,1 кГц) и *разрядностью отсчетов* (для CD Audio это 16 бит). В случае *стереозвука* функция `sound` получает в качестве аргумента матрицу `m x 2`, в которой каждый столбец содержит данные для одного из стереоканалов.

Проиллюстрируем применение перечисленных функций. Допустим, на диске имеется записанный стандартными для платформы Windows средствами стереофонический звуковой файл `'sound123.wav'`, причем его качество соответствует стандарту CD Audio. Считываем звуковую информацию из этого файла в состоящую из двух столбцов числовую матрицу `V`

```
[V,f,b] = wavread('sound123.wav');
```

а в скалярную переменную `f` при этом читается частота дискретизации, в переменную `b` – разрядность. Так как в файл `'sound123.wav'` звуковая информация была записана с качеством CD Audio, то переменная `f` должна принять значение 44 100 (герц), а переменная `b` должна быть равна 16 (бит на один отсчет). Проверяем и убеждаемся, что это действительно имеет место (см. рис. 5.7).

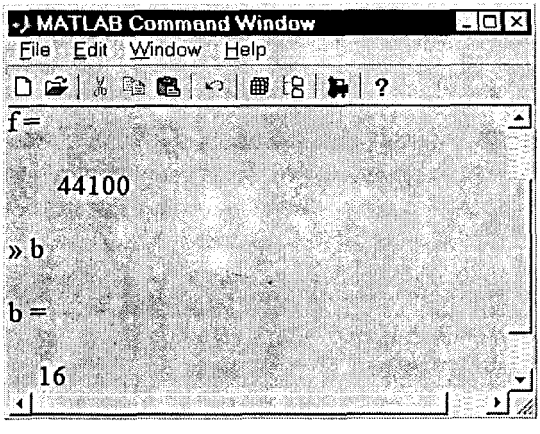


Рисунок 5.7

Теперь можно воспроизвести звук:

```
sound( V, f, b )
```

указав как сам поток звуковых отсчетов в виде числового массива (матрицы `m x 2`) `V`, так и его характеристики в виде частоты дискретизации и разрядности отсчетов.

Каких-либо встроенных средств по разумному автоматическому преобразованию массива звуковых отсчетов `V` в ядре системы MATLAB нет, но они всегда могут найтись среди многочисленных пакетов расширения этой системы.

В любом случае нам никто не запрещает попробовать свои собственные силы в этом деле. В качестве простейшего примера посмотрим, что получится, если осуществить небольшое *зашумление* звукового сигнала:

```
[m,n] = size( V ); s = 0.05; V1 = zeros( m,1 );
for k=1:m, V1(k) = V(k,1)+s*randn(1); end
```

Снова вызываем функцию sound и «слушаем вектор» V1, у которого имеется только один столбец, в который занесен зашумленный звук, полученный из первого стереоканала исходного звука. Заранее было ясно, что качество звука (в музыкальном смысле) от таких преобразований может только понизиться.

Чтобы сохранить результаты любых экспериментов со звуком, нужно записать информацию в файл, для чего следует применить предназначенную для этого функцию wavwrite. В частности, наш эксперимент с зашумлением сохраним в файле 'sound124.wav':

```
wavwrite( V1, f, b, 'sound124.wav' );
```

Здесь первым аргументом является матрица (для монофонического звука это вектор) звуковых отсчетов, вторым – частота дискретизации, третьим – разрядность отсчетов, а последним – имя файла.

Разумеется, надо отдавать себе ясный отчет в цене таких «мультимедийных удобств». Несжатые стереофонические звуковые файлы высокого качества (CD Audio) занимают гигантские объемы памяти: 1 минута звучания требует около 10 Мбайт.

Сценарии и М-файлы

В предыдущем подразделе, посвященном анимации, приходилось вводить с клавиатуры в командное окно системы MATLAB довольно большие фрагменты кода на М-языке. Если потребуется осуществить анимацию (или другую столь же объемную задачу) в ином сеансе работы с пакетом MATLAB, то придется повторить весь ввод с клавиатуры. Повторный ввод большого набора инструкций не только утомителен, но и чреват возникновением ошибок, которые нужно будет преодолевать. Ясно, что в рамках пакета MATLAB крайне желателен инструмент для запоминания полезных фрагментов кода изрядного размера, чтобы иметь возможность избежать его повторного ввода. Такой инструмент есть и называется он *сценарием* работы с пакетом MATLAB.

Под сценарием понимают текстовый файл, содержащий инструкции на М-языке, подлежащие исполнению в автоматическом пакетном режиме. Этот файл может иметь произвольное имя, но расширение имени фиксированно – оно состоит из одной буквы m. Любой текстовый файл, содержащий произвольный код на М-языке и имеющий оговоренное выше расширение, принято называть *М-файлом*.

Создать такой файл можно с помощью любого текстового редактора, но система MATLAB располагает собственным текстовым редактором, окно которого показано на рис. 5.8.

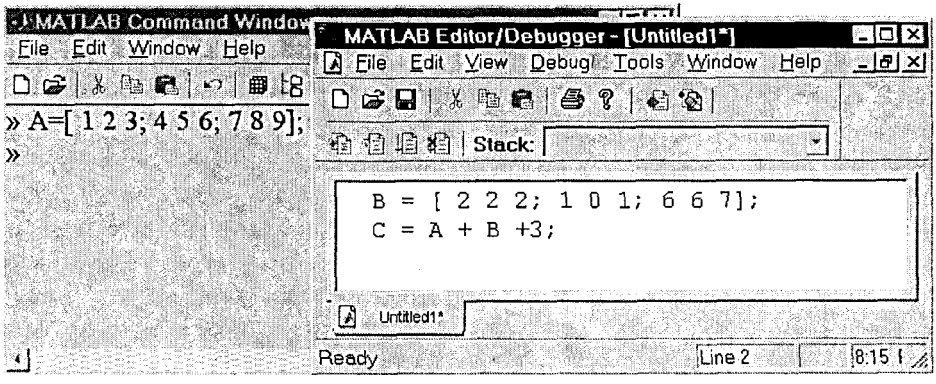


Рисунок 5.8

Этот редактор обеспечивает такие дополнительные удобные визуальные средства, как, например, цветовое выделение ключевых слов М-языка. Так что лучше всего пользоваться именно этим редактором. Он вызывается из командного окна системы MATLAB командой меню `File|New|M-file` (или самой левой кнопкой на полосе инструментов, на которой изображен чистый белый лист бумаги) и работает в своем собственном окне.

Как могли заметить внимательные читатели, в тексте сценария на показанном выше рисунке используется переменная `A`, которая определяется в командном окне системы MATLAB. Это можно делать потому, что переменные, *определяемые в командном окне*, и переменные, *определяемые в сценариях*, составляют *единое рабочее пространство* системы MATLAB. Ниже мы еще вернемся к этому важному вопросу.

После создания текста сценария его надо сохранить в М-файле на диске. М-файлы не надо путать с MAT-файлами, изученными ранее в гл. 1 и в которых хранятся переменные из рабочего пространства системы MATLAB.

Каталог на диске для хранения М-файлов может быть любым, но путь к этому каталогу обязательно должен быть известен системе MATLAB. Пакет MATLAB хранит сведения обо всех известных ему каталогах. Для создания нового каталога нужно сначала выполнить команду меню (командного окна) `File|Set Path...`, с помощью которой вызывается диалоговое окно с именем `Path Browser` (просмотрщик путей доступа к файлам) (см. рис. 5.9).

В этом окне показывается список всех зарегистрированных в системе MATLAB путей доступа. Для добавления нового каталога в список путей доступа служит следующая команда меню этого окна:

```
Path | Add to path..
```

которая, в свою очередь, порождает еще целый ряд диалоговых окон, работа с которыми практически очевидна.

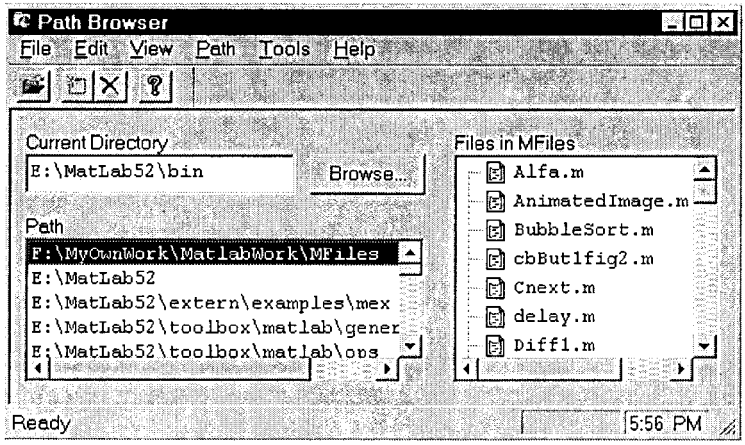


Рисунок 5.9

После того как мы добавили в список путей доступа новый каталог, в нем можно сохранить файл с созданным в редакторе сценарием. Чтобы задействовать код, расположенный в М-файле, нужно ввести в командном окне имя М-файла, содержащего сценарий работы (*явно указывать при этом расширение файла нельзя!!!* – возникнет ошибочная ситуация), и нажать клавишу Enter.

Еще раз напомним, что сценарий обрабатывает как свои собственные переменные, так и переменные, определенные до вызова сценария в командном окне системы MATLAB и хранящиеся в ее рабочем пространстве. С другой стороны, все переменные, созданные во время работы сценария, остаются в рабочем пространстве системы MATLAB и после окончания выполнения сценария.

Это свойство является одновременно и сильной, и слабой чертой сценариев. Сильной чертой является *полная интеграция с рабочим пространством*, что и делает сценарии средством автоматизации интерактивного сеанса работы пользователя с пакетом MATLAB. Эта же интеграция является слабой чертой, если предполагается оформить в виде сценария отдельный фрагмент решения некоторой задачи, который следует всегда вызывать в одной и той же программной обстановке, а это невозможно в силу переменной программной обстановки в рабочем пространстве системы MATLAB. Кроме того, сценарию нельзя в момент его вызова передать в виде параметров дополнительную информацию, не содержащуюся в рабочем пространстве.

Перечисленные слабые стороны сценариев преодолеваются с помощью М-функций, которые будут изучаться в следующей главе. Сейчас же отметим, что М-функции, в свою очередь, плохо приспособлены для нужд автоматизации интерактивного сеанса работы, так как их пространство переменных полностью изолировано от рабочего пространства системы MATLAB.

Одним из типовых случаев применения сценариев является запись в М-файл текста, скопированного из командного окна (или сохраненного в текстовом файле командой diary). Если среди содержимого такого файла будет некоторое количество ненужных или ошибочно записанных инструкций, их следует удалить из текста файла, после чего можно вызывать полученный таким образом сценарий на выполнение. В результате в рабочем пространстве системы MATLAB будут восстановлены все переменные предыдущего сеанса работы, а кроме них будут *воссозданы все графические объекты* (если они были ранее созданы). В этом отношении такие сценарии превосходят даже пару команд save - load, так как с помощью последних невозможно восстановить графические объекты.

При автоматизации выполнения анимационных работ следует учесть некоторые тонкие нюансы. Например, ниже показан текст сценария, в который мы поместили все команды для решения задачи о движении бусинки по окружности, рассмотренные ранее в предыдущем подразделе и вводившиеся с клавиатуры в интерактивном режиме (ввод каждой строки инструкций заканчивался нажатием клавиши Enter):

```
% -- Animation script  -------
x = -1 : 0.01 : 1;
y1 = sqrt(1 - x.*x);
y2 = -y1;
plot(x,y1,'k',x,y2,'k');
axis square;
hold on
%-----------------------------
x = 1; y = 0;
h = plot(x, y, '.r');
set(h,'EraseMode','xor','MarkerSize',18);
%-----------------------------
t=0; dt=0.01; a=0.5;
while 1
   t=t+dt;
   x=cos(a*t); y=sin(a*t);
   set(h,'Xdata',x,'Ydata',y);
   drawnow
end
%- Rem -> To STOP, press Ctrl-C or Ctrl-Break
```

В этот текст мы добавили несколько строк *комментариев* (часть из них – в виде штриховой разметки текста для улучшения зрительного разделения функционально разных частей; любой комментарий начинается со знака %), изменили пространственное расположение кода (теперь нет никакой необходимо-

сти записывать все элементы цикла в одну строку), но самое главное, мы добавили одну важную команду – команду drawnow.

Эта команда совсем не нужна в интерактивном режиме, когда команды вводятся по очереди с клавиатуры и их ввод заканчивается нажатием клавиши Enter. А теперь, когда они расположены в тексте сценария, они исполняются в автоматическом режиме и никаких дополнительных сигналов от нажатия клавиши Enter в систему MATLAB не поступает. Это приводит к тому, что без команды drawnow динамической перерисовки графического окна не происходит и нет никакой анимации. Это и есть тот самый нюанс в использовании сценариев, о котором мы упоминали выше. На это следует обратить особое внимание и не забывать добавлять команду drawnow, означающую посылку системе MATLAB специального сигнала с требованием немедленной перерисовки графического окна.

Аналитические вычисления с помощью пакета расширения Symbolic Math Toolbox

Сколь широки ни были бы возможности *ядра* системы MATLAB по предоставлению готовых услуг в области вычислений и компьютерной графики, всегда найдутся новые частные проблемы и целые предметные области, которые остались неохваченными.

Для гибкого реагирования на такие ситуации архитектура пакета MATLAB построена таким образом, что имеется очень удобная техника встраивания в общую систему новых программных решений. Причем это встраивание происходит *прозрачным* (незаметным) для пользователя образом. После инсталляции *пакета расширения* (по-английски – *Toolbox*) функциональные возможности этого пакета как бы сливаются с базовыми возможностями ядра системы MATLAB и ими становится возможным пользоваться абсолютно традиционными способами.

Пакеты расширения для системы MATLAB производят как *фирма-изготовитель* самого пакета MATLAB (The MathWorks Inc., USA), так и *сторонние производители*. Изготовить такой пакет с точки зрения техники программирования не представляет никаких принципиальных затруднений, о чем и будет рассказано в настоящем пособии далее в главах, посвященных программированию. Однако истинная ценность любого пакета расширения определяется, в первую очередь, актуальностью и сложностью решаемых с его помощью задач. С этой точки зрения изготовить пакет расширения для системы MATLAB так, чтобы он заслужил внимание серьезных пользователей, крайне сложно. Почти наверняка такое решение должно быть следствием глубоких научно-технических и программных разработок в некоторой важной и интересной предметной области.

В настоящее время существуют десятки официально распространяемых пакетов расширения, среди которых существуют специальные математические пакеты для решения систем дифференциальных уравнений с частными производными (пакет Partial Differential Equations Toolbox), для решения задач статистики (Statistics Toolbox) и оптимизационных задач (Optimization Toolbox), для решения задач обработки изображений (Image Processing Toolbox; Wavelet Toolbox), для решения задач нечеткой математики (Fuzzy Logic Toolbox) и многих других задач. Существуют инженерные пакеты для моделирования различных динамических систем, электрических цепей и сигналов (Simulink; Control System Toolbox; Signal Processing Toolbox). Даже имеются пакеты расширения, посвященные экономике (Financial Toolbox).

Данное пособие *не рассматривает* всех пакетов расширения для системы MATLAB. Оно посвящено всестороннему (в связи с ограниченным объемом издания – не исчерпывающему) изложению готовых возможностей ядра системы MATLAB, а также многочисленным вопросам программирования в рамках этой системы, и даже вне рамок системы MATLAB, но в связи с последней. Если некоторый пакет расширения для системы MATLAB вызывает повышенный интерес со стороны пользователей, то его детальному описанию должно быть посвящено отдельное пособие. В любом случае каждый «уважающий себя» и пользователей пакет расширения должен поставляться вместе с электронной подсистемой помощи, из которой всегда можно узнать основные сведения о пакете и о специфических (если таковые имеются) приемах работы с ним.

Однако мы рассмотрим основные возможности пакета расширения Symbolic Math Toolbox (это его официальное название), позволяющего в рамках системы MATLAB осуществлять *аналитические вычисления и аналитические преобразования* выражений. Этот пакет лицензирован у фирмы Waterloo Maple Software, Canada, и включен в состав дистрибутива пакета MATLAB на правах пакета расширения. Он столь органично дополняет и расширяет возможности ядра системы MATLAB, что пропустить рассказ об этом пакете расширения было бы неверно. Возможности пакета Symbolic Math Toolbox поистине гигантские, поэтому мы изучим лишь основные и наиболее характерные из них. Пакет снабжен встроенной системой электронной помощи, что в принципе позволяет получить технические сведения по всем аспектам работы с ним.

Помимо выполнения *аналитических преобразований* пакет Symbolic Math Toolbox позволяет выполнить *арифметические вычисления с контролируемой точностью*, которую можно заказать заранее.

Начнем с краткого рассказа про вычисления с контролируемой точностью. Пакет расширения Symbolic Math Toolbox предоставляет для этой цели две функции – `digits` и `vpa`. Первая из этих функций устанавливает требуемую точность в количестве верных десятичных знаков после запятой, а вторая функция осуществляет вычисления с заданной точностью. Аббревиатура `vpa` означает Variable Precision Arithmetic, что переводится как «арифметика с переменной точностью».

Вот пример, иллюстрирующий работу этих функций:

```
digits( 30 );
vpa( pi )
ans =
3.14159265358979323846264338328
```

в котром мы очень просто получили прекрасный результат – число pi с 30 значащими знаками. Так же легко (для нас, пользователей) достигается фантастическая точность для другого знаменитого иррационального числа – числа e:

```
digits(100 );
vpa( exp( 1 ) )
ans =
2.71828182845904509079559829842764884233473144531
```

Вернемся к основной обязанности пакета расширения Symbolic Math Toolbox – к выполнению аналитических преобразований. Допустим, что требуется выполнить сложение (символьное, а не численное) алгебраического выражения x + y и выражения 3y. Нам совершенно очевидно, что результат (при любых допустимых числовых значениях независимых переменных x и y) должен быть равен x + 4y. Получить такой результат средствами ядра пакета MATLAB нельзя, так как там осуществляются только численные расчеты, а в сформулированной задаче вообще не заданы никакие численные значения для переменных x и y, входящих в исходные выражения. Вот тут-то на помощь и приходит Symbolic Math Toolbox:

```
sym('x+y') + sym('3*y')
ans =
x+4*y
```

Этот пакет осуществляет сложение над объектами нового типа – объектами sym. Такие объекты получаются после вызова одноименной функции – конструктора таких объектов. Над объектами типа sym производятся манипуляции в соответствии с правилами алгебры и математического анализа.

В предыдущем примере можно было бы сначала завести две переменные типа sym, значениями которых были бы алгебраические выражения x + y и 3 * y:

```
symbV1 = sym('x+y'); symbV2 = sym('3*y');
```

Тип созданных переменных с именами symbV1 и symbV2 всегда можно узнать при помощи команды whos (см. рис. 5.10).

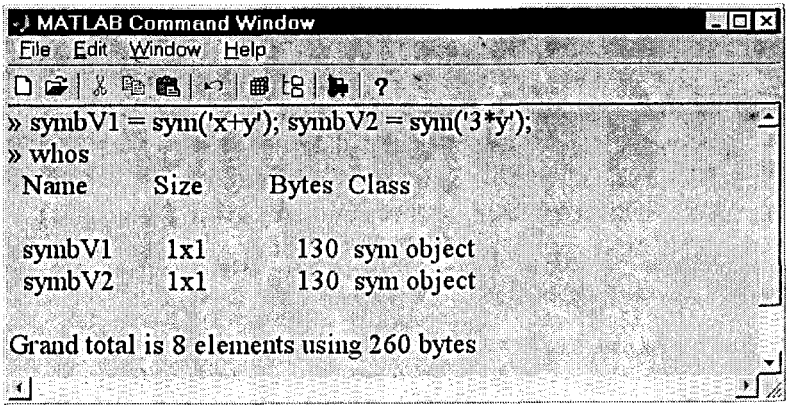


Рисунок 5.10

Отсюда видно, что объекты (переменные) типа sym занимают в памяти 130 байт. Итак, мы имеем две переменные типа sym. Для таких переменных операция сложения выполняется не численно, а по законам алгебры:

```
symbRes = symbV1 + symbV2;
symbRes =
x+4*y
```

Зададимся вопросом, какое имя будет наиболее удачным для символьной переменной со значением x? Наверное, такой переменной лучше всего присвоить имя x:

```
x = sym( 'x' );
```

В результате получается выражение несколько, на первый взгляд, странного вида, но мы только что объяснили его происхождение, так что дальше недоразумений возникать не должно.

В рамках пакета расширения Symbolic Math Toolbox имеется возможность осуществлять над переменными типа sym гигантское количество операций и применять к ним множество интересных функций. Наше пособие посвящено обзору всех возможностей пакета MATLAB, поэтому мы здесь не можем рассматривать аналитические преобразования слишком подробно. Вкратце опишем лишь основные возможности.

Функция simplify осуществляет упрощение символьных выражений:

```
symX = sym('x^2-2*x*y+y^2'); symY = sym('x-y');
simplify( symX/symY)
ans =
x-y
```

Функция expand призвана раскрывать алгебраические и функциональные выражения:

```
A = sym('sin(x+y)');
expand( a )
ans =
sin(x)*cos(y)+cos(x)*sin(y)
```

Ясно, что так можно запрашивать подсказки по забытым формулам алгебры или тригонометрии. Можно просто перемножать многочлены:

```
A = sym('(x+y)*(x-y)*(2*x-3*y)');
expand( a )
ans =
2*x^3-3*x^2*y-2*y^2*x+3*y^3
```

С помощью функции factor можно раскладывать многочлены на простые множители, а целые числа – в произведение простых чисел:

```
factor(sym('x^5 - 1'))
ans =
(x-1)*(x^4+x^3+x^2+x+1)
factor(sym('123456789'))
ans =
(3)^2*(3803)*(3607)
```

Функция subs осуществляет подстановку новых выражений для указанных символьных переменных:

```
syms x y a b
subs(x*y,[x,y],[a+b,a-b])
ans =
(a+b)*(a-b)
```

В последнем примере мы применили функцию syms для краткой записи целого набора эквивалентных выражений:

```
x=sym('x'); y=sym('y');a=sym('a');b=sym('b');
```

Затем мы с помощью функции subs аналитически подставили в выражение x*y вместо x выражение a+b, а вместо y – выражение a-b.

Функция det, оперируя символьными матрицами, аналитически вычисляет детерминант (определитель) этой матрицы:

```
syms x y a b
det([x,y;a,b])
ans =
x*b-y*a
```

Аналитически можно найти и обратную матрицу:

```
syms x y a b
inv([x,y;a,b])
ans =
[ -b/(-x*b+y*a),  y/(-x*b+y*a)]
[  a/(-x*b+y*a), -x/(-x*b+y*a)]
```

Вообще, имеется большое количество функций, осуществляющих *аналитические операции линейной алгебры*. О них всегда можно справиться по встроенной в пакет Symbolic Math Toolbox документации, сосредоточенной в файле \help\pdf_doc\symbolic\symbolic_tb.pdf.

Особенно выигрышными с эмоциональной точки зрения кажутся многим операции *символьного дифференцирования и символьного интегрирования*. Первая из них выполняется функцией diff, а вторая – функцией int. Вот соответствующие примеры:

```
y=sym('x^2-4*x-7');
diff( y, 'x' )
ans =
2*x-4
```

Выражение diff(y,'x') означает «продифференцировать y по x». А теперь пример на интегрирование:

```
y=sym('1/sin(x)');
int( y, 'x' )
ans =
log(csc(x)-cot(x))
```

Можно аналитически вычислять *пределы*:

```
limit(sym('sin(x)/x'))
ans =
1
```

Представляют значительный интерес операции аналитического разложения функций в ряды Тейлора:

```
y=sym('sin(x)');
taylor(y,0)
ans =
x-1/6*x^3+1/120*x^5
```

Здесь фактически представлен частный случай ряда Тейлора – так называемый ряд Маклорена, когда ряд находится для окрестности точки 0.

Имеется возможность управлять порядком разложения:

```
taylor(y,0,8)
```

Результат такой операции изображен на рис. 5.11.

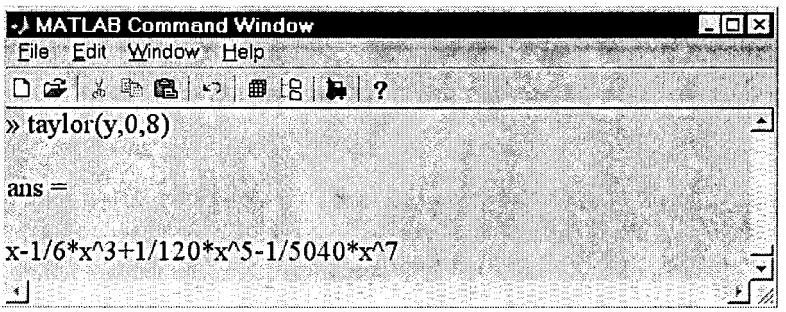


Рисунок 5.11

Пакет расширения Symbolic Math Toolbox позволяет находить *аналитические решения для алгебраических и дифференциальных уравнений.* Вот соответствующие примеры.

Сначала решим простейшее квадратное уравнение:

```
solve(sym('a*x^2+b*x+c'))
ans =
[ 1/2/a*(-b+(b^2-4*a*c)^(1/2))]
[ 1/2/a*(-b-(b^2-4*a*c)^(1/2))]
```

Теперь решим дифференциальное уравнение гармонических колебаний:

```
y'' = -y; y(0)=0; y'(0)=1;
```

Вот как это записывается для выполнения аналитического поиска решения:

```
dsolve('D2y=-y','y(0)=0','Dy(0)=1','x')
ans =
sin(x)
```

Из представленного краткого обзора хорошо видна гигантская мощь пакета расширения Symbolic Math Toolbox, значительно увеличивающего универсальность и практическую ценность пакета MATLAB в целом.

Справочная подсистема пакета MATLAB

Пакет MATLAB снабжен мощной и разветвленной справочной подсистемой. Самым простым способом получить информацию справочного характера о той или иной функции (команде) системы MATLAB можно, введя с клавиатуры команду

```
help Имя_Объекта_Интереса
```

после чего в командное окно выводится в достаточно сжатом виде текстовая информация о запрошенном объекте. Например, команда

```
help drawnow
```

выводит в командное окно системы MATLAB следующую информацию о функции drawnow, рассмотренной нами ранее в связи со сценариями и анимационными задачами (см. рис. 5.12).

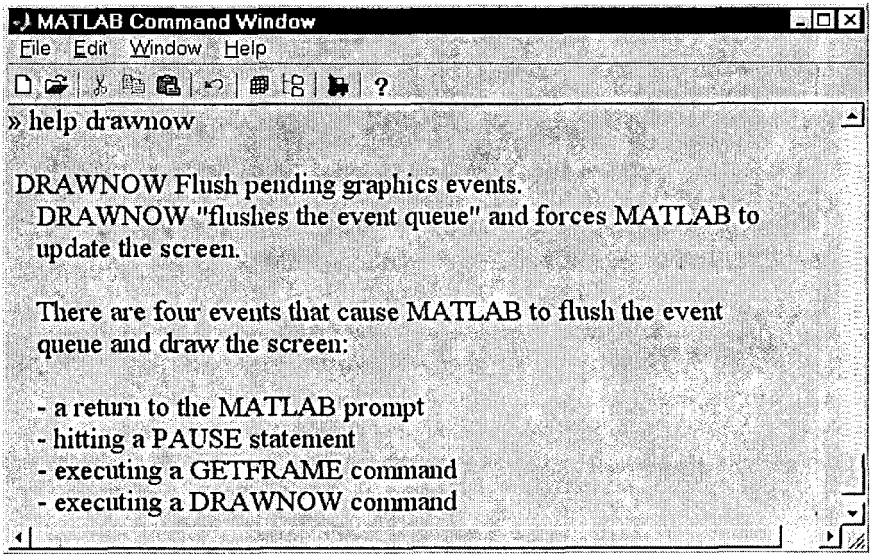


Рисунок 5.12

Объектом, для которого запрашивается справочная информация, может быть любая команда или встроенная функция системы MATLAB, а также все ее графические объекты. Информация по функциям включает в себя все варианты вызова функции с разным числом аргументов.

Другим способом вызова той же самой справочной информации является выполнение команды меню Help|Help Window, в результате осуществления которой появляется отдельное окно (в левом поле ввода этого окна следует ввести имя команды drawnow и нажать клавишу Enter) с той же информацией (см. рис. 5.13).

Здесь важным дополнительным элементом является список, в поле которого видна надпись See also («Смотри также»). Развернув этот список, обнаруживаем надпись More drawnow help (HTML), выбирая которую, переходим к более развернутой справочной информации об этой функции. Эта информация представлена в стандартной форме WWW (основной формат Internet) и показывается в окне Internet-браузера (на рис. 5.14 показано окно браузера IE 5.0):

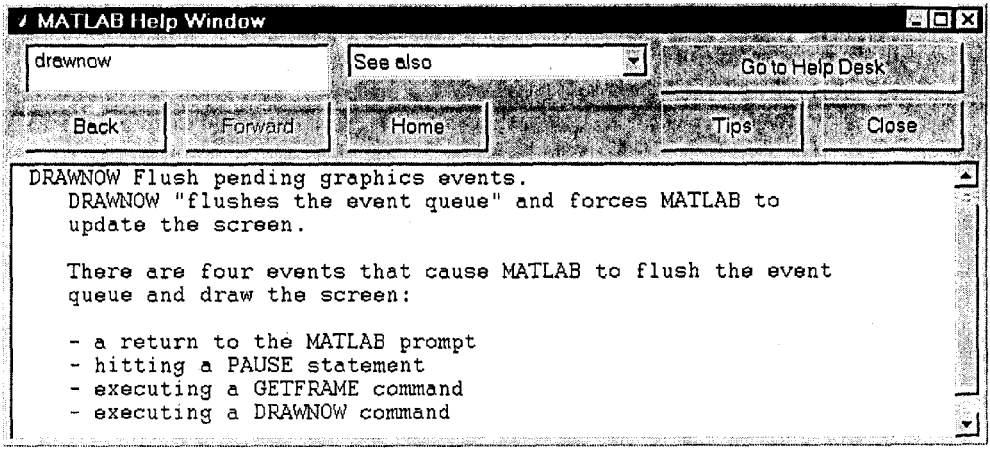


Рисунок 5.13

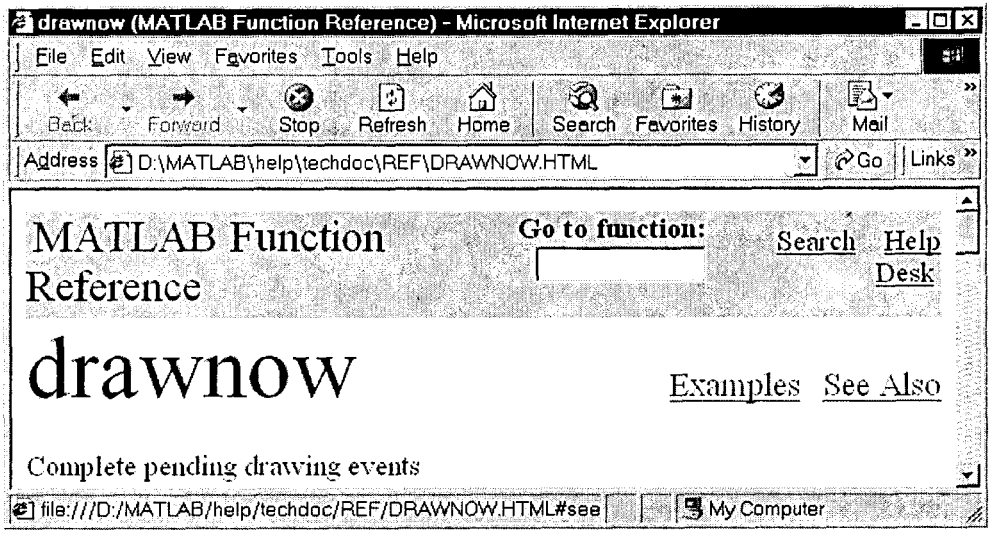


Рисунок 5.14

Представленная в HTML-форме справочная документация системы MATLAB является наиболее точной технической документацией по этой системе. Наличие гиперссылок позволяет легко переходить к другим, близким по содержанию статьям. Зачастую статьи здесь сопровождаются примерами использования объектов, о которых запрошена информация. Заодно из окон браузера можно узнать о том, в каких именно каталогах на диске расположены справочные HTML-файлы. Как видно из предыдущего рисунка, этим каталогом является каталог D:\MATLAB\HELP\TECHDOC\REF, где имя одного из подкаталогов TECHDOC

означает место, где расположена техническая (TECHnical) документация (DOCumentation) по системе MATLAB.

Есть, однако, еще один слой справочной информации. Это так называемые электронные книги в формате PDF (Portable Document Format – формат переносимых документов). PDF-файлы можно просматривать с помощью любого из PDF-вьюеров (viewer – просмотрщик). Ими являются программы Acrobat Reader и Acrobat Exchange фирмы Adobe. Например, на рис. 5.15 показано окно программы Acrobat Exchange с информацией из электронной книги GRAPHG.PDF, посвященной изучению графической подсистемы пакета MATLAB:

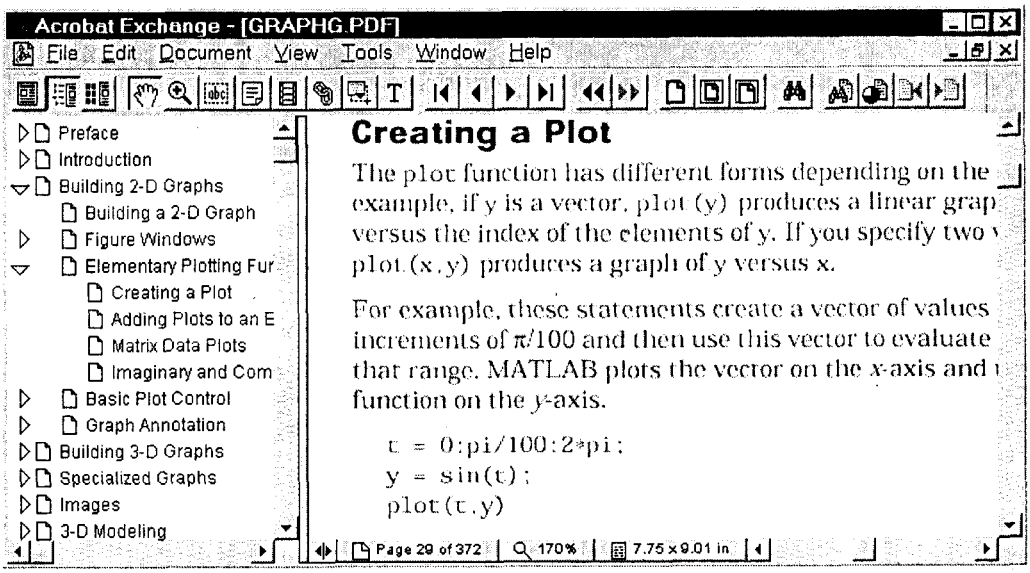


Рисунок 5.15

Информация в электронных книгах полностью отформатирована, ее удобно читать на дисплее компьютера, и она готова к распечатке на принтере. Изложение ведется довольно связным образом и представляет собой достаточно качественный обзор предмета. Лишь по точности и строгости изложения она уступает технической HTML-документации.

Столь большой набор документации пакета MATLAB свидетельствует как о качестве изготовления данного программного пакета, так и о его необычайной широте и глубине. Из последнего утверждения вытекает, что наверняка во всей этой встроенной справочной информации остаются « белые пятна», а также присутствуют неудачно изложенные фрагменты. Данное учебное пособие отчасти предназначено для ликвидации указанных недостатков справочной подсистемы пакета MATLAB.

Часть 2

Программирование в среде системы MATLAB

Глава 6

Программирование функций на М-языке

Синтаксис определения и вызова М-функций

В предыдущей главе изучались сценарии, которые разделяют свои переменные с рабочим пространством системы MATLAB. Такая тесная интеграция не позволяет реализовать полностью изолированный и независимый от рабочего пространства фрагмент кода на М-языке. Кроме того, сценариям невозможно передать входные параметры.

Чтобы реализовать независимый и изолированный фрагмент кода, решающий некоторую фиксированную задачу и принимающий в виде входных параметров начальную информацию, нужно оформить этот фрагмент в виде *функции* системы MATLAB.

Функции, как и сценарии, состоят из набора инструкций М-языка, и их так же записывают в текстовые файлы с расширением `m`. В итоге для лаконичности их называют *М-функциями*. Сейчас мы подробно рассмотрим все особенности устройства М-функций.

Текст М-функции должен начинаться с *заголовка*, после которого следует *тело функции*. Заголовок определяет *интерфейс* функции (способ взаимодействия с ней) и имеет следующий вид:

```
function [Ret1, Ret2,…] = FName(par1, par2,…)
```

В нем объявляется функция (с помощью ключевого слова function) с именем FName, которая принимает *входные параметры* par1, par2,... и вырабатывает *выходные* (*возвращаемые*) *значения* Ret1, Ret2... Многоточия не являются элементами синтаксической конструкции, и мы их здесь применяем только для того, чтобы наглядно подчеркнуть тот факт, что функции могут иметь разное число входных параметров и выходных значений. Вот иллюстрирующие примеры на эту тему:

```
function FName1
function FName2(par1, par2, par3)
function Ret1 = FName3(par1, par2)
function [Ret1, Ret2, Ret3] = FName4(par1)
```

Здесь в первой строке представлен заголовок функции FName1, не имеющей ни входных параметров, ни выходных значений. Функция FName2 принимает три входных параметра и так же не имеет возвращаемых значений. Далее, функция FName3 принимает два и возвращает одно значение. И наконец, функция FName4 имеет один входной параметр и три выходных значения.

Часто для краткости входные параметры называют *аргументами функции*, а возвращаемые (выходные) значения функции называют просто ее *значениями*.

Указанное в заголовке имя функции должно совпадать с именем файла (без учета расширения.m), в который записывается текст функции. Рассогласование имени функции и имени файла не допускается.

Тело функции состоит из инструкций М-языка, с помощью которых в итоге вычисляются возвращаемые значения. Тело функции следует за заголовком функции. Заголовок функции плюс тело функции в совокупности составляют *определение функции*. Таким образом, в М-файл записывается именно определение функции. Размещать М-файл с определением функции нужно в одном из каталогов диска, входящих в список доступа пакета MATLAB. Об этом мы уже рассказывали ранее, когда изучались сценарии.

Если в М-файл поместить определения сразу нескольких функций, то вызывать из командного окна системы MATLAB (или из функций другого М-файла) можно будет только ту из них, имя которой совпадает с именем М-файла. Таким образом, только одна функция из М-файла видима вне этого файла. Остальные функции вызываются изнутри данного М-файла и выполняют вспомогательную работу.

Вот пример ситуации, когда в файле ManyFunc.m содержатся определения двух функций (определение функции заканчивается либо исчерпанием инструкций, либо заголовком определения следующей функции):

```
function ret1 = ManyFunc( x1, x2 )
ret1 = x1 .* x2 + AnotherFunc( x1 );
function ret2 = AnotherFunc( y )
ret2 = y .* y + 2 * y + 3;
```

Здесь определены функции ManyFunc и AnotherFunc. Однако извне можно вызывать только функцию ManyFunc. Функцию AnotherFunc может вызывать функция ManyFunc и больше ничто. Функцию ManyFunc можно назвать основной функцией в файле ManyFunc.m, а функцию AnotherFunc – *вспомогательной функцией* или *подфункцией* (по-английски – *subfunction*).

Существует еще один способ ограничить возможности вызова функции. Допустим, мы хотим, чтобы некоторую М-функцию могли вызывать только М-функции из заданного каталога и больше ничто. Тогда у этого каталога нужно создать дочерний подкаталог с именем private и поместить туда указанную функцию. Все функции из каталога private видимы только функциям из родительского каталога.

Как входные параметры, так и возвращаемые значения могут быть в общем случае массивами различных типов, размерностей и размеров. Например, следующая функция с именем MatrProc1:

```
function [ A, B ] = MatrProc1( X1, X2, x )
% MatrProc1 calculates MatrixProduction
% plus (or by) one scalar
%---------------------------
A = X1 .* X2 * x;
B = X1 .* X2 + x;  %The last line of the code
```

рассчитана на «прием» двух числовых массивов X1 и X2 одинакового размера и одного скаляра x. Эти массивы в теле функции сначала перемножаются поэлементно, после чего результат такого перемножения дополнительно умножается на скаляр x. Таким образом порождается первый из выходных массивов, то есть массив A. Одинаковые размеры входных массивов X1 и X2 гарантируют выполнимость операции их поэлементного умножения. Второй выходной массив (с именем B) отличается от первого тем, что получается сложением со скаляром (а не умножением).

В тексте функции MatrProc1 помимо инструкций, вычисляющих возвращаемые функцией значения, присутствуют также комментарии, начинающиеся со знака %. Как известно, в любом языке программирования комментарии не являются исполняемыми инструкциями, а служат лишь для целей документирования процесса программирования. Они поясняют программисту смысл того или иного участка кода. Без комментариев трудно бывает понять отдельные места в сложных программах, когда прошло изрядное время с момента первоначальной разработки программы (функции).

В системе MATLAB особую роль играют несколько строк комментариев, располагающихся сразу же за заголовком функции. Эти комментарии предназначены не только для программистов, но и для конечных пользователей, желающих ознакомиться с краткой информацией по разработанной кем-то другим функции. Таким образом, в этих строках должна быть расположена краткая

справка по хорошо документированной функции. Получить эту справку может любой пользователь функции, если выполнит команду

```
help MatrProc1
```

в результате выполнения которой в командное окно поступит следующая информация (см. рис. 6.1):

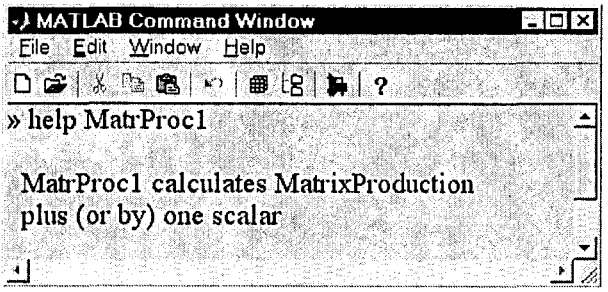


Рисунок 6.1

Эти же начальные строки комментариев участвуют в поиске функции по содержащимся в ней текстовым фрагментам. Такой поиск осуществляется командой

```
lookfor текст -all
```

Наконец, командой

```
type имя_функции
```

«распечатывается» (в командном окне) весь текст М-функции вместе со всеми ее комментариями.

Теперь продолжим изучение других аспектов синтаксиса М-функций, а именно тех из них, что связаны с использованием разработанных функций. *Вызов* созданной нами функции осуществляется из командного окна системы MATLAB или из текста какой-либо другой функции. Разные М-функции с *совпадающими именами* (совпадают имена их М-файлов) могут располагаться в разных каталогах на диске компьютера. В результате при вызове М-функций система MATLAB должна руководствоваться четко определенным критерием выбора конкретной функции. Критерий состоит в заданной последовательности отбора кандидатов. Приоритет всегда отдается внутренней для М-файла функции, если вызов осуществляется внутри этого файла и если внутренняя функция с этим именем существует. В случае неудачи поиск продолжается в частном подкаталоге `private` и только после этого начинает просматриваться весь список путей доступа пакета MATLAB вплоть до первой встретившейся при этом функции с совпадающим именем (это имя М-файла).

Синтаксис вызова функции традиционен для большинства языков программирования: записывается имя функции, после которого в круглых скобках через запятую перечисляются *фактические входные параметры*, со значениями кото-

рых и будут произведены вычисления. Фактические параметры могут быть заданы числовыми (и другими) значениями, именами переменных (уже имеющими конкретные значения), а также выражениями.

Если фактический параметр задан именем некоторой переменной, то реальные вычисления будут производиться с *копией* этой переменной (а не с ней самой). Это называется *передачей параметров по значению*.

На рис. 6.2 показан вызов из командного окна системы MATLAB ранее созданной нами для примера функции `MatrProc1`.

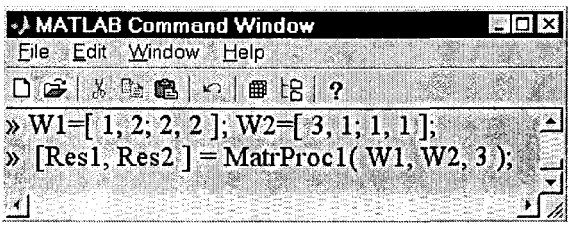


Рисунок 6.2

Здесь имена фактических входных параметров (`W1` и `W2`) и переменных, в которых записываются результаты вычислений (`Res1` и `Res2`), не совпадают с именами аналогичных переменных в определении функции `MatrProc1`. Очевидно, что совпадения и не требуется, тем более что у третьего входного фактического параметра нет имени вообще! Чтобы подчеркнуть это возможное отличие, имена входных параметров и выходных значений в определении функции называют *формальными*.

В рассмотренном примере вызова функции `MatrProc1` из двух входных квадратных матриц `2 x 2` получаются две выходные матрицы `Res1` и `Res2` точно таких же размеров:

```
Res1 =
     9   6
     6   6
Res2 =
     6   5
     5   5
```

Вызвав функцию `MatrProc1`

```
[r1,r2] = MatrProc1( [1 2 3; 4 5 6 ],[ 7 7 7; 2 2 2 ],1 );
```

с двумя входными массивами размера `2 x 3`, получим две выходные матрицы размера `2 x 3`. То есть одна и та же функция `MatrProc1` может обрабатывать входные параметры различных размеров и размерностей! Можно вместо массивов применить эту функцию к скалярам (это все равно массивы размера `1 x 1`).

Теперь рассмотрим вопрос о том, можно ли использовать функцию `MatrProc1` в составе выражений так, как это делается с функциями, возвра-

щающими единственное значение? Оказывается, это делать можно, причем в качестве значения функции, применяемого для дальнейших вычислений, используется первое из возвращаемых функцией значений. Следующее окно системы MATLAB иллюстрирует это положение (см. рис. 6.3).

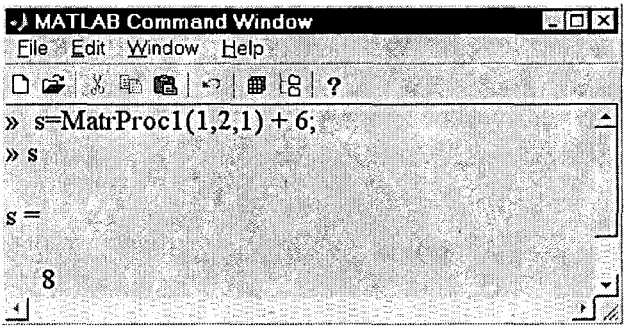


Рисунок 6.3

При вызове с параметрами 1, 2, 1 функция MatrProc1 возвращает два значения: 2 и 3. Для вычисления всего выражения используется первое из них, так что переменная s становится равной 8.

Так как вызов любой функции можно осуществить написав произвольное выражение в командном окне системы MATLAB, то всегда можно совершить ошибку, связанную с несовпадением типов фактических и формальных параметров. MATLAB не выполняет никаких проверок на эту тему, а просто передает управление функции. В результате могут возникнуть ошибочные ситуации. Чтобы избежать (по возможности) возникновения таких ошибочных ситуаций, нужно в тексте М-функций осуществлять *проверку входных параметров*. Например, в функции MatrProc1 легко осуществить выявление ситуации, когда размеры первого и второго входных параметров различны. Для написания такого кода требуется полный набор конструкций управления, из которых мы пока что изучили лишь операторы цикла (в гл. 5).

Конструкции управления

В любом языке программирования, в том числе и в М-языке, встроенном в систему MATLAB, имеются специальные конструкции, которые задаются с помощью зарезервированных ключевых слов этого языка и служат для управления порядком выполнения операций.

Такие конструкции часто называют *операторами управления*. К ним относятся *операторы ветвления* и *операторы цикла*. Операторы цикла мы изучили ранее в гл. 5. Теперь изучим операторы ветвления.

К операторам ветвления в М-языке относятся *условный оператор* и *оператор переключения*. Условный оператор использует ключевые слова if (если), else

(иначе), elseif (иначе если), end (конец всей конструкции) и может выступать в одной из следующих трех форм. Во-первых,

```
if условие
...
end
```

Во-вторых,

```
if условие
...
else
...
end
```

и, наконец, в форме

```
if условие1
...
elseif условие2
...
else
...
end
```

в которой ветвей с ключевым словом elseif может быть много.

Область действия условного оператора начинается ключевым словом if, а заканчивается ключевым словом end. Под условием понимается произвольное выражение (чаще всего это выражение включает в себя операции сравнения и логические операции), истинность или ложность которого понимается как отличие или равенство нулю.

Если условие истинно, то выполняются команды, стоящие после строки с ключевым словом if. Если условие ложно, то эти команды пропускаются и либо переходят к следующему за условным оператору (первая форма), либо проверяют еще одно условие в строке с ключевым словом elseif (третья форма условного оператора), или выполняются без дополнительных проверок команды, стоящие после строки с ключевым словом else (вторая из приведенных выше форм). Более подробных объяснений по работе условного оператора не требуется, так как фактический смысл ключевых слов говорит сам за себя.

Однако требуются еще кое-какие разъяснения, связанные с тотальным использованием в М-языке системы MATLAB массивов. Их можно использовать и в условных выражениях, входящих в условные операторы М-языка системы MATLAB. В тех случаях, когда значением таких выражений будет массив, истинность условия наступает, когда истинны (не равны нулю) все элементы массива. Если хоть один элемент такого массива будет равен нулю, то условие

считается ложным. Кроме того, ложность имеет место при использовании пустых массивов.

Приведем иллюстрирующий работу условного оператора фрагмент кода:

```
A = [ 1 2; 4 0 ];
if A
  b = 1;
else
  b = 2;
```

в результате выполнения которого переменная b получит значение 2, так как матрица A содержит один нулевой элемент и все условие считается ложным.

Кстати, запись `if A` по своему действию полностью эквивалентна записи

```
if A ~= 0
```

и записи

```
if all( A( : ) )
```

Условные операторы часто используются в теле М-функций совместно с инструкцией `return` для осуществления *досрочного завершения* функции и выхода из нее. В следующей функции вычисляется факториал целого положительного числа n:

```
function res = MyFactorial( n )
 res = 1;
 if n == 1
  return
 else
   for i = 2:n
    res = res .* i;
   end
 end
```

Если входной параметр данной функции равен 1, то с помощью оператора `return` осуществляется досрочный выход из функции, так как правильный результат (`res = 1;`) уже сформирован.

Когда оператор `return` не используется, нормальное завершение функции наступает по исчерпании инструкций из тела функции.

Условные операторы также широко применяются внутри операторов цикла совместно с инструкцией `break` для досрочного выхода из цикла. Иногда такое условие выхода из цикла может быть единственным, как в следующем примере:

```
prod = 1; i = 1;
while 1
 prod = prod .* i;
```

```
    i = i + 1;
    if i > 8
     break
    end
   end
```

где условие в заголовке цикла всегда истинно, так что цикл был бы бесконечным, если бы не инструкция `break`, работающая совместно с условным оператором. В результате цикл здесь конечен и вычисляет произведение всех целых чисел от единицы до восьми (то есть это факториал восьми).

Теперь продолжим изучать операторы ветвления. Другим оператором ветвления является оператор переключения. Он использует ключевые слова `switch` (переключить), `case` (случай), `otherwise` (иначе) и имеет следующую конструкцию:

```
switch выражение
 case  значение1
 ...
 case { значение2, значение3 }
 ...
 ...
 otherwise
 ...
end
```

Сначала вычисляется вырабатывающее скалярное числовое значение `выражение`, а затем полученный результат сравнивается с набором значений `значение1`, `значение2`, `значение3` и т. д. В случае совпадения с одним из значений выполняется нижестоящая ветка. Если нет совпадения ни с каким из перечисленных значений, то выполняется ветка, стоящая после ключевого слова `otherwise`. Строк с ключевым словом `case` может быть много, а строка с ключевым словом `otherwise` одна.

Отметим одну важную деталь: после выполнения той или иной ветви оператора переключения никакие другие ветви этого оператора (например, нижележащие) не выполняются. Поэтому в М-языке системы MATLAB не надо предпринимать никаких специальных усилий для обхода нижележащих ветвей оператора переключения (в отличие, например, от языков C/C++).

Интерактивное взаимодействие М-функций с пользователем

Только в самых простых случаях не требуется интерактивного взаимодействия с пользователем во время выполнения М-функций. В таких случаях пользователь остается в неведении как о ходе вычислений, так и о достигнутых промежуточных результатах.

Очень часто бывает полезно контролировать ход выполнения М-функций. Для интерактивного взаимодействия с пользователем в М-языке предусмотрен ряд специальных функций. В частности, функция `disp` применяется для вывода промежуточных результатов в командное окно системы MATLAB. На основе анализа пользователем качества промежуточных результатов может приниматься решение о продолжении вычислений или об их прекращении.

Функцию `disp` вызывают с единственным аргументом, который может быть числовым или символьным массивом (вектор, матрица). Ниже показан пример циклических повторяющихся вычислений конечных отрезков расходящегося гармонического ряда со все большим числом входящих в них слагаемых:

```
S = 0; i = 1;
while 1
  S = S + 1 / i;
  str = sprintf('i=%d  S=%f', i, S );
  disp( str )
  i = i + 1;
  if rem(i,100) == 0
   ans = input( 'Stop?  Answer=', 's' );
   if ans == 'y'
      return
   end
  end
end
```

Здесь функцией `disp` все значения промежуточных (частичных) сумм ряда выводятся в командное окно системы MATLAB. Кроме того, через каждые сто слагаемых функцией `input` выводится надпись, предписывающая пользователю ответить на вопрос, не пора ли останавливать вычисления (вообще говоря, бесконечные). Если пользователь вводит при этом с клавиатуры латинский символ `'y'` (первая буква английского слова yes), то вычисления прекращаются, так как выполняется оператор досрочного выхода из функции `return`.

Непосредственный ввод информации с клавиатуры и присвоение введенного значения переменной `ans` также осуществляется функцией `input`. В качестве второго аргумента (параметра) функции `input` используется символ `'s'`, озна-

чающее, что функция input должна принять символьные (а не числовые) значения. Еще поясним использование новой для нас функции sprintf, которая помещает в выходную строку значения своих аргументов (кроме первого аргумента) в формате, указанном первым аргументом. Этот формат полностью аналогичен формату, ранее описанному в гл. 3 в связи с функцией fprintf, осуществляющей форматный вывод информации в текстовые файлы.

В результате разработанная функция позволяет пользователю полностью контролировать процесс вычислений, наблюдать за промежуточными результатами и прекращать вычисления по мере необходимости. На рис. 6.4 показано командное окно системы MATLAB, в котором присутствует текущая выходная информация вышеразработанной программы, из которой видно, что, например, для достижения текущей суммой ряда (она ничем сверху не ограничена) значения 10, нужно просуммировать 12 367 членов этого ряда:

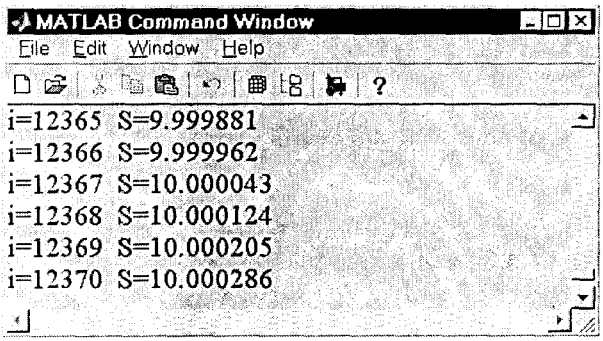


Рисунок 6.4

в то время как для достижения значения 5 достаточно всего 83 слагаемых.

Для вывода предупредительных сообщений (удобное средство для отладки функций) вместо функции disp лучше использовать функцию warning, так как ее вывод легко подавить из командного окна, выполнив команду

```
warning off
```

Возобновление вывода функцией warning предупредительных сообщений осуществляется командой

```
warning on
```

Если в теле М-функции присутствует код обнаружения тех или иных ошибочных ситуаций, то завершать выполнение М-функций в таких случаях целесообразно вызовом функции error, которая не только выполняет такое завершение абсолютно корректно, но и выводит поясняющее ошибку сообщение в командное окно системы MATLAB. Например, ниже показана функция, вычисляющая величину, обратную к входному числовому скалярному параметру:

```
function y = Inverse( x )
```

```
if x == 0
 error( 'Division by zero' )
else
 y = 1 ./ x;
end
```

Она проверяет, не равен ли нулю входной параметр, и если равенство имеет место, то выполняется функция error. Вот что при этом видит пользователь в командном окне (см. рис. 6.5).

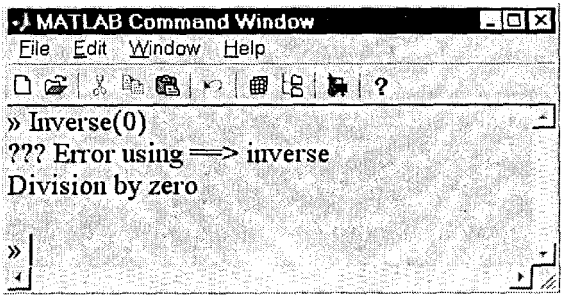


Рисунок 6.5

Если требуется временно приостановить выполнение М-функции, чтобы, например, спокойно посмотреть и проанализировать промежуточную информацию, то следует в код функции вставить вызов функции pause (без входных параметров). Дальнейшее выполнение М-функции возобновляется после нажатия любой клавиши на клавиатуре.

Завершая краткое изложение функций системы MATLAB, ориентированных на интерактивный ввод-вывод, упомянем еще функцию menu, реализующую очень наглядный графический ввод одного из альтернативных значений. Например, строка кода с вызовом функции menu

```
n = menu('Enter your choice','Dog','Cat');
```

помещенная в тело некоторой М-функции, вызывает появление на дисплее компьютера следующего окна с надписью Enter your choice и двумя кнопками (см. рис. 6.6).

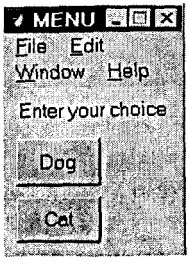


Рисунок 6.6

Если пользователь нажмет левую клавишу мыши, указав на кнопку Dog, то переменная n получит значение 1, а если кнопку Cat, то переменная n станет равной 2. Таким образом, возвращаемое функцией menu числовое значение есть номер выбранного пользователем пункта из списка всех доступных вариантов (*меню выбора*).

Специальным случаем интерактивного взаимодействия с пользователем является ввод им имени функции или символьного представления выражения, подлежащих дальнейшему вычислению. В системе MATLAB для выполнения (вызова) функций, имя которых заранее неизвестно и содержится в качестве значения в текстовой строке, а также для вычисления выражений по их символьному представлению предназначены две специальные функции: eval и feval.

Допустим, пользователь ввел с клавиатуры в строковую переменную str1 следующее содержание:

```
'A = [ 1 2 3; 4 5 6; 7 8 9];'
```

что означает, что он хочет создать числовую матрицу с именем A и с девятью числовыми элементами, указанными в представленной выше строке текста. Эту работу выполняет функция eval:

```
eval( str1 );
```

после чего и появится числовая переменная с именем A и указанным выше значением. Если пользователь введет строку

```
'x1 = sin( 1.25 );'
```

то функция eval, выполнив синтаксический разбор этой строки, вычислит значение функции sin(1.25) и поместит полученное значение 0.9490 в переменную x1.

Если в процессе работы функции eval возникнет ошибочная ситуация, то в командном окне системы MATLAB появится сообщение о характере ошибки. Например, если мы ошибочно определим строку s следующим образом:

```
s = 'sin( 1.25';
```

и вызовем в командном окне функцию eval(s), то увидим следующее сообщение:

```
Improper function reference. a ',' or ')' is expected.
```

означающее, что не хватает либо запятой, либо закрывающей круглой скобки (последнее является действительной ошибкой).

Если же такая ошибочная ситуация возникнет в теле М-функции, то в командное окно системы MATLAB будет выдано лишь сообщение о том, в какой именно строке М-функции возникла ошибочная ситуация, но никакой диагностики сообщено не будет. Чтобы все-таки получить эту строковую диагностиче-

скую информацию в теле М-функции, нужно вызывать функцию eval с двумя аргументами, указав строковым значением второго аргумента имя другой (нами же разработанной) М-функции, предназначенной для обработки ошибок. В последней функции (то есть в функции обработки ошибок) нужно вызывать функцию lasterr, которая и возвратит эту диагностическую информацию.

Вот соответствующий пример М-функции, где предусмотрен вызов собственной функции MyErrorCatch для обработки ошибок, могущих возникнуть в процессе выполнения функции eval:

```
function y = ErrorTest
s = input('Enter string:', 's' );
y = eval( s, 'MyErrorCatch' );
```

Если с клавиатуры будет введено рассмотренное выше ошибочное выражение с вычислением синуса, то управление будет передано в функцию MyErrorCatch:

```
function z = MyErrorCatch
err = lasterr;
%error processing:…
z = 2;
```

в которой строка err получит в качестве значения символьное выражение 'Improper function reference. a »,» or «)» is expected'. Проанализировав далее содержимое переменной err, можно распознать характер ошибочной ситуации и предпринять некоторые исправляющие действия. Здесь же, для краткости изложения, мы просто возвращаем двойку. В результате эта единица становится и возвращаемым значением функции ErrorTest, в которой, собственно, и возникла ошибочная ситуация. Таким образом, функция ErrorTest всегда возвращает какое-либо значение: когда выражение для вычисления синуса вводится с клавиатуры без ошибок, происходит вычисление этой функции и возвращаемое значение по модулю меньше единицы, а в случае каких-либо ошибок ввода возвращается двойка.

По сравнению с функцией eval, которая хорошо подходит для вычисления сложных выражений, функция feval более удобна для вычисления одиночной функции, которой при этом можно передать дополнительные параметры:

```
feval( str, x1, x2,…,xN )
```

Здесь строка str задает имя функции, подлежащей вычислению при значениях аргументов x1, x2, ..., xN. Вот простой иллюстрирующий пример использования функции feval:

```
FunName = { 'cos', 'sin', 'tan' };
m = input('Enter function index: ');
x = feval( FunName{m}, 0.5 );
```

Если с клавиатуры будет введено целое число 2, то с помощью функции `feval` будет вычислено значение `sin(0.5)`, равное `0.4794`.

Локальные, глобальные и статические переменные

М-функции располагают собственным пространством переменных, изолированным от рабочего пространства системы MATLAB. Поэтому совпадение имен переменных из рабочего пространства и имен внутренних переменных М-функций не приводит ни к каким коллизиям.

Переменные, которые используются в теле М-функции и не совпадают с именами формальных параметров этой функции, называются *локальными*. Их область действия полностью ограничена рамками тела данной М-функции. Подругому говоря, они видимы лишь в пределах М-функции. Из рабочего пространства системы MATLAB и из других М-функций они не видны и не достижимы. Аналогично локальные внутри некоторой функции переменные не видны из другой М-функции и из рабочего пространства.

Основным каналом передачи информации из командного окна системы MATLAB в М-функцию и из одной функции в другую является механизм параметров функции. Другим механизмом передачи информации в функцию являются *глобальные переменные*.

Чтобы рабочая область системы MATLAB и несколько М-функций могли совместно использовать некоторую переменную с заданным именем, ее всюду нужно объявить как *глобальную* с помощью ключевого слова `global`.

На рис. 6.7 показан пример, когда переменная `b` объявляется в командном окне как глобальная.

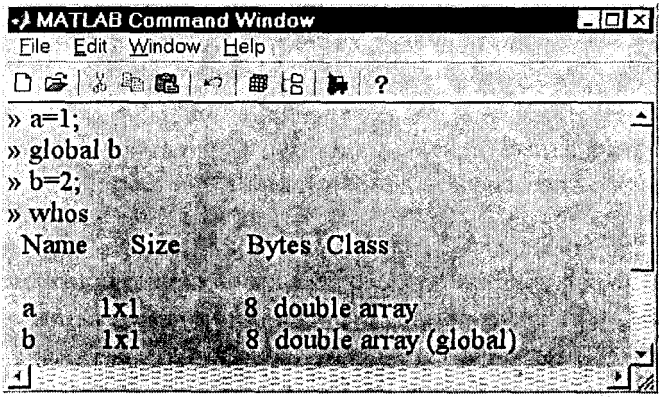


Рисунок 6.7

Команда whos показывает информацию для всех переменных рабочей области системы MATLAB, а глобальные переменные отмечает при этом ключевым словом global. Команда

```
whos global
```

выдает справочную информацию только по глобальным переменным.

Глобальные переменные (в отличие от других переменных) *автоматически инициализируются пустыми массивами*:

```
global c
c =
  []
```

Таким образом, глобальные переменные всегда имеют хоть какие-то значения. Глобальные переменные, явно инициализированные в командном окне системы MATLAB, можно смело использовать в теле М-функции, не производя в последней никаких начальных присваиваний этой переменной.

На рис. 6.8 показан пример, когда переменная glVarS объявляется глобальной в тексте функции FuncWithGlobVar и в командном окне системы MATLAB.

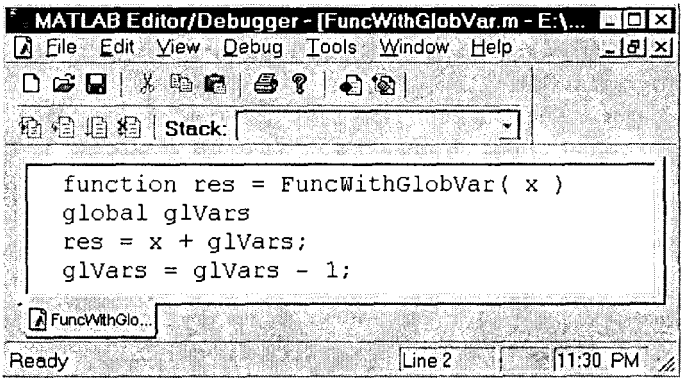


Рисунок 6.8

В командном окне системы MATLAB переменная glVars явно инициализируется значением 5, после чего вызывается функция FuncWithGlobVar, в теле которой значение этой переменной используется в вычислениях (см. рис. 6.9)

После выполнения функции FuncWithGlobVar глобальная переменная glVars изменила свое первоначальное значение и стала равной 4. Так как у глобальных переменных «глобальная» область действия, то, чтобы случайно (по ошибке) не переопределить ее где-либо, желательно давать таким переменным мнемонические (более длинные и осмысленные) имена.

При повторном вычислении функции FuncWithGlobVar с тем же самым входным параметром, равным 1, она возвратит уже иное значение, равное 5. Та-

кой эффект, когда одинаковый вызов функции приводит к разным возвращаемым значениям, связан с использованием глобальных переменных и с их свойством сохранять свои значения между разными вызовами функции.

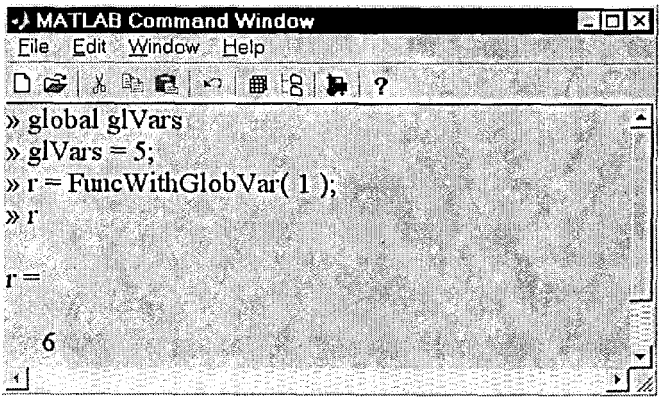


Рисунок 6.9

Такого же эффекта можно добиться и с локальными переменными, если объявить их с ключевым словом `persistent`. Тогда эти переменные, оставаясь попрежнему недоступными вне М-функции, сохраняют свои значения между вызовами этой функции. Такие *стабильные*, или *статические*, локальные переменные неявно инициализируются пустыми массивами при первом (в рамках заданного сеанса работы с пакетом MATLAB) входе в М-функцию.

Например, функция

```
function y = MyPersistFun( x )
persistent var;
if isempty(var)
  var = 1;
end
y = var .* x;
var = y;%save for future call
```

имеет статическую (стабильную) переменную `var`, которая неявно инициализируется пустым массивом (инициализация осуществляется только *один раз* при первом входе в функцию, чем эта операция и отличается от операции присваивания, которая выполняется *каждый раз* при любом входе в функцию). После этого в условном операторе переменной `var` присваивается единица. Все это происходит при первом входе в эту функцию. При последующих заходах в функцию `MyPersistFun` переменная `var` уже не является пустой, так что поток управления (последовательность выполнения операторов) обходит условный оператор стороной.

Из представленных объяснений деталей работы функции MyPersistFun следует, что циклический вызов функции MyPersistFun из командного окна (или из другой М-функции)

```
for i = 1:5
  res = MyPersistFun( i );
end
```

приведет к вычислению факториала пяти:

```
res =
   120
```

Так как статическая переменная var обладает свойством сохранять свои значения между вызовами функции MyPersistFun, то повторное исполнение указанного выше циклического вызова этой функции приведет уже к другому значению переменной res:

```
res =
   14400
```

поскольку ранее полученное значение 120 последовательно умножается на 2, 3, 4 и 5. Так что представленное решение – не лучшее решение задачи о вычислении факториала. Это просто формальный пример на применение статических локальных переменных, то есть локальных переменных с атрибутом persistent. Тем не менее вернуть ситуацию в исходное состояние можно, не только закрыв сеанс работы с пакетом MATLAB, но и просто выполнив команду

```
clear all
```

К настоящему моменту мы привели несколько примеров самых разных реализаций алгоритма вычисления факториала целого положительного числа. Функции, реализующие разные алгоритмы решения одной и той же задачи, могут отличаться друг от друга своим размером, скоростью выполнения и другими характеристиками. Обсуждению этого вопроса посвящен следующий подраздел.

Рекурсивные функции. Производительность М-функций

Прежде чем начать сравнивать между собой функции, реализующие разные алгоритмы решения одинаковых задач, рассмотрим еще один (последний) вариант решения задачи о вычислении факториала целого положительного числа. Этот вариант основан на применении *рекурсивных функций.*

Функция называется рекурсивной, если в ее теле осуществляется вызов самой себя. Вот абсолютно формальный пример описанной ситуации:

```
function y = SimpleRecursF( x )
...
z = SimpleRecursF( t ) + x .* x;
...
```

Так как в теле функции SimpleRecursF осуществляется вызов этой же функции (с другим аргументом), то по определению функция является рекурсивной. В общем случае писать рекурсивные функции сложнее нерекурсивных, так как здесь следует тщательно прослеживать последовательность вложенных вызовов таких функций, чтобы установить завершение этого процесса. Очень легко совершаются ошибки, связанные с зацикливанием рекурсивных вызовов, так что окончательного выхода из такой функции не происходит вообще, и компьютер мог бы «зависнуть», если бы не то обстоятельство, что в системе MATLAB (в отличие от программирования на языке С) все М-функции выполняются под полным контролем самой системы, которая и ограничивает предельное количество рекурсивных вызовов. С одной стороны, это положительное свойство, а с другой – это слишком сильное вторжение в процесс выполнения рекурсивных функций, что ограничивает их самостоятельную роль.

И вообще роль рекурсивных функций в системе MATLAB существеннно ниже, чем в системах программирования на языках C/C++, так как наиболее интересные случаи базируются на применении указателей, которых в М-языке системы MATLAB нет вовсе.

Тем не менее рекурсивные функции допустимы в рамках пакета MATLAB, и мы сейчас их кратко рассмотрим на, уже набившем наверное, оскомину примере вычисления факториала положительного целого числа. Последний можно определить рекурсивно (рекуррентно) как произведение этого числа на факториал числа, меньше заданного на единицу. Вот реализация рекуррентного определения факториала на М-языке в виде рекурсивной М-функции:

```
function res = MyRecursFact( n )
if n == 1
  res = 1;
  return
else
  res = n * MyRecursFact( n - 1 );
end
```

Здесь гарантией завершения цепочки рекурсивных вызовов является условный оператор, так как вызовы осуществляются с уменьшающимися значениями аргументов, и когда те достигнут значения 1, начнется обратная цепочка возвратов из этих экземпляров функций с последовательным умножением возвращае-

мых значений на последовательно увеличивающиеся положительные числа, что и приведет в итоге к вычислению факториала.

Разработанная рекурсивная функция прекрасно вычисляет факториалы различных целых чисел:

```
MyRecursFact( 57 )
ans =
  4.0527e+076
MyRecursFact( 163 )
ans =
 2.0044e+291
```

и т. д. Следующий же результат:

```
MyRecursFact( 200 )
ans =
   Inf
```

означает, что столь большое значение факториала не может поместиться в отведенную под число типа `double` память компьютера, так что приходится заменять истинное числовое значение на символическую константу `Inf`, означающую «машинную бесконечность», но сама рекурсивная функция работает при этом нормально. В то же время результат следующего вызова этой функции с очевидностью подтверждает, что система MATLAB действительно накладывает ограничение на глубину рекурсивных вызовов (это мы упомянали выше, подчеркнув, что тем самым преодолевается проблема бесконечной рекурсии) (см. рис. 6.10).

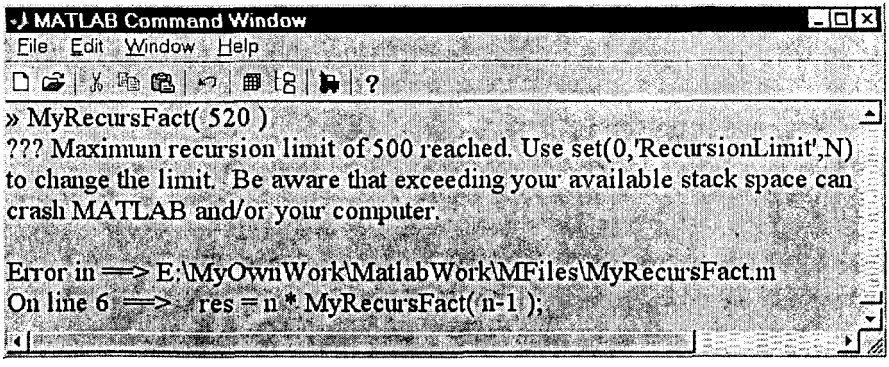


Рисунок 6.10

Из сопутствующего диагностического сообщения следует, что предел глубины вызовов по умолчанию равен 500. Если нужно увеличить эту величину, то следует выполнить команду

```
set(0,'RecursionLimit',N)
```

где N – новое значение предела для любых рекурсивных функций, действующее в рамках текущего сеанса работы с пакетом MATLAB.

Заканчиваем на этом краткий рассказ про рекурсивные функции (напоминаем, что их роль существенно ниже, чем в системах программирования на языках C/C++) и переходим к вопросу об эффективности М-функций вообще и разных реализаций алгоритмов решения одной и той же задачи в частности.

Ясно, что разные реализации решения одной и той же задачи отличаются друг от друга самыми разнообразными свойствами, такими, например, как размер функций, их «читаемость», сложность разработки (в смысле времени, затраченном на разработку и отладку функции). Разные реализации отличаются даже по такому неуловимому понятию, как «красота реализации». Все перечисленные характеристики вполне очевидны, мы займемся только эффективностью (в смысле быстродействия выполнения М-функции). Пакет MATLAB поставляет удобное средство для приблизительного замера этой характеристики – набор команд `tic` и `toc`. Это средство применяется следующим образом:

```
tic, FunctionCall, toc
```

то есть в командной строке вызов некоторой функции обрамляется с двух сторон этими командами, в результате чего после вычисления функции в командное окно выдается не только результат вычисления, но и условное время вычисления. По этому условному времени можно сравнивать разные реализации функций между собой. Следует, однако, иметь в виду, что время выполнения замеряется приблизительно и зависит от ряда неконтролируемых факторов, так что такие вызовы следует повторить несколько раз, а полученные замеры усреднить.

Сейчас сравним на практике две реализации решения задачи о вычислении факториала: разработанную и представленную выше функцию `MyFactorial`, основанную на операторе цикла, и функцию `MyRecursFact`, основанную на рекурсии. В результате выясним, что эффективнее – циклы или рекурсия. Вот практические результаты соответствующих замеров:

```
tic, MyRecursFact( 100 ), toc
ans =
   9.3326e+157
elapsed_time =
   0.0710
tic, MyFactorial( 100 ), toc
ans =
   9.3326e+157
elapsed_time =
   0.0700
```

Видно, что результаты замеров условного времени практически одинаковые. Так что функции вычисления факториалов на основе оператора цикла и на основе рекурсии сравнимы по эффективности.

Однако бывают случаи, когда разные реализации имеют колоссально разные эффективности. Таким типичным случаем для М-языка системы MATLAB является возможность избежать использования операторов цикла с помощью применения операции задания диапазона значений и других эффективных групповых операций М-языка (множественная индексация, например).

В частности, следующий фрагмент:

```
A = 1 : 0.1 : 2000;
```

формирующий вектор A из 19 991 элемента в 10 000 раз эффективнее фрагмента

```
i=1;e=1; while i<=19991, A(i)=e;e=e+0.1;i=i+1; end
```

основанного на применении оператора цикла для индивидуального прописывания элементов вектора A. Поэтому советуют при программировании М-функций старательно избегать применения массивных циклических вычислений на базе операторов цикла М-языка. Это можно сделать, применив вместо них эффективные групповые операции М-языка. Если же этого сделать нельзя, то есть избежать циклов не удается, то тогда стоит подумать о реализации циклических алгоритмов на языках C/C++, поскольку в системе MATLAB имеется принципиальная возможность подключить для совместной работы даже такие программные модули. Об этом будет подробно рассказано в гл. 8.

Наконец, расскажем о повышении эффективности выполнения М-функций за счет специальной настройки среды MATLAB. Дело в том, что при первой загрузке содержимого М-файла в память компьютера система MATLAB осуществляет синтаксический разбор кода, переводит его во внутренний формат, называемый *псевдокодом* или *P-кодом*, после чего сохраняет полученный псевдокод в памяти компьютера в течение всего сеанса работы с пакетом MATLAB. В результате первый вызов М-функции осуществляется медленнее (обычно ненамного медленнее) повторных вызовов. Можно несколько повысить эффективность первого выполнения М-функции, выполнив команду

```
pcode имя_М-функции(файла)
```

Эта команда транслирует содержимое М-файла в псевдокод и записывает на диск файл с тем же именем, но с расширением .p, после чего загрузка М-функции будет выполняться именно из этого файла и первое исполнение функции пройдет быстрее. Кроме того, можно будет стереть файл с расширением .m, а содержимое оставшегося файла с псевдокодом будет невозможно прочитать в текстовых редакторах, так как формат этих файлов бинарный. В результате можно будет скрыть от любопытных глаз «тайные» детали разработанных вами алгоритмов.

М-функции с переменным числом входных параметров и выходных значений

В определениях М-функций, которые мы рассматривали до сих пор, всегда было строго фиксированное конечное число входных параметров и выходных значений. Однако бывают ситуации, когда полезно иметь одну функцию, которая штатно обрабатывала бы переменное число входных параметров и выходных значений. Например, неплохо иметь функцию, которая может суммировать длины произвольного числа векторов. Такая функция должна уметь распознавать реальное количество параметров, с которыми она вызвана.

В М-языке системы MATLAB такая возможность базируется на использовании массива ячеек. В определении М-функции параметр, через который передается заранее неизвестное число входных аргументов, нужно обозначить ключевым словом `varargin`. Таким ключевым словом обозначается массив ячеек, в который упакованы эти параметры. Функция всегда может узнать истинное число аргументов, упакованных в параметре `varargin`, применив для этого функцию `length`.

Ниже представлен код функции, вычисляющей сумму квадратов длин произвольного количества вектор-строк:

```
function SumLen = NumLength( varargin )
n = length( varargin ); SumLen = 0;
for k = 1 : n
  SumLen = SumLen+varargin{k}(1)^2+varargin{k}(2)^2;
end
```

Если аргумент `varargin` не единственный в списке параметров, то он должен стоять последним. В рассмотренном примере с помощью фигурных скобок мы извлекаем содержимое отдельной ячейки, то есть вектор, а с помощью дальнейшей индексации круглыми скобками извлекаем первую и вторую координаты вектора.

При вызове функции `NumLength` не нужно (и нельзя) упаковывать входные числовые вектор-строки в массив ячеек, так как MATLAB делает это сам. Достаточно перечислить их в качестве фактических параметров через запятую:

```
NumLength( [ 1 2], [3 4] )
ans =
30
```

Теперь вызовем функцию `NumLength` с другим числом параметров:

```
NumLength( [ 1 2], [3 4], [ 5 6] )
ans =
91
```

Функция легко обрабатывает оба этих случая, правильно вычисляя суммарную длину входных векторов.

В определении М-функции переменное число возвращаемых значений упаковывается в массив ячеек, обозначаемый ключевым словом varargout:

```
function varargout = MyFunc3( X )
```

Здесь в массив ячеек с именем varargout можно в теле функции упаковать произвольное число выходных значений. Допустим, что на вход функции MyFunc3 может подаваться в качестве единственного входного параметра массив разных размерностей и размеров. Мы хотим возвращать несколько скаляров, каждый из которых является размером входного массива вдоль одного из его измерений. Так как количество измерений заранее неизвестно, то его можно определить в теле функции динамически и на ходу упаковать все эти скаляры в единственную выходную переменную varargout. Вот решение этой задачи:

```
function varargout = MyFunc3( X )
n = ndims( X );
for i = 1 : n
  varargout(i) = {size(X,i)};
end
```

Здесь функция size(X,i) вычисляет размер массива X вдоль i-го направления (измерения). Ниже показаны два примера использования этой функции:

```
A = [ 1 2 3; 4 5 6];
[ m, n ] = MyFunc3( a );
```

Здесь скаляры m и n примут соответственно значения 2 и 3. А теперь сформируем трехмерный массив C и вызовем для него функцию MyFunc3:

```
B = [ 4 5 6; 9 8 7];
C(:,:,1) = A; C(:,:,2) = B;
[m,n,k]= MyFunc3( C );
```

Скаляры m, n и k примут значения 2, 3, 2. В них помещены размеры трехмерного массива C вдоль всех его измерений. Если бы нас интересовали размеры этого массива вдоль только первых двух измерений, то мы могли бы вызвать функцию MyFunc3 в следующем формате:

```
[m,n]= MyFunc3( C );
```

который является абсолютно корректным с точки зрения синтаксиса М-языка, но при котором третье вырабатываемое функцией MyFunc3 выходное значение теряется.

Контроль входных параметров и выходных значений М-функции

Несовпадение типов и числа фактических (с которыми реально вызывается функция) и формальных параметров приводит к неправильной работе М-функций. Но «пользователь» М-функции (это может быть как другая функция, так и человек) всегда может ошибиться при ее вызове. Поэтому желательно встраивать внутрь кода М-функций проверку типа и количества входных параметров, а также количества выходных значений, в которые «вызывающая сторона» пытается поместить результаты работы функции.

Рассмотрим проблему на примере указанной выше функции `MatrProc1`. Эта функция предполагала использовать в качестве первого и второго аргументов массивы одинаковых размеров. Если пользователь по ошибке задаст фактические параметры в виде массивов разных размеров, то в процессе выполнения функции возникнет ошибка. Чтобы избежать этого, можно в теле функции `MatrProc1` организовать проверку размеров первого и второго параметров:

```
function [ A, B ] = MatrProc1( X1, X2, x )
n1 = ndims(X1); n2 = ndims(X2);
%--- parameters control ----------------------------
if n1 ~= n2
 error('Different dimensions')
else
 for i=1:n1
  if size(X1,i)~=size(X2,i)
  error('Different sizes of 1st and 2nd parameters')
  end
 end
end
%---  calculations  ---------------------------------
A = X1 .* X2 * x;
B = X1 .* X2 + x;
```

Теперь при вызове функции `MatrProc1` с неправильными размерами первого и второго аргументов стандартная функция системы MATLAB `error` будет корректно останавливать всю работу и выводить в командное окно системы MATLAB наши диагностические сообщения (аргументы функции `error`), после чего пользователю останется лишь повторно вызвать функцию `MatrProc1`, но уже с правильными параметрами.

Для большей надежности нужно еще добавить проверку третьего параметра на скалярность (в системе MATLAB скаляры являются матрицами `1 x 1`), что можно выполнить следующим фрагментом кода:

```
[ m ,n ] = size( x );
if ( m ~= 1 | n ~= 1 )
  error('3-d parameter must be scalar')
end
```

Наконец, неплохо проверить общее число параметров, с которыми функция была вызвана. Для этой цели в системе MATLAB специально предусмотрена переменная с именем `nargin`. Ее значением является количество аргументов, фактически переданное функции при ее вызове. Тогда проверка на число параметров выполняется следующим образом:

```
if nargin ~= 3
  error('Bad number of parameters')
end
```

Более того, в системе MATLAB предусмотрена переменная `nargout`, содержащая число возвращаемых значений в текущей форме вызова этой функции. Например, вызов

```
[ s1, s2, s3 ] = MatrProc1( x1, x2, x)
```

предполагает получить аж три возвращаемых значения, в то время как из определения функции следует, что возвращаемых значений у этой функции два. Чтобы предотвратить такой формат вызова функции `MatrProc1` и предупредить пользователя функции о несоответствии числа ожидаемых возвращаемых значений их номинальному числу, можно в теле функции осуществить проверку переменной `nargout` следующим образом:

```
if nargout ~= 2
  error('Must be 2 return values')
end
```

Осуществленные нами проверки приводят к тому, что функцию можно вызвать только с правильным числом входных параметров и возвращаемых значений. Однако ранее мы встречались со встроенными функциями системы MATLAB, которые могли быть вызваны с разным числом входных параметров (и это очень типично). В результате фактически разные варианты работы близкого типа выполняются под одним и тем же именем функции, что весьма наглядно и удобно. Таковой, например, является функция `plot`, имя которой говорит о построении графиков функций. Если бы разные варианты вызовов этой функции пришлось бы осуществлять под разными именами, то от наглядности не осталось бы и следа.

Таким образом, целесообразно при разработке М-функций допускать многовариантность работы при разном числе входных аргументов; это нужно предусмотреть при проверке их числа, и вместо прекращения работы функции следует реализовать разные ветви выполнения. То же касается и числа возвращаемых

значений. В заголовке определения М-функции нужно при этом использовать максимально возможное число как первых, так и вторых.

Так, функция TestFunc2

```
function [ res1, res2] = TestFunc2( var1, var2 )
switch nargin
 case 1
  if nargout == 1, res1 = var1 * 2;
  elseif nargout == 2, res1=var1 * 2; res2=var1 + 3;
  else  error('Must be 1 or 2 return values');
  end
 case 2
  if nargout == 1, res1 = var1 .* var2;
  elseif nargout == 2, res1 = var1 .* var2; res2 = var1 + 3;
  else  error('Must be 1 or 2 return values');
  end
 otherwise error('Must be 1 or 2 parameters');
end
```

допускает много вариантов заранее предусмотренных форматов вызовов.

Для краткости мы здесь опустили проверку размеров входных параметров, подробно рассмотренную выше. Кроме того, мы здесь намеренно для большей наглядности выполняем некоторую лишнюю работу: на практике система MATLAB самостоятельно отслеживает ситуацию превышения числа параметров и возвращаемых значений над номинальным их числом.

В заключение отметим, что степень подробности проверок зависит от предназначения функции. Если М-функция пишется для собственного потребления, проверки могут быть менее строгими, так как выполнение М-функций осуществляется в режиме интерпретации под полным контролем системы MATLAB. Ошибочные ситуации автоматически обрабатываются самой этой системой и в командное окно выдается соответствующее диагностическое сообщение.

Но если предполагается передача функции для внешнего потребления, то проверки нужно сделать более жесткими, так как внешнему пользователю трудно разобраться во всех деталях работы вашей функции.

Система MATLAB штатно поставляется с большим числом встроенных М-функций. Текст этих функций выводится в командное окно командой

```
type имя_функции
```

так что всегда можно как изучить детали реализации функцией некоторого алгоритма, так и научиться разным приемам программирования, в частности способам и глубине проверок входных параметров и выходных значений.

Рассмотрим текст функции repmat, введя и исполнив команду

```
type repmat
```

в результате чего получаем ее полный текст:

```
function B = repmat(A,M,N)
%REPMAT Replicate and tile an array.
% B = REPMAT(A,M,N) replicates and tiles the matrix a
% to produce the M-by-N block matrix B.
%
% B = REPMAT(A,[M N]) produces the same thing.
%
% B = REPMAT(A,[M N P ...]) tiles the array a to
% produce a M-by-N-by-P-by-... block array. a can be N-D.
%
% REPMAT(A,M,N), when a is a scalar, is commonly used to
% produce an M-by-N matrix filled with A's value.
% This can be much faster than A*ONES(M,N)
% when M and/or N are large.
%
% Example:
%   repmat(magic(2),2,3)
%   repmat(NaN,2,3)
%
% See also MESHGRID.
%
%   Copyright (c) 1984-98 by The MathWorks, Inc.
%   $Revision: 1.11 $  $Date: 1997/11/21 23:30:13 $

if nargin < 2
   error('Requires at least 2 inputs.')
elseif nargin == 2
  if prod(size(M))==1
     siz = [M M];
  else
     siz = M;
  end
else
  siz = [M N];
end

if length(A)==1
% This produces the same answer as B = A(ones(siz));
% but uses less memory.
 B = ...
```

```
 reshape(A(ones(1,siz(1)),ones(1,prod(siz(2:end)))),siz);
elseif ndims(A)==2 & length(siz)==2
  [m,n] = size(A);
  mind = (1:m)';
  nind = (1:n)';
  mind = mind(:,ones(1,siz(1)));
  nind = nind(:,ones(1,siz(2)));
  B = A(mind,nind);
else
  Asiz = size(A);
  Asiz = [Asiz ones(1,length(siz)-length(Asiz))];
  siz = [siz ones(1,length(Asiz)-length(siz))];
  for i=length(Asiz):-1:1
    ind = (1:Asiz(i))';
    subs{i} = ind(:,ones(1,siz(i)));
  end
  B = A(subs{:});
end
```

Текст функции `repmat` весьма полезен для изучения. Во-первых, здесь показан эффективный алгоритм решения задачи о повторении (*репликации*) заданной матрицы `A` в вертикальном направлении `M` раз и в горизонтальном направлении `N` раз. Вот пример работы функции `repmat` (см. рис. 6.11).

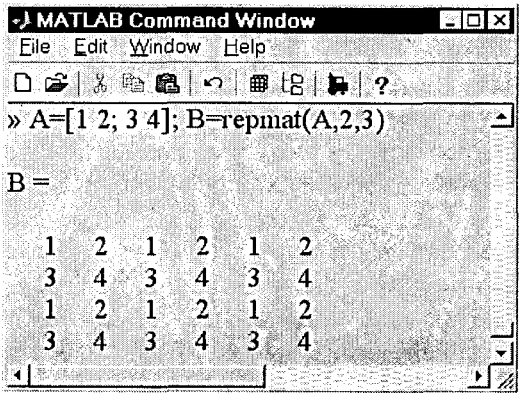


Рисунок 6.11

Для решения этой задачи здесь активно используются высокоэффективные операции М-языка, такие, как множественная индексация и индексация матрицами. Этот сложный для первичного изучения вопрос был нами подробно рассмотрен в гл. 1. Если сейчас он у вас по-прежнему вызывает некоторые затруднения в понимании, то вернитесь к материалу гл. 1 для повторного изучения.

Во-вторых, данный пример показывает, что весьма значительный процент от общего объема кода занимают многочисленные проверки входных параметров и выходных значений.

В-третьих, видно, что очень большое значение уделяется документированию кода. Это важно как для сторонних пользователей функции, так и для самих разработчиков, поскольку помогает им в процессе разработки и отладки алгоритма и деталей кодировки.

Практические советы по разработке и отладке М-функций

Разрабатывая функцию, вы в первую очередь разрабатываете алгоритм решения некоторой задачи, после чего переводите его на формальный язык кодирования, которым и является М-язык. Несмотря на довольно высокую наглядность М-языка (отсутствуют низкоуровневые конструкции, близкие к машинным командам), все равно это формальный язык. По прошествии времени детали разработок забудутся и для модификации функции придется все вспоминать снова.

Чтобы упростить процесс дальнейшей модификации функции, а также ее отладки на стадии, когда еще не удалось добиться правильной работы, в текст функции вставляют комментарии. Объем таких комментариев может быть весьма большим, что мы выше наблюдали на примере штатной функции `RepMat`.

Напомним еще раз, что комментарии могут занимать отдельные строки, начинающиеся с символа `%`, после которого следует текст комментария. Также комментарии можно располагать в конце любой строки кода, поскольку интерпретатор М-языка, встретив знак `%`, считает все символы после него просто комментарием (а не командами, подлежащими переводу в машинную форму и исполнению).

Особую роль в системе MATLAB имеют комментарии, располагающиеся в смежном наборе строк сразу за заголовком определения функции. Весь этот набор строк выводится в командное окно системы MATLAB при исполнении команды

```
help имя_М-функции
```

Поскольку такую команду в первую очередь будут применять пользователи функции (а не разработчики), то желательно расположить в этих комментариях описательную информацию и сведения о правильном вызове этой функции.

Теперь подробно остановимся на вопросе об отладке М-функций, то есть на приемах, с помощью которых можно выявить месторасположение ошибок и их причину. Система MATLAB осуществляет серьезную помощь в этом процессе. В частности, при возникновении ошибки в процессе выполнения М-функции в командное окно выводится приблизительное диагностическое сообщение

(не следует переоценивать качество такой диагностики) и номер строки, в которой с точки зрения системы MATLAB произошла ошибка.

Другим, более развитым способом отладки функции является применение *точек останова* (*Breakpoints*) и *пошаговое выполнение* тела функции. Для этого применяют встроенные возможности редактора-отладчика системы MATLAB. О том, что уже многократно применявшийся нами редактор (в нем набираем текст функций и с помощью меню сохраняем в файле) заодно является и *отладчиком*, говорит даже заголовок его окна:

```
Matlab Editor / Debugger
```

так как `debugger` в переводе с английского означает «отладчик».

Чтобы поставить *точку останова* на какой-либо строке кода функции, туда нужно поместить курсор и нажать клавишу F12 (повторное нажатие этой клавиши убирает точку останова). Вместо нажатия этой клавиши можно выполнить команду меню

```
Debug | Set/Clear Breakpoint
```

но все же быстрее это можно выполнить нажатием клавиши. После этого в строке слева появляется красный кружок, указывающий на то, что в данной строке проставлена точка останова. После этого, не закрывая окна редактора/отладчика (Editor/Debugger), переключаем фокус ввода с клавиатуры в командное окно системы MATLAB и запускаем обычным образом функцию на выполнение. После этого и произойдет останов выполнения функции прямо на строке, в которой поставлена точка останова.

Теперь мы можем просматривать фактические значения входных параметров функции, текущие значения глобальных и локальных переменных, а также значения выражений. Чтобы просмотреть значение переменной, достаточно подвести курсор к ее имени в тексте функции, после чего на экране появится всплывающий желтый прямоугольник со значением переменной внутри его (см. рис. 6.12).

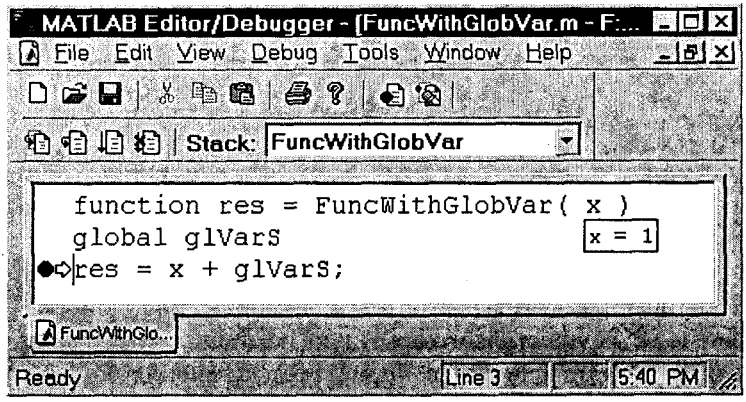


Рисунок 6.12

Далее, нажимая клавишу F10, мы можем выполнять функцию построчно, каждый раз проверяя результаты такой пошаговой работы функции. В результате всегда можно «окружить ошибку» и выявить ее причину.

Изменив текст функции и устранив выявленную ошибку, запускаем функцию на выполнение, в результате чего либо удостоверяемся в ее правильной работе, либо находим новую ошибку. Желательно продумать методику отладки, запуская функцию на выполнение с разными значениями аргументов и разными значениями глобальных переменных. В результате такого итерационного отладочного процесса приходят к правильно работающим функциям.

Глава 7

Примеры конкретных разработок М-функций

Функции, работающие со временем и датами

В состав пакета MATLAB входит изрядное число стандартных функций, возвращающих сведения о текущем времени и текущей дате. Сначала кратко рассмотрим эти функции, а затем разработаем на их базе собственную М-функцию, осуществляющую достаточно сложную (с учебной точки зрения) обработку дат и времени.

Начнем с функции `clock`, которая возвращает текущие год, месяц, день, час, минуту, секунду в качестве последовательных элементов числового вектора `1 x 6`. На рис. 7.1 показан результат работы этой функции.

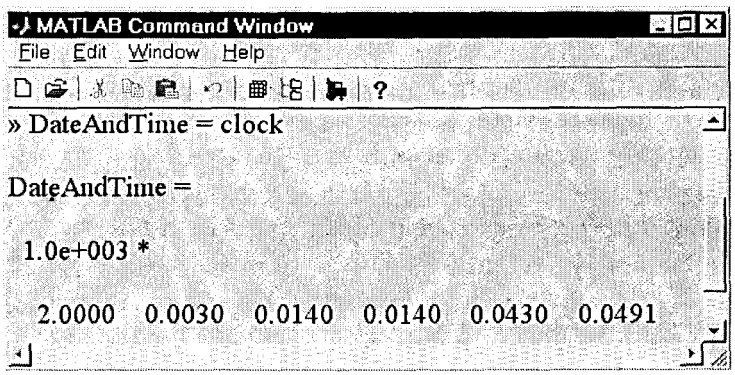


Рисунок 7.1

Первым элементом числового вектора `DateAndTime` является значение текущего года (2000 год), затем указан текущий месяц года – третий (то есть март), после чего идут текущие день, час, минута и секунда. Такой формат задания даты и времени называется *векторным форматом*. Легко заметить, что первые пять элементов этого вектора всегда являются целыми и только шестой элемент может быть дробным числом. Если и для секунд достаточно целых показаний, то тогда целесообразно вызывать функцию `clock` в виде аргумента округляющей числовые значения функции `fix`:

```
fix( clock )
ans =
   2000   3   14   14   56   37
```

В отличие от рассмотренной функции clock функция now возвращает дату и время в другом формате – так называемом внутреннем числовом формате (то есть в виде одного дробного числа). Эта функция возвращает число, целая часть которого задает дату, а дробная часть – время.

Дата представляется в виде номера текущего дня, отсчитанного от фиксированной базовой даты (это начало нулевого года). Таким образом, 1 января 0000 года имеет номер 1, 2 февраля 0000 года имеет номер 33, дата 14 марта 2000 года имеет номер 730559 и т. д.

Время задается в виде дробной части возвращаемого значения. Величина этой дробной части показывает долю от полного дня (24 часа ровно), которую составляет при отсчете от полуночи текущее время. Например, если выполнить команды

```
format long
now
ans =
    7.305596406160417e+005
```

то получается дробное число, целая часть которого равна 730559 (то есть 14 марта 2000 года), а дробная часть 0.640... означает текущее время в часах, равное

```
24 * 0.640... = 15.36...
```

что соответствует 15 часам и приблизительно 22 минутам.

Ясно, что выражения fix(now) или floor(now) возвращают только дату, а выражение rem(now, 1) возвращает только время.

Для возврата одной даты (без текущего времени) в текстовом формате предназначена функция date (см. рис. 7.2).

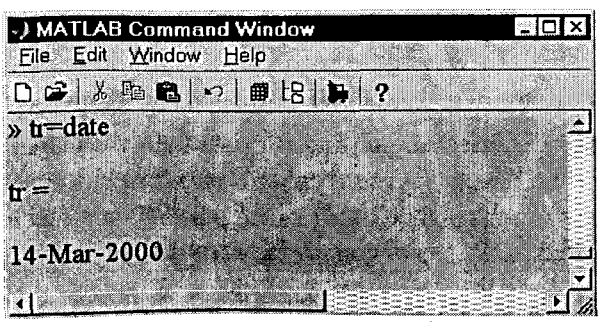


Рисунок 7.2

Преобразовываются даты из одного формата в другой функциями datenum, datestr и datevec. Функция datenum преобразовывает входной строковый

формат даты в выходной внутренний числовой формат. Функция datestr осуществляет обратное преобразование:

```
datenum('14-Mar-2000')
ans =
    730559
datestr(730559)
ans =
14-Mar-2000
```

Если входной параметр функции datestr содержит дробную часть, то вырабатываемая этой функцией строка содержит также сведения о времени. Например,

```
datestr(730559.423567)
ans =
14-Mar-2000 10:09:56
```

Имеется возможность управлять форматом выходной строки с помощью второго (необязательного) параметра функции datestr. Например,

```
datestr(730559.423567, 2)
ans =
03/14/00
```

Выяснить все возможные варианты строковых форматов можно, выполнив команду

```
help datestr
```

Функция datevec расщепляет входной параметр (может иметь как внутренний числовой формат, так и строковый формат) на шесть числовых элементов векторного формата:

```
format short
datevec(7.305596406160417e+005)
ans =
 1.0e+003 *
 2.0000    0.0030    0.0140    0.0150    0.0220    0.0292
```

К стандартным функциям, которые работают со временем и датами, относятся также функции tic, toc, cputime, etime, calendar, weekday, eomday, часть из которых нам уже известна (tic, toc), другие функции элементарны (например, функция calendar выводит в командное окно календарь на запрошенный или текущий месяц), а по остальным легко получить встроенную в систему MATLAB справку.

Мы же теперь воспользуемся изученными стандартными функциями для работы с датами и временем, чтобы разработать и написать код собственной М-функции. Вот практическая ситуация, когда требуется наше вмешательство. По малозначительному недосмотру разработчиков пакета MATLAB стандартная

функция datenum не проверяет входную дату в строковом формате на корректность. В итоге возможна следующая не совсем красивая ситуация (см. рис. 7.3).

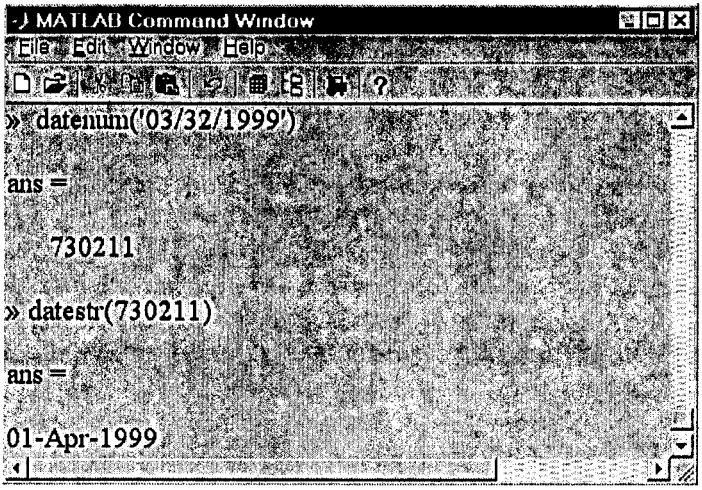


Рисунок 7.3

Мы ввели несуществующую дату 32 марта 1999 года, которую функция datenum «молча съела», преобразовав ее в число 730211. Это число, в свою очередь, функцией datestr, обратной к исходной функции datenum, преобразовывается уже в 1 апреля 1999 года. Ясно, что ситуация двусмысленная. Лучше всего не допускать в функциях типа datenum неправильных значений входного параметра, когда в качестве числа месяца указывается недопустимое значение. Напишем собственную функцию MyStrDateToNumDate, которая проверяет входной параметр на корректность.

В М-файл с именем MyStrDateToNumDate поместим не только функцию с таким же именем, но и еще три вспомогательные внутренние функции (их часто в терминах системы MATLAB называют подфункциями).

Поясним одну маленькую деталь, связанную с *григорианским календарем*: последний год столетия считается *високосным* (по-английски это *leap year*), только если он кратен 400. Остальные детали алгоритма решения поставленной задачи достаточно очевидны и легко извлекаются непосредственно из текста функций:

```
function res = MyStrDateToNumDate( D )
% Our own version with input date checking

if IsCorrect( D )
  res = datenum( D )
else
  error('Incorrect input date!')
end
```

```
%-----  subfunction IsCorrect -----------

function ret = IsCorrect( D )
[Y,Month,Day,H,Minute,Sec]=MyDateVec( D );
if Day <= RightNumOfDays( Month, Y )
  ret = 1; %true
else
  ret = 0; %false
end

%--- subfunction RightNumOfDays ----------

function num = RightNumOfDays( M, Y )
if M==4 | M==6 | M==9 | M==11, num = 30;
elseif M==2
  if IsLeapYear( Y ), num = 29;
  else num = 28;
  end
else num = 31;
end

%------  subfunction IsLeapYear  ---------

function ret = IsLeapYear( Y )
ret = rem(Y,400)==0 | rem(Y,100)~=0 & rem(Y,4)==0;
```

Работа основной (видимой извне) функции MyStrDateToNumDate опирается на функцию IsCorrect, а та, в свою очередь, использует функции RightNumOfDays и IsLeapYear, имена которых говорят (на английском языке) сами за себя: первая из них определяет правильность даты в месяце на основе сведений о високосности года. Последняя информация поставляется функцией IsLeapYear.

Очень важная деталь: вместо стандартной функции datevec, которая расщепляет строковые форматы даты на элементы вектора [Y,Month,Day,H,Minute,Sec], мы применяем собственную функцию MyDateVec, поскольку именно стандартная функция datevec, игнорируя неправильность формата входной даты, преспокойно осуществляет некорректное расщепление (с переносом дат на следующий месяц).

Собственную функцию обработки строковых данных MyDateVec мы не помещаем в файл MyStrDateToNumDate.m, а располагаем ее в файле MyDateVec.m. Эту функцию мы рассмотрим в следующем подразделе, посвященном программированию функций, обрабатывающих текстовые данные.

Рассмотренный пример иллюстрирует простую мысль, что как бы ни хороша была «фирменная система» (в частности, система MATLAB), всегда остается обширное поле для самостоятельного творчества.

Обработка текстов

Штатные функции обработки текстов (строк), входящие в стандартную поставку системы MATLAB, были нами ранее рассмотрены в гл. 3. Этот набор является достаточно полным для выполнения типичных мелкомасштабных символьных операций. Однако если требуется решить специфическую крупномасштабную задачу по обработке текстов, то нужно разрабатывать собственные функции, которые могут в своей работе опереться как на базовые операции М-языка, так и на стандартные строковые функции.

В качестве примера мы будем разрабатывать собственную функцию MyDateVec, о которой зашла речь в предыдущем подразделе. Заранее скажем, что практически полезные и надежные в работе функции по обработке текстовой информации обычно являются весьма и весьма сложными. Кто сомневается в этом, может ознакомиться с текстом стандартной функции datevec, которая и выполняет преобразование даты из текстового формата в набор вещественных чисел. Вот этот текст, полученный с помощью команды type datevec:

```
function [y,mo,d,h,mi,s] = datevec(t)
%DATEVEC Date components.
% C = DATEVEC(T) separates the components of date strings
%   and date numbers into date vectors containing [year
%   month date hour mins secs] as columns. If T is a date
%   string, it must be in one of the date formats
%   0,1,2,6,13,14,15,16 (as defined by DATESTR). Date
%   strings with 2 character years are interpreted as if
%   they are in the current century.
%
% [Y,M,D,H,MI,S] = DATEVEC(T) returns the components of
%   the date vector as individual variables.
%
% Examples
%     d = '12/24/1984';
%     t = 725000.00;
%     c = datevec(d) or c = datevec(t) produce
%         c = [1984 12 24 0 0 0].
%     [y,m,d,h,mi,s] = datevec(d) returns y=1984, m=12,
%                                 d=24, h=0, mi=0, s=0.
%
```

```
%   See also DATENUM, DATESTR, CLOCK.
%   Copyright (c) 1984-98 by The MathWorks, Inc.
%   $Revision: 1.15 $  $Date: 1997/11/21 23:48:03 $

if nargin < 1
  error('Not enough input arguments.');
end

[date_row,date_col] = size(t);

if isstr(t)
  pm = -1; %means am or pm is not in datestr
  dts = zeros(date_row,6);
  siz = [date_row 1];
  for count = 1 : date_row
   % Convert date input to date vector
   % Initially, the six fields are all unknown.
   c(1,1:6) = NaN;
   d = [' ' lower(t(count,:)) ' '];
   % Replace 'a ', 'am', 'p ' or 'pm' with ': '.
   p = max(find(d == 'a' | d == 'p'));
   if ~isempty(p)
    if (d(p+1)=='m'|d(p+1)==' ') & d(p-1) ~= lower('e')
      pm = (d(p) == 'p');
      if d(p-1) == ' '
        d(p-1:p+1) = ':  ';
      else
        d(p:p+1) = ': ';
      end
    end
   end

   % Any remaining letters must be in the month field;
   % interpret and delete them.
   p = find(isletter(d));
   if ~isempty(p)
     k = min(p);
     if d(k+3) == '.', d(k+3) = ' '; end
     M = ['jan'; 'feb'; 'mar'; 'apr'; 'may'; 'jun'; ...
          'jul'; 'aug'; 'sep'; 'oct'; 'nov'; 'dec'];
     c(2) = find(all((M == d(ones(12,1),k:k+2))'));
     d(p) = setstr(' '*ones(size(p)));
```

```
end

% Find all nonnumbers.
p = find((d < '0' | d > '9') & (d ~= '.'));

% Pick off and classify numeric fields, one by one.
% Colons delinate hour, minutes and seconds.
k = 1;
while k < length(p)
 if d(p(k)) ~= ' ' & d(p(k)+1) == '-'
   f = str2num(d(p(k)+1:p(k+2)-1));
   k = k+1;
 else
   f = str2num(d(p(k)+1:p(k+1)-1));
 end
 if ~isempty(f)
  if d(p(k))==':' | d(p(k+1))==':'
   if isnan(c(4))
    c(4) = f; %hour
    if pm==1&f~=12%Add 12 if pm spec.and hour isn't 12
      c(4) = f+12;
    elseif pm == 0 & f == 12
      c(4) = 0;
    end
   elseif isnan(c(5))
    c(5) = f; %minutes
   elseif isnan(c(6))
    c(6) = f; %seconds
   else  error(['Too many time fields in ' t])
   end
  elseif isnan(c(2))
   if f > 12 error([num2str(f) ...
               ' is too large to be a month.'])
   end
   c(2) = f;  %month
  elseif isnan(c(3))
   c(3) = f;  %date
  elseif isnan(c(1))
   if (f >= 0) & (p(k+1)-p(k) == 3)
     clk = clock;
     c(1)=f+floor(clk(1)/100)*100;%year in cur. cent.
   else
```

```
        c(1) = f; %year
       end
      else
       error(['Too many date fields in ' t])
      end
     end
     k = k+1;
    end %while

    if sum(isnan(c)) >= 5
      error(['Cannot parse date ' t])
    end

    % If the any of the day fields have been set, set an
    % unspecified year to the current year
    if isnan(c(1)) & any(~isnan(c(2:3)))
     clk = clock; c(1) = clk(1);
    end

    % If any field has not been specified, set it to zero.
    p = find(isnan(c));
    if ~isempty(p), c(p) = zeros(1,length(p)); end

    dts(count,:) = c;
   end %for
   c = dts;
else
   siz = size(t);
   c = dvcore(86400*t);
end

%Make sure time part is properly rounded, the day number
%is within range, and the first five fields are integers.
maxc = ones(size(c,1),1)*[24 60 60];
[e,col] = find(any((c(:,4:6) >= maxc)')' | ...
                any((c(:,3)>eomday(c(:,1),c(:,2)))')'|...
                any((c(:,1:5) ~= floor(c(:,1:5)))')');
if ~isempty(e),
 dn = datenum(c(e,1),c(e,2),c(e,3),c(e,4),c(e,5),c(e,6));
 t = datevec(dn);
 if dn < 1, %Time only
   c(e,4:6) = t(:,4:6);
```

```
 else
   c(e,:) = t;
 end
end

if nargout <= 1,  y = c;
else
  y = reshape(c(:,1),siz);
  mo = reshape(c(:,2),siz);
  d = reshape(c(:,3),siz);
  h = reshape(c(:,4),siz);
  mi = reshape(c(:,5),siz);
  s = reshape(c(:,6),siz);
end
```

Этот текст весьма поучителен в плане техники программирования М-функций, обрабатывающих строки. Наша же задача сейчас состоит в том, чтобы разработать собственную версию этой функции, которая не осуществляла бы ненужного преобразования месяцев в случае неправильного задания даты, о чем и шла речь в предыдущем пподразделе.

Наша разработка будет отличаться от представленной стандартной функции исправлением этого недостатка. Но еще она будет сильно упрощенной версией, так как в учебнике в целях наглядности и компактности изложения невозможно обсудить и детально реализовать слишком объемную разработку.

Одна из самых серьезных проблем, с которыми приходится сталкиваться при обработке текстов, заключается в том, что на вход функции может быть подан абсолютно произвольный набор символов, отличающийся от ожидаемого в каком угодно отношении. Именно обработка любых нештатных отклонений во входном тексте и представляет собой очень сложную задачу. В значительной степени поэтому представленный выше код стандартной функции `datevec` является таким громоздким.

Мы сознательно (в учебных целях) сильно упростим себе жизнь тем, что предположим, что на вход нашей собственной функции `MyDateVec` будут подаваться только абсолютно корректные и однозначные текстовые представления даты в формате

```
'mm/dd/yyyy'
```

то есть сначала расположены два числовых символа месяца, затем через косую черту идут два символа дня месяца и, наконец, снова через косую черту – четыре числовых символа года. Это означает, что мы свободны от проверки возможных неконтролируемых отклонений от заданного формата, и это самым радикальным образом упрощает задачу. Еще раз повторим, что такое упрощение переводит разрабатываемую нами функцию в разряд чисто учебных иллюстративных

функций. Однако наша цель, связанная с получением функции, осуществляющей проверку корректности входных дат, будет при этом достигнута.

Вот код нашей функции:

```
function [y,m,d,x,y,z] = MyDateVec( strData )
% tutorial function with correct date checking

sv = findstr( strData, '/' );
%--- string representation of m,d,y ------
strM = strData( 1 : (sv(1)-1) );
strD = strData( (sv(1)+1) : (sv(2)-1) );
strY = strData( (sv(2)+1) : end );
%--- convert them to numbers -------------
m = str2num( strM );
d = str2num( strD );
y = str2num( strY );
x=0;y=0;z=0;
```

Конечно, совсем нетрудно было бы поставить простейшие проверки типа `ischar(strData)` или `isempty(sv)` и т. д. Но в нашем предположении о безошибочности входных данных они не нужны, а в общем случае произвольных отклонений от штатного формата их все равно окажется недостаточно.

Теперь запишем функцию `MyDateVec` в файл `MyDateVec.m` и вызовем разработанную в предыдущем подразделе функцию `MyStrDateToNumDate` (см. рис. 7.4).

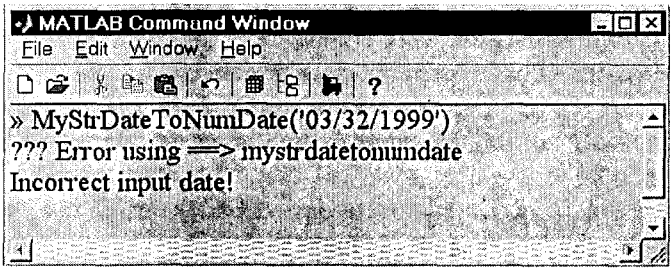


Рисунок 7.4

Функция `MyStrDateToNumDate` не пропускает на обработку некорректные входные даты в отличие от стандартной функции `datenum` (см. рисунок в предыдущем подразделе). Именно эту цель мы и преследовали, разрабатывая функцию `MyStrDateToNumDate`, так что нужный результат достигнут.

Функции для работы с файлами данных

В гл. 3 были изучены стандартные функции системы MATLAB для записи информации в бинарные и текстовые файлы. Это очень удобные в использовании функции, так как с помощью компактных выражений с этими функциями может быть выполнен большой объем работы. Это важно для работы в интерактивном режиме.

Тем не менее стандартные файловые функции системы MATLAB не могут осуществлять непосредственную запись в файлы массивов структур и ячеек. В этом случае для каждого конкретного типа массива структур или ячеек нужно написать собственную М-функцию, выполняющую эту работу. Так как только разработчик знает устройство этих массивов, то бинарные файлы с их содержимым имеют так называемый *закрытый*, или *частный* (*private*, *custom*), формат.

В гл. 3, когда мы изучали массивы ячеек, был для примера создан массив ячеек `MyCellArray`. Напомним его устройство:

```
MyStruct = struct('field1',[ 1 2 3],'field2','Hello');
MyCellArray( 1, 1 ) = { 'Bonjour!' };
MyCellArray( 1, 2 ) = { [ 1 2 3; 4 5 6; 7 8 9 ] };
MyCellArray( 2, 1 ) = { MyStruct };
MyCellArray( 2, 2 ) = { [ 9 7 5 ] };
```

Итак, массив `MyCellArray` размером `2 x 2` состоит из четырех элементов разных типов. Предположим, что создается достаточно большая программа для обработки массива ячеек такого типа. В процессе обработки размер массива не меняется, как не меняются и типы входящих в них элементов. Допустимы лишь следующие изменения, которые мы сейчас перечислим.

Элемент `MyCellArray(1,1)` всегда содержит текстовую строку (то есть вектор-строку символов), у которой могут меняться как ее длина, так и конкретное содержимое. Элемент `MyCellArray(1,2)` содержит числовые квадратные матрицы `m x m`, где целое число `m` может быть разным на разных стадиях работы. Элемент `MyCellArray(2,1)` всегда является структурой `MyStruct`, у которой могут меняться длины ее числовых и текстовых полей. И наконец, элемент `MyCellArray(2,2)` является числовой вектор-строкой произвольной длины.

Мы будем записывать содержимое такого массива ячеек, жестко опираясь на сформулированный формат (устройство) этого массива. В частности, мы всегда записываем в бинарный файл элементы массива в перечисленном выше порядке. Помимо содержимого каждого элемента в файл будет записываться вспомогательная информация, характеризующая переменные атрибуты элементов (длины, размеры).

Непосредственно перед содержимым элемента `MyCellArray(1,1)` будет записываться количество элементов в его текстовой строке. Перед вторым элементом (порядок записи представлен выше) прописываем размер квадратной мат-

рицы, перед третьим пишем подряд два числа – размеры числового и текстового полей структуры, и перед четвертым пишем размер числовой вектор-строки.

После представленных объяснений приступаем к написанию двух функций. Функция WriteMyCellArray пишет в бинарный файл MyBinCellFile.qwe (расширение файла выбрано произвольно) содержимое массива ячеек, а читает это содержимое из файла и восстанавливает сам массив в памяти компьютера функция ReadMyCellArray.

Текст функции WriteMyCellArray очевиден, так как в точности соответствует представленным выше пояснениям:

```
function WriteMyCellArray( Ar )
%write custom binary file

f1 = fopen( 'MyBinCellFile.qwe', 'wb' );

%1.-- (1,1) element: ---------
len1 = length( Ar{1,1} );
fwrite( f1, len1, 'float64' );
fwrite( f1, double(Ar{1,1}), 'float64' );

%2.-- (1,2) element: ---------
[m,n] = size( Ar{1,2} );
fwrite( f1, m, 'float64' );
fwrite( f1, Ar{1,2}, 'float64' );

%3.-- (2,1) element: ---------
k1 = length( Ar{2,1}.field1 );
k2 = length( Ar{2,1}.field2 );
fwrite( f1, k1, 'float64' );
fwrite( f1, k2, 'float64' );
fwrite( f1, Ar{2,1}.field1, 'float64' );
fwrite( f1, double(Ar{2,1}.field2), 'float64' );

%4.-- (2,2) element: ---------
len2 = length( Ar{2,2} );
fwrite( f1, len2, 'float64' );
fwrite( f1, Ar{2,2}, 'float64' );

fclose( f1 );
```

Функция ReadMyCellArray, предназначенная для чтения содержимого массива ячеек заданной конструкции из бинарного файла, еще и возвращает реконструированный массив:

```
function CellAr = ReadMyCellArray
```

```
%read custom binary file

f1 = fopen( 'MyBinCellFile.qwe', 'rb' );

%1.-- (1,1) element: ---------
len1 = fread( f1, [1,1], 'float64' );
CellAr(1,1)={ char(fread(f1,[1,len1],'float64')) };

%2.-- (1,2) element: ---------
m = fread( f1, [1,1], 'float64' );
CellAr(1,2)={ fread(f1,[m,m],'float64') };

%3.-- (2,1) element: ---------
k = fread( f1, [1,2], 'float64' );
vd = fread( f1, [1,k(1)], 'float64' );
vs = char( fread( f1, [1,k(2)], 'float64' ) );
CellAr(2,1)={ struct('field1',vd,'field2',vs) };

%4.-- (2,2) element: ---------
len2 = fread( f1, [1,1], 'float64' );
CellAr(2,2)={ fread(f1,[1,len2],'float64') };

fclose( f1 );
```

Осталось опробовать разработанные функции в работе. Сначала записываем созданный выше массив ячеек `MyCellArray` функцией `WriteMyCellArray` в файл `MyBinCellFile.qwe` на диск:

```
WriteMyCellArray( MyCellArray );
```

Затем удостоверяемся в наличии такого файла на диске компьютера, очищаем командой `clear` содержимое рабочего пространства системы MATLAB и восстанавливаем переменную `MyCellArray` чтением указанного выше файла (см. рис. 7.5).

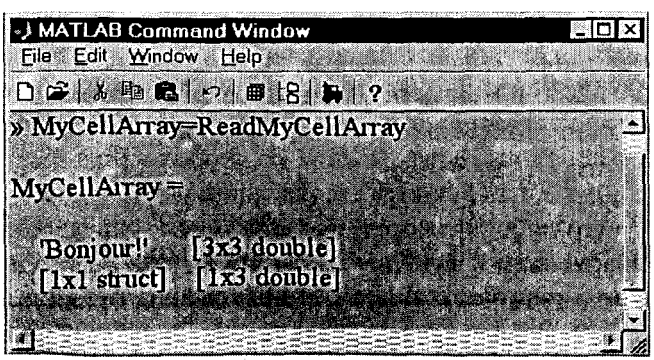


Рисунок 7.5

Отсюда видно, что прочитанная из файла переменная имеет правильную структуру. Теперь проверяем значения отдельных элементов этого массива ячеек:

```
MyCellArray{1,1}
ans =
Bonjour!
MyCellArray{1,2}
ans =
   1  2  3

   4  5  6

   7  8  9
MyCellArray{2,1}
ans =
   field1: [1 2 3]

   field2: 'Hello'
MyCellArray{2,2}
ans =
   9  7  5
```

Значения прочитанных элементов массива ячеек совпадают с их первоначальными значениями, которые записывались в файл. Таким образом, созданные нами функции вполне работоспособны. Но их нельзя воспринимать в качестве универсальных программных средств по разным причинам, из которых на поверхности лежит отсутствие необходимых проверок в теле этих функций (например, отсутствует проверка реального открытия файла – а вдруг этот файл отсутствует на диске).

Динамическое построение графика функции

Для построения двумерных (плоских) графиков функций, то есть графиков функций одной вещественной переменной, мы традиционно применяли самую высокоуровневую функцию `plot`, которая выполняет огромный объем работы самостоятельно и не требует дополнительных усилий со стороны пользователя (программиста). За это функция `plot` пользуется заслуженной популярностью.

Как всегда, недостатки чего-либо являются продолжением определенных достоинств. Не является исключением и функция `plot`. Эта функция не годится для решения задачи о гладком динамическом построении графика функции, ко-

гда линия графика на глазах пользователя должна плавно расти и визуально отражать факт разворачивающихся во времени вычислений.

Самый лучший способ самостоятельно удостовериться в этом заключается в разработке тестовой функции DynDrawOnPlot, использующей функцию plot для динамического построения графика функции. Будем строить график функции sin(cos(sin(cos(x)))) на отрезке от 0 до 10. Вот текст функции DynDrawOnPlot:

```
function DynDrawOnPlot
%test function of plot operation
%-- Initial acts: ----------------
dx=0.001; x=0;
y=sin(cos(sin(cos(x))));
plot([x x],[y y]);
X = x;
Y = y;
%-- Cycle of dynamic graphing: ---
for i=1:10000
  xn=x+dx;
  yn=sin(cos(sin(cos(xn))));
  X=[X xn];
  Y=[Y yn];
  plot(X,Y);
  x=xn;
  y=yn;
  drawnow;
end
```

Ужаснейшее мерцание графического окна и порождаемая этим чрезвычайная медленность перерисовки делают представленную функцию абсолютно неприемлемой. Итак, применять функцию plot для динамического построения графика нецелесообразно.

Для решения указанной задачи нужно использовать графические объекты line, представляющие собой наборы отрезков прямых линий. Объекты типа line порождаются функциями, называемыми *конструкторами* этих графических объектов. Отличительная особенность функций-конструкторов заключается в том, что их имена совпадают с именами объектов. То есть в данном случае речь идет о функциях line. Этим функциям в качестве аргументов задаются массив первых координат (x-координат) концов отрезков прямых и массив вторых координат (y-координат) концов отрезков. Например, пусть массивы

```
X = [ 0, 1, 2 ]; Y = [ 0, 1, 0];
```

задают три точки с координатами (0,0), (1,1) и (2.0). Тогда функция

```
line( X, Y )
```

строит два отрезка прямых линий так, как это показано на рис. 7.6.

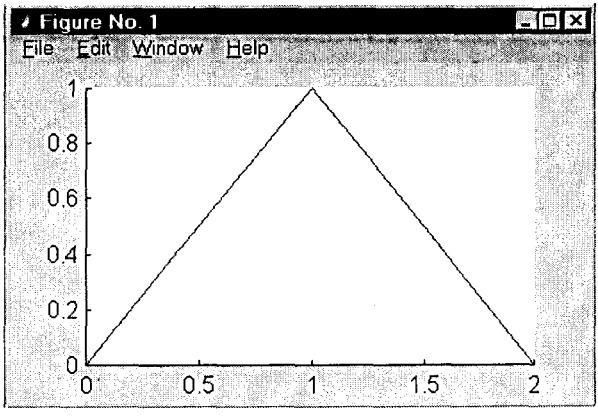


Рисунок 7.6

Этой же функцией нетрудно построить и одиночную точку в том случае, когда массивы X и Y содержат по одному числовому элементу (то есть являются по существу скалярами). Например, выражение

```
line( 1, 0.5, 'Color', 'red' );
```

ставит на предыдущем рисунке красную точку под центральной вершиной треугольника.

Применим графический объект `line` в нашей новой функции `DynDrawOnLine1` для динамического построения графика все той же функции `sin(cos(sin(cos(x))))`. В этом случае можно и нужно фиксировать пределы по осям координат, чтобы они «не прыгали» во время постепенного построения графика. Далее, чтобы не было моргания, задаем свойство `'EraseMode'` графических объектов `line` равным `'none'` (такое свойство при использовании функции `plot` задать невозможно, так как оно игнорируется функцией `plot`). Затем в цикле создаем множество (набор) объектов `line`, вызывая каждый раз конструктор `line` с координатами очередного отрезка прямой:

```
function H=DynDrawOnLine1
%-- Initial acts: ----------------
axis([0 10 0 1]);
dx=0.001; x=0;
y=sin(cos(sin(cos(x))));
H=[];
%-- Cycle of dynamic graphing: ---
for i=1:10000
  xn=x+dx;
  yn=sin(cos(sin(cos(xn))));
  X=[x,xn];
```

```
    Y=[y,yn];
    h=line(X,Y,'EraseMode','none');
    H=[H,h];
    x=xn;
    y=yn;
    drawnow;
  end
```

Вызываем разработанную функцию и наблюдаем во времени процесс построения графика функции, протекающий без каких-либо мерцаний графического окна. Построив график, мы можем изменить размер графического окна мышью, после чего графическое окно самостоятельно масштабирует наш график. Вот как выглядит графическое окно после такого масштабирования (см. рис. 7.7).

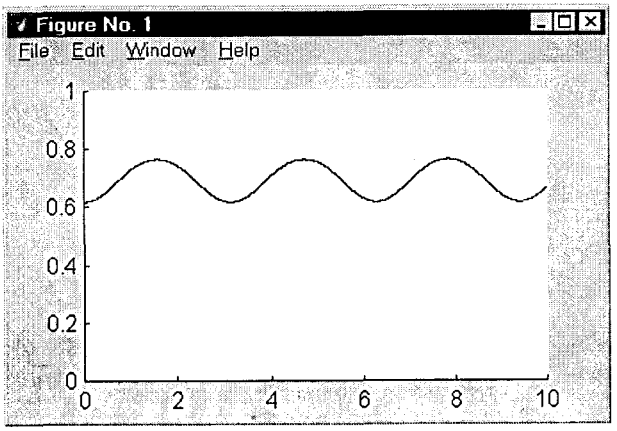


Рисунок 7.7

Итак, мы реализовали динамическое построение графика функции, осуществляя в цикле создание новых объектов типа `line`. Общее количество построенных объектов `line` может быть запрошено через вектор, хранящий все их описатели. Этот вектор возвращается функцией `DynDrawOnLine1` (см. рис. 7.8).

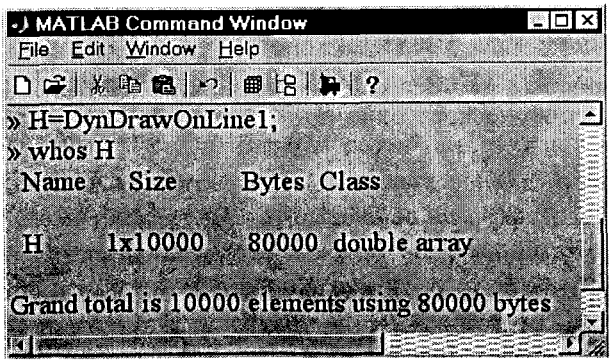


Рисунок 7.8

Отсюда видно, что всего создано 10 000 объектов типа line. В официальной документации к системе MATLAB ничего не сказано о том, есть ли ограничения на количество создаваемых графических объектов, но можно предположить, что излишнее их увеличение в любом случае не способствует эффективности. Так что у представленного решения задачи о динамическом построении графика функции есть некоторые небольшие недостатки. Кроме того, немалое количество дочерних (по отношению к объекту axes) объектов line приводит также к относительно медленной перерисовке содержимого графического окна при изменении его размеров.

Теперь представим другое решение на базе графических объектов line, в котором будет создаваться единственный такой объект. С помощью функции set можно будет динамически изменять его свойства XData и YData, представляющие собой массивы координат точек вершин строящейся ломаной. Вот текст соответствующей функции DynDrawOnLine2:

```
function DynDrawOnLine2
%-- Initial acts: ----------------
axis([0 10 0 1]);
dx=0.001; x=0;
y=sin(cos(sin(cos(x))));
h=line([x x],[y y],'EraseMode','none');
X=x;Y=y;
%-- Cycle of dynamic graphing: ---
for i=1:10000
  xn=x+dx;
  yn=sin(cos(sin(cos(xn))));
  X=[X,xn];
  Y=[Y,yn];
  set(h,'XData',X,'YData',Y );
  x=xn;
  y=yn;
  drawnow;
end
```

Эта функция строит график очень медленно, зато после завершения первичного построения перерисовка графического окна при изменении его размеров происходит очень быстро.

Если же перерисовка не нужна вообще, то можно заранее программно задать нужные размеры графического окна и осуществить следующее быстрое первичное рисование графика (по-прежнему на базе создания единственного объекта line):

```
function DynDrawOnLine3(w,h)
%-- Initial acts: ----------------
```

```
figure('Units','pixels','Position',[100,100,w,h]);
axis([0 10 0 1]);
dx=0.001; x=0;
y=sin(cos(sin(cos(x))));
h=line([x x],[y y],'EraseMode','none');
%-- Cycle of dynamic graphing: ---
for i=1:10000
  xn=x+dx;
  yn=sin(cos(sin(cos(xn))));
  X=[x,xn];
  Y=[y,yn];
  set(h,'XData',X,'YData',Y );
  x=xn;
  y=yn;
  drawnow;
end
```

Здесь быстрое первичное рисование определяется тем, что ранее нарисованные порции графика исключаются из текущего содержимого (то есть из свойств XData и YData) объекта line. Эти порции остаются в пределах графического окна в виде картинки, то есть набора пикселов, но уже не являются содержимым графического объекта line. Можно так образно выразиться, что мы графическим объектом line скользим вдоль строящейся кривой. В результате чего и достигается быстрое первичное построение, но становится невозможной перерисовка.

Все представленные способы запрограммировать задачу о динамическом построении графика функции имеют как достоинства, так и недостатки. В разных конкретных случаях можно предпочесть тот или иной из них.

Вращение трехмерных графиков

Ранее в гл. 2 мы строили красивые, но статичные трехмерные графики (графики вещественных функций двух переменных). Наглядность визуального представления таких графиков только усилится, если осуществить их динамическое вращение вокруг некоторой оси. Тогда создастся полное впечатление, что мы осматриваем поверхность графика функции с разных сторон.

Осуществить такое вращение в системе MATLAB очень просто, так как существует специально предназначенная для этого функция rotate. Эта функция осуществляет однократное вращение графического объекта с описателем H на угол ALPHA вокруг заданной оси вращения. Ось вращения можно задать либо двумя углами THETA и PHI (сферические координаты), либо тремя декартовыми

координатами X, Y, Z вектора этой оси. В обоих случаях ось вращения проходит через начало координат. Поясним дополнительно, что угол PHI задает угол возвышения над плоскостью xy, а угол THETA есть угол между проекцией вектора направления вращения на плоскость xy и положительным направлением оси x.

Итак, существуют два основных шаблона вызова функции rotate:

```
rotate( H, [THETA,PHI], ALPHA )
rotate( H, [X,Y,Z], ALPHA )
```

Если вводить эти выражения с клавиатуры и выполнять их нажатием клавиши Enter, то отрисовка (визуальное вращение) будет производиться немедленно. Если же эти выражения выполняются в порядке своей очереди из текста М-функции, то для немедленной перерисовки графического окна требуется осуществить вызов функции drawnow (не забывайте об этом).

Вот простейший пример пользовательской функции, которая, опираясь на системную функцию rotate, осуществляет достаточно длительное непрерывное вращение графика функции peaks относительно вертикальной оси:

```
function MyRotate1( speed, time, w, h )
figure('Units','Pixels','Position',[100,100,w,h]);
[X,Y,Z] = peaks(30);
hS = surfl(X,Y,Z); shading interp; colormap(copper);
for i=1:time
  rotate( hS, [0, 0, 1], speed );
  drawnow;
end
```

Здесь параметр speed задает угол единичного дискретного вращения, осуществляемого системной функцией rotate, что при непрерывном вращении выглядит как скорость такого вращения. Параметр time задает длительность показа, а параметры w и h – размеры графического окна. Всегда можно подобрать входные параметры функции MyRotate так, чтобы вращение получилось достаточно плавным и наглядно демонстрировало бы график функции двух вещественных переменных со всех сторон.

Например, на дисплее компьютера с разрешением 1024 x 768 неплохой результат получается в случае

```
MyRotate1( 10, 100, 300, 300 );
```

так как при этом достигается достаточно плавный показ вращающегося графика. Отдельный кадр этого «ролика» показан на рис. 7.9.

Еще более плавного показа вращения графика можно добиться, если промежуточные кадры записывать с помощью функции getframe в предварительно выделенный для этого буфер в памяти компьютера, а потом осуществлять максимально плавный показ функцией movie, которой можно даже указать темп

показа (количество кадров в секунду). Всю эту работу выполняет функция MyRotate2:

```
function MyRotate2( speed, time, w, h )
figure('Units','Pixels','Position',[100,100,w,h]);
[X,Y,Z] = peaks(30);
hS = surfl(X,Y,Z); shading interp; colormap(copper);
Buf = moviein(time);
%--- prepare frames for film -------------

for i=1:time
  rotate( hS, [0, 0, 1], speed );
  Buf(:,i)=getframe;
end

%-- show film 10 times with 30 fr/sec -----
movie( Buf, 10, 30 );
clear Buf;
```

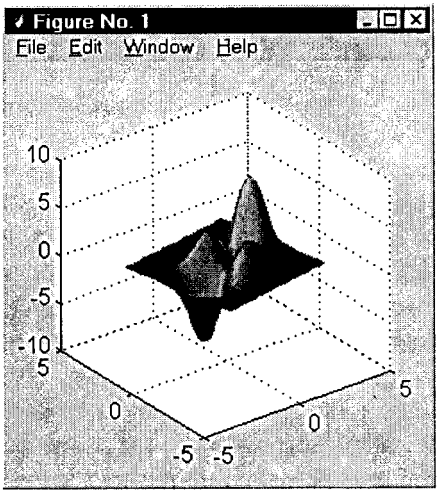


Рисунок 7.9

Очень плавный показ со скоростью 30 кадров в секунду (при наличии достаточно хорошего видеоадаптера) осуществляется следующим вызовом функции MyRotate2:

```
MyRotate2( 5, 72, 300, 300 );
```

где мы задаем дискретный шаг вращения в 5 °, просчитываем все кадры для одного полного оборота (72 раза по 5 °), после чего 10 раз подряд прокручиваем полученный «ролик».

Глава 8
Программирование функций на языке С

Интерфейс MEX-функций с системой MATLAB

Когда стоящую перед нами задачу не удается решить с помощью функций, встроенных в систему MATLAB, приходится разрабатывать собственные функции. Разработка ведется на некотором языке программирования. До сих пор нами разрабатывались функции на М-языке – внутреннем языке программирования пакета MATLAB. Код М-функций, как известно, сохраняется в *текстовых файлах*, имеющих расширением букву m. Именно в таком состоянии эти функции готовы к применению: система MATLAB загружает их в память, преобразовывает в некоторый *промежуточный псевдокод* (*Р-код*), который уже и выполняется далее в *режиме интерпретации*, когда каждая синтаксически законченная конструкция Р-кода заменяется на соответствующий набор машинных инструкций.

У М-языка имеется масса достоинств. Это наглядный и простой язык *высокого уровня* (очень далекий от машинного языка), похожий во многих отношениях на самый распространенный в мире язык программирования BASIC. Работа с М-функциями в интерпретируемом режиме (без промежуточной стадии полной компиляции в машинный код) создает очень быстро реализуемую цепочку «разработка – запуск – отладка (исправление ошибок) – новый запуск» и т. д. М-язык сверхкомпактен по отношению к массивам и операциям над ними: все математические функции работают с массивами так же легко, как и со скалярными величинами. И наконец, богатейший набор встроенных математических и графических функций завершают перечисление чрезвычайно привлекательных особенностей внутреннего языка пакета MATLAB.

И тем не менее у любого языка программирования всегда имеются недостатки, являющиеся продолжением его достоинств. Высокоуровневый характер М-языка не позволяет реализовывать алгоритмы работы со сверхсложными структурами данных (деревья и ветвящиеся списки). Интерпретируемый характер этого языка, способствующий быстроте разработки, препятствует увеличению быстродействия во время выполнения и т. д. Кроме того, как быть, если некоторая функция уже разработана на другом языке программирования. Перера-

ботка на М-язык может оказаться весьма трудоемкой (если вообще возможной), особенно, если нет поддержки со стороны оригинальных разработчиков.

Ясно, что любой математический пакет имел бы существенно большую область применимости, если бы он предоставлял возможность вести комплексные разработки, когда разные части прикладной пользовательской программы разрабатываются на *разных языках программирования* с помощью *различных инструментальных средств*. Именно таковым и является пакет MATLAB! Он позволяет разрабатывать отдельные функции на популярнейших компилируемых языках программирования C, C++ и Фортран. Тем самым уже накопленный фонд программ на этих языках может быть легко подключен к текущему проекту, в котором лишь часть функций должна программироваться на М-языке.

В данном учебном пособии мы будем рассматривать разработку функций только на языке C, предоставив читателю возможность самостоятельно изучить по встроенной системе помощи аналогичные вопросы, касающиеся Фортрана. Ситуация же с языком C++ практически не отличается от рассматриваемой нами в связи с языком C.

Мы здесь выбрали язык C, как наиболее яркий антипод М-языку. Язык программирования C является наиболее *низкоуровневым* (близким по смыслу к машинным инструкциям) и потому самым гибким в вариантах его применения и самым эффективным в смысле быстродействия и экономии использования памяти компьютера. Безусловно, этот язык явно не тянет на чемпиона в легкости его изучения и применения. Однако, являясь самым распространенным языком *системного программирования* и будучи хорошо переносимым между разными программными и аппаратными платформами, он является чемпионом в номинации «Программирование без ограничений» (Programming Unlimited)!

В рамках терминологии пакета MATLAB все функции, разработанные на внешнем по отношению к пакету MATLAB языке программирования, принято называть MEX-функциями (M – это Matlab, а EX – это EXternal, то есть «внешний»).

Еще раз повторим, что мы будем разрабатывать MEX-функции только на языке программирования C, оставив аналогичные вопросы по языкам Фортран и C++ для самостоятельного изучения читателями по встроенной в пакет MATLAB системе помощи. Кроме того, предполагается, что читатель знаком с языком программирования C и средством разработки Microsoft Visual C++ (версий 5 или 6).

На платформе Windows готовые к применению в рамках пакета MATLAB MEX-функции представляют собой скомпилированный машинный код, погруженный в файлы динамических библиотек (`Dynamic Link Libraries`; эти файлы имеют расширение `dll`). Одна MEX-функция соответствует одному файлу динамической библиотеки. При этом имя MEX-функции, используемое для ее вызова в выражениях системы MATLAB, совпадает с именем файла динамической библиотеки.

Например, если мы в командном окне системы MATLAB записываем выражение

```
res = a .* B + MyMexFunction1( a + B );
```

где `MyMexFunction1` – имя MEX-функции, то на диске компьютера в любом известном пакету MATLAB каталоге должен находиться бинарный файл с машинными инструкциями `MyMexFunction1.dll`, разработанный и скомпилированный в среде Microsoft Visual C++. Как раз по имени MEX-функции пакет MATLAB ищет необходимый файл, загружает его в память компьютера в свое адресное пространство, после чего содержимое этого файла становится продолжением «родного машинного кода» пакета MATLAB. В этом смысле MEX-функции *являются бинарными расширениями* кода пакета MATLAB, в то время как M-функции являются его *текстовыми расширениями*.

Если в одном и том же каталоге встречаются как MEX- так и M-функции с одинаковым именем, то интерпретатор пакета MATLAB отдает предпочтение MEX-функции, в то время как команда `help` выбирает M-функцию. Это позволяет документировать информацию по MEX-функции в содержимом одноименной M-функции.

Теперь начнем рассматривать вопросы, связанные с практическими аспектами разработки MEX-функций на языке программирования С. Как реально скомпилировать файл динамической библиотеки в среде компилятора Microsoft Visual C++, будет рассказано в следующем подразделе. Сейчас же сосредоточимся на необходимых фрагментах исходного С-кода.

Для обеспечения возможности взаимодействия ядра системы MATLAB с MEX-функцией, расположенной в динамической библиотеке, последняя с точки зрения языка С должна представлять собой совокупность С-функций, одна из которых имеет фиксированное имя `mexFunction` (здесь нельзя менять даже регистр букв), заданный набор входных параметров и заданное возвращаемое значение.

Итак, повторим еще раз. После компиляции MEX-функция является файлом с расширением `dll`. Имя этого файла и определяет имя MEX-функции, которая вызывается по этому имени из командного окна или из M-функций пакета MATLAB. С точки зрения системы MATLAB такая MEX-функция полностью определяется одним своим именем и представляет собой некоторую неделимую программную единицу. В то же время на этапе разработки MEX-функция представляет собой С-код, состоящий из любого числа С-функций, из которых заданным прототипом обладает только одна функция – функция `mexFunction`.

Именно эту функцию и *экспортирует* динамическая библиотека, то есть имя функции `mexFunction` и ее относительный адрес внутри библиотеки могут быть прочитаны загрузчиком операционной системы. Именно эту функцию и вызывает на самом деле пакет MATLAB в ответ на требование вычислить MEX-функцию. Фактическое имя MEX-функции требуется лишь для поиска и загрузки содержимого файла динамической библиотеки.

Окончательно подытоживаем сказанное: имя MEX-функции совпадает с именем файла динамической библиотеки. Это произвольное имя, выбираемое разработчиком по своему вкусу. Точка входа в динамической библиотеке имеет фиксированное имя – `mexFunction`. Это имя всегда одно и то же, и его менять нельзя.

Ясно, что при разработке MEX-функций особая роль принадлежит С-функции `mexFunction`. Это роль *интерфейса* между внутренним миром пакета MATLAB и внутренним устройством программ на языке С. Функция `mexFunction` принимает входные данные от пакета MATLAB. Она же и возвращает ему результаты работы.

Самый минимальный С-код проекта по созданию MEX-функции пакета MATLAB содержит единственную функцию `mexFunction`. Мы сейчас для простоты ограничимся этим случаем, причем в теле функции `mexFunction` не будем выполнять вообще никаких действий. Таким образом мы реализуем «пустую» (в смысле полезных действий) MEX-функцию, которую так и назовем – `MyEmptyMexFunction`, что в переводе значит «моя пустая функция».

Вот этот минимальный С-код:

```
#include 'mex.h'
void mexFunction( int           nOut,
                  mxArray*      pOut[],
                  int           nIn,
                  const mxArray* pIn[]  )
{
}
```

из которого четко виден заданный прототип интерфейсной функции `mexFunction`. Здесь нужно дать пояснения по типу данных `mxArray`. Это С-структура, определенная в файле `mex.h`, поставляемом вместе с пакетом MATLAB и располагающемся в каталоге `\matlab\extern\include`, где `matlab` – основной каталог системы MATLAB.

Любой массив системы MATLAB представим в виде структуры `mxArray`. Отдельные поля этой структуры хранят информацию о типе массива, его размерности и размерах. Есть и поле, указывающее на область памяти с собственно данными (элементами массива, которые хранятся линейно упорядоченными по столбцам).

Параметры функции `mexFunction` относятся как к входным параметрам вызываемой из пакета MATLAB MEX-функции (они помечены суффиксом `Out`), так и к возвращаемым этой функцией значениям (эти параметры помечены суффиксом `In`).

Так как наша пустая функция не будет принимать никаких входных параметров и не будет возвращать никаких результатов, то нам в теле функции ничего не надо с ними делать. Если мы вызовем эту нашу MEX-функцию из командно-

го окна системы MATLAB (или из некоторой M-функции) с правильным синтаксисом, то есть без входных параметров и возвращаемых значений, то на входе mexFunction будут следующие значения ее входных параметров:

```
nOut = 0
pOut = NULL
nIn = 0
pIn = NULL
```

Разработанную таким образом пустую функцию мы будем использовать в качестве тренировочного материала для прослеживания всего процесса разработки и вызова MEX-функции. Такую MEX-функцию действительно можно будет вызвать из среды пакета MATLAB, и этот вызов отработает нормально без возникновения ошибочных результатов. Состоять этот вызов будет в передаче управления бинарному коду MEX-функции, после чего этот код тут же вернет управление (так как он не содержит никаких полезных действий) в точку вызова.

Теперь пора перейти к вопросам, связанным с компиляцией этой пустой MEX-функции (точнее, с компиляцией соответствующего ей C-кода) и получением файла динамической библиотеки. Для этого мы воспользуемся компилятором Microsoft Visual C++.

Создание и компиляция DLL-проекта в среде Microsoft Visual C++

Работа в среде компилятора Microsoft Visual C++ начинается с создания проекта соответствующего типа. В нашем случае мы должны создать проект типа «Win32 Dynamic-Link Library» и поместить его в каталог с именем MyEmptyMexFunction. То есть имя каталога, в котором будут располагаться файлы проекта, по умолчанию совпадает с именем целевого файла динамической библиотеки.

На рис. 8.1 показано диалоговое окно New оболочки Microsoft Developer Studio (это графическая оболочка компилятора Microsoft Visual C++), в котором следует выбрать тип проекта и имя каталога.

Далее добавляем в проект файл main.c, содержащий пустую функцию mexFunction, описанную в предыдущем подразделе, а также файл MyEmptyMexFunction.def. Последний файл содержит всего лишь одну строку, сообщающую компилятору сведения об экспортируемых из динамической библиотеки функциях (см. рис. 8.2).

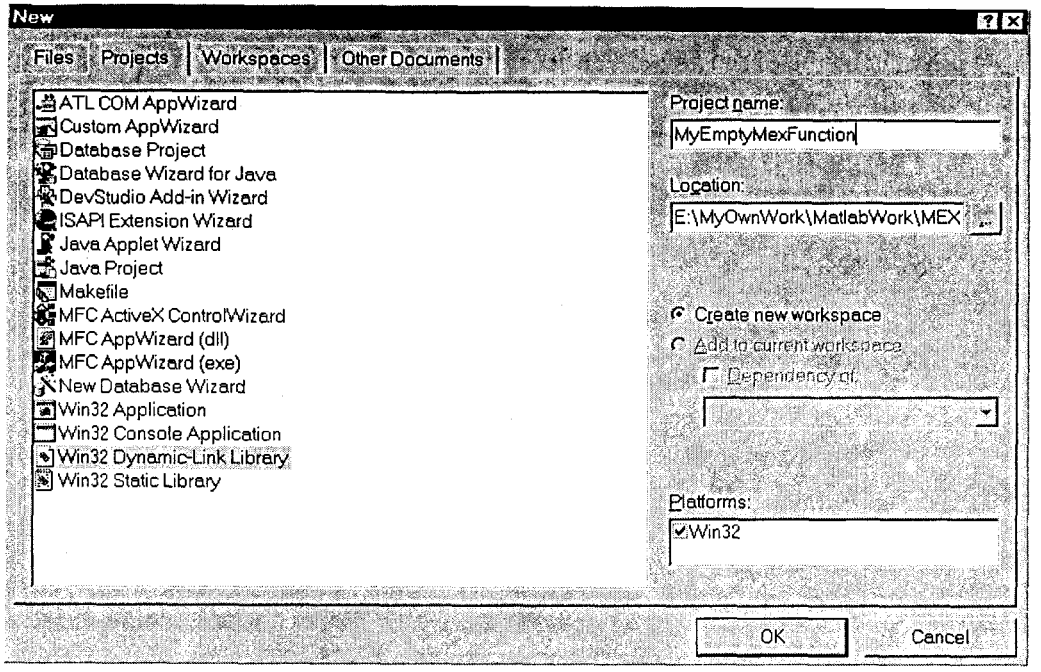


Рисунок 8.1

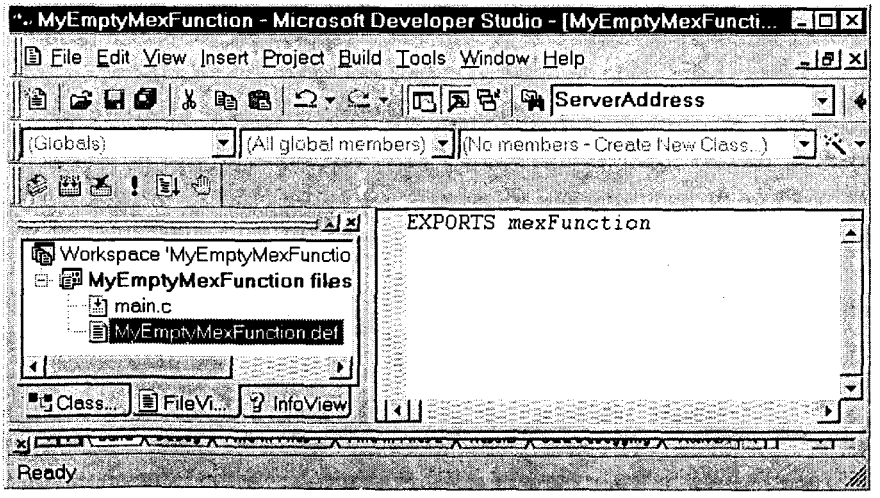


Рисунок 8.2

Теперь нужно вручную настроить некоторые параметры проекта, используя команду меню `Project|Settings...`, в результате чего появляется диалоговое окно, в котором при активной странице `C/C++` в комбинированном списке

Category нужно выбрать позицию Preprocessor, после чего в редактируемой строке Additional include directories нужно прописать путь к каталогу, в котором хранится файл mex.h (см. рис. 8.3). Этим каталогом является подкаталог \extern\include главного каталога пакета MATLAB.

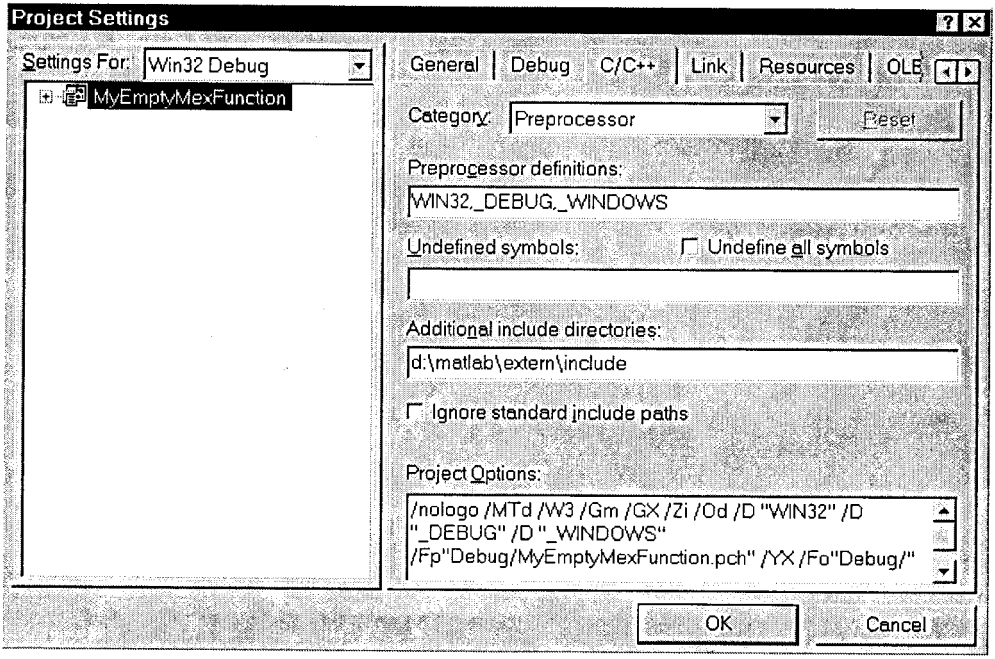


Рисунок 8.3

Далее в том же самом диалоговом окне Project Settings переходим на страницу Link и в конец редактируемой строки Object/library modules добавляем (к имеющемуся уже содержимому) надпись matlab.lib. Завершаем работу с диалоговым окном нажатием кнопки OK.

Теперь проект настроен полностью. Не хватает только наличия указанного выше файла matlab.lib в корневом каталоге нашего проекта, то есть в каталоге MyEmptyMexFunction. Очевидно, что такой файл должен быть скопирован в указанный каталог. Незадача же в том, что файл matlab.lib не входит в штатную поставку пакета MATLAB (из соображений экономии дискового пространства). Нам придется изготовить этот файл самим, после чего мы будем его всегда копировать в главные каталоги всех наших будущих проектов по созданию MEX-функций.

Чтобы изготовить файл matlab.lib, вызовем утилиту командной строки (это MS-DOS Prompt на платформе Win9x или Cmd.exe на платформе Windows NT) и перейдем в подкаталог \extern\include главного каталога пакета MATLAB, после чего введем команду, показанную на рис. 8.4.

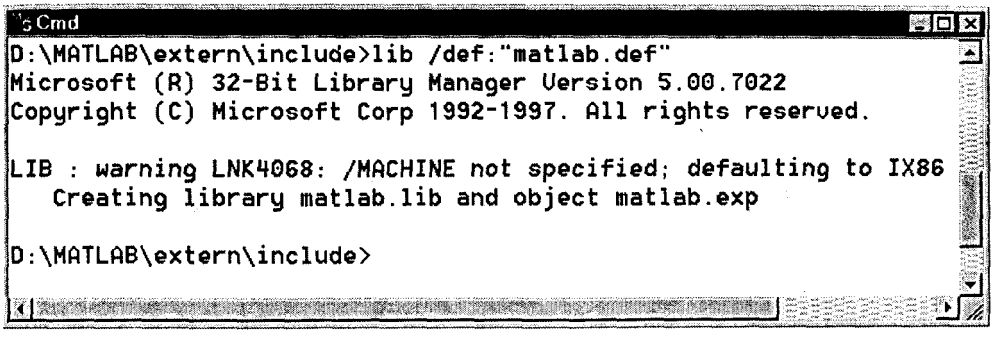


Рисунок 8.4

Этого достаточно, чтобы в подкаталоге `\extern\include` появился необходимый нам файл `matlab.lib`. Копируем его в каталог нашего проекта `MyEmptyMexFunction` и начинаем компилировать проект нажатием клавиши F7 (или командой меню `Build | Build MyEmptyMexFunction.dll`). Если не было допущено никаких отклонений от представленных нами описаний всех действий (то есть не допущено ошибок), то проект успешно скомпилируется и в каталоге `MyEmptyMexFunction\Debug` появится файл `MyEmptyMexFunction.dll`.

Теперь нужно этот файл скопировать в любой каталог, путь к которому прописан в списке путей доступа пакета MATLAB (или добавить каталог `MyEmptyMexFunction\Debug` в список доступа). После чего мы можем из командного окна вызвать разработанную нами MEX-функцию `MyEmptyMexFunction` (см. рис. 8.5).

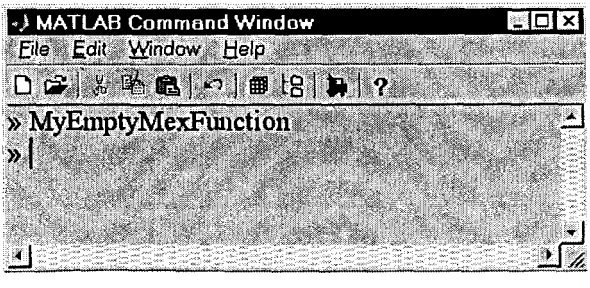


Рисунок 8.5

Отсутствие сообщений об ошибках свидетельствует о том, что система MATLAB успешно нашла файл с указанной функцией и выполнила ее. Так как эта функция ничего не делает, то никаких результатов при этом не получается. Однако на примере такой пустой MEX-функции мы смогли, не отвлекаясь на другие детали, проследить весь процесс изготовления MEX-функций.

Вызов функций MATLAB API

Сейчас сосредоточимся на содержательной стороне работы MEX-функций. Основной задачей интерфейсной функции `mexFunction` является прием данных, пришедших из системы MATLAB, а также формирование правильных структур данных, которые можно будет возвратить назад в систему MATLAB. Обмен данными происходит через параметры функции `mexFunction`.

Поэксплуатируем еще немного ранее созданную пустую MEX-функцию `MyEmptyMexFunction`. На ее примере покажем, какие параметры получает внутренняя интерфейсная `mexFunction` в случае разных вариантов вызова MEX-функции из системы MATLAB. Мы ранее говорили, что все эти параметры нулевые при вызове `MyEmptyMexFunction` без входных параметров и без возвращаемых значений, то есть при том варианте вызова, который показан на последнем рисунке.

Теперь рассмотрим такой вызов этой функции, при котором вызывающий передает функции `MyEmptyMexFunction` три входных параметра, а также надеется получить три возвращаемых значения (переменные `r`, `s` и `t`) (см. рис. 8.6). Так как наша функция реально не возвращает никаких значений, то система MATLAB выдает в командное окно соответствующее предупреждение (оно предназначено для информирования пользователя о том, что он напрасно надеется получить от данной функции какие-то результаты).

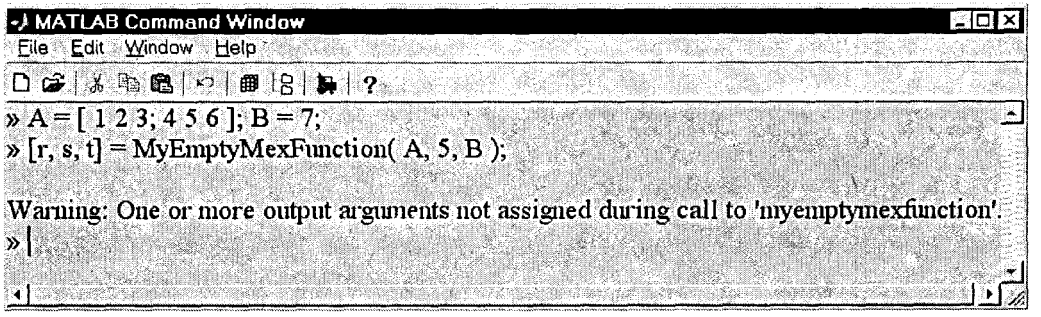


Рисунок 8.6

В этом случае при входе в функцию `mexFunction` оба параметра `nOut` и `nIn` равны трем. Указатель `pIn` указывает на область памяти, содержащую три ненулевых указателя на структуры типа `mxArray`, то есть на массивы пакета MATLAB (напоминаем, что все типы данных системы MATLAB являются массивами того или иного рода). Конкретно первый указатель из этой области – указатель `pIn[0]` указывает в нашем случае на массив `A`, следующий указатель – `pIn[1]` указывает на безымянный массив из одного элемента, равного 5. Наконец, последний указатель из этого блока памяти – `pIn[2]` указывает на массив `B`. Подыт-

тожим сказанное: все четыре указателя – pIn, pIn[0] (это то же самое, что и *pIn), pIn[1] и pIn[2] не равны NULL. Они все указывают на предварительно выделенные для дальнейшей работы блоки памяти. Мы теперь можем через эти указатели подбираться к входным массивам, пришедшим из пакета MATLAB.

Разобравшись со входными указателями, переходим к выходным указателям. Указатель pOut теперь указывает на блок памяти, хранящий три указателя на mxArray. Но все три последних равны NULL, так как именно мы сами внутри mexFunction должны выделить необходимую память под выходные массивы.

Все сказанное про параметры функции mexFunction чрезвычайно важно, поэтому сразу же закрепим полученные сведения на конкретном практическом примере. Для этого создадим MEX-функцию MyMexF2, которая работает с одним входным параметром (и игнорирует все остальные входные параметры, если пользователь функции удосужится их поставить). Эта функция возвращает единицу, если входной параметр не имеет тип double, и возвращает удвоенную величину первого параметра в противном случае. Ясно, что это не очень интересная с прикладной точки зрения функция, но нам она нужна в учебных целях для закрепления сведений о входных параметрах функции mexFunction.

Чтобы MEX-функция могла удобно работать с типами данных пакета MATLAB, а также выдавать сообщения в командное окно системы MATLAB (и выполнять ее команды и функции), в пакете MATLAB штатно присутствует набор специальных функций, которые с указанными целями можно вызывать из C-кода разрабатываемых MEX-функций. Такие функции в совокупности составляют функциональный набор, который принято называть *MATLAB API* (*Application Program Interface*).

Мы будем постепенно знакомиться с конкретными функциями MATLAB API. Для выполнения из MEX-функций любых действий, направленных в саму систему MATLAB, предназначены функции с префиксом (приставкой) mex, а для разбора структур данных пакета MATLAB предназначены функции с префиксом mx.

Вот C-текст для построения MEX-функции MyMexF2 (это же имя должен иметь проект Microsoft Developer Studio для получения файла MyMexF2.dll), в котором используется ряд функций MATLAB API:

```
#include 'mex.h'
//Prototype:
void MyDouble(int,int,double*,double*);
//------ Main interface function: ----------------
void mexFunction( int             nOut,
                  mxArray*        pOut[],
                  int             nIn,
                  const mxArray*  pIn[]  )
{
 int m, n; double *pI, *pO;
```

```
  if( nIn == 0 ) mexErrMsgTxt('Input required');
  if( nOut > 0 )
  {
    if( !mxIsDouble(pIn[0]) )
    {
      pOut[0]=mxCreateDoubleMatrix(1,1,mxREAL);
      *(mxGetPr(pOut[0])) = 1;
    }
    else
    {
      m = mxGetM( pIn[0] ); n = mxGetN( pIn[0] );
      pOut[0]=mxCreateDoubleMatrix(m,n,mxREAL);
      pI = mxGetPr( pIn[0] );
      pO = mxGetPr( pOut[0] );
      MyDouble( m, n, pI, pO );
    }
  }
}
//-------  Our own auxiliary function: ------------
void MyDouble( int m, int n, double* pI, double* pO )
{
  int i;
  for( i=0; i < n*m; i++ )
    pO[i] = 2 * pI[i];
}
```

В представленном C-коде вызываются следующие функции MATLAB API: `mexErrMsgTxt`, `mxIsDouble`, `mxCreateDoubleMatrix`, `mxGetPr`, `mxGetM` и `mxGetN`.

Первая из перечисленных функций имеет префикс `mex` и направлена на выполнение некоторого действия в самой системе MATLAB. Конкретно функция `mexErrMsgTxt` выдает в командное окно системы MATLAB сообщение об ошибке, заключающейся в отсутствии входных параметров (наша функция должна вызываться минимум с одним входным параметром), и корректно завершает выполнение нашей MEX-функции.

Остальные функции имею префикс `mx` и направлены, как мы уже знаем, на работу со структурой `mxArray` – основным типом данных пакета MATLAB (это есть внутреннее представление любых массивов системы MATLAB).

У функции `mxIsDouble` один входной параметр – он является указателем на структуру `mxArray`, то есть указателем на массив системы MATLAB. Данная функция возвращает единицу («истину» в смысле языка программирования C),

если массив имеет тип double, и возвращает нуль (логическая «ложь») в противном случае.

Функции mxGetM и mxGetN принимают по одному такому же параметру (указателю на структуру mxArray). Первая из них возвращает число строк во входной матрице, а вторая функция возвращает число столбцов.

Функция mxCreateDoubleMatrix выделяет память под структуру mxArray, соответствующую числовой матрице с размерами, которые указываются через ее первый и второй параметры. Третий параметр является условным числовым флагом, сигнализирующим о создании чисто вещественной или комплексной числовой матрицы. Флаг mxREAL заставляет функцию mxCreateDoubleMatrix создавать вещественную числовую матрицу. При этом следует четко понимать, что функция mxCreateDoubleMatrix не заполняет созданную матрицу числовыми элементами. Это нужно выполнять в виде отдельной работы. В частности, сначала нужно получить доступ к собственно данным матрицы, то есть к ее элементам.

Такую операцию получения доступа к элементам массивов системы MATLAB выполняет функция mxGetPr. Эта функция возвращает указатель на область памяти, в которой в линейно упорядоченной форме (по столбцам) хранятся все элементы массива. Конкретно функция mxGetPr возвращает указатель на действительные составляющие в общем случае комплексных элементов. Указатель на мнимые части элементов возвращается функцией mxGetPi. Действительные и мнимые части элементов хранятся порознь в виде отдельных блоков памяти. В каждом из этих блоков данные упорядочены линейно по столбцам. Так как мы создаем вещественные матрицы, то нам достаточно работать только с функцией mxGetPr.

Теперь уже можно объяснить работу нашей MEX-функции. Сначала в теле интерфейсной функции mexFunction мы проверяем количество входных параметров, с которыми вызвана из системы MATLAB наша MEX-функция. Если она вызвана вообще без параметров, то выдаем в командное окно системы MATLAB сообщение об ошибке (Input required – «Требуются входные параметры») и завершаем работу. Все это делается с помощью MATLAB API-функции mexErrMsgTxt.

Дальнейшая работа выполняется только в случае наличия одного или большего числа входных параметров. Мы договорились, что наша функция все входные параметры, кроме первого, будет игнорировать (при этом не будет возникать никаких ошибочных ситуаций). Первый же входной параметр она будет тестировать на его тип. Если этот параметр не является массивом типа double, то наша MEX-функция просто возвращает единицу. Вот как достигается эта цель:

```
pOut[0]=mxCreateDoubleMatrix(1,1,mxREAL);
*(mxGetPr(pOut[0])) = 1;
```

Мы создаем с помощью функции mxCreateDoubleMatrix чисто вещественную матрицу (флаг mxREAL) размером 1 x 1. Подбираемся к ее единственному

элементу с помощью функции mxGetPr, которая возвращает указатель типа double* на этот элемент. По этому указателю мы и прописываем требуемую единицу. Ничего больше для возврата этого единственного значения делать не нужно. Интерфейсная функция mexFunction, заканчивая свою работу, всегда возвращает результаты системе MATLAB через указатели pOut[0], pOut[1] (если он был задействован) и т. д.

В случае же, когда первый входной параметр является вещественной числовой матрицей, мы создаем структуру mxArray такого же размера, после чего в отдельной C-функции MyDouble прописываем все ее элементы значениями, в два раза превосходящими соответствующие элементы входной матрицы. Чем и достигаем поставленной цели.

Окончательно создаем в Microsoft Developer Studio проект MyMexF2, помещаем в него представленный выше C-текст, настраиваем проект так, как было рассказано в предыдущем подразделе, компилируем его и получаем файл динамической библиотеки MyMexF2.dll.

Помещаем этот файл в любой каталог, доступный пакету MATLAB, и начинаем вызывать MEX-функцию MyMexF2 в самых разных конфигурациях (задавая различное количество и тип входных переменных и выходных значений). Вот некоторые результаты (см. рис. 8.7).

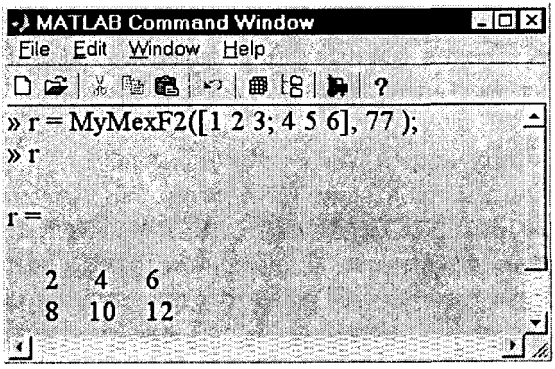


Рисунок 8.7

Здесь мы передали созданной нами MEX-функции MyMexF2 два входных параметра, из которых первый является вещественной числовой матрицей 2 x 3. Наша функция отработала штатно и вернула массив такой же структуры, но с удвоенными элементами.

А вот вызов функции MyMexF2 со строковым входным аргументом:

```
str = 'Hello, MEX-function';
res = MyMexF2( str );
res =
   1
```

Видно, что созданная нами функция распознала ситуацию, когда первый параметр не является числовым массивом, и вернула единицу в числовом массиве res, размер которого равен 1 x 1 (см. рис. 8.8).

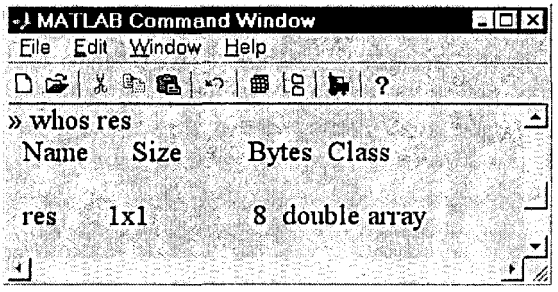


Рисунок 8.8

Ситуация с вызовом функции MyMexF2, когда ее аргументом (параметром) будет массив ячеек или структур, абсолютно очевидна: функция вернет единицу. Но вот что будет, если вызвать эту функцию с числовым массивом размерности три? Попробуем это сделать (хотя про себя сразу отметим, что мы разрабатывали эту функцию в расчете на двумерные числовые массивы – матрицы):

```
W(:,:,1)=[1,2;3,4]; W(:,:,2)=[5,6;7,8];
res = MyMexF2( W );
res =
   2  4  10  12
   6  8  14  16
```

Видно, что функция «переварила» эту ситуацию и «не сломалась», но в любом случае тут не получается возврата той же самой структуры, что и входной параметр. Если бы мы сразу вспомнили при разработке функции MyMexF2 про многомерные числовые массивы, то их можно было бы обрабатывать штатно по некоторой схеме. А без этого остается надеяться, что по крайней мере не произойдет аварийного завершения работы функции.

Это одна из причин, которые могут вызывать ошибочные ситуации при выполнении кода MEX-функций. Существуют также многие другие причины, по которым разрабатываемые MEX-функции могут работать не так, как ожидается. В таком случае нужно проводить отладку этих функций.

Отладка MEX-функций

Отладка MEX-функций, понимаемая как процесс исправления ошибок, может заключаться в чем угодно, хоть в многократном визуальном изучении текста функции с целью обнаружить в нем логические и синтаксические ошибки. Однако в технике программирования под отладкой чаще всего понимают

некоторые автоматизированные методики. Самым удобным приемом является прием приостановки выполнения кода функции с изучением значений переменных в этот момент времени.

Как это делается в случае M-функций, мы уже изучали. Там мы опирались на возможности редактора/отладчика, входящего в состав пакета MATLAB. MEX-функции являются чужеродным элементом для системы MATLAB, которая лишь взаимодействует с такими функциями, но не берет на себя управление их выполнением. Поэтому и отладка этих функций выполняется не средствами пакета MATLAB, а средствами оболочки Microsoft Developer Studio. Вот как это делается.

Нужно в Developer Studio открыть соответствующий проект и в тексте интерфейсной функции `mexFunction` поставить напротив некоторой строки C-кода так называемую метку точки останова (Breakpoint). Это делается при помощи клавиши F9, после чего напротив выбранной строки появляется красный кружок, указывающий на наличие точки останова именно в этой строке.

Далее нужно выполнить некоторые дополнительные настройки в `Project | Settings`, показанные на рис. 8.9.

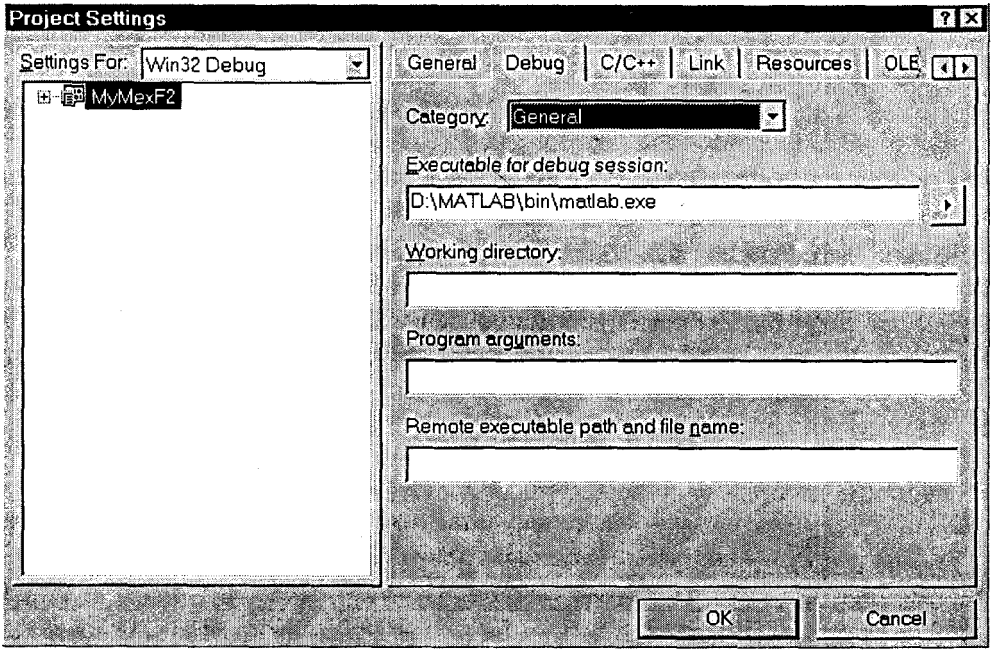


Рисунок 8.9

Нужно на странице `Debug` в редактируемом поле `Executable for debug session` прописать путь к исполняемому модулю системы MATLAB, коим является файл `matlab.exe`.

После этого нужно нажать клавишу F5, означающую работу с открытым проектом в режиме отладки. Но так как наш проект представляет собой библиотеку (а не самостоятельное приложение), то сначала загружается основной программный модуль, который и указывается как `Executable for debug session`. В нашем случае это сама система MATLAB, то есть исполняемый файл `matlab.exe`. После запуска системы MATLAB из его командного окна мы вызываем нашу MEX-функцию (показан пример, когда в качестве параметра передается трехмерный числовой массив), и тут-то и происходит останов на той строке исходного С-кода, где мы ранее поставили точку останова (см. рис. 8.10). Целесообразно ставить ее на одну из первых строк интерфейсной функции `mexFunction`, так как в самом начале следует проверить, с какими входными параметрами она вызвана.

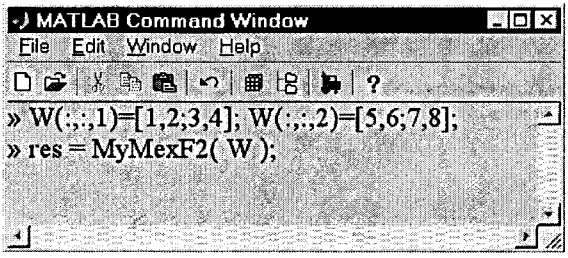


Рисунок 8.10

На рис. 8.11 видна точка останова, а также видно, как происходит просмотр значений переменных при наведении на них курсора мыши (появляется маленькое всплывающее окно со значением переменной).

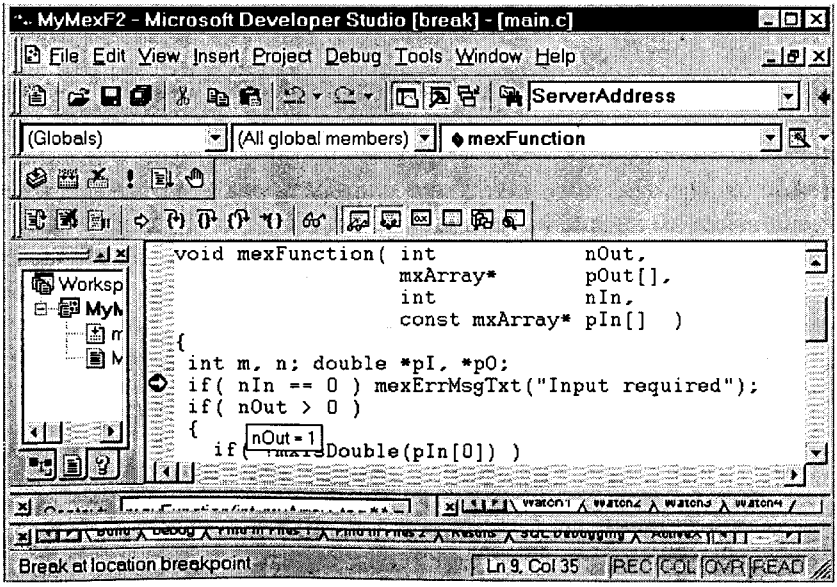


Рисунок 8.11

Теперь мы можем буквально собственными глазами увидеть, что же на самом деле происходит, когда нашей MEX-функции передается в качестве параметра трехмерный массив. Для этого ставим точку останова на строке, где вычисляются размеры «матрицы» m и n. Выполняем эту строку нажатием клавиши F10 (пошаговое исполнение инструкций исходного C-кода) и смотрим значения переменных m и n. Для матриц m является количеством строк, и в данном случае для этой переменной получается значение 2, что совпадает с количеством строк в обеих страницах входного трехмерного массива. Для переменной же n, которая у матриц равна количеству столбцов, получается значение, равное 4. Теперь ясно, что после этого создается новая матрица 2 x 4, которая и заполняется удвоенными элементами входного массива. Работа нашей MEX-функции в случае входного трехмерного массива полностью ясна.

Если мы хотим добавить к штатному поведению нашей MEX-функции корректную обработку многомерных числовых массивов, то вместо функций mxGetM и mxGetN нужно использовать MATLAB API-функции mxGetNumberOfDimensions и mxGetDimensions. Первая из этих функций возвращает количество измерений массива, а вторая функция возвращает массив размеров вдоль каждого из измерений. Например, код

```
int N; int* pd;
...
num = mxGetNumberOfDimensions( pIn[0] );
pd = mxGetDimensions( pIn[0] );
pOut[0]=mxCreateNumericArray(N,pd,mxDOUBLE_CLASS,mxREAL);
```

в рассмотренном нами для примера конкретном случае трехмерного массива W приведет к созданию трехмерного выходного массива размером 2 x 2 x 2, который далее можно заполнить удвоенными значениями элементов входного массива.

Как бы тщательно ни разрабатывались MEX-функции, этап отладки остается практически неизбежным. Поэтому овладение мастерством отладки необходимо для приобретения высокой квалификации. Такие приемы, как просмотр не только имеющихся в C-коде переменных, но и произвольных выражений с ними (с помощью диалогового окна, вызываемого клавишами Shift+F9), простановка условных точек останова, и другие приемы, известные опытным программистам, работающим с компилятором Visual C++, пригодятся и при разработке MEX-функций для пакета MATLAB.

Примеры конкретных разработок MEX-функций

Освоив некоторое количество функций MATLAB API, можно перейти к рассмотрению конкретных примеров MEX-функций, работающих с разного рода массивами пакета MATLAB.

Начнем с простого случая, когда разрабатываемая MEX-функция должна принять на входе два числовых скаляра, которые будут использоваться как размеры матрицы, создающейся внутри этой MEX-функции и отправляемой назад в систему MATLAB.

Этот пример очень простой, но он иллюстрирует новые для нас MATLAB API-функции обработки скаляров. Кроме того, он демонстрирует некоторую защиту, связанную с проверкой входных параметров, а также еще раз показывает *необязательный*, но логически понятный *стиль программирования*, когда внутри интерфейсной функции `mexFunction` выполняется вся работа, связанная с приемом и переработкой *специфических* для системы MATLAB данных (тип `mxArray`), а все остальные C-функции проекта ничего такого не знают. Последние даже выглядят абсолютно традиционно для C-функций, не знакомых с пакетом MATLAB. Более того, эти функции могли быть разработаны ранее и по другому поводу и только сейчас они подключаются для работы в составе пакета MATLAB.

Итак, в нашей MEX-функции мы будем создавать числовую матрицу заданного размера `M x N` и будем заполнять ее некоторыми числовыми значениями (в возрастающем порядке в смысле упорядочения по столбцам). Вот исходный C-код проекта, по-прежнему расположенный в пределах единственного файла `main.c` (часто все функции, кроме интерфейсной функции `mexFunction`, выносятся в отдельные файлы проекта):

```
#include 'mex.h'

// Prototypes:
void SetData( double*,int,int );

////////////////////////////////////////////////////
void mexFunction( int nOut,       mxArray* pOut[],
                  int nIn,  const mxArray* pIn[]   )
{
  // We ignore extra input parameters!
  int M, N;
  double* pElem;

  if( (mxGetN( pIn[0] ) != 1) ||
```

```
      (mxGetM( pIn[0] ) != 1) ||
      (mxGetN( pIn[1] ) != 1) ||
      (mxGetM( pIn[1] ) != 1)    )
    mexErrMsgTxt('Only scalars must be!');

  M = (int)mxGetScalar( pIn[0] );
  N = (int)mxGetScalar( pIn[1] );

  pOut[0] = mxCreateDoubleMatrix( M, N, mxREAL );
  pElem   = mxGetPr( pOut[0] );

  SetData( pElem, M, N );
}
////////////////////////////////////////////////////
void SetData( double* p, int m, int n )
{
  int i;
  for( i=0; i < m*n; i++ )
  {
    *p = (double)i;
    p++;
  }
}
//////////////---the end of the code---///////////////
```

Мы знаем, что по существу скалярные числовые величины в системе MATLAB являются матрицами 1 x 1 и внутренне представимы структурами типа mxArray. Поэтому единственный числовой элемент этих массивов еще нужно извлечь из структуры mxArray, прежде чем можно будет его использовать. Это выполняется с помощью MATLAB API-функции mxGetScalar. Именно эту функцию мы и применили в представленном выше коде.

После этого мы создаем с помощью функции mxCreateDoubleMatrix числовую матрицу нужного размера, а с помощью функции mxGetPr получаем указатель типа double* на выделенный при этом блок памяти, предназначенный для хранения числовых значений элементов матрицы. Заполняем же этот блок памяти некоторыми конкретными значениями элементов мы в отдельной функции – в функции SetData. Эта отдельная C-функция выглядит абсолютно традиционно для C-функций, не знакомых с системой MATLAB. Это объясняется тем, что в ней отсутствуют вызовы MATLAB API-функций и нет никакой работы со структурой mxArray.

Ранее мы подробно рассказывали о том, что нужно сделать, чтобы превратить исходный C-код в целевой файл динамической библиотеки, представляющей со-

бой MEX-функцию пакета MATLAB. Поэтому мы не будем это повторять, а только сообщим, что именуем проект (рабочий каталог) в Microsoft Developer Studio как SetMatrix. Это значит, что целевая MEX-функция будет иметь такое же имя. Ниже показан пример вызова этой функции из командного окна системы MATLAB:

```
A = SetMatrix( 3, 4 );
A =
  0  3  6  9

  1  4  7  10

  2  5  8  11
```

Итак, MEX-функция SetMatrix создает числовую матрицу заданного размера и заполняет ее элементы целыми значениями от нуля и выше.

Теперь рассмотрим другой проект, в котором реализуем более осмысленную задачу о вычислении средних арифметических значений строк числовых матриц. Назовем такую MEX-функцию AverVector, поскольку она возвращает вектор-столбец средних по строкам входной матрицы значений (Aver – сокращение от average, то есть «средний»).

Продемонстрируем сначала весь С-код этого проекта, после чего обсудим некоторые детали:

```
#include 'mex.h'

// Prototype:
void Aver( double*,double*,int,int);

////////////////////////////////////////////////////
void mexFunction( int nOut,      mxArray* pOut[],
                  int nIn, const mxArray* pIn[]   )
{
  double* pDataIn;
  double* pDataOut;
  int     M, N;

  if( nIn !=1 ) mexErrMsgTxt( 'One input required.' );
  if( !mxIsDouble(pIn[0]) || mxIsComplex(pIn[0]) )
    mexErrMsgTxt( 'Input must be a noncomplex double
                   matrix only.' );

  M = mxGetM( pIn[0] );
  N = mxGetN( pIn[0] );
```

```
  pOut[0] = mxCreateDoubleMatrix( M, 1, mxREAL );

  pDataIn  = mxGetPr( pIn[0] );
  pDataOut = mxGetPr( pOut[0] );

  Aver( pDataIn, pDataOut, M, N );
}
//////////////////////////////////////////////////////
void Aver( double* pin, double* pout, int m, int n )
{
   int i, j; double sum;

   for( i = 0; i < m; i++ )
   {
     sum = 0;
     for( j=0; j < n; j++ )
       sum = sum + *( pin + i + j * m );

    *( pout + i ) = sum / ((double)n);
   }
}
//////////---the end of the code---/////////////////
```

Здесь функция `mxIsDouble` возвращает логическую истину только тогда, когда входная матрица имеет тип `double`, а функция `mxIsComplex` дополнительно требует для этого, чтобы числовая матрица имела комплексные значения. Поэтому входную проверку проходит и далее поступает на обработку только входная числовая матрица с действительными элементами.

Сама по себе представленная обработка входной матрицы, когда вычисляются средние арифметические значения по ее строкам, а результаты записываются в элементы выходного вектор-столбца, весьма проста и не требует дополнительных комментариев за исключением оговорки, что *язык С* сам по себе *достаточно сложен* в таком компоненте, как *активное использование указателей*, что мы и делаем в С-функции `Aver`. Но с этим ничего нельзя поделать: сложности при изучении языка С оборачиваются преимуществами в работе получающихся с его помощью программ.

А вот и пример использования MEX-функции `AverVector`, из которого видно, что разработанная нами MEX-функция `AverVector` справляется с возложенными на нее обязанностями (см. рис. 8.12).

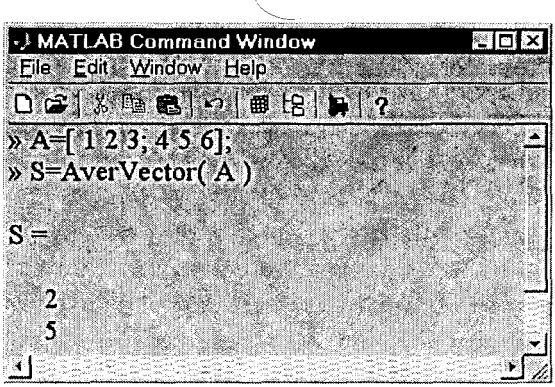


Рисунок 8.12

Перейдем теперь от разработок MEX-функций, работающих с массивами типа double, к функциям обработки символьных данных. Разработаем MEX-функцию RevString, которая будет «выворачивать» содержимое входной строки наоборот (последний символ станет первым, предпоследний – вторым и т. д.).

Большой разницы в коде MEX-функций, работающих с текстовыми строками, по сравнению с MEX-функциями, обрабатывающими вещественные числа, нет. Тем не менее для данного проекта нам потребуется ряд дополнительных MATLAB API-функций, которые мы еще не изучали. Приведем весь текст проекта, а потом обсудим использованные в нем новшества:

```
#include 'mex.h'

//////////////////////////////////////////////////////
void Reverse( char* inStr, int len, char* outStr )
{
  int i;

  for( i=0; i < len-1; i++ )
    *( outStr + i ) = *( inStr + len - i - 2 );
}
//////////////////////////////////////////////////////
void mexFunction( int nOut,      mxArray* pOut[],
                  int nIn, const mxArray* pIn[]   )
{
  char* StringIn;
  char* StringOut;
  int len, status;

//------  Input and output checking: -----------
  if(nIn!=1) mexErrMsgTxt('One input required.');
```

```
    else if(nOut > 1)
      mexErrMsgTxt('Too many output arguments.');

    if( mxIsChar( pIn[0]) != 1 )
      mexErrMsgTxt('Input must be a string.');

    if(mxGetM( pIn[0])!=1 )
      mexErrMsgTxt('Input must be a row vector.');

  //-----  Allocate buffers for strings:  ----------
    len = mxGetN(pIn[0]) + 1;
    StringIn  = mxCalloc( len, sizeof(char) );
    StringOut = mxCalloc( len, sizeof(char) );

    status = mxGetString( pIn[0], StringIn, len );
    if(status != 0)
     mexWarnMsgTxt('Not enough space for string.');

    Reverse( StringIn, len, StringOut );

    pOut[0] = mxCreateString( StringOut );
  }
  ////////////---the end of the code---/////////////////
```

Ну, прежде всего, в данном случае мы текст внутренней (неэкспортируемой из динамической библиотеки) функции `Reverse` поместили перед текстом интерфейсной функции `mexFunction`, что не представляет собой ничего необычного, но, тем не менее, позволяет обойтись без прототипа этой функции (что всегда имело место в предыдущих примерах). В самой функции `Reverse` имеет место типичное для языка C тотальное применение указателей для работы с символьными массивами (в терминологии языка C это массивы типа `char`).

Входная текстовая строка извлекается из массива `mxArray` с помощью MATLAB API-функции `mxGetString`, которая помещает эту строку в заранее подготовленный буфер, память для которого выделяется другой MATLAB API-функцией – функцией `mxCalloc`. В случае недостаточного размера буфера мы выдаем в командное окно системы MATLAB предупреждающее сообщение с помощью функции `mexWarnMsgTxt`.

Наконец, выходное значение данной MEX-функции, которое является массивом системы MATLAB типа `char`, создается с помощью MATLAB API-функции `mxCreateString`, аргументом для которой служит массив языка C типа `char`, в котором располагаются выходные символы.

Для проверки работы полученной MEX-функции RevString подадим ей на вход строку 'Hello, World'. Результат работы представлен на рис. 8.13.

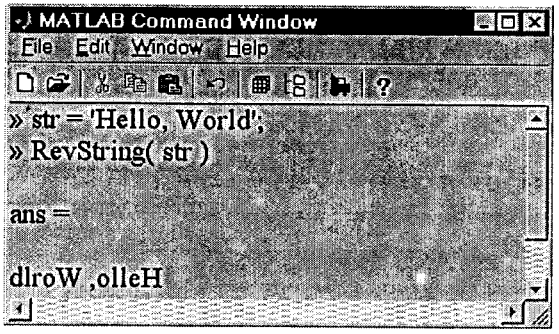


Рисунок 8.13

В завершение цепи примеров рассмотрим MEX-функцию, принимающую на входе массив ячеек. Мы знаем, что массив ячеек является «массивом массивов», так как его элементы сами являются массивами системы MATLAB. С точки зрения MEX-функций это означает, что MATLAB API-функция mxGetCell возвращает указатель на тип mxArray, который требует дальнейшей обработки.

Создадим MEX-функцию, которая получив на входе массив ячеек, изучает его структуру и передает полученную информацию в командное окно системы MATLAB с помощью MATLAB API-функции mexPrintf.

Для простоты ограничимся случаем, когда входной массив ячеек имеет размерность два, то есть является матрицей. Исходя из этого ограничения следующий C-код достаточен для разбора содержимого (на первом уровне возможных многоуровневых вложений массивов ячеек в другие массивы ячеек) входного массива (матрицы) типа cell:

```
#include 'mex.h'

// Prototype:
void MyPrint( int, int, mxArray* );

////////////////////////////////////////////////////
void mexFunction( int nOut,      mxArray* pOut[],
                  int nIn, const mxArray* pIn[]   )
{
  int i, j, ndim, M, N;
  mxArray* pAr;

//------  Input and output checking: -----------
  if(nIn!=1) mexErrMsgTxt('One input required.');
  else if(nOut > 0)
```

```
    mexErrMsgTxt('No output required.');

  ndim = mxGetNumberOfDimensions( pIn[0] );
  if( ndim != 2 )
    mexErrMsgTxt('Only matrices accepted.');

  if( !mxIsCell( pIn[0]) )
    mexErrMsgTxt('Input must be a cell only.');

//-----  :  ----------
M = mxGetM( pIn[0] );
N = mxGetN( pIn[0] );

for( i = 0; i < M; i++ )
 for( j = 0; j < N; j++ )
 {
     pAr = mxGetCell( pIn[0], i+j*M );
     MyPrint( i, j, pAr );
 }
}
////////////////////////////////////////////////////////
void MyPrint( int i, int j, mxArray* pAr )
{
  if( mxIsChar( pAr ) )
    mexPrintf('%d-%d element is Char\n',i+1,j+1);
  else if( mxIsDouble( pAr ) )
    mexPrintf('%d-%d element is Double\n',i+1,j+1);
  else if( mxIsComplex( pAr ) )
    mexPrintf('%d-%d element is Complex\n',i+1,j+1);
  else if( mxIsCell( pAr ) )
    mexPrintf('%d-%d element is Cell\n',i+1,j+1);
else if( mxIsStruct( pAr ) )
    mexPrintf('%d-%d element is Struct\n',i+1,j+1);
  else if( mxIsUint8( pAr ) )
    mexPrintf('%d-%d is Uint8\n',i+1,j+1);
else if( mxIsSparse( pAr ) )
    mexPrintf('%d-%d is Sparse\n',i+1,j+1);
  else
    mexPrintf('%d-%d element - Unknown\n',i+1,j+1);
}
/////////////---the end of the code---/////////////////
```

На вход данной функции допускаются только двумерные массивы ячеек. С помощью MATLAB API-функции `mxGetCell` мы извлекаем отдельный элемент входного массива. Для массива ячеек элементы имеют тип `mxArray`. Обратим внимание на то, что в функции `mxGetCell` отдельные элементы индексируются единственным индексом, значение которого нужно назначать исходя из упорядочения элементов массивов системы MATLAB по столбцам. Именно поэтому элемент из `i`-й строки и `j`-го столбца извлекается с помощью следующего выражения (индексы в языке C начинаются с нуля):

```
pAr = mxGetCell( pIn[0], i+j*M );
```

Указатель `pAr` на структуру `mxArray` далее передается в функцию `MyPrint`, где с помощью функций `mxIsChar`, `mxIsDouble`, `mxIsComplex`, `mxIsCell`, `mxIsStruct`, `mxIsUint8` и `mxIsSparse` определяется, какие данные содержатся в этой структуре.

Так как в языке C индексы массивов начинаются с нуля, а в M-языке они начинаются с единицы, то, чтобы наблюдать в командном окне системы MATLAB привычные значения индексов, мы в функции `MyPrint` увеличиваем значения индексов на единицу при их выводе в командное окно функцией `mexPrintf`.

Протестируем работу созданной только что MEX-функции `MexCell1` на примере простого массива типа `cell` (см. рис. 8.14).

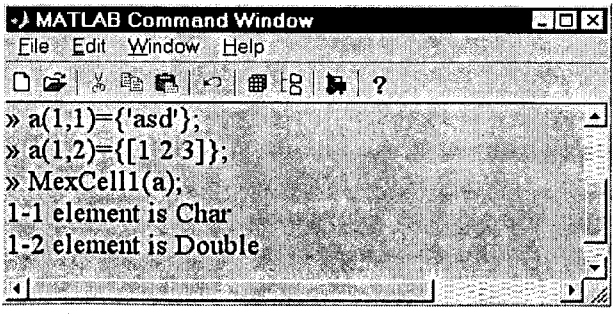


Рисунок 8.14

Если же применить функцию `MexCell1` к массиву ячеек `MyCellArray`

```
MyStruct = struct('field1',[ 1 2 3],'field2','Hello');
MyCellArray( 1, 1 ) = { 'Bonjour!' };
MyCellArray( 1, 2 ) = { [ 1 2 3; 4 5 6; 7 8 9 ] };
MyCellArray( 2, 1 ) = { MyStruct };
MyCellArray( 2, 2 ) = { [ 9 7 5 ] };
```

который мы ранее создавали в гл. 3 при изучении массивов типа `cell`, то результат работы этой функции покажет, что распознаются и ситуации, когда

некоторые элементы входного массива ячеек сами имеет сложную структуру, например являются массивами структур:

```
MexCell1( MyCellArray )
1-1 element is Char
1-2 element is Double
2-1 element is Struct
2-2 element is Double
```

Таким же образом распознаются ситуации, когда элементами массива ячеек в свою очередь являются массивы ячеек.

Заканчивая рассмотрение конкретных примеров программирования MEX-функций, отметим, что со всеми не рассмотренными нами MATLAB API-функциями всегда можно ознакомиться по встроенной в систему MATLAB подсистеме справочной информации. Например, можно воспользоваться информацией из PDF-файлов, расположенных в подкаталоге \help\pdf_doc\matlab\Api. Напомним, что PDF-файлы просматриваются программами Acrobat Reader или Acrobat Exchange.

Вызов функций и команд системы MATLAB из MEX-функций

Мы уже говорили в предыдущем подразделе, что помимо MATLAB API-функций с префиксом mx имеются также MATLAB API-функции с префиксом mex. Последние предназначены для выполнения команд и функций среды MATLAB. В примерах предыдущего подраздела мы на практике использовали такого рода функции mexErrMsgTxt, mexWarnMsgTxt и mexPrintf.

Однако концептуально наиболее важной функцией такого типа (помимо интерфейсной функции mexFunction) является функция mexCallMATLAB. Эта функция позволяет изнутри С-кода разрабатываемых нами MEX-функций вызывать другие MEX-функции, любые М-функции и вообще все команды и функции среды MATLAB. Отсюда ясно, что важность функции mexCallMATLAB трудно переоценить. Именно эта функция позволяет самым тесным образом интегрировать MEX-функции в среду MATLAB, сделав их самым непосредственным расширением этой программной среды.

В качестве примера использования функции mexCallMATLAB рассмотрим сейчас пример MEX-функции, которая ничего не принимает на входе, но самостоятельно создает квадратную матрицу типа double, заполняет ее элементы конкретными значениями, после чего вызывает встроенную в среду MATLAB функцию det для вычисления определителя полученной матрицы. Далее наша MEX-функция выведет результат вычисления определителя в командное окно системы MATLAB опять-таки с помощью функции mexCallMATLAB.

Эта MEX-функция, которую мы назовем MyMexDeterminant, всю работу производит в своем собственном коде и не возвращает в среду MATLAB никаких значений. Поэтому мы проверяем параметры nIn и nOut, чтобы гарантировать правильный вызов этой MEX-функции из командного окна системы MATLAB (то есть без входных параметров и без возвращаемых значений).

Перед тем как писать код MEX-функции MyMexDeterminant, представим прототип функции mexCallMATLAB:

```
int mexCallMATLAB( int nOut, mxArray*pOut[],
                   int nIn,  mxArray*pIn[],
                   const char* name   );
```

У этой функции всего пять параметров. Первые четыре из них совпадают с параметрами функции mexFunction, что абсолютно очевидно, потому что это и есть интерфейсная функция среды MATLAB (как же иначе можно было бы вызвать из MEX-функции другую MEX-функцию). Пятый параметр представляет имя функции (команды), подлежащей выполнению.

Теперь мы готовы к написанию кода MEX-функции MyMexDeterminant (напомним еще раз, что имя MEX-функции совпадает с именем проекта в Microsoft Developer Studio и используется при вызове этой функции, но не встречается в ее исходном С-коде):

```
#include 'mex.h'
#include <memory.h>

////////////////////////////////////////////////////
void mexFunction( int nOut,      mxArray* pOut[],
                  int nIn, const mxArray* pIn[]   )
{
  int ret; double x; char str[128];
  mxArray* pAr[1];  mxArray* pRes[1];
  double pS[] = {1,4,7,2,5,8,3,6,9};

//------  Input and output checking: -----------
  if(nIn!=0) mexErrMsgTxt('None input required.');
  else if(nOut != 0)
    mexErrMsgTxt('None output required.');

// 3-by-3 matrix creation and element setting:
  pAr[0] = mxCreateDoubleMatrix(3, 3, mxREAL);
  memcpy( mxGetPr( pAr[0] ), pS, sizeof( pS ) );

// We ask MATLAB to calculate determinant.
```

```
// The result will be in pRes.
  ret = mexCallMATLAB( 1, pRes, 1, pAr, 'det' );
  if( ret != 0 ) mexErrMsgTxt('det failure');

// Now we show the result in Command window:
// this is equivalent to 'disp' command:
  x = *( mxGetPr( pRes[0] ) );
  sprintf( str, 'determinant = %f', x );
  mxDestroyArray( pRes[0] );
  pRes[0] = mxCreateString( str );
  mexCallMATLAB( 0, NULL, 1, pRes, 'disp' );

// the last memory cleanup (good practice!):
mxDestroyArray( pAr[0] );
mxDestroyArray( pRes[0] );
}
/////////////---the end of the code---///////////////
```

По тексту функции можно дать несколько пояснений. Прежде всего, заполнение созданного массива типа double значениями его элементов можно выполнить единственным вызовом функции memcpy, так как эта функция за один раз копирует содержимое блока памяти с подготовленными элементами в блок памяти, отведенный под элементы в структуре mxArray.

Вызов функции mexCallMATLAB для вычисления определителя (детерминанта) матрицы достаточно очевиден:

```
ret = mexCallMATLAB( 1, pRes, 1, pAr, 'det' );
```

У нас тут имеются единственный входной параметр и единственное выходное значение. Входной параметр передается через массив pAr (этот массив состоит из одного элемента, являющегося указателем на структуру mxArray), а выходное значение будет прописано в массив pRes. Последний массив также состоит из единственного указателя на mxArray. Именно так эти переменные и были определены в нашем коде:

```
mxArray* pAr[1];  mxArray* pRes[1];
```

Важно проверять возврат функции mexCallMATLAB, так как операция вычисления определителя не всегда возможна (вдруг вы по ошибке передали для вычисления неквадратную матрицу, например).

Из результирующего массива pRes[0] мы извлекаем значение вычисленного определителя и помещаем его в переменную x типа double, после чего используем значение этой переменной для формирования строки текста,

содержащего это значение. Делается это с помощью стандартной библиотечной С-функции sprintf.

Далее мы уничтожаем за ненадобностью ранее сформированный массив pRes[0] (фактически освобождаем выделенную под него память), а затем по этому же указателю (не пропадать же добру) создаем новый массив системы MATLAB, но этот массив уже имеет тип char. Он-то и будет служить входным параметром для функции mexCallMATLAB, с помощью которой подготовленное текстовое сообщение и выводится в командное окно системы MATLAB (см. рис. 8.15).

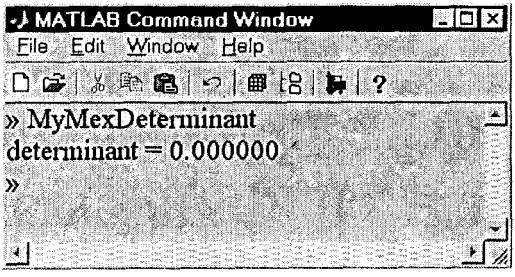


Рисунок 8.15

Вот этот фрагмент кода, который осуществляет показанный на рисунке вывод:

```
pRes[0] = mxCreateString( str );
mexCallMATLAB( 0, NULL, 1, pRes, 'disp' );
```

Так как для исполняемой таким образом функции disp системы MATLAB возвращаемые значения не используются, то мы и прописываем первые два параметра функции mexCallMATLAB соответственно нулем и значением NULL.

В конце текста нашей MEX-функции хорошим решением будет самостоятельное (чтобы не перегружать пакет MATLAB дополнительной работой) освобождение памяти, выделенной под структуры mxArray (то есть под массивы системы MATLAB):

```
// the last memory cleanup (good practice!):
mxDestroyArray( pAr[0] );
mxDestroyArray( pRes[0] );
```

Выполняется эта работа с помощью MATLAB API-функции mxDestroyArray, синтаксис вызова которой абсолютно очевиден.

Часть 3

Создание законченных приложений

Глава 9

Законченные приложения на базе графического интерфейса пользователя системы MATLAB

Графические окна системы MATLAB и элементы управления

Для решения крупной задачи обычно требуется разработка множества M- и MEX-функций, которые вызываются в некоторой заданной последовательности наряду со стандартными функциями системы MATLAB. Помимо этого нужно вводить исходные числовые и текстовые данные, а также визуализировать промежуточные и конечные результаты в той или иной форме: часть результатов можно представить в текстовом виде, а часть – в графическом виде. Может также потребоваться несколько графических окон для наглядной графической визуализации.

Непосредственный разработчик, знакомый со всеми нюансами, в состоянии дирижировать таким довольно сложным «ансамблем». Однако повторить эту работу стороннему пользователю непросто, ведь она требует точных знаний команд и функций системы MATLAB, а также основ программирования в рамках этой системы. Даже самому разработчику для выполнения многочисленных повторяющихся заданий было бы удобнее иметь некоторый *интегрирующий меха-*

низм, позволяющий удобным и наглядным способом выполнять все виды разнородных работ из единого *командного центра*.

Для этих целей в системе MATLAB применяются уже изученные нами ранее графические окна вместе с *графическими элементами управления*. Последние располагаются на поверхности графических окон и позволяют вводить и читать числовую и текстовую информацию, нажатием кнопок инициировать выполнение любых функций (стандартных, М-функций или MEX-функций) с последующим автоматическим показом результатов вычислений на поверхности графического объекта `axes` того же графического окна. Здесь же реализуется возможность получить любую *справочную информацию* с помощью команды *меню* этого же графического окна.

В результате графические окна системы MATLAB, в которых реализованы все необходимые элементы управления и меню, становятся удобным интегрирующим механизмом, упрощающим использование всего ансамбля стандартных и самостоятельно разработанных функций, позволяющих решать сложные задачи и визуализировать результаты. Но самое главное заключается в том, что работа в таком графическом окне становится наглядной, удобной и простой. Из командного окна системы MATLAB потребуется лишь один раз вызвать М-функцию, создающую такое графическое окно, после чего любой сторонний пользователь сможет самостоятельно с ним работать.

Степень простоты и удобства работы зависит, конечно, от самоочевидности реализованного пользовательского интерфейса и от полноты и ясности средств документирования и оперативной помощи. В любом случае описанный интегрирующий механизм на базе графических окон и элементов управления позволяет создавать *законченные отчуждаемые приложения*, которые можно передавать сторонним пользователям для последующего многократного применения.

Простота создания графических элементов управления и меню в рамках пакета MATLAB позволяет дополнительно охарактеризовать его как *средство быстрой разработки* научных проектов.

В следующем подразделе мы будем систематически изучать графические объекты управления и практически создадим объект `axes`, предназначенный для построения графиков функций (мы с ним уже знакомы по темам, связанным с визуализацией), а также несколько объектов общего типа `uicontrol`. К последним относятся *командные кнопки*, *текстовые поля* с возможностью редактирования текста и без такой возможности, *переключатели* и *списки*.

А в текущем подразделе мы в основном рассмотрим общие свойства всех графических объектов системы MATLAB. При этом мы еще раз повторим некоторые из уже известных нам графических объектов и создадим один элемент управления – командную кнопку.

Как мы уже знаем, все графические объекты выполнены в системе MATLAB по *объектной технологии* и характеризуются присущим им *набором свойств*. Меняя последние, мы модифицируем их внешний вид и поведение. Чтобы изменить свойства графических объектов, нужно получить доступ к ним по их *описателю* (числу, уникально идентифицирующему конкретный объект).

Описатели графического объекта возвращают *функции-конструкторы*, создающие эти объекты. Имена функций-конструкторов совпадают с именами графических объектов. Запомнив описатели в глобальных переменных, мы всегда впоследствии будем иметь возможность простого доступа к ним.

Другим способом получения описателей является использование специальных функций системы MATLAB, таких, как функция `gcf` (`get current figure` – получить описатель текущего графического окна). Функция-конструктор `figure` создает графическое окно стандартного (заданного по умолчанию) цвета. Если нам требуется изменить цвет на красный, то это легко выполнить, получив описатель окна функцией `gcf`, а затем присвоив новое значение свойству `'Color'`:

```
hFig = gcf;
set( hFig, 'Color', 'red' );
```

или еще короче:

```
set( gcf, 'Color', 'red' )
```

Изо всех графических объектов системы MATLAB нас сейчас будут интересовать графические элементы управления. Эти объекты (они являются окнами в смысле операционной системы Windows) на поверхности графического окна создаются по иерархической схеме «родитель – потомство». Родительским окном для элементов управления служит само графическое окно системы MATLAB.

Для поиска описателей графических объектов удобно применить функцию `findobj`:

```
hArray=findobj(hParent,'имя_свойства',значение_свойства)
```

так как эта функция отыскивает все объекты, являющиеся потомками объекта с описателем `hParent` и имеющие для свойства `'имя_свойства'` значение, указанное в параметре `значение_свойства`. Она возвращает их описатели в массиве `hArray`.

Этой функцией широко пользуются для доступа к описателям конкретных элементов управления, которые являются потомками графического окна системы MATLAB, так что вместо `hParent` можно использовать `gcf`. В качестве удобного для поиска свойства часто употребляется свойство `'Tag'` (*ярлык*), значением которого является произвольно присваиваемый элементу управления текстовый идентификатор. Если мы разным объектам присвоим разные идентификаторы в качестве значений свойства `'Tag'`, то не будет проблем с поиском описателей этих объектов.

А теперь для примера создадим на поверхности графического окна командную кнопку:

```
hF1 = figure;
uicontrol( hF1,'Style','pushbutton',...
           'String', 'MyButton1',...
           'Position', [ 10 10 70 30 ]  );
```

Элементы управления в системе MATLAB имеют тип `uicontrol`. Они создаются функцией-конструктором `uicontrol`, у которой первым параметром идет описатель родительского окна, а затем по очереди перечисляются имена и значения свойств, которым мы явно придаем собственные значения (а остальные, менее важные для нас свойства, получают значения по умолчанию). В итоге имеем графическое окно, в котором явственно видна кнопка (см. рис. 9.1). Эта кнопка визуально действует безупречно – с помощью левой клавиши мыши она нажимается (виден процесс заглубления поверхности кнопки) и отжимается, но при этом не происходит никаких действий в качестве последствий нажатия. Это происходит потому, что мы еще не приписали этой кнопке функций, выполнение которых должно быть *реакцией* на нажатие.

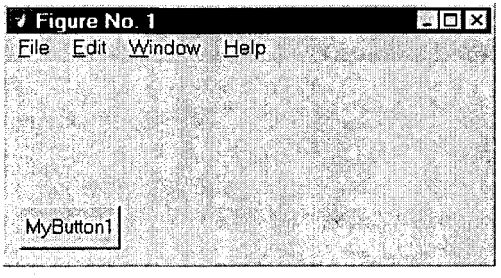


Рисунок 9.1

В функции `uicontrol`, создающей элемент управления, самым важным параметром после описателя родительского окна является свойство `'Style'`, так как оно задает *тип* управляющего элемента. Задав для этого свойства значение `'pushbutton'`, мы создали именно кнопку (а не редактируемое поле или что-нибудь еще).

Имена двух других свойств говорят сами за себя: `String` задает надпись на поверхности кнопки (в данном случае это `MyButton1`), а `Position` имеет значением вектор-строку из четырех чисел и задает положение управляющего элемента относительно левого нижнего угла графического окна. Если более конкретно, то положение левого нижнего угла кнопки относительно левого нижнего угла графического окна задают первые два элемента числовой строки. Третий же элемент этой строки задает ширину кнопки, а четвертый – высоту кнопки.

Создание основных элементов управления

В предыдущем подразделе мы создали командную кнопку. Теперь будем создавать другие элементы управления.

Для ввода и редактирования входной информации (числовой и текстовой) предназначены *текстовые поля с возможностью редактирования*. Они создаются командой

```
uicontrol( hParent, 'Style', 'edit', … );
```

где многоточие означает дополнительные свойства, одно из которых, `'Position'`, нужно указывать практически обязательно, иначе управляющий элемент будет расположен в неудачном месте. Теперь если в дополнение к ранее созданной кнопке выполнить команду

```
uicontrol( hF1, 'Style', 'edit',…
          'Position', [ 100 10 70 30 ],…
          'BackgroundColor', 'white',…
          'HorizontalAlignment', 'left' );
```

то графическое окно примет вид, изображенный на рис. 9.2.

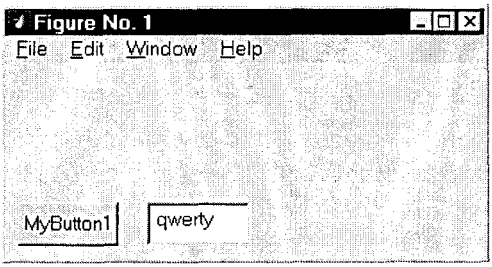


Рисунок 9.2

Для редактирующего элемента мы прописали свойство `'BackgroundColor'` (цвет фона – сделали его белым вместо темно-серого по умолчанию) и свойство `'HorizontalAlignment'` (тип выравнивания текста в горизонтальном направлении – сделали *выравнивание по левой границе* вместо *выравнивание по центру*, имеющему место по умолчанию).

Заметим здесь, что свойство `'BackgroundColor'` *нельзя произвольно задавать для кнопок* (требование операционной системы Windows).

Как и в случае с кнопкой, созданный редактирующий элемент сразу же проявляет свою функциональность: в нем можно мышью поставить тестовый курсор (переместить на него *фокус ввода с клавиатуры*) и вводить произвольный текст и числа с клавиатуры. На рисунке видно, что мы успели ввести «магическое» буквосочетание `qwerty`.

Теперь создадим *текстовое поле без редактирования* (часто называют статическим текстом). На поверхности этого элемента можно расположить произвольный поясняющий текст. Этот текст можно изменять программно, но его нельзя ввести с клавиатуры. Вот пример вызова функции `uicontrol`, создающей нередактируемое текстовое поле, которое расположено чуть выше редактирующего элемента:

```
uicontrol( hF1, 'Style', 'text',…
          'Position', [ 100 50 70 30 ],…
          'BackgroundColor', 'white',…
```

```
        'String','Hello, World!',...
        'HorizontalAlignment', 'center' );
```

Теперь наше графическое окно содержит уже три элемента управления (см. рис. 9.3).

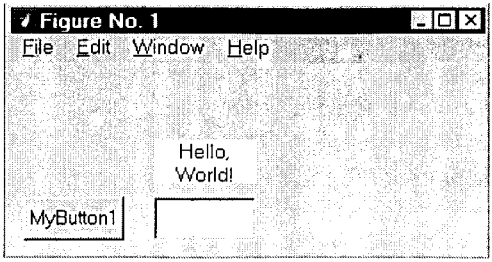


Рисунок 9.3

Как видно из приведенного фрагмента, для создания нередактируемого элемента свойство `'Style'` должно иметь значение `'text'`.

В отличие от ранее созданного редактирующего элемента, здесь мы выбрали тип выравнивания текста *по центру* (присвоив значение `'center'` свойству `HorizontalAlignment`), так как оно более уместно в данном конкретном случае общей надписи над редактирующим элементом.

Далее создадим элемент управления, называемый *флажком* (*checkbox*):

```
uicontrol( hF1, 'Style', 'checkbox',...
        'Position',[10 100 150 30],...
        'String','Important option',...
        'HorizontalAlignment', 'left' );
```

Графическое окно примет вид, изображенный на рис. 9.4.

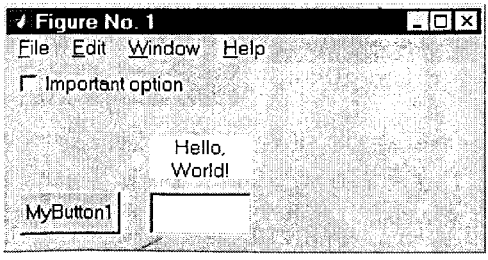


Рисунок 9.4

Этот элемент управления может пребывать *в одном из двух состояний*: *отмеченном* или *неотмеченном*. В первом из них в квадрате должна стоять *галочка* (отметка), которую проставляют щелчком левой кнопки мыши на этом элементе управления. Повторный щелчок убирает галочку.

Наконец, создадим *список*, содержащий несколько имен функций:

```
uicontrol( hF1, 'Style', 'listbox',…
           'Position',[200 10 100 80],…
           'String',{'sin','cos','tan'},…
           'HorizontalAlignment', 'left' );
```

Так как список (свойство 'Style' должно быть 'listbox') содержит сразу несколько строковых значений для свойства 'String', а должно быть одно значение, то это единственное значение является *массивом ячеек* (каждый элемент этого массива есть текстовая строка, то есть массив типа char).

Внутри списка щелчком мыши можно *выделить* (подсветить) какую-либо из его строк. На рис. 9.5 показано, что выделена вторая строка.

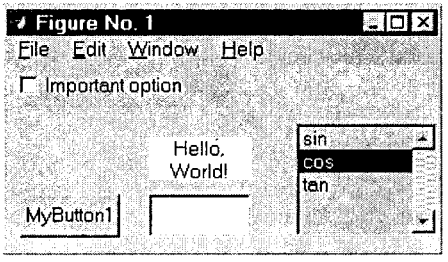


Рисунок 9.5

В дальнейшем программным способом эта выделенная строка может быть извлечена из списка.

В созданное нами окно мы еще дополнительно поместим графический объект axes, предназначенный для показа графиков. Он не является графическим объектом типа uicontrol. Прежде чем перейти в следующем подразделе к встраиванию в наше окно объекта axes, перечислим и проиллюстрируем остальные графические элементы управления: togglebutton, radiobutton, slider, frame. Имеется еще один элемент управления – элемент popupmenu, но мы его не будем рассматривать, так как он является просто другим вариантом списка – так называемым *раскрывающимся списком* (в обычном состоянии он свернут в одну строку).

Элемент управления togglebutton является кнопкой, отличающейся от элемента pushbutton своим поведением. Этот вариант кнопки может стабильно находиться в одном из двух состояний: *нажатом* или *отжатом*. С точки зрения внешнего вида он соответствует обычной командной кнопке. Проиллюстрируем все сказанное практическим примером. Вот код, который создает в новом графическом окне (Figure No. 2) кнопку togglebutton:

```
hF2 = figure;
uicontrol( hF2,'Style','togglebutton',…
           'String', 'MyButton1',…
           'Position', [ 10 10 70 30 ]  );
```

На рис. 9.6 эта кнопка показана в нажатом состоянии, поскольку в ненажатом она полностью совпадает по внешнему виду с командной кнопкой.

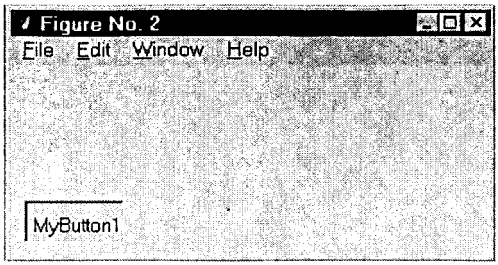


Рисунок 9.6

В нажатом состоянии такая кнопка может пребывать сколь угодно долго, пока она не будет возвращена в ненажатое состояние очередным щелчком левой клавиши мыши. Новый же щелчок мышью снова переведет такую кнопку в нажатое состояние. Таким образом, этот элемент управления может пребывать в одном из двух состояний, каждое из которых легко наблюдается пользователем и сообщает ему о выборе одного из двух вариантов поведения данного приложения. Характеристика, выбор которой осуществляется такой кнопкой, зависит от конкретного приложения. Поясняющие надписи могут быть расположены рядом с кнопкой.

Другим графическим элементом управления, стандартно предназначенным для осуществления выбора *одного из нескольких вариантов* (от двух и выше) той или иной характеристики, является элемент `radiobutton`. Вот код, создающий три элемента управления стиля `radiobutton`, из которых только один может находиться в отмеченном состоянии в каждый конкретный момент времени:

```
uicontrol( hF2,'Style','radiobutton',...
           'String', 'Third',...
           'Position', [ 120 10 70 30 ]  );
uicontrol( hF2,'Style','radiobutton',...
           'String', 'Second',...
           'Position', [ 120 50 70 30 ]  );
uicontrol( hF2,'Style','radiobutton',...
           'String', 'First',...
           'Position', [ 120 90 70 30 ]  );
```

На рис. 9.7 показано изображение графического окна `Figure No.2`, в котором помимо ранее созданной кнопки стиля `togglebutton` располагаются также три элемента управления `radiobutton`, причем в отмеченном состоянии (точка в пределах маленького кружка) находятся сразу два из этих элементов.

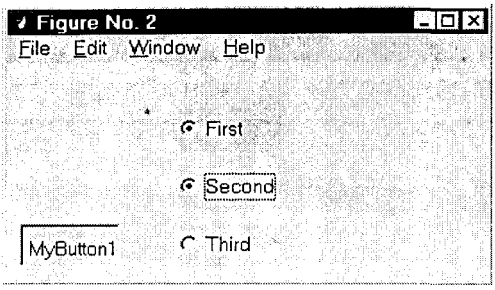


Рисунок 9.7

Отсюда видно, что стандартная функциональность элементов управления `radiobutton`, допускающая возможность единственного выбора, не обеспечивается автоматически сразу же после создания группы таких элементов. Эту функциональность *требуется обеспечить программно*, когда с каждым таким элементом связывается *программа-обработчик события* (нажатия левой клавиши мыши на элементе), которая должна гасить отметки на всех остальных элементах управления стиля `radiobutton`.

Набор элементов стиля `radiobutton` желательно визуально сгруппировать, для чего следует применить графический элемент управления (реально он ничем не управляет, кроме внешнего вида) стиля `frame`. Его нужно создать *перед созданием* элементов `radiobutton`, а последние нужно располагать на его поверхности:

```
uicontrol( hF2,'Style','frame',…
           'Position', [ 115 5 80 120 ] );
uicontrol( hF2,'Style','radiobutton',…
           'String', 'Third',…
           'Position', [ 120 10 70 30 ] );
uicontrol( hF2,'Style','radiobutton',…
           'String', 'Second',…
           'Position', [ 120 50 70 30 ] );
uicontrol( hF2,'Style','radiobutton',…
           'String', 'First',…
           'Position', [ 120 90 70 30 ] );
```

Вот какой при этом получается внешний вид графического окна (см. рис. 9.8).

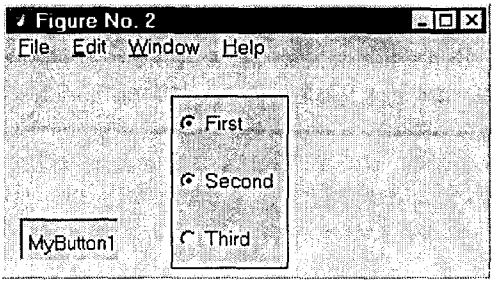


Рисунок 9.8

Еще раз отметим, что элемент управления `frame` должен создаваться *раньше* располагающихся над его поверхностью других элементов управления. Если поступить наоборот, то элемент `frame` накроет собой вышеперечисленные элементы управления и они будут не видны. Про это свойство говорят в таких терминах, что элемент `frame` является *непрозрачным*.

Ну и, наконец, мы подошли к последнему графическому элементу управления, который обычно называют *ползунком* (по-английски – *slider*). Этот элемент управления имеет стиль `slider`:

```
uicontrol( hF2,'Style','slider',…
           'Position', [ 10 90 70 30 ] );
```

Теперь наше иллюстративное графическое окно `Figure No. 2` включает еще один элемент управления и принимает следующий вид, изображенный на рис. 9.9.

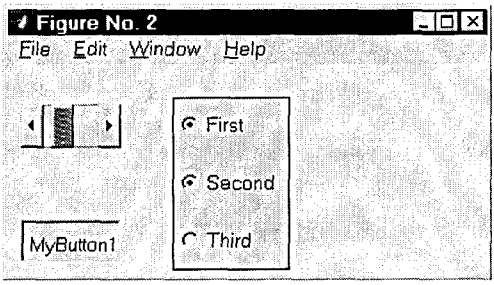


Рисунок 9.9

Графический элемент управления `slider` позволяет мышью перемещать свой ползунок, относительное положение которого задает некоторое входное число. Самое левое положение ползунка соответствует свойству `Min` этого элемента управления, а самое правое положение – свойству `Max`. Оба этих свойства нужно заранее приписать элементу управления для задания диапазона изменения входной величины. Если явно не задать числовых значений для этих свойств, то по умолчанию будут приняты величины соответственно `0` и `1`.

Завершая рассказ о графических элементах управления, добавим еще одну немаловажную деталь их оформления. В случае идеально спроектированного *пользовательского интерфейса* (под которым можно понимать совокупность всех элементов управления и их функциональное предназначение) его применение является для пользователя самоочевидным. В остальных случаях (а их, естественно, большинство) будет полезным снабдить каждый создаваемый элемент управления *всплывающей подсказкой* (по-английски – *tooltip*), которая действительно как бы всплывает над экраном при наведении курсора мыши на данный элемент.

Процесс снабжения графического элемента управления соответствующей ему всплывающей подсказкой весьма прост – при создании графического элемента управления функцией-конструктором `uicontrol` явно задавайте текстовое значение для его свойства `TooltipString`, которое и будет содержанием всплывающей подсказки (кроме *пассивных элементов управления* стилей `frame` и `text`).

Графический объект axes

С графическим объектом `axes` мы знакомились, когда обсуждали вопрос о построении графиков функций. Теперь, когда мы в пределах графического окна системы MATLAB собираем весь набор управляющих и отображающих элементов, работа с объектом `axes` по-прежнему является одной из наиболее важных.

Создается объект `axes` с помощью функции `axes` (конструктор объекта). На нашем учебном окне `Figure No. 1` осталось мало свободного места, однако мы ухитримся втиснуть туда графический объект `axes`, для чего специально уменьшим *размер шрифта* (`'FontSize'`), используемого для надписей на осях координат объекта `axes`:

```
axes( 'Parent', hF1, 'Color', [ 1 1 1],…
      'Units', 'points',…
      'Position', [ 12 33 40 22 ],…
      'FontSize', 6 );
```

В результате получается следующая картинка, не претендующая на оптимальность в размещении объектов в пределах графического окна (см. рис. 9.10).

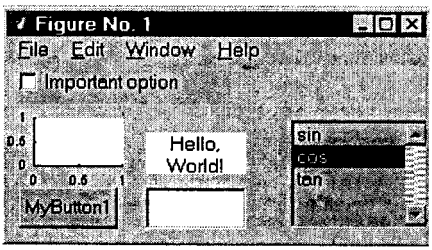


Рисунок 9.10

Все команды, которые создают само графическое окно, а также создают объект axes и элементы пользовательского интерфейса, удобно поместить в один М-файл. В итоге от пользователя потребуется минимальное количество действий – набрать в командном окне системы MATLAB имя этого М-файла и нажать клавишу Enter.

Приведем пример М-файла, создающего в графическом окне один объект axes, один список, четыре редактируемых поля для ввода числовой информации и две командные кнопки, не считая нескольких текстовых полей (без возможности редактирования), содержащих подписи, пояснения и разграничительные линии. Вот содержимое этого файла:

```
function MyFig1
%----------- global variables ------------------

global hFig1 hAxes
global hEd1 hEd2 hEd3 hEd4
global hList
global hBut1 hBut2
global hTxt1 hTxt2 hTxt3 hTxt4 hTxt5 hTxt6 hTxt7

%----------- figure and axes -------------------

hFig1 = figure
hAxes = axes( 'Parent',hFig1,'Color', [ 1 1 1],...
              'Units', 'points',...
              'Position', [ 20 60 230 230 ],...
              'FontSize', 10  );

%----------- four edits ------------------------
hEd1  = uicontrol( hFig1,'Style','edit',...
                   'BackgroundColor',[1 1 1],...
                   'Position',[435 400 190 30],...
                   'HorizontalAlignment','left' );
hEd2  = uicontrol( hFig1,'Style','edit',...
                   'BackgroundColor',[1 1 1],...
                   'Position',[435 320 190 30],...
                   'HorizontalAlignment','left' );
hEd3  = uicontrol( hFig1,'Style','edit',...
                   'BackgroundColor',[1 1 1],...
                   'Position',[435 230 190 30],...
                   'HorizontalAlignment','left' );
hEd4  = uicontrol( hFig1,'Style','edit',...
```

```
                    'BackgroundColor',[1 1 1],...
                    'Position',[435 150 190 30],...
                    'HorizontalAlignment','left' );

%---------- list control ----------------------

hList = uicontrol( hFig1,'Style','listbox',...
                    'Position',[485 15 140 90],...
  'String',{'sin','cos','MyFunct1','MyFunct2','f3'},...
                    'HorizontalAlignment', 'left' );

%---------- two pushbuttons -------------------

hBut1 = uicontrol( hFig1,'Style','pushbutton',...
                    'Position',[60 25 140 40],...
                    'String', 'GO' );
hBut2 = uicontrol( hFig1,'Style','pushbutton',...
                    'Position',[240 25 140 40],...
                    'String', 'CLEAR' );

%------------ 7 text fields ---------------------

hTxt1 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0.7 0.7 0.7],...
                    'Position', [90 475 240 20],...
                    'String','Euler solution' );
hTxt2 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0.7 0.7 0.7],...
                    'Position', [435 440 60 20],...
                    'String','X initial' );
hTxt3 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0.7 0.7 0.7],...
                    'Position', [435 360 60 20],...
                    'String','Y initial' );
hTxt4 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0.7 0.7 0.7],...
                    'Position', [435 260 60 20],...
                    'String','X final' );
hTxt5 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0.7 0.7 0.7],...
                    'Position', [435 180 60 20],...
```

```
                    'String','N' );
hTxt6 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0 0 0],...
                    'Position', [430 5 1 900] );
hTxt7 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0.7 0.7 0.7],...
                    'Position', [435 140 240 20],...
                    'String','Choose function' );
%----------------- the end of file --------------------
```

Отметим, что из семи созданных текстовых полей (без возможности редактирования) шесть применяются в виде поясняющих надписей, расположенных над редактируемыми текстовыми полями ввода, над списком и над объектом `axes`. Седьмое же текстовое поле (его описатель `hTxt6`) не содержит никаких надписей, так как мы не задаем явно его свойство `String`. Оно по умолчанию равно пустой строке.

Возникает законный вопрос, зачем же нужно такое текстовое поле, если в нем нет никакого текста? Ответ на этот вопрос заключается в том, что графический элемент управления стиля `text` можно использовать для нанесения на поверхность графического окна системы MATLAB различных элементов оформления, например разграничительных линий. Именно такую вертикальную линию мы и формируем, создавая текстовое поле с помощью кода

```
hTxt6 = uicontrol( hFig1,'Style','text',...
                    'BackgroundColor',[0 0 0],...
                    'Position', [430 5 1 900] );
```

в котором задаем *цвет фона управляющего элемента* отличным от цвета фона графического окна (цвет `[0 0 0]` – это черный цвет, а цвет графического окна по умолчанию серый). Кроме того, мы задаем ширину этого элемента в один пиксел.

В результате получается графическое окно, изображенное на рис. 9.11.

Это окно предназначено для численного решения дифференциального уравнения

```
y'(x) = F(x);
Xinitial < x < X final;
Yinitial = y( Xinitial );
```

где функция `F(x)` выбирается из списка, расположенного в правом нижнем углу графического окна. Остальные параметры задаются с клавиатуры в редактируемых полях. Кнопка `GO` предназначена для начала расчетов по методу Эйлера и показа результата в виде графика на поверхности объекта `axes`. Кнопка `CLEAR` позволяет очистить объект `axes` от содержимого.

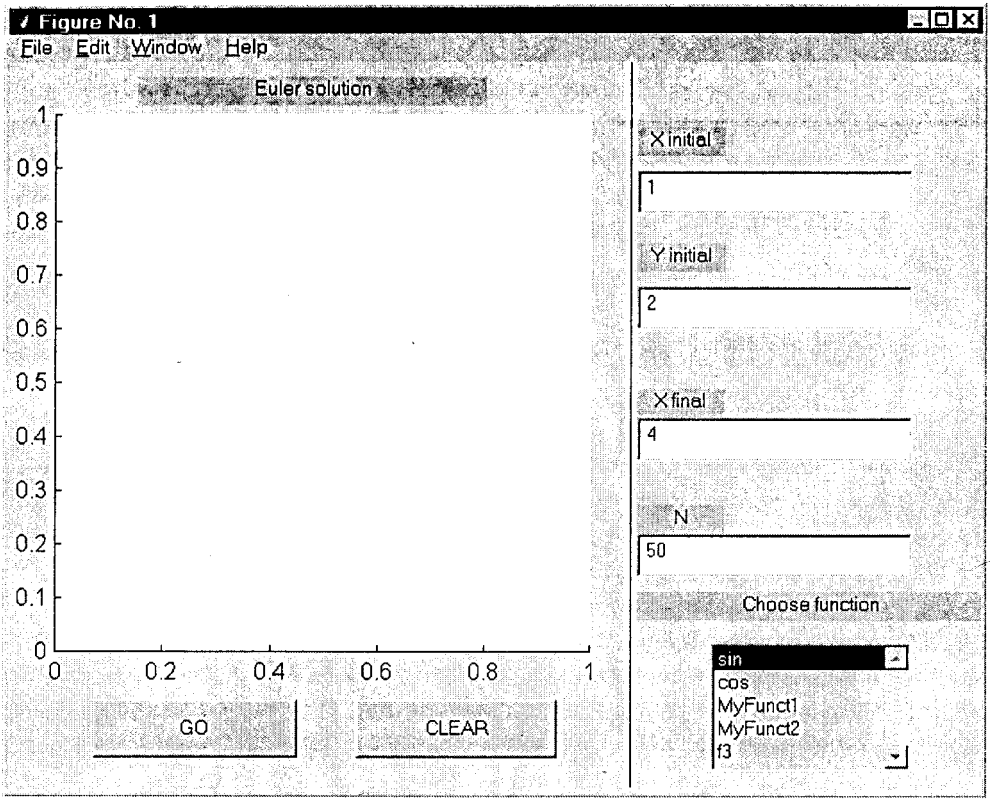


Рисунок 9.11

Как мы видим из рисунка, предусмотрены расчеты со стандартными функциями `sin(x)` и `cos(x)`, а также с собственными функциями `MyFunct1(x)`, `MyFunct2(x)` и `f3(x)`. Последние должны быть записаны в М-файлы `MyFunct1.m`, `MyFunct2.m`, `f3.m`.

Число `N` вводится перед расчетами с клавиатуры и определяет шаг интегрирования, который равен

```
( Xfinal - Xinitial ) / N.
```

Чтобы запустить программы расчетов, нужно не только написать алгоритмы решения на М-языке и сохранить их тексты в М-файлах, но и связать эти программы с командными кнопками.

Callback-функции

Функции, которые связываются с графическими элементами управления, называют *callback-функциями* (их не вызывают наши М-функции; они вызываются средой MATLAB у нас «за спиной», отсюда и происхождение их названия).

Чтобы связать наши командные кнопки с callback-функциями, которые будут вызываться средой MATLAB при «нажатии» на эти кнопки, при их создании функцией `uicontrol` нужно прописать свойство `'Callback'`, указав ему в качестве значения имя соответствующей М-функции. Эти callback-функции можно также назвать *функциями обработки событий*, связанных с элементами управления. Это значит, что в ранее представленном М-файле `MyFig1.m` нужно изменить строки, касающиеся создания кнопок. Вот этот измененный код, позволяющий связать кнопки с их callback-функциями:

```
hBut1 = uicontrol( hFig1, 'Style', 'pushbutton',...
                   'Position', [60 25 140 40],...
                   'String', 'GO',...
                   'Callback', 'MyGO' );
hBut2 = uicontrol( hFig1, 'Style', 'pushbutton',...
                   'Position', [240 25 140 40],...
                   'String', 'CLEAR',...
                   'Callback', 'MyCLEAR' );
```

Если функция, которая будет вызываться при нажатии на кнопку `GO`, будет записана в файле `MyGO.m`, то в качестве значения свойства `'Callback'` обязательно нужно написать значение в виде `'MyGO'` и *нельзя* писать `'MyGO.m'` (возникнет ошибочная ситуация).

Внеся указанные выше изменения в файл `MyFig1.m`, перейдем к написанию функций `MyGO` и `MyCLEAR`. Начнем с более простой функции `MyCLEAR`. Вот текст этой функции:

```
function MyCLEAR
%---------------
global hAxes
axes( hAxes );
cla;
```

Ранее мы уже в файле `MyFig1.m` обозначили все описатели созданных нами графических объектов как *глобальные*. Таким образом, у нас имеется очень простой к ним доступ. Все функции, где эти описатели маркированы как глобальные, имеют к ним непосредственный доступ. Именно так все и происходит в функции `MyCLEAR` (ее мы записываем в файл `MyCLEAR.m`).

Поясним текст этой функции. Здесь команда cla стирает содержимое текущего (активного) объекта axes. Предварительный вызов функции

```
axes( hAxes )
```

позволяет гарантировать, что будет стерто содержимое именно объекта axes с описателем hAxes, так как такой вызов делает активным объект, описатель которого указан в качестве параметра. У нас в данной конфигурации имеется всего один объект axes, так что этот вызов не является обязательным, но он иллюстрирует решение проблемы в случае нескольких объектов axes.

Теперь займемся функцией MyGO, которая запускается по нажатии кнопки GO. Эта функция опрашивает редактируемые поля, в результате чего инициализируются необходимые для счета переменные, после чего производит вычисление решения дифференциального уравнения в соответствии с простейшим алгоритмом Эйлера:

```
function MyGO
global hAxes  hEd1  hEd2  hEd3  hEd4  hList

%-- get string information from edit fields ---

str1=get( hEd1,'String' ); str2=get( hEd2,'String' );
str3=get( hEd3,'String' ); str4=get( hEd4,'String' );

%-- convert strings to numbers  ---------------

x0 = str2num( str1 ); y0 = str2num( str2 );
xf = str2num( str3 ); N  = str2num( str4 );

%-------  get function name  ------------------

index   = get( hList, 'Value' );
cellArr = get( hList, 'String' );
funName    = cellArr{ index };

%------------  Euler algorithm  ---------------

dx = ( xf - x0 ) / N;
X=x0; Y=y0;

for k=1:N
   Y=[ Y, Y(end) + feval( funName,  X(end) ) * dx ];
   X=[ X, X(end) + dx ];
end
```

```
%------  graph plotting  ---------------------

axes( hAxes );
plot( X , Y );

%--------------  the end of file  -------------
```

С помощью функций `get`, указывая им в качестве первого аргумента описатели редактирующих полей, мы получаем значения свойства `'String'`, то есть текстовое содержимое этих полей, которое вводится пользователем с клавиатуры. Это и есть входные данные. Только пока что числовые данные представлены в строковой (текстовой) форме.

Для получения имени функции, представляющей правую часть дифференциального уравнения, сначала у списка запрашиваем через свойство `'Value'` индекс подсвеченной (выбранной или отселектированной) строки, затем читаем значение свойства `'String'`. Последнее для списка является массивом ячеек, откуда и выбираем имя функции с помощью операции индексирования массива ячеек фигурными скобками.

С помощью функций `str2num` преобразовываем текстовые величины в числа. Затем устанавливаем начальные значения для массива `X` значений аргумента и массива `Y` значений функции на сетке. *Шаг сетки* (шаг интегрирования) `dx` вычисляется делением длины отрезка интегрирования на количество шагов `N`, которое пользователь вводит с клавиатуры. Наращивание массивов осуществляется операцией конкатенации, в то время как очередное значение функции вычисляется согласно алгоритму метода Эйлера и равно

```
feval( funName, X(end) ) * dx
```

Ключевое слово системы MATLAB `end` в индексирующих выражениях означает ссылку на последний элемент массива, что нам и требуется в данном случае. С помощью функции `feval` осуществляется вызов функции с именем `funName`, прочитанным из списка, и ей передается аргумент `X(end)` (текущее значение независимого аргумента).

В конце концов осуществляется построение графика вычисленного решения дифференциального уравнения с помощью функции `plot`. Перед вызовом этой функции с помощью `axes( hAxes )` гарантируется, что наш объект типа `axes` является активным, и именно в нем функция `plot` строит график функции.

Проверим работу приложения для пользовательских функций `MyFunct1`, `MyFunct2` и `f3`. Текст функции `MyFunct1`

```
function ret = MyFunct1( x )
ret = x ^ 3;
```

записываем в файл `MyFunct1.m`. Текст функции `MyFunct2`

```
function ret = MyFunct2( x )
ret = 1 / sin( x );
```

записываем в файл `MyFunct2.m`. Наконец, текст функции `f3`

```
function ret = f3( x )
ret = sin( x * x );
```

записываем в файл `f3.m`.

Начинаем с простой функции `cos(x)` (см. рис. 9.12).

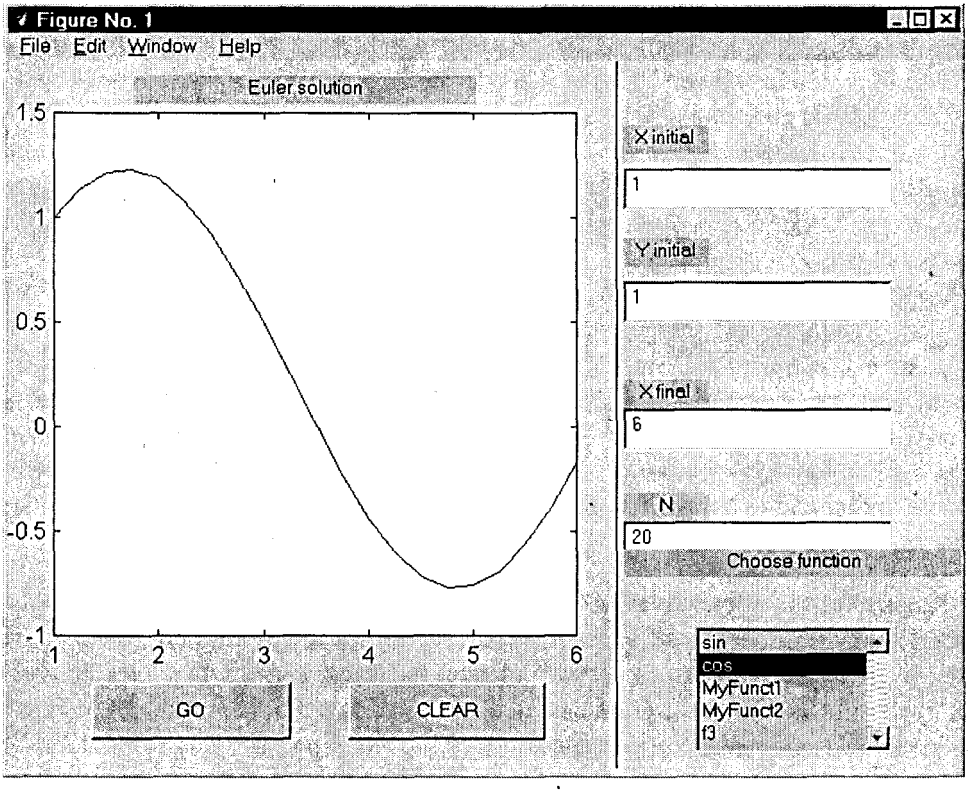


Рисунок 9.12

Так как дифференциальное уравнение

`y'(x)=cos(x)`

имеет аналитическое решение

```
y(x) = Yinitial - sin( Xinitial ) + sin(x)
```

то в полученном графике легко узнать график функции `sin(x)`, сдвинутый вдоль вертикальной оси вверх.

Также очевидные решения получаются для функций `sin` и `MyFunct1`, так как и в этих случаях существуют тривиальные аналитические решения. В случае выбора функции `MyFunct2` аналитическое решение существует в виде довольно замысловатой функции

```
Yinitial + ln( |  tg( x / 2 ) / tg( Xinitial / 2 ) | )
```

а в случае функции `f3` аналитического решения не существует и единственным способом остаются непосредственные численные расчеты.

Для дифференциального уравнения

```
y'(x) = f3(x)
```

где функция `f3(x) = sin(x^2)`, не существует аналитического решения. Единственным способом получить решение этого дифференциального уравнения остается способ численного решения. Это мы и делаем с помощью разработанного нами приложения, опираясь на самый простой численный метод – метод Эйлера.

Рисунок 9.13, на котором показано наше графическое окно, иллюстрирует полученное таким образом решение для представленного выше дифференциального уравнения.

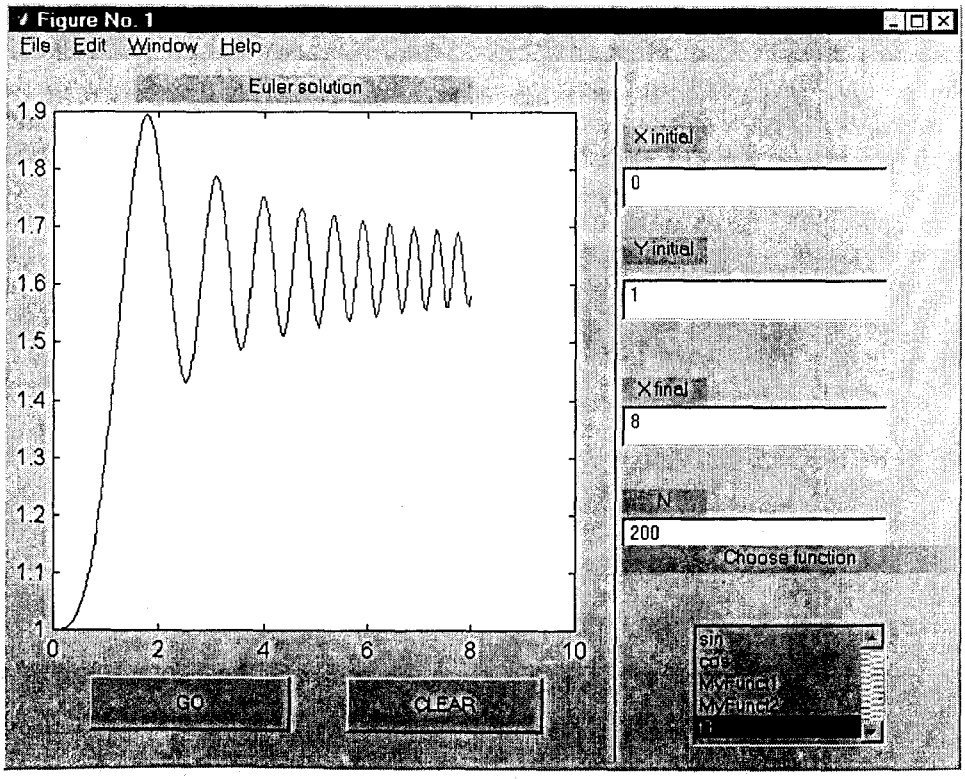


Рисунок 9.13

Созданное нами законченное приложение в рамках среды MATLAB удобно применять также для иллюстрации процесса сходимости приближенного решения к точному решению по мере увеличения числа шагов интегрирования N (то есть уменьшения шага интегрирования).

Применение утилиты guide для формирования пользовательского интерфейса

Создание графических окон с элементами управления возможно не только с помощью *ручного* написания соответствующих М-файлов, изобилующих вызовами функций-конструкторов типа `uicontrol`, `axes` и т. д. Этот способ, который мы рассмотрели подробно, является *наилучшим с точки зрения максимального контроля разработчика над процессом создания графических окон.*

В случае недостаточной квалификации разработчика (или необходимости очень быстрого выполнения работы) ему на помощь может прийти специальное *визуальное средство разработки*, входящее в состав пакета MATLAB – так называемая *утилита guide*. По команде `guide`, которая вводится в командном окне системы MATLAB, на экран дисплея выводятся два окна. Одно из них является привычным для нас графическим окном-заготовкой, а другое является главным окном утилиты `guide` (в заголовке этого окна стоит надпись `Guide Control Panel`) и содержит *палитру графических элементов управления* (см. рис. 9.14).

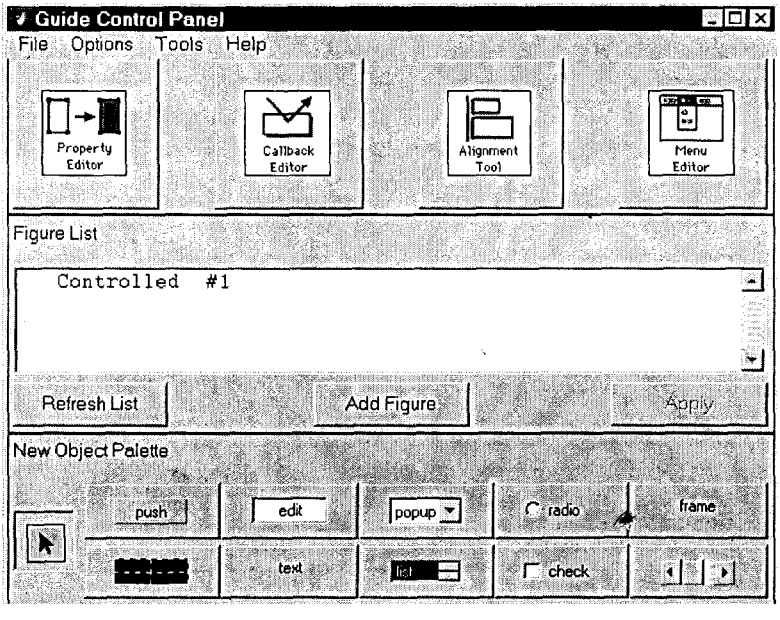


Рисунок 9.14

Эта палитра графических элементов управления расположена в нижней части окна. Из представленного рисунка хорошо видно, что все элементы снабжены названиями, кроме *ползунка* (его графическое изображение с очевидностью отражает внешний вид этого элемента) и объекта `axes` (пунктиром символически показаны оси координат).

С помощью мыши можно методом буксировки «перетаскивать» эти элементы на создаваемое собственное графическое окно, что и показано на рис. 9.15

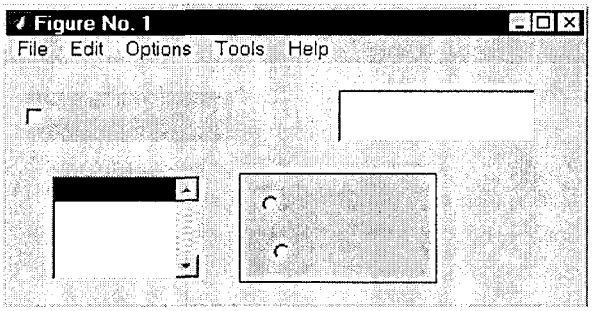


Рисунок 9.15

Белые штрихи по краям клиентской части графического окна показывают, что это проектируемое окно находится под управлением утилиты `guide`. Этот факт отражается также и в средней части окна `Guide Control Panel` в виде надписи

```
Controlled #1
```

Далее, когда элемент управления уже размещен на поверхности графического окна, двойной щелчок мышью на его поверхности вызывает специальный редактор свойств этого элемента `Graphics Property Editor`, что и показано на рис. 9.16 для элемента стиля `checkbox`.

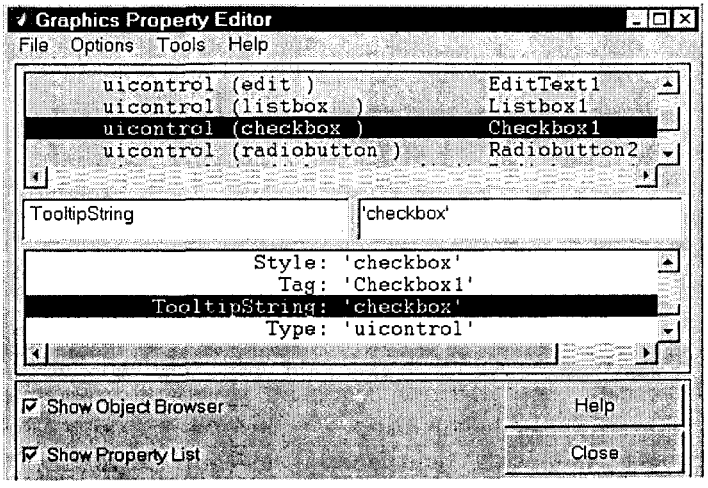


Рисунок 9.16

Здесь в средней части окна расположен список всех свойств данного графического элемента управления (для которого это окно редактора и вызвано). На рисунке показано, что в данный момент выбрано для редактирования свойство `TooltipString` и ему присвоено значение `'checkbox'` (это не очень интересная и малоинформативная всплывающая подсказка, так как и без того ясно, что это элемент стиля `checkbox`).

Таким образом прописываются все необходимые свойства (тут же легко можно, пролистав список, посмотреть все умолчательные значения свойств элементов управления). В частности, здесь для свойства `Callback` вписывается имя М-функции, служащей в качестве callback-функции для элемента управления. Саму эту функцию разрабатывают, отлаживают и записывают в М-файл самым обычным образом. На этом завершается формирование функциональности разрабатываемого интегрированного графического окна, после чего можно закрывать утилиту `guide`. Перед фактическим закрытием этой утилиты вам будет задан вопрос, намерены ли вы сохранить результаты работы в М-файле на диске. Естественно, надо будет ответить положительно и, выбрав имя для этого файла, сохранить в нем созданные за вас утилитой `guide` все команды и функции, которые в предыдущем подразделе мы писали самостоятельно.

Если потребуется что-то переделать в созданном утилитой `guide` интегрированном графическом окне, нужно будет сначала из командного окна системы MATLAB вызвать на выполнение созданную в предыдущем сеансе М-функцию, после чего выполнить команду `guide`, и эта утилита снова возьмет под управление графическое окно, подлежащее переделке. По окончании работы не забудьте сохранить все изменения в том же самом М-файле.

Изучение средства `guide` в полном объеме (то есть во всех деталях и с указанием полезных практических приемов) удобнее осуществлять на практике (самостоятельной или под руководством преподавателя на практическом занятии, повторяя демонстрируемые действия). Поэтому мы здесь не будем больше останавливаться на этом вопросе.

Динамическая перестройка элементов управления

Читателю после беглого прочтения может показаться, что мы слишком пренебрежительно отнеслись к средству быстрой разработки – к утилите `guide`. Он может счесть ее вполне конкурентоспособной в сравнении с ручным методом. Вынуждены разочаровать такого читателя – это не так.

Сначала очень кратко остановимся на некоторых очевидных недостатках работы с утилитой `guide`. Этим инструментом довольно неудобно выравнивать элементы управления друг относительно друга, в то время как при ручном способе достаточно просто прописать одинаковые значения координат для

свойства Position этих элементов управления. Есть и другие подобного рода мелкие недостатки. Но более существенным является проблема нахождения описателей для всех созданных элементов управления (бывают случаи, когда не срабатывает даже специально предназначенная для этого функция findobj). В случае ручной разработки мы преспокойно все описатели объявляли как глобальные – и проблема переставала существовать.

По мере роста сложности композиции графических элементов управления в пределах одного графического окна последняя проблема становится все более острой, особенно при плотном взаимодействии этих элементов управления между собой, когда требуется интенсивное обращение к свойствам элементов из различных callback-функций.

И наконец, в проектах, где требуется сложная *динамическая перестройка элементов управления* в процессе работы всего приложения, метод работы с утилитой guide становится практически бесполезным.

Поэтому мы сейчас приступим к разработке такого специфического проекта, опираясь на самый мощный, ручной способ формирования интегрированного графического окна.

Не будем слишком мудрствовать и реализуем проект, в котором на этапе построения графика функции объект axes расширяется и перекрывает часть из элементов управления, но при этом появляется новая кнопка, нажатие на которую возвращает все в исходное положение. Основная идея этого учебного проекта состоит в преодолении типичной проблемы нехватки свободного места при большом количестве необходимых элементов управления. В других случаях требуется динамически отображать в пределах одного и того же графического окна разное число объектов axes. Возможны и иные мотивы для динамической перестройки числа графических объектов и их взаимного расположения.

В качестве иллюстративной математической задачи воспользуемся ранее рассмотренным в гл. 4 численным методом решения жестких дифференциальных уравнений, реализуемым «решателем» ode15s. В гл. 4 мы вызывали эту функцию из командного окна, а результаты ее работы показывали в отдельном графическом окне, которое появляется как реакция на вызов функции plot. Теперь мы все это объединим в единый управляющий центр на базе интегрированного графического окна с элементами управления.

Если бы не необходимость сделать учебный проект достаточно компактным (иначе он будет плохо обозримым), можно было бы в рамках одного графического окна развернуть большое количество элементов управления, позволяющих на первом этапе задавать множество входных параметров: виды уравнений, начальные значения, длины интервалов интегрирования, различные численные методы, различные их программные реализации и т. д. На втором этапе динамически показывался бы процесс решения, причем размер графической области показа был бы максимально большим, чтобы помимо него оставалось лишь минимум места для одной кнопки с надписью RETURN. Кроме того, можно было бы предусмотреть несколько вариантов показа связанной с процессом решения ин-

формации: число вычислений функций в правых частях системы уравнений, погрешности на каждом из шагов интегрирования. Последнюю информацию тоже можно было бы показывать в виде некоторого графика. Такие задачи со столь сложной композицией либо решаются созданием большого числа графических окон (что может запутать пользователя), либо должны реализовываться в пределах единственного окна, но с динамической перестройкой «всего и вся». И хотя последнее решение трудно для разработчика, для пользователя оно является предпочтительным, ведь от него не потребуется никаких усилий.

Мы сейчас в нашем учебном проекте реализуем лишь малую часть перечисленного, но это будут принципиальные моменты, которые и нужно проиллюстрировать в их практическом воплощении.

В принципе в динамической перестройке нет ничего сложного, все базируется на единственном свойстве элементов управления и объекта `axes`. Это свойство `Visible`, которому можно придавать два значения – `'on'` или `'off'`. В первом случае объект виден, а во втором случае – нет. По умолчанию этому свойству всегда присваивается значение `'on'`.

В итоге на одном и том же физическом месте в пределах графического окна можно создать несколько дочерних объектов. Далее, управляя свойством `Visible` этих объектов, а также их размерами и положением и изменяя некоторые другие их свойства, мы и реализуем задуманное.

Ниже представлен код, задача которого состоит в том, чтобы создать все необходимые объекты, как видимые в первый момент, так и невидимые. Невидимыми здесь являются объект `axes` и кнопка `hBut2`. Они будут показываться позже во время вычислений. До этого момента в графическом окне должны быть видны редактируемые поля для ввода данных, надписи над ними, центральная надпись о предназначении всего окна, а также кнопка `hBut1`, нажатие на которую инициирует вычисления.

Вот этот код, который мы размещаем в файле `MyDynamicFig.m`:

```
function MyDynamicFig

%---------- global variables -----------------

global hFig
global hAxes
global hEd1 hEd2 hEd3
global hBut1 hBut2
global hTxt1 hTxt2 hTxt3 hTxt4

%---------- figure and axes ------------------

hFig  = figure( 'Position', [100,100,550,300] );
```

```
hAxes = axes( 'Parent',hFig,'Color', [1 1 1],...
              'Units', 'pixels',...
              'Position', [20,55,510,230],...
              'FontSize', 10,...
              'Visible', 'off' );

%---------- three edits ----------------------

hEd1  = uicontrol( hFig,'Style','edit',...
                   'BackgroundColor',[1 1 1],...
                   'Position',[435 220 100 30],...
                   'HorizontalAlignment','left' );
hEd2  = uicontrol( hFig,'Style','edit',...
                   'BackgroundColor',[1 1 1],...
                   'Position',[435 150 100 30],...
                   'HorizontalAlignment','left' );
hEd3  = uicontrol( hFig,'Style','edit',...
                   'BackgroundColor',[1 1 1],...
                   'Position',[435 80 100 30],...
                   'HorizontalAlignment','left' );

%------------- two pushbuttons -------------------

hBut1 = uicontrol( hFig,'Style','pushbutton',...
                   'Position',[435 15 100 40],...
                   'String', 'GO',...
                   'Callback', 'GoCallback' );

%-- not visible button:
hBut2 = uicontrol( hFig,'Style','pushbutton',...
                   'Position',[240,5,90,25],...
                   'String', 'RETURN',...
                   'Callback', 'ReturnCallback',...
                   'Visible', 'off' );

%------------- 4 text fields ---------------------

hTxt1 = uicontrol( hFig,'Style','text',...
                   'BackgroundColor',[0.7 0.7 0.7],...
                   'Position', [20 15 400 270],...
                   'FontName', 'Arial',...
```

```
                    'FontSize', 20,…
                    'String',{'van der Pol equation',…
                              'ode15s solver'} );
hTxt2 = uicontrol( hFig,'Style','text',…
                    'BackgroundColor',[0.7 0.7 0.7],…
                    'Position', [435 255 60 20],…
                    'String','Initial y1' );
hTxt3 = uicontrol( hFig,'Style','text',…
                    'BackgroundColor',[0.7 0.7 0.7],…
                    'Position', [435 185 60 20],…
                    'String','Initial y2' );
hTxt4 = uicontrol( hFig,'Style','text',…
                    'BackgroundColor',[0.7 0.7 0.7],…
                    'Position', [435 115 60 20],…
                    'String','Interval' );

%----------------- the end of file --------------------
```

Функция `MyDynamicFig` создает графическое окно, первоначальный вид которого показан на рис. 9.17.

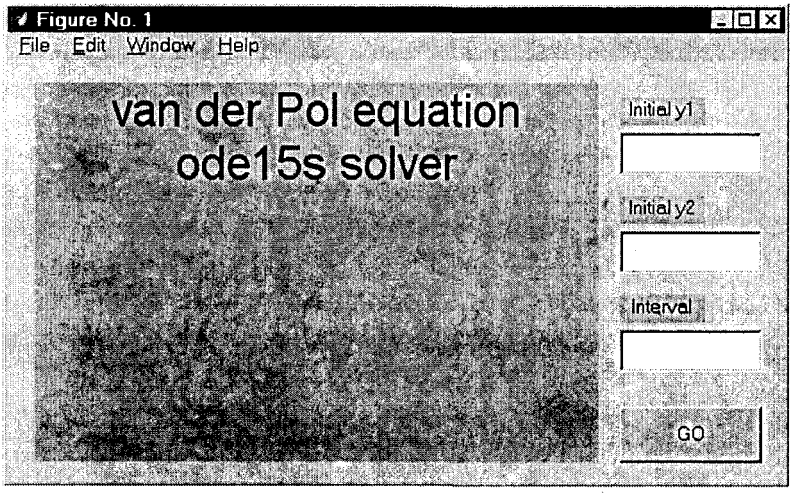


Рисунок 9.17

Вводим в редактируемые текстовые поля для начальных значений независимых переменных и длины интервала интегрирования соответственно значения 2, 0 и 3000. После чего нажимаем на кнопку `GO`, которой соответствует следующая callback-функция:

```
function GoCallback
```

```
%--------- global variables ------------------

global hAxes
global hEd1 hEd2 hEd3
global hBut1 hBut2
global hTxt1 hTxt2 hTxt3 hTxt4

%-- get string information from edit fields ---

str1=get( hEd1,'String' ); str2=get( hEd2,'String' );
str3=get( hEd3,'String' );

%-- convert strings to numbers --------------

y1In = str2num( str1 ); y2In = str2num( str2 );
len = str2num( str3 );

%----------- Numerical solution -------------

[X,Y] = ode15s('MyVanDerPol',[0,len],[y1In,y2In]);

%---------- controls reordering ------------

set( hTxt1, 'Visible', 'off' );
set( hTxt2, 'Visible', 'off' );
set( hTxt3, 'Visible', 'off' );
set( hTxt4, 'Visible', 'off' );

set( hEd1, 'Visible', 'off' );
set( hEd2, 'Visible', 'off' );
set( hEd3, 'Visible', 'off' );

set( hBut1, 'Visible', 'off' );
set( hBut2, 'Visible', 'on' );

%------------ graph plotting ----------------

plot( X, Y(:,1) );
set( hAxes, 'Visible', 'on' );

%-------------- the end of file ------------
```

Нельзя забывать, что для работы «решателя» ode15s требуется функция с именем MyVanDerPol, которую мы разработали ранее в гл. 4. Эта функция обязательно должна присутствовать на диске компьютера в одном из каталогов, доступных системе MATLAB.

После того как отработает функция GoCallback, в нашем графическом окне будет показываться только график полученного решения и единственная кнопка для возврата в исходное состояние (см. рис. 9.18).

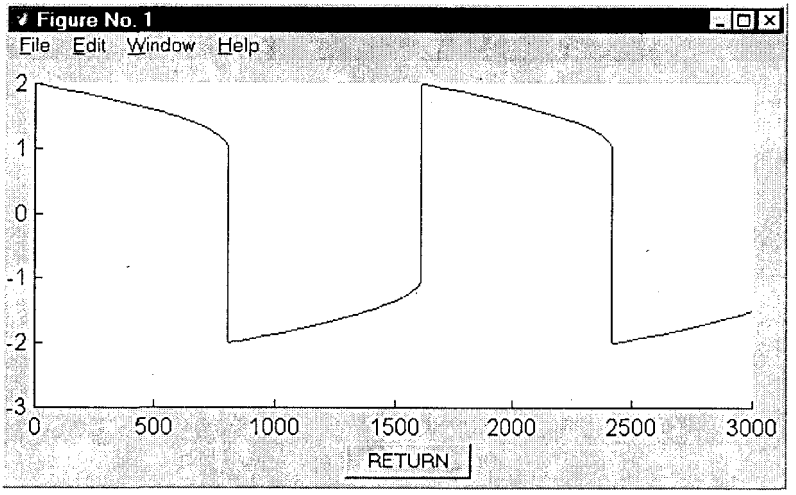


Рисунок 9.18

При нажатии на кнопку RETURN вызывается ее callback-функция ReturnCallback:

```
function ReturnCallback

%---------  global variables  -------------------

global hAxes
global hEd1 hEd2 hEd3
global hBut1 hBut2
global hTxt1 hTxt2 hTxt3 hTxt4

%----------  controls reordering  -------------

set( hTxt1, 'Visible', 'on' );
set( hTxt2, 'Visible', 'on' );
set( hTxt3, 'Visible', 'on' );
set( hTxt4, 'Visible', 'on' );
```

```
set( hEd1, 'Visible', 'on' );
set( hEd2, 'Visible', 'on' );
set( hEd3, 'Visible', 'on' );

set( hBut1, 'Visible', 'on' );
set( hBut2, 'Visible', 'off' );

axes( hAxes ); cla;
set( hAxes, 'Visible', 'off' );

%-------------  the end of file  -------------
```

Эта функция все возвращает в исходное состояние: скрывает объект axes и кнопку RETURN и показывает все остальные графические элементы управления. После этого вычисления можно повторять, вводя новые исходные данные.

Например, при начальных значениях y1 = 3 и y2 = 0 получается решение, график которого показан на рис. 9.19.

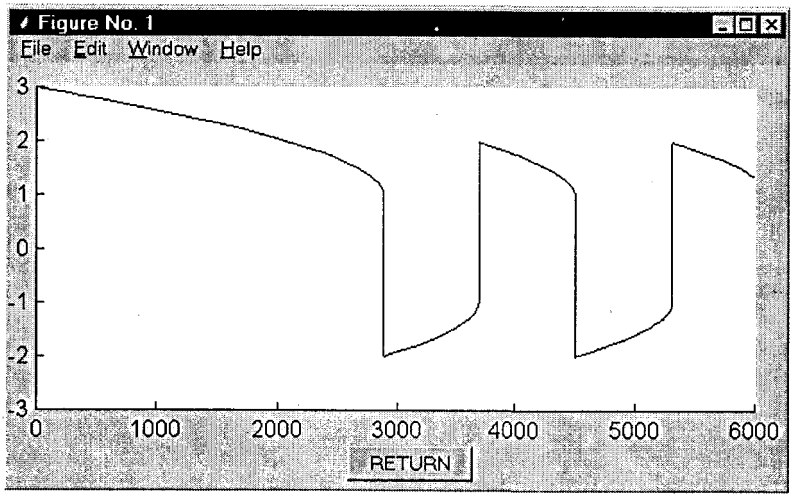


Рисунок 9.19

Чтобы был хорошо виден режим выхода на периодические релаксационные колебания, пришлось для переменной, помеченной в исходном интегрированном окне как Interval (это длина интервала интегрирования), ввести увеличенное значение 6000.

В итоге можно сделать вывод, что даже простой учебный пример, оформленный в виде удобных и наглядных интегрированных графических окон системы MATLAB, может оказать большую пользу в изучении математических вопросов.

Использование манипулятора мышь в графических окнах пакета MATLAB

В интегрированных графических окнах манипулятор мышь используется очень интенсивно. Ведь именно с помощью этого устройства мы нажимаем на командные кнопки, устанавливаем фокус ввода с клавиатуры в то или иное редактируемое поле ввода и т. д.

Но это еще не все. Можно задействовать мышь для инициирования различных действий в те моменты, когда нажимается ее левая клавиша при позиционировании указателя на том или ином графическом объекте. Для этого нужно всего лишь прописать свойство `ButtonDownFcn` этого объекта, назначив ему имя соответствующей М-функции.

Такое свойство имеется у всех графических объектов системы MATLAB, включая все элементы управления, объект `axes`, объекты `line`, `patch`, `surface` и т. д.

Например, когда мышью щелкают по объектам типа `line`, можно организовать изменение цвета линий, их толщины и других характеристик. Точно так же можно менять цвет фона графического окна (объект `figure`) или объекта `axes`. Стоит специально отметить, что для объектов `figure` или `axes` щелчки мышью должны позиционироваться на свободных местах этих объектов, где не располагаются их дочерние объекты.

В предыдущем подразделе мы разработали интегрированное графическое окно с динамически изменяемым интерфейсом пользователя. В частности, там для возврата к исходному состоянию элементов управления была задействована специальная кнопка с надписью `RETURN`, которая больше ни для каких целей не использовалась. Так вот, вместо этой кнопки можно было бы применить щелчок мышью по объекту `axes` (или `figure`) для инициализации обработчика события, связанного с его свойством `ButtonDownFcn`. Все же стоит признать, что вариант с кнопкой является для пользователя более наглядным, а самоочевидность пользовательского интерфейса является одним из наивысших приоритетов.

Поэтому мы не будем ничего переделывать в примере из предыдущего подраздела, а разработаем новый пример, специально предназначенный для иллюстрации работы с мышью. Создадим графическое окно с объектом `axes`, в котором будем проставлять жирные красные «снежинки» в тех местах, где происходит щелчок мышью.

Вот текст функции, создающей такое окно:

```
function MyTestForMouse
global hAxes

%---------- figure and axes ------------------
```

```
hFig  = figure( 'Position', [100,100,550,300],…
                'CloseRequestFcn', 'MyCloseF' );

hAxes = axes( 'Parent',hFig,'Color', [ 1 1 1],…
              'Units', 'pixels',…
              'Position', [20,55,510,230],…
              'XLimMode', 'manual',…
              'YLimMode', 'manual',…
              'ZLimMode', 'manual',…
              'XLim', [0,1],'YLim',[0,1],'ZLim',[0,1],…
              'ButtonDownFcn', 'MyButtnDownInAxes' );

%----------------- the end of file -------------------
```

а далее представлен код функции MyButtnDownInAxes, являющейся обработчиком события, связанного с «мышиным щелчком» на объекте axes:

```
function MyButtnDownInAxes
global hAxes
global h

%-----  get mouse coordinates  -----

v = get( hAxes, 'CurrentPoint' );% v is 2x3 matrix
x = v(1,1); y = v(1,2);

%---------------  put the red asterisk  ----------------

% first asterisk:
if isempty(h)
  h = line(x,y);
  set( h, 'Marker', '*', 'Color', 'red' );
  set( h, 'EraseMode', 'none', 'MarkerSize', 18 );
end

% another asterisks:
set( h, 'XData', x, 'YData', y );

%-------------  the end of the file  ----------
```

Кроме того, нам несколько неожиданно потребовалась функция (мы ее назвали MyCloseF), которая вызывается при закрытии графического окна hFig:

```
function MyCloseF
global h

% our specific action:
h=[];

% usual action:
delete( get( 0, 'CurrentFigure' ) )
```

Необходимость в последней функции возникла из-за того, что мы в функции MyButtnDownInAxes применили глобальную переменную h, которая при первом запуске этой функции всегда должна быть пустым массивом. После того как наше графическое окно закрывается, глобальная переменная h сохраняет свое уже непустое значение в рабочем пространстве системы MATLAB, отведенном для хранения глобальных переменных. Тогда если в том же самом сеансе работы с пакетом MATLAB мы снова запустим наш проект, то функция MyButtnDownInAxes при первом щелчке мышью отработает с ошибкой, так как в этот момент глобальная переменная h вовсе не является пустым массивом. Поэтому очень важно при закрытии графического окна присваивать переменной h значение пустого массива. Функцию, которая всегда будет вызываться при закрытии нашего графического окна, нужно предварительно при создании этого графического окна зарегистрировать в качестве значения свойства 'CloseRequestFcn', что и было нами выполнено в тексте функции MyTestForMouse.

Отметим теперь некоторые детали, связанные с созданием объекта axes, на поверхности которого и происходят щелчки мышью, порождающие появление красных «снежинок» (звездочек; по-английски – asterisks).

Для этого объекта мы в функции-конструкторе axes зарегистрировали обработчик для нажатия левой клавиши мыши, прописав для свойства 'ButtonDownFcn' значение 'MyButtnDownInAxes', являющееся именем нами разработанной функции.

Кроме того, чтобы не было лишней перестройки диапазонов значений, приписанных осям координат, мы зафиксировали их вручную, установив соответствующие значения для свойств 'XLimMode', 'YLimMode', 'ZLimMode', 'XLim', 'YLim', 'ZLim'.

В функции MyButtnDownInAxes мы различаем первое нажатие клавиши мыши и последующие нажатия с целью более гладкой прорисовки, исключающей излишние и крайне неприятные мерцания. Для этого при первичном создании объекта line мы приписываем его свойству 'EraseMode' значение 'none' (и кроме того, заодно мы прописываем другие свойства, связанные с конкретным видом получающегося изображения).

Фрагмент кода, который непосредственно отвечает за получение координат точки, где произошел щелчок мышью, повторен ниже:

```
v = get( hAxes, 'CurrentPoint' );% v is 2x3 matrix
x = v(1,1); y = v(1,2);
```

В качестве значения свойства `'CurrentPoint'` возвращается матрица, строки которой представляют собой координаты двух точек. В нашем конкретном случае *двумерных изображений, лежащих в плоскости экрана дисплея*, x- и y-координаты обеих точек совпадают. Мы используем этот факт и берем для дальнейших действий координаты `v(1,1)` и `v(1,2)`.

Ну и, наконец, ниже показан пример «художественного творчества» с применением разработанного нами оригинального инструмента рисования (см. рис. 9.20).

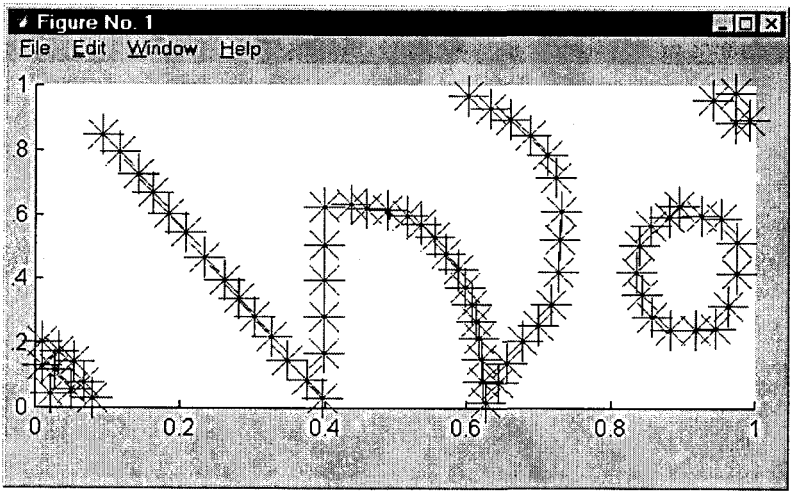


Рисунок 9.20

У нашего инструмента нет никаких прецизионных средств выставления точных значений координат, однако если поточнее «прицелиться», то можно нарисовать что-нибудь интересное.

Создание меню

Применение *меню* в современных графических приложения Windows является стандартным и практически обязательным элементом пользовательского интерфейса. Поэтому в рамках графических окон системы MATLAB предусмотрено значительное число приемов и способов работы с меню.

Меню в системе MATLAB представлено двумя типами графических объектов – `uimenu` и `uicontextmenu`, каждый из которых создается одноимен-

ной функцией-конструктором. Естественно, что каждый из этих объектов обладает некоторым набором свойств, управляя значениями которых, мы и создаем необходимые нам варианты меню.

Оставим контекстное меню на потом и начнем рассмотрение с меню, которое во всех стандартных окнах платформы Windows располагается в верхней части окна, сразу за его заголовком. Оно представляет собой полоску с набором элементов, каждый из которых может быть текстом или рисунком. Все графические окна системы MATLAB автоматически снабжаются стандартным набором пунктов меню верхнего уровня: `File`, `Edit`, `Wimdow`, `Help`. Эти пункты меню видны всегда. Если же щелкнуть мышью по одному из этих пунктов, то появится набор пунктов меню следующего уровня. Глубина вложения уровней меню может быть любой и зависит только от практической необходимости и вкуса разработчика. На рис. 9.21 для примера показано выпадающее меню для пунка `Edit` графических окон системы MATLAB.

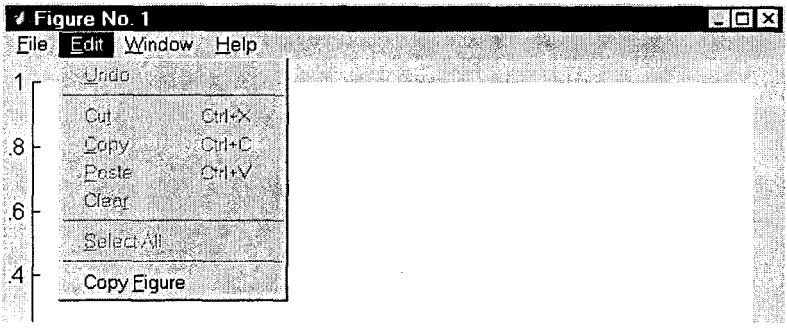


Рисунок 9.21

Отсюда видно, что пункты меню могут находиться в двух состояниях: готовом к работе (по-английски – enabled) и в пассивном состоянии (disabled). Переключение между этими двумя состояниями осуществляется программно в зависимости от конкретного контекста ситуации: если действие, инициируемое некоторым пунктом меню, в этот момент неуместно, то такой пункт меню программно переводится в пассивное состояние, которое визуально средствами операционной системы Windows показывается серым цветом, чтобы была видна разница с активными черными пунктами меню.

Мы можем при разработке интегрированных графических окон *добавлять новые пункты меню верхнего уровня* к стандартному набору, рассмотренному выше. К каждому из таких новых пунктов мы может присовокупить *выпадающее меню*, содержащее нужный нам набор пунктов, которые либо вызывают при нажатии на них мышью некоторые действия, либо показывают новые выпадающие меню. Таким образом, иерархия разрабатываемой системы меню может быть любой и сколь угодно сложной.

Мы не будем иллюстрировать практическим примером создание системы меню очень разветвленной иерархии, поскольку этот вопрос не сложен с практической точки зрения, однако весьма громоздок при его изложении в учебном посо-

бии. Вместо этого мы ограничимся простым случаем, в котором повторим ранее разработанный проект по изучению решений дифференциальных уравнений Ван-дер-Поля. Однако вместо создания специальной кнопки, предназначенной для возврата системы управляющих элементов окна в исходное состояние, мы сейчас для этой же цели применим команду меню верхнего уровня.

Естественно, что нам не придется переделывать весь проект, поэтому мы приведем лишь те фрагменты, которые подвергаются изменениям.

В начальной, создающей само графическое окно функции `MyDynamicFig` теперь не надо создавать кнопку `hBut2`, зато надо будет приписать самому графическому окну, то есть объекту `figure`, новый пункт меню верхнего уровня, что делается следующим образом:

```
global hMenu
%----------- figure and axes ------------------

hFig  = figure( 'Position', [100,100,550,300] );
hMenu = uimenu( hFig, 'Label', '&Return',...
                'Callback', 'ReturnCallback',...
                'Enable', 'off' );
```

Новый пункт меню мы скрывать (а потом проявлять) не будем, так как он не занимает излишнего места, чтобы такая необходимость могла возникнуть. Однако в начальном положении его действие *неуместно*, поэтому мы переводим его сразу же в пассивное состояние, задавая для свойства `'Enable'` значение `'off'`.

Действие, инициируемое по выбору этого пункта меню (оно будет выполняться только в том случае, когда пункт меню находится в активном состоянии), прописывается в виде имени соответствующей функции для свойства `'Callback'`. Мы здесь применяем ту же самую функцию `ReturnCallback`, которая ранее была использована в качестве callback-функции кнопки `hBut2`. Так как самой такой кнопки больше не существует, то в callback-функциях `GoCallback` и `ReturnCallback` нужно закомментировать по одной строке, которые содержат описатель `hBut2`.

Дополнительно в этих функциях вместо обработки кнопки `hBut2` теперь нужно обрабатывать меню `hMenu`: в функции `GoCallback` этот пункт меню нужно переводить в активное состояние:

```
set( hMenu, 'Enable', 'on' );
```

а в функции `ReturnCallback` осуществлять обратное действие:

```
set( hMenu, 'Enable', 'off' );
```

Естественно, что в обеих этих функциях нужно добавить по строчке, объявляющей переменную hMenu как глобальную:

```
global hMenu
```

Наконец, объясним, какой смысл имеет знак &, стоящий перед буквой R в строке, задающей текст нового пункта меню. Этот знак приводит к тому, что буква R при изображении меню будет показываться в подчеркнутом виде. Это сигнализирует пользователю, что вместо мыши можно применить клавиатурную комбинацию Alt+R с тем же самым эффектом.

Вот и все изменения, которые пришлось внести в ранее разработанный проект, чтобы заменить динамически перестраиваемую кнопку на новый пункт меню графического окна, которое теперь будет выглядеть следующим образом (см. рис. 9.22).

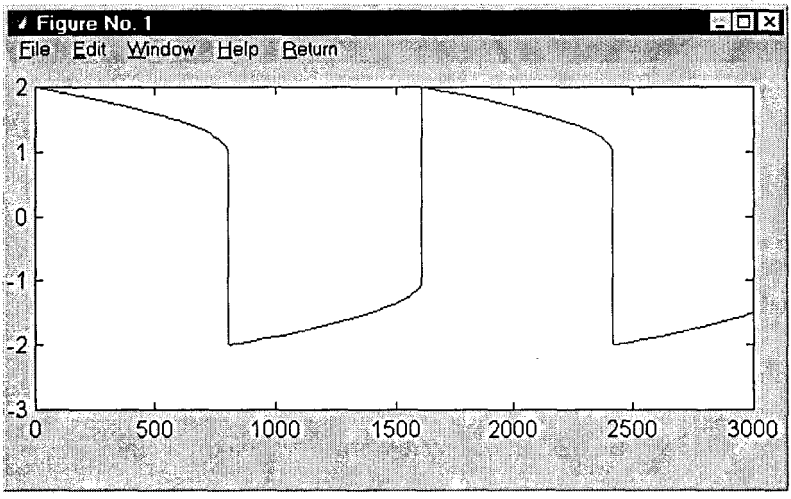


Рисунок 9.22

Если в этом окне, в котором теперь уже нет кнопки RETURN, выбрать (нажать) пункт меню Return, то произойдет возврат в исходное состояние с набором редактируемых текстовых полей ввода исходных числовых данных.

Теперь рассмотрим другой вариант меню – так называемое *контекстное меню*, которое в системе MATLAB представлено графическим объектом uicontextmenu.

Этот тип меню инициируется по нажатии *правой клавиши мыши* на любом графическом объекте в пределах графического окна. Ему не соответствует никакая статически выделенная надпись в обычной строке меню верхнего уровня, расположенной сразу же под заголовком окна.

Так как контекстное меню активизируется в связи с некоторым графическим объектом, то его основное предназначение состоит в показе или изменении свойств этого графического объекта.

Разработаем простой иллюстрирующий пример, который оформим в виде М-функции MyContextMenu:

```
function MyContextMenu

global hLine

%-------  create the context menu  ------------

hMenuCt = uicontextmenu;

%--defining of context menu items with their callbacks--

uimenu(hMenuCt, 'Label','red',     'Callback', 'Clbk1' );
uimenu(hMenuCt, 'Label','green',   'Callback', 'Clbk2' );
uimenu(hMenuCt, 'Label','magenta','Callback', 'Clbk3' );

%--create the line and connect it to the context menu:

x = 0:0.01:3; y = sin( x );
hLine = plot( x, y, 'UIContextMenu', hMenuCt );

%--------------  the end of M-function  --------------
```

Теперь по существу создания контекстного меню. Сначала функцией-конструктором uicontextmenu создаем соответствующий графический объект и запоминаем в переменной hMenuCt его описатель. Затем функциями uimenu создаем обычные пункты меню, привязанные в качестве дочерних объектов к только что созданному контекстному меню.

Каждый пункт меню привязываем к его собственной callback-функции. Вот эти очень короткие callback-функции, текст каждой из которых записываем в отдельный М-файл:

```
function Clbk1
global hLine
set( hLine, 'Color', 'red');

function Clbk2
global hLine
set( hLine, 'Color', 'green' );

function Clbk3
global hLine
set( hLine, 'Color', 'magenta' );
```

Так как контекстное меню появляется только в связи с некоторым другим графическим объектом, то мы создаем график функции, то есть объект типа `line`, и привязываем наше контекстное меню к этому графику.

После этого можно будет по нажатии правой клавишей мыши на линии графика функции вызывать контекстное меню, с помощью которого пользователь сможет изменять цвет этой линии. Рис. 9.23 иллюстрирует эту ситуацию.

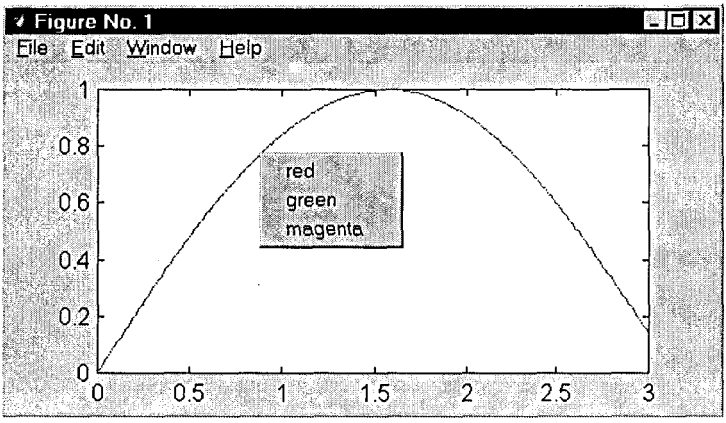


Рисунок 9.23

Естественно, что после появления всплывающего контекстного меню выбор того или иного пункта из него осуществляется нажатием левой клавиши мыши.

Глава 10

Взаимодействие внешних приложений с системой MATLAB

Взаимодействие приложений Windows с MATLAB Engine

До сих пор мы работали целиком и полностью с единственным приложением – с пакетом MATLAB. Даже когда мы разрабатывали некоторые функции на языке C, мы компилировали их в динамические библиотеки, которые для работы загружаются в адресное пространство работающей системы MATLAB, после чего становятся неотделимы от нее. Можно сказать, что MEX-функции системы MATLAB (их-то и разрабатывают на языке C/C++) являются прямым расширением этой системы.

Как быть, если мы разрабатываем собственное *изолированное приложение* Windows, в котором требуется осуществлять серьезные математические вычисления?

Пакет MATLAB имеет убедительный ответ на этот вопрос, так как предоставляет прямой интерфейс взаимодействия вашего приложения с этим пакетом. Способ взаимодействия отдельного приложения с системой MATLAB называется *взаимодействием с подсистемой MATLAB Engine*. В некотором смысле это просто принятая в рамках системы MATLAB терминология. Про приложение, которое взаимодействует с системой MATLAB по технологии MATLAB Engine говорят, что это *приложение MATLAB Engine*. Всякая терминология хороша тем, что позволяет точно формулировать детали разработки и обстоятельства функционирования объектов обсуждения.

С точки зрения исходных текстов разрабатываемого на языке C изолированного приложения его взаимодействие с системой MATLAB осуществляется путем вызова специальных функций из библиотеки MATLAB Engine (это небольшая динамическая библиотека `libeng.dll` размером 18 Кбайт), а функции из этой библиотеки и осуществляют непосредственное взаимодействие с приложением `matlab.exe` (размер этого файла равен 3668 Кбайт). Библиотека MATLAB Engine во время работы загружается в адресное пространство вашего приложения, не слишком утяжеляя его из-за своего небольшого размера. На этапе компиляции же требуется так называемая библиотека импорта `libeng.lib`. Как получить такую библиотеку, а также как реально осуществить компиляцию проек-

та для получения изолированного приложения MATLAB Engine, будет рассказано ниже в специально посвященном этому подразделе.

Сейчас же просто перечислим немногочисленные функции из библиотеки MATLAB Engine:

1. `engClose( Engine* ep )`
2. `engEvalString( Engine* ep, const char* string )`
3. `engGetArray( Engine* ep, const char* name )`
4. `engOpen( const char* startcmd )`
5. `engOutputBuffer( Engine* ep, char* p, int n )`
6. `engPutArray( Engine* ep, const mxArray* mp )`

Итак, в этой библиотеке всего шесть функций (есть еще некоторое количество устаревших функций, доставшихся в наследство от предыдущих версий пакета MATLAB, – их не рекомендуется использовать в новых приложениях).

Глядя на представленные прототипы функций, замечаем как уже известные нам внутренние типы данных системы MATLAB, такие, как `mxArray`, так и новый тип данных `Engine`. Чтобы воспользоваться этим типом данных, нужно включать (директивой препроцессора `include`) в исходный код разрабатываемого проекта заголовочный файл `engine.h`.

Взаимодействие с системой MATLAB начинается вызовом функции `engOpen`, которая и возвращает правильный указатель на тип `Engine`:

```
Engine* pEng;
pEng = engOpen( NULL );
```

причем всегда нужно проверять возврат на `NULL`. В последнем случае вызов функции `engOpen` прошел неудачно (например, по причине невозможности запустить на выполнение приложение `matlab.exe`) и следует закрыть приложение, так как взаимодействовать с системой MATLAB по технологии MATLAB Engine уже невозможно.

Если же вызов функции `engOpen` прошел удачно, то есть получен *ненулевой указатель*, то после этого можно вызывать другие функции с префиксом `eng`, а также функции с префиксом `mx`, которые мы раньше активно использовали при разработке MEX-функций (см. гл. 8). Однако теперь нельзя использовать функции с префиксом `mex`.

Когда работа с системой MATLAB заканчивается, нужно вызвать функцию `engClose`, чтобы осуществить корректное отсоединение и освободить занятые ресурсы:

```
engClose ( pEng );
```

В промежутке между вызовами функций engOpen и engClose можно вызывать остальные функции библиотеки MATLAB Engine, то есть функции engPutArray, engGetArray, engEvalString и engOutputBuffer.

Функция engPutArray помещает массив mxArray в рабочее пространство системы MATLAB, после чего над этим массивом можно производить вычисления непосредственно в этом рабочем пространстве при помощи функции engEvalString. Чтобы получить из рабочего пространства копию массива (например, результата вычислений) и разместить ее в адресном пространстве нашего приложения, нужно вызвать функцию engGetArray.

Наконец, функция engOutputBuffer позволяет задать буфер в памяти нашего процесса, куда будет поступать при работе функции engEvalString поток сообщений, обычно поступающий в командное окно системы MATLAB.

Итак, мы очень кратко описали работу всех функций библиотеки MATLAB Engine. Теперь проиллюстрируем их работу на конкретных примерах. Мы ограничимся компактными учебными примерами, так что весь C-код всегда будет помещаться в единственный файл с именем main.c. Мы также будем создавать только приложения Win32 Console Application, причем делается это лишь для простоты. На самом деле абсолютно приемлемы и даже желательны приложения с полноценным графическим интерфейсом Windows типа Win32 Application.

Всего мы рассмотрим три примера. В первом из них мы в своем приложении сформируем матрицу, затем функцией engPutArray передадим ее в рабочее пространство системы MATLAB. Затем с помощью функции engEvalString вычислим в рабочем пространстве собственные значения этой матрицы и возвратим результаты в наше адресное пространство с помощью функции engGetArray. Вот текст первого примера:

```
#include <stdio.h>
#include <memory.h>
#include <conio.h>

#include 'engine.h'

double data[6] = {1,2,3,4,5,6};

int main( void )
{
  Engine* pEng = NULL;
  mxArray* pArr = NULL;
  mxArray* pV = NULL;
  double* pReal  = NULL;
  double* pImage = NULL;
```

```
  int M, i;

// Create real matrix
  pArr = mxCreateDoubleMatrix( 3, 2, mxREAL );
  mxSetName( pArr, 'A' );


// Set its elements:
  memcpy( mxGetPr( pArr ), data, 6*sizeof(double) );

// Start Engine session:
  pEng = engOpen( NULL );
  if( pEng == NULL )
  {
    printf( '\nCan't start Engine session!' );
    return 1;
  }

// Put array into MATLAB environment
// and calculate eigen values of it:
  engPutArray( pEng, pArr );
  engEvalString( pEng, 'Vec = eig( A*A')' );
  pV = engGetArray( pEng, 'Vec' );

// Stop Engine session:
  engClose( pEng );

// Get double values from
// possibly complex mxArray v:
  pReal  = mxGetPr( pV );
  pImage = mxGetPi( pV );
  M = mxGetM( pV );

// Print results:
  printf('Real parts of eig vector\n');
  for( i=0; i< M; i++)
  {
    printf('%lf\n', pReal[i] );
  }

  if( pImage )
```

```
    {
      printf('\nImage parts of eig vector\n');
      for( i=0; i< M; i++)
      {
        printf('%lf\n', pImage[i] );
      }
    }

  // Free mxArray memory buffers:
    mxDestroyArray( pArr );
    mxDestroyArray( pV );

  // Screen delay:
    printf('Press any key to exit');
    getch();
    return 0;

  }
```

В следующем подразделе будет детально рассказано, как скомпилировать реальное Win32 Console-приложение на базе представленного текста, которое составляет содержимое файла `main.c`. Это единственный исходный файл, необходимый для компиляции. Там же будет рассказано, как следует отлаживать такое приложение.

Сейчас же покажем, как выглядит окно такого приложения после того, как все вычисления будут произведены (см. рис. 10.1).

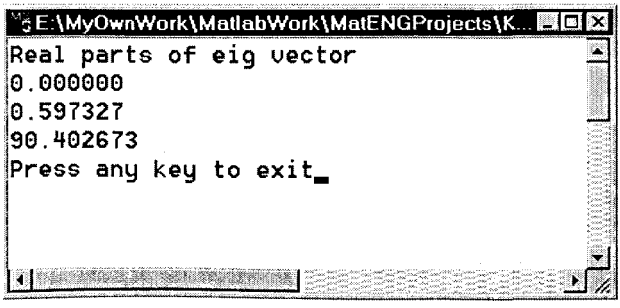


Рисунок 10.1

Отсюда видно, что получены три собственных числа. После нажатия любой клавиши окно закроется.

Весь представленный выше код достаточно очевиден после того, как мы кратко описали предназначение всех функций библиотеки MATLAB Engine. Помимо них еще используются функции с префиксом `mx`, большинство из кото-

рых нам также хорошо известно, так как мы достаточно потренировались с ними на примерах из гл. 8.

Однако есть все-таки одна новая функция, на которой следует остановить свое внимание. Это функция mxSetName. Это важная функция, без которой вся представленная выше работа не получится. Можно сказать, что эта функция нивелирует некоторое принципиальное различие между миром программ на языке C и миром программ на M-языке. В первом из них объекты типа mxArray (массивы системы MATLAB) адресуются указателями, которых нет вообще в M-языке. Там есть только обычные имена для массивов. Так вот функция mxSetName и призвана снивелировать это различие, точнее, перебросить мостик между двумя способами обращения к массивам (матрицам) в этих двух мирах.

Фрагмент из вышеприведенной программы:

```
// Create real matrix
  pArr = mxCreateDoubleMatrix( 3, 2, mxREAL );
  mxSetName( pArr, 'A' );
```

показывает, что сначала создается массив 3 x 2 и он адресуется указателем pArr. После этого с помощью функции mxSetName для массива вводится «человеческое» (в отличие от использования «нечеловеческих» указателей в языке C) имя A. Когда позже этот массив будет внедрен в рабочее пространство системы MATLAB для дальнейшей его обработки, то сама эта обработка в рабочем пространстве системы MATLAB будет происходить над переменной с именем A. Вот соответствующий фрагмент кода:

```
engPutArray( pEng, pArr );
engEvalString( pEng, 'Vec = eig( A*A')' );
pV = engGetArray( pEng, 'Vec' );
```

Здесь в первой строке массив, адресуемый в C-коде указателем pArr, внедряется в рабочее пространство системы MATLAB, где с ним уже работают как с переменной A, что прекрасно видно из второй строки фрагмента. Далее, поскольку результат вычислений поименован там как Vec, именно под этим именем мы и извлекаем его назад из рабочего пространства функцией engGetArray. Результат такого извлечения, то есть некоторая область памяти, далее адресуется в C-коде указателем pV.

Надо полагать, что представленных подробнейших пояснений по поводу назначения функции mxSetName вполне достаточно.

Теперь перейдем ко второму примеру. Он будет небольшой модификацией первого примера, направленной на то, чтобы проиллюстрировать назначение и работу функции engOutputBuffer.

Отметим один более-менее очевидный недостаток в коде первого примера, для чего воспроизведем следующий фрагмент:

```
engEvalString( pEng, 'Vec = eig( A*A')' );
```

```
  pV = engGetArray( pEng, 'Vec' );

// Stop Engine session:
  engClose( pEng );

// Get double values from
// possibly complex mxArray v:
  pReal  = mxGetPr( pV );
```

Первой строкой здесь запрашивается выполнение некоторого действия в рабочем пространстве системы MATLAB. Результат этого действия извлекается с помощью функции `engGetArray`. Но что будет, если в процессе выполнения запрошенной операции произошла ошибка? Тогда невозможно получить правильное значение для указателя `pV`, адресующего результат вычислений. Из-за этого функция `mxGetPr`, использующая этот указатель, отработает с ошибкой.

С одной стороны, этот недостаток представленного кода легко исправить. Для этого надо лишь перед использованием указателя `pV` проверять его на равенство значению `NULL`:

```
if( pV != NULL )
  pReal  = mxGetPr( pV );
```

и тогда все будет работать безошибочно.

Однако остается *проблема получения диагностической информации об ошибках*, которая всегда выводится в командное окно системы MATLAB. Пусть, к примеру, для вычисления собственных значений выбрана неквадратная матрица. Вот что при этом мы наблюдаем в командном окне системы MATLAB (см. рис. 10.2).

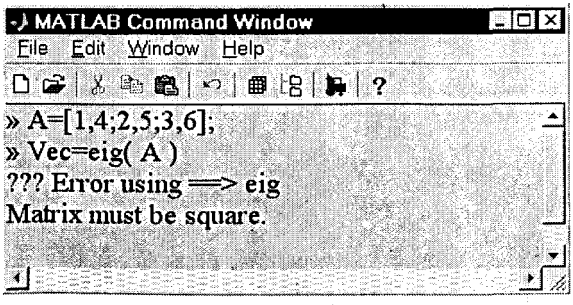


Рисунок 10.2

Задачей второго примера будет передача этой диагностической информации в распоряжение С-кода приложения типа MATLAB Engine. Как мы уже рассказывали выше, для этой цели служит функция `engOutputBuffer`. Она задает буфер в памяти, куда и будут поступать сообщения из системы MATLAB.

Добавим в код первого примера несколько строк кода, которые позволят отслеживать ошибочные ситуации, возникающие в процессе вычислений в рабочем пространстве системы MATLAB, а также позволят получить всю диагностическую информацию оттуда. Чтобы не возникло недоразумений и сложностей в восприятии нового варианта текста из первого примера, приведем весь этот код в полном объеме:

```
#include <stdio.h>
#include <memory.h>
#include <conio.h>

#include 'engine.h'

#define BUFSIZE 256
double data[6] = {1,2,3,4,5,6};

int main( void )
{
  Engine* pEng = NULL;
  mxArray* pArr = NULL;
  mxArray* pV = NULL;
  double* pReal  = NULL;
  double* pImage = NULL;
  int M, i;
  char pBuf[BUFSIZE];

// Create real matrix
  pArr = mxCreateDoubleMatrix( 3, 2, mxREAL );
  if( pArr == NULL )
  {
    printf('\nCan't create matrix!');
    return 1;
  }
  mxSetName( pArr, 'A' );


// Set its elements:
  memcpy( mxGetPr( pArr ), data, 6*sizeof(double) );

// Start Engine session:
  pEng = engOpen( NULL );
  if( pEng == NULL )
  {
```

```
    printf( '\nCan't start Engine session!' );
    return 2;
  }

// Put array into MATLAB environment
// and calculate eigen values of it:
  engPutArray( pEng, pArr );
  engOutputBuffer( pEng, pBuf, BUFSIZE );
  engEvalString( pEng, 'Vec = eig( A*A')' );
  pV = engGetArray( pEng, 'Vec' );

// Stop Engine session:
  engClose( pEng );

// Let's see the OutputBuffer:
  printf( '%s', pBuf );

// Get double values from
// possibly complex mxArray v:
  if( pV != NULL )
  {
    pReal  = mxGetPr( pV );
    pImage = mxGetPi( pV );
    M = mxGetM( pV );

    // Print results:
    printf('Real parts of eig vector\n');
    for( i=0; i< M; i++)
    {
      printf('%lf\n', pReal[i] );
    }

    if( pImage )
    {
      printf('\nImage parts of eig vector\n');
      for( i=0; i< M; i++)
      {
        printf('%lf\n', pImage[i] );
      }
    }
  }
```

```
// Free mxArray memory buffers:
  mxDestroyArray( pArr );
  if( pV != NULL ) mxDestroyArray( pV );

// Screen delay:
  printf('Press any key to exit');
  getch();
  return 0;

}
```

В этот текст включены все проверки указателей на равенство их значению NULL, но самое главное изменение заключается в применении функции engOutputBuffer.

Эту функцию нужно вызвать перед вызовом функции engEvalString, в результате чего весь вывод информации в командное окно системы MATLAB будет продублирован в заранее приготовленный буфер нашей С-программы:

```
engOutputBuffer( pEng, pBuf, SIZEBUF );
```

Буфер размером SIZEBUF адресуется указателем pBuf. Третий параметр этой функции задает число байтов информации, которое будет направлено в буфер. Если размер диагностической информации, выводимой в командное окно системы MATLAB, будет больше, чем заданное нами число SIZEBUF, то ничего страшного не произойдет: в соответствии со сказанным только первые SIZEBUF байт будут скопированы в предоставленный буфер памяти. Также никаких проблем не возникает, если реально переданное число байтов информации будет меньше, чем истинный размер буфера.

На рис. 10.3 показано окно разработанного приложения с выведенной в него информацией из рабочего пространства системы MATLAB.

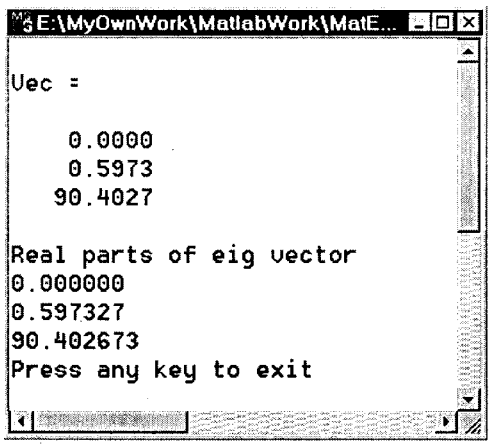


Рисунок 10.3

В случае штатной ситуации, когда вычисления в рабочем пространстве системы MATLAB осуществляются корректно, вывод информации, предоставленной функцией `engOutputBuffer`, бывает излишним, что и видно из последнего рисунка.

Чтобы не было избыточной информации, распечатку буфера лучше осуществлять после проверки наличия ошибочных ситуаций. Вот единственное изменение кода, которое потребуется для этого:

```
if( pV == NULL )
  printf( '%s', pBuf );
```

После этого наша программа отработает без выдачи излишней информации. В то же время, если таковая возникнет, например если мы попробуем в среде MATLAB вычислить следующее выражение:

```
engEvalString( pEng, 'Vec = eig( a )' );
```

(которое нужно подставить в приведенный выше код второго примера вместо вызова функции `engEvalString` с правильным выражением), то в окно нашего приложения поступит уже крайне полезная информация об ошибочной ситуации, так как других способов продиагностировать ее нет (см. рис. 10.4).

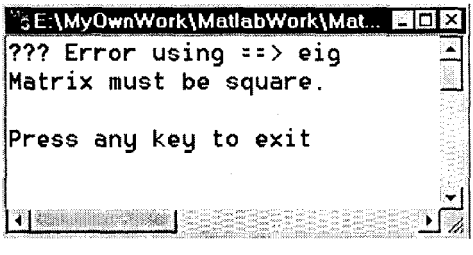


Рисунок 10.4

Теперь разработаем третий пример приложения типа MATLAB Engine, который будет демонстрировать выполнение системой MATLAB графических манипуляций по требованию из нашего приложения. В этом примере нет ничего особенного, кроме того, что он с очевидностью демонстрирует возможность получать графические изображения, выполненные системой MATLAB, по приказу из внешних приложений типа MATLAB Engine. Вот текст исходного кода третьего примера:

```
#include <stdio.h>
#include <memory.h>
#include <conio.h>

#include 'engine.h'

double data[10] = { 0.1,0.2,0.3,0.4,0.5,
                    0.6,0.7,0.8,0.9,1.0 };
```

```
int main( void )
{
  Engine* pEng = NULL;
  mxArray* pArr = NULL;

// Create real vector
  pArr = mxCreateDoubleMatrix( 1, 10, mxREAL );
  if( pArr == NULL )
  {
    printf('\nCan't create vector!');
    return 1;
  }
  mxSetName( pArr, 'x' );

// Set its elements:
  memcpy( mxGetPr( pArr ), data, 10*sizeof(double) );

// Start Engine session:
  pEng = engOpen( NULL );
  if( pEng == NULL )
  {
    printf( '\nCan't start Engine session!' );
    return 2;
  }

// Put vector into MATLAB environment
// and calculate y=sin(x). Then plot graph:
  engPutArray( pEng, pArr );
  engEvalString( pEng, 'y = sin( x );plot(x,y);' );

// Free mxArray memory buffers:
  mxDestroyArray( pArr );

// Screen delay:
  printf('Press any key to exit');
  getch();

// Stop Engine session:
  engClose( pEng );
  return 0;
}
```

После запуска на выполнение этого приложения, вскоре будет также запущена система MATLAB, которая и породит следующее графическое окно (см. рис. 10.5).

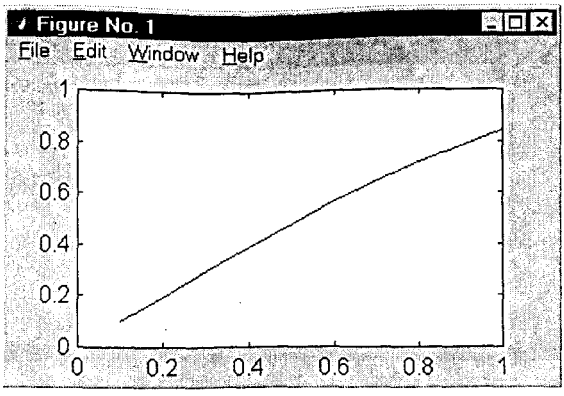


Рисунок 10.5

Обратите внимание на то, что закрытие сессии (сеанса) MATLAB Engine функцией `engClose` следует осуществлять после вызова функции `getch`, ожидающей ввода с клавиатуры, тем самым обеспечивающей пользователя достаточным временем для наблюдения графических изображений. После же вызова функции `engClose` приложение MATLAB заканчивает работу и закрывает все свои графические окна.

После этого примера возникает впечатление, что из приложения MATLAB Engine можно выполнить в среде MATLAB все, что можно выполнить, находясь непосредственно в командном окне системы MATLAB. Однако на сегодняшний день это не так. Вплоть до версии MATLAB 5.2 из приложений типа MATLAB Engine можно было работать только с матрицами. Так называемые новые типы данных версий MATLAB 5.x (многомерные массивы, структуры и ячейки, разреженные матрицы) из приложений MATLAB Engine недоступны.

Создание и компиляция EXE-проекта в среде Microsoft Visual C++

Недостаточно просто написать C-код для приложения типа MATLAB Engine. Нужно еще выбрать тип проекта Microsoft Visual C++, выполнить изменение свойств проекта и, наконец, скомпилировать его.

Допустимыми типами проектов для создания приложений MATLAB Engine являются проекты `Win32 Console Application` и `Win32 Application`. Последние позволяют компилировать приложения с развитым графическим интерфейсом пользователя, в то время как первые проекты ограничиваются работой с простыми окнами, ориентированными на текстовый вывод. Именно

эти окна (в рамках терминологии системы Windows они и называются консольными) мы приводили выше в качестве иллюстрации работы приложений MATLAB Engine, получающихся из представленных там примеров исходных С-кодов. Мы и далее будем работать только с проектами типа `Win32 Console Application` исключительно из соображений их меньшего объема, что важно в процессе изучения системы MATLAB.

Создав проект указанного типа, включаем в него единственный файл (например, под именем `main.c`, хотя выбор имени абсолютно произволен). Если тут же начать компилировать проект, то Developer Studio выдаст сообщения об ошибках, поскольку мы еще не выполнили должную настройку проекта. Она заключается в следующем.

Прежде всего, нужно прописать путь к включаемому файлу `engine.h`. Это все тот же каталог `\extern\include`, который мы уже указывали в свойствах проекта по созданию MEX-функций (см. гл. 8).

Далее в списке библиотечных файлов проекта нужно вручную записать имена двух библиотек импорта, что показано на рис. 10.6.

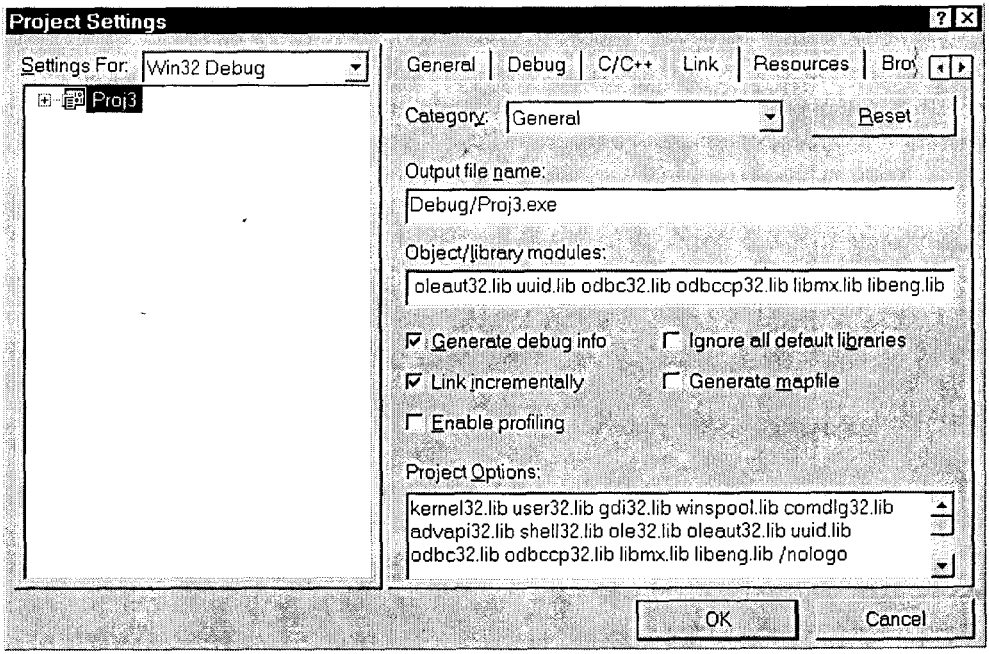


Рисунок 10.6

В конце строки ввода `Object/library modules` здесь присутствуют имена `libmx.lib` и `libeng.lib` этих библиотек. Их нужно ввести вручную с клавиатуры.

Сами эти библиотечные файлы, однако, не поставляются в готовом виде вместе с пакетом MATLAB. Их требуется изготовить самостоятельно. В гл. 8 мы

уже сами изготавливали аналогичный по смыслу и назначению файл `matlab.lib`. Мы сейчас точно таким же способом, который использовался в гл. 8, изготовим недостающие библиотечные файлы `libmx.lib` и `libeng.lib`.

Содержимое следующего командного окна Windows показывает необходимые действия по изготовлению первого из них (см. рис. 10.7).

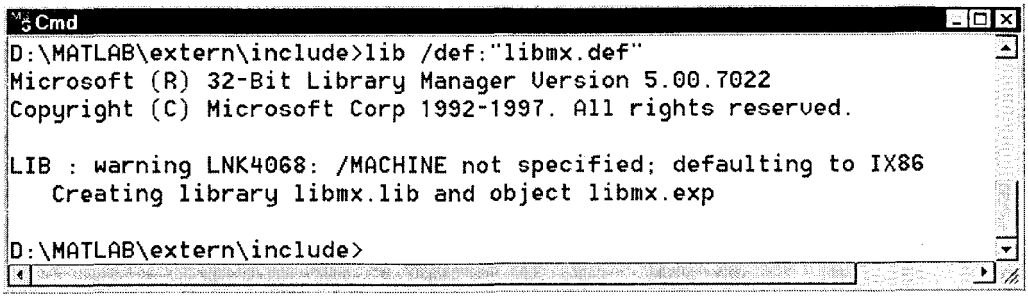


Рисунок 10.7

Аналогично получается второй недостающий библиотечный файл, если только имя `libmx.def` заменить на `limeng.def`.

Затем полученные таким образом библиотечные файлы нужно скопировать в главный каталог нашего проекта в среде Developer Studio, после чего все готово к компиляции. Нажав клавишу F7, получаем в результате процесса компиляции целевой исполняемый файл с расширением `exe` (а имя его совпадает с выбранным именем каталога проекта). Этот целевой файл уже можно запускать на выполнение (если под отладчиком среды Developer Studio – то клавишей F5). Он сам в процессе работы запустит приложение MATLAB, запросит у него выполнение некоторой работы, а результаты возможных вариантов работы мы уже наблюдали в предыдущем подразделе.

Теперь несколько слов об отладке приложений MATLAB Engine. Для отладки эти приложения запускаются клавишей F5 (если, конечно, они собраны в Debug-, а не в Release-варианте) с предварительно поставленными в их тексте точками останова (Breakpoints). После осуществления останова выполнения программы на одной из таких точек можно клавишей F10 осуществлять построчное продвижение. При этом очень легко наблюдать процесс запуска приложения MATLAB в фоновом режиме. Это, однако, не мешает щелчком мыши по иконке этого приложения развернуть его главное (командное) окно и начать тут же вводить команды и выполнять их.

Таким образом легко убедиться в роли такой библиотечной функции системы MATLAB, как функция `mxSetName`, роль которой мы детально обсуждали выше. Однако в процессе отладки в справедливости сказанного вы можете убедиться сами. Например, закомментируйте строку в исходном C-коде с вызовом функции `mxSetName` и перекомпилируйте весь проект. Затем запустите приложение под отладчиком и остановитесь на вызове функции `engPutArray`. По идее после

работы этой функции в рабочем пространстве системы MATLAB должна появиться внедряемая туда с помощью этой функции переменная. Вот это и можно легко проверить. Разверните окно системы MATLAB и введите команду

```
who
```

которая и покажет список всех переменных из рабочего пространства системы MATLAB. При закомментированной функции `mxSetName` вы внедряемой функции не обнаружите. Это естественно, так как функцию в рамках системы MATLAB можно показать только по ее имени, а мы не приписали в С-коде ей никакого имени, так как это делается закомментированной функцией `mxSetName`.

Если вернуться и раскомментировать вызов этой функции, перекомпилировать и запустить проект снова под отладчиком, остановиться где надо, то, зайдя в командное окно системы MATLAB и выполнив команду `who`, мы теперь обнаружим внедряемую нами переменную в списке переменных из рабочего пространства системы MATLAB.

Остальные приемы отладки приложений типа MATLAB Engine не отличаются от приемов отладки любых других приложений Windows.

С-библиотеки математических функций системы MATLAB

Разработав приложение типа MATLAB Engine, вы, возможно, захотите передать его для работы сторонним пользователям. Это вполне разумная идея, особенно если на языке С разработан удобный и самоочевидный графический интерфейс, а также присутствуют подробные HELP-файлы.

Тут, однако, следует иметь в виду, что приложение MATLAB Engine невозможно запустить на компьютере, на котором не установлено приложение MATLAB. В этом случае пользователь получит сообщение об ошибке, гласящее, что не удается найти библиотеки `libeng.dll` и `libmx.dll`. Если просто скопировать эти библиотеки на другой компьютер (например, в каталог System32) и снова запустить на нем разработанное приложение MATLAB Engine, то теперь будут затребованы другие библиотеки, так что конца этому не будет и ничего не получится.

Как же быть, если требуется разработать на языке С приложение, которое использует всю математику, присущую системе MATLAB, но, тем не менее, *полностью независимо от самого приложения MATLAB*? Ответ здесь заключается в том, что тогда нужно разрабатывать на языке С приложение Windows, которое обращается уже не к библиотекам MATLAB Engine (это файлы `libeng.dll` и `libmx.dll`), а к *математическим библиотекам*, поставляемым вместе с пакетом MATLAB. В данном подразделе рассмотрим функции из этих библиотек, а в следующем подразделе разработаем на их основе ряд учебных проектов.

Сразу оговоримся, что приложения, опирающиеся на математические библиотеки пакета MATLAB, помимо ограничений, свойственных приложениям

типа MATLAB Engine, еще не могут реализовывать графические возможности этого пакета.

Тем не менее мощь математических библиотек, поставляемых с пакетом MATLAB и которые можно вызывать из программ, разработанных на языке C, весьма велика. Суммарно математические библиотеки системы MATLAB содержат более 350 функций. Все математические библиотеки являются библиотеками динамической компоновки и располагаются в каталоге \bin.

Полный набор индивидуальных библиотечных файлов, составляющих логически единую математическую C-библиотеку пакета MATLAB, представлен ниже:

```
libmmfile.dll
libmatlb.dll
libmcc.dll
libmat.dll
libmx.dll
libut.dll
```

Эти библиотеки экспортируют основные математические (и некоторые вспомогательные) функции пакета MATLAB. Если изолированное от ядра пакета MATLAB (то есть от приложения matlab.exe) приложение хочет вызывать эти функции, то при его компиляции нужно предоставить компоновщику (часть компилятора в широком смысле этого термина) соответствующие библиотеки импорта. Эти библиотеки, то есть файлы libmmfile.lib, libmatlb.lib, libmcc.lib, libmat.lib, libmx.lib и libut.lib, изготавливаются с помощью утилиты lib.exe и соответствующих файлов с расширением def. Этот процесс аналогичен ранее рассмотренным примерам получения lib-файлов.

Библиотека libmat.dll содержит функции для работы с MAT-файлами, то есть с файлами закрытого формата, в которых система MATLAB сохраняет свое рабочее пространство по команде save.

С помощью утилиты dumpbin.exe, входящей в состав пакета Microsoft Visual C++, можно получить список функций, экспортируемых той или иной библиотекой. Выходную информацию удобно сохранять в текстовом файле (см. рис. 10.8).

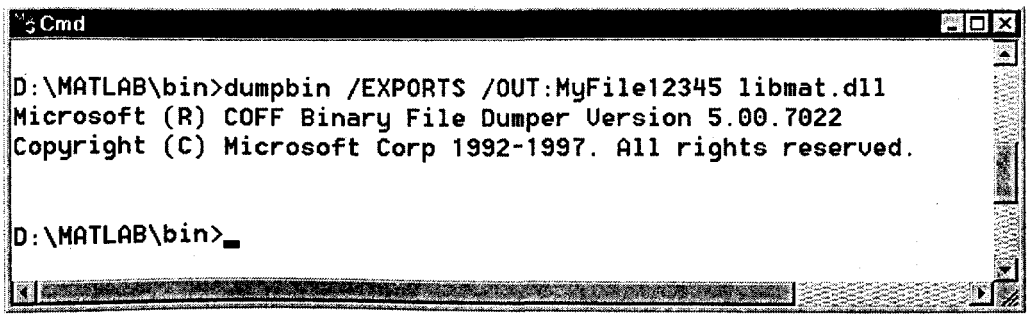


Рисунок 10.8

В частности, в представленном примере мы сохраняем выходную информацию в файле MyFile12345. Просмотреть содержимое такого файла можно любым текстовым редактором и узнать список экспортируемых функций:

```
1  0  FMATCLOSE            (00001076)
2  1  FMATDELETEMATRIX     (000010EB)
3  2  FMATGETDIR           (00001085)
4  3  FMATGETFULL          (00001115)
5  4  FMATGETMATRIX        (00001098)
6  5  FMATGETNEXTMATRIX    (000010DC)
7  6  FMATGETSTRING        (0000119A)
8  7  FMATOPEN             (00001010)
9  8  FMATPUTFULL          (00001158)
10 9  FMATPUTMATRIX        (000010C7)
11 a  FMATPUTSTRING        (0000123A)
12 B  matClose             (0000152B)
13 C  matCreateMATFile     (0000129F)
14 D  matDeleteArray       (00001642)
15 E  matDeleteMatrix      (00001733)
16 F  matGetArray          (000015D7)
17 10 matGetArrayHeader    (00001605)
18 11 matGetDir            (0000161E)
19 12 matGetFp             (00001567)
20 13 matGetFull           (00001744)
21 14 matGetMatrix         (00001921)
22 15 matGetNextArray      (000015C2)
23 16 matGetNextArrayHeader  (000015F0)
24 17 matGetNextMatrix     (00001943)
25 18 matGetString         (00001870)
26 19 matOpen              (0000147B)
27 1A matPutArray          (0000158A)
28 1B matPutArrayAsGlobal  (000015A6)
29 1C matPutFull           (000017DE)
30 1D matPutMatrix         (00001932)
31 1E matPutNextArray      (0000156E)
32 1F matPutString         (000018DB)
```

В списках экспортируемых функций помимо порядковых десятичных и шестнадцатеричных номеров функций даются также их байтовые смещения от начала файла.

Показанным способом всегда можно узнать список экспортируемых функций из любой библиотеки, входящей в группу математических С-библиотек пакета MATLAB.

Одной из наиболее важных библиотек этой группы является библиотека libmatlb.dll. Вот часть списка экспортируемых ею функций:

```
..
123  99   mlfReshape              (00044011)
124  9A   mlfRound                (00041B94)
173  9B   mlfSave                 (00044461)
125  9C   mlfScalar               (00043BD8)
126  9D   mlfSchur                (00043830)
127  9E   mlfSetErrorHandler      (0004117C)
162  9F   mlfSetLibraryAllocFcns  (000449E6)
3    A0   mlfSetLibraryCalloc     (00042C27)
1    A1   mlfSetLibraryFree       (00042C5E)
2    A2   mlfSetLibraryMalloc     (00042C48)
128  A3   mlfSetPrintHandler      (00041573)
129  A4   mlfSetstr               (00044081)
130  A5   mlfSign                 (00041BBE)
131  A6   mlfSin                  (00041BE8)
132  A7   mlfSize                 (00043617)
133  A8   mlfSort                 (0004386B)
134  A9   mlfSprintf              (000430F4)
135  AA   mlfSqrt                 (00041C0F)
136  AB   mlfSscanf               (000431FB)
137  AC   mlfStrcmp               (00042148)
138  AD   mlfStrncmp              (00042D97)
139  AE   mlfStrrep               (00044183)
140  AF   mlfSum                  (000424B0)
141  B0   mlfSvd                  (00043669)
142  B1   mlfTan                  (00041C36)
..
```

Из представленного списка хорошо видно, что функции из этой библиотеки имеют префикс mlf и коррелируют с сответствующими функциями, которые вызываются в М-языке без этого префикса. Например, для обычной функции М-языка, такой, как sin, в этой библиотеке имеется функция аналогичного действия с именем mlfSin. Для функции sort в пару ставится библиотечная функция mlfSort и т. д. Вообще, в библиотеке libmatlb.dll *сгруппированы функции, используемые на практике наиболее часто*, так что при создании многих приложений можно будет ограничиться лишь двумя библиотеками: libmatlb.dll и libmx.dll. Суммарный объем этих двух библиотек небольшой и составляет всего 445 Кбайт.

Последняя из этих двух библиотек нужна всегда, так как математические функции системы MATLAB оперируют на массивах этой системы, а работу с такими массивами как раз и обеспечивает библиотека `libmx.dll`. Мы уже использовали эту библиотеку как при создании MEX-функций, так и при написании приложений типа MATLAB Engine, поскольку везде приходится работать с массивами системы MATLAB.

Как было сказано выше, суммарно математические библиотеки пакета MATLAB содержат более 350 функций, поэтому их хотя бы краткое поименное обсуждение здесь невозможно. В то же время в этом нет практической необходимости, так как действие большинства функций вполне понятно из-за их хороших мнемонических имен. Кроме того, в случае необходимости по каждой функции можно получить справочную информацию, содержащуюся в файле `\help\pdf_doc\mathlib\Cmath_ug.pdf`.

Поэтому мы сразу перейдем к практическому примеру, иллюстрирующему создание приложений, использующих математическую мощь пакета MATLAB, но являющихся вполне независимыми от ядра этого пакета, то есть от приложения `matlab.exe`. Будем для краткости называть такие приложения *изолированными* от `matlab.exe`.

Изолированные от matlab.exe приложения Windows

Такие приложения создаются на языке C и с помощью компилятора Microsoft Visual C++ компилируются в исполняемые файлы, для дальнейшей работы которых требуется наличие на компьютере (например, в том же каталоге, где находится исполняемый файл изолированного приложения) динамических математических библиотек пакета MATLAB, о которых было рассказано в предыдущем подразделе.

Последнее означает, что для переноса изолированных приложений на другие компьютеры, на которых пакет MATLAB отсутствует, их следует поставлять вместе с указанными математическими библиотеками.

Рассмотрим простой пример, в котором вычислим `sin(x)`, опираясь на математические библиотеки системы MATLAB, а не на математическую библиотеку компилятора Visual C++. На таком примере продемонстрируем передачу параметров в функции математической библиотеки системы MATLAB, а также рассмотрим создание проекта и его настройки, необходимые для успешной компиляции.

Вот текст первого примера, в котором исключительно ради краткости убраны все проверки возвращаемых значений (разумеется, их нужно применять в реально работающих приложениях):

```
#include <stdio.h>
```

```
#include <memory.h>
#include <conio.h>

#include 'matlab.h'

double x[] = {1,3,5,7,2,4,6,8};

int main( void )
{
  mxArray* X = NULL;
  mxArray* Y = NULL;
  int i, M, N;

// Create arguments matrix X:
  X = mxCreateDoubleMatrix( 4, 2, mxREAL );

// Set its 8 elements:
  memcpy( mxGetPr( X ), data, 8*sizeof(double) );

// Calculate sin for all values at one time:
  Y = mlfSin( X );


// Get elements of mxArray Y:
  pEl  = mxGetPr( Y );
  M = mxGetM( Y );
  N = mxGetN( X );


// Print results of calculations:
  for( i=0; i< M*N; i++)
  {
    printf('%lf\n', pEl[i] );
  }

// Free all mxArray memory buffers:
  mxDestroyArray( X );
  mxDestroyArray( Y );

// Screen delay:
  printf('Press any key to exit');
```

```
    getch();
    return 0;
}
```

Из этого примера видно, что математические функции из библиотеки системы MATLAB сохраняют важное свойство всех функций системы MATLAB – *способность осуществлять массовые (групповые) вычисления*. Здесь мы одним вызовом функции `mlfSin` вычислили сразу восемь скалярных значений функции `sin` для восьми различных скалярных значений аргумента.

В остальном представленный код нам должен быть абсолютно понятен, так как в нем нет ничего нового по сравнению с примерами, которые мы разрабатывали ранее в связи с приложениями MATLAB Engine. Обратим только внимание на необходимость удалять все буферы в памяти компьютера, выделенные под структуры `mxArray`. Причем это нужно делать как для массивов, созданных нами явно функцией `mxCreateDoubleMatrix`, так и для массивов, являющихся возвращаемыми значениями функций из математической библиотеки системы MATLAB. В нашем коде таким массивом является массив `Y` значений синуса. Память под этот массив была выделена в процессе работы библиотечной функции `mlfSin`, а освободить эту память должны мы сами:

```
mxDestroyArray( X );
mxDestroyArray( Y );
```

Иначе мы допустим ситуацию, когда после завершения работы нашего приложения память, выделенная в процессе ее работы, не будет освобождена, что перегружает операционную работу дополнительными обязанностями, а это нельзя признать хорошим стилем программирования.

Чтобы из представленного исходного С-кода получить исполняемый модуль, нужно создать проект `Win32 Console Application`, прописать в свойствах проекта путь к включаемому файлу `matlab.h`, добавить в список библиотек проекта еще две библиотеки: `libmx.lib` и `libmatlb.lib`. Изготовить эти библиотеки нужно самостоятельно так, как мы всегда поступали в таких случаях (см., например, подраздел посвященный приложениям типа MATLAB Engine), после чего скопировать их в главный каталог нашего проекта. После всего этого нужно запустить процесс компиляции нажатием клавиши F7.

Запустив скомпилированное приложение, получим результаты вычислений восьми значений синуса в консольном окне, показанном на рис. 10.9.

Как мы заявляли ранее, полученное приложение абсолютно независимо от самого приложения `matlab.exe`. Его можно перенести вместе с библиотеками `libmx.dll` и `libmatlb.dll`, а также еще двумя вспомогательными библиотеками `libut.dll` и `libmat.dll` (около 100 Кбайт в сумме) на другой компьютер, на котором пакет MATLAB не установлен, а работоспособность нашего приложения от этого не пострадает. В этом смысле изолированные от `matlab.exe` приложения поддаются наиболее простому способу распространения среди раз-

личных групп пользователей. Естественно, сохраняется зависимость от операционной платформы (невозможно, например, запустить приложение, скомпилированное под Microsoft Windows, на платформе Macintosh).

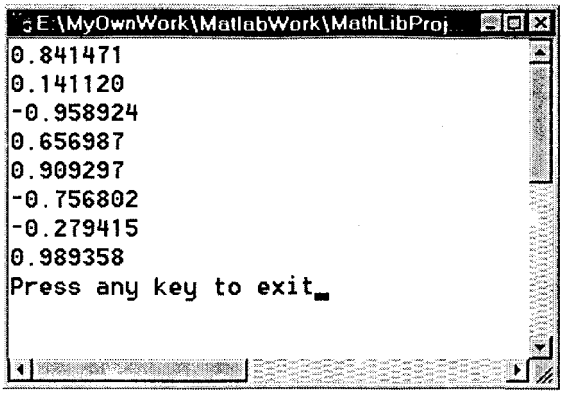


Рисунок 10.9

Приложение

Создание новых типов данных. Классы и объекты

Очевидно, что необходимость в создании новых типов данных, встраиваемых в М-язык системы MATLAB, возникает не так уж часто. Это прерогатива крупных проектов, в которых такие операции осуществляются с целью максимальной интеграции проекта в ядро системы MATLAB. Так, в основном (но не всегда) поступают разработчики универсальных и важных с практической точки зрения *пакетов расширения* (*toolboxes*) для системы MATLAB.

Создание новых типов данных в системе MATLAB основывается на понятии *класса* этой системы. Это достаточно сложный во всех своих аспектах вопрос, детальное изучение которого уместно лишь в профессионально ориентированных учебных пособиях. Настоящее учебное пособие, скорее, ориентировано на то, чтобы познакомить читателя максимально широко с разными сторонами работы с универсальным математическим пакетом MATLAB, поэтому мы здесь не будем очень подробно знакомиться с классами системы MATLAB. Ограничимся лишь достаточно беглым их изложением.

Как мы уже сказали, механизм классов в любом языке программирования позволяет создавать новые типы данных. Экземпляры этих новых типов принято называть *объектами классов*. Объекты призваны *скрывать детали реализации* (внутренние данные и методы), а общение с объектами осуществляется через тщательно отобранные открытые методы, составляющие в своей совокупности *интерфейс взаимодействия* с объектами класса.

В системе MATLAB концепция объектно-ориентированного программирования позволяет реализовывать классы на трех принципах: *сокрытия внутренних данных и методов*, *переопределения операций* и *наследования классов*.

Универсальная теоретическая концепция объектно-ориентированного программирования здесь реализована не в полной мере, так как отсутствуют механизмы, аналогичные виртуальным функциям классов языка C++.

Все основные типы массивов системы MATLAB (числовые и символьные массивы произвольных размерностей, разреженные массивы, структуры и ячейки) представляют собой встроенные классы, а конкретные переменные являются объектами этих классов. Дополнительно имеется возможность вводить новые классы, а также переопределять и доопределять методы всех существующих классов.

Для создания нового класса объектов нужно *спроектировать структуру* системы MATLAB, которая будет хранить данные, принадлежащие объекту,

и определить функции-методы работы с этими объектами. Эти функции определяются в обычных М-файлах, которые должны быть помещены в специальную папку, имя которой начинается с символа @, а в остальном должно совпадать с именем структуры (класса), причем эта папка должна входить в папку, доступ к которой уже определен в списке путей доступа системы MATLAB. Саму папку-контейнер методов добавлять в путь MATLAB не нужно.

Как мы знаем, язык программирования системы MATLAB, то есть М-язык, не имеет операторов для объявления типов переменных, в том числе деклараций (объявлений) новых классов и типов. Поэтому любой объект – представитель некоторого класса создается в момент вызова функции-конструктора этого класса. Следовательно, для создания объекта нужно создать хотя бы один метод (конструктор) в упомянутой папке-контейнере его класса.

Все поля структуры, хранящей данные класса, являются *скрытыми* (*private*), то есть эти поля доступны только из методов данного класса. Напрямую в выражениях их использовать нельзя.

Рассмотрим пример конструктора, создающего объекты класса «отношение». Эти объекты будут представлять собой отношение двух вещественных чисел, при этом не требуется, чтобы числитель и знаменатель отношения были целыми числами.

Конструктор класса должен находиться в файле `@ratio/ratio.m`:

```
function r = ratio(a,b)
%Ratio a/b class constructor.
% r = ratio(a,b)

if nargin == 0
   r.a = 0;
   r.b = 1;
   r = class(r, 'ratio')
elseif nargin == 1
   if isa(a,'ratio')
      r = a;
   elseif isa(a,'double')
      r.a = a(1);
      if length(a) > 1
         r.b = a(2);
      else
         r.b = 1;
      end
      r = class(r,'ratio');
   end
else
   r.a = a(1);
```

```
    r.b = b(1);
    r = class(r,'ratio');
end
```

Представленный конструктор создает отношение заданных чисел a и b. При этом анализируется целый ряд возможных способов задания отношения – из пары чисел, из другого отношения, из элементов вектора или нулевое отношение, если аргументы не заданы.

Сначала в конструкторе проверяется количество аргументов, и если оно равно нулю, то создается нулевое отношение. Точнее, создается структура `r`, содержащая отношение чисел `0` и `1`. Затем из этой структуры конструируется сам объект при помощи встроенной функции `class()`. У этой функции есть два обязательных параметра. Первый параметр – структура, которая будет представлять данные объекта, а второй – текстовая строка, содержащая имя создаваемого класса. Это имя должно совпадать с именем конструктора и папки, его содержащей (с добавлением символа `@` в начале имени папки).

Если при вызове конструктора был задан единственный аргумент, то проверяется тип этого аргумента. В том случае, когда был задан объект того же класса (так называемый *copy-constructor*), создается его копия при помощи операции присваивания. Проверка класса переданного объекта осуществляется по его имени при помощи функции `isa()`.

Если был передан обычный массив, то он копируется, а отношение конструируется из первых двух его элементов. Если передано одно число, то знаменатель полагается равным нулю. Наконец, если были переданы два параметра, то отношение конструируется из первых элементов переданных векторов.

Подобная последовательность действий – *создание структуры и порождение класса из нее* – является обязательной для конструктора объектов. Естественно, что структура, хранящая данные объекта, может быть создана сколь угодно сложной – с произвольным набором полей.

Отметим, что за пределами методов данного класса функциями `class()` и `isa()` тоже можно активно пользоваться. В этом случае первая из них может принять только один аргумент – объект, а возвращает текстовую строку – имя класса заданного объекта. Например, для заданного вещественного вектора эта функция вернет строку типа `'double'`.

Часто бывает необходимо *преобразовать объект одного класса к другому классу*. Например, может понадобиться преобразовать созданное нами отношение обратно к типу вещественного числа.. Чтобы обеспечить такую возможность, нужно создать для исходного класса специальную *функцию-конвертор*. Имя этой функции (и имя ее М-файла) должно совпадать с именем класса, к которому она будет преобразовывать исходный объект. Для нашего примера понадобится следующая функция :

```
function c = double( r )
% @ratio/double.m.
```

```
c = r.a / r.b;
```

Теперь мы можем преобразовывать числа в отношения и обратно:

```
P = ratio( 1, 2 );
double( p )
ans =
   0.5000
```

Другим частым преобразованием является *преобразование к текстовому виду* (метод `char()`) для нужд распечатки объекта. В примере с отношением этот метод (`@ratio/char.m`) мог бы порождать строку `'1/2'`. Вот этот метод:

```
function s = char(r)
% @ratio/char.m.
s = [ num2str(r.a) ' / ' num2str(r.b) ];
```

Преобразование к тексту используется в другом методе, которой желательно реализовать в каждом создаваемом новом классе. Это метод `display()` – он вызывается всякий раз, когда системе MATLAB нужно распечатать (вывести в командное окно) содержимое объекта. В основном это происходит, когда в командной строке введено выражение, не завершенное точкой с запятой.

```
function display(r)
% @ratio/display.m
disp(' ');
disp([inputname(1),' = '])
disp(' ');
disp([' ' char(r)])
disp(' ');
```

Одним из важнейших способов управления поведением объекта служит *переопределение* для него *основных математических операций*. Делается это так же, как и в предыдущем случае, при определении конструкторов и конверторов. Нужно завести в папке, содержащей методы класса, М-файл с именем, соответствующим имени переопределяемой операции, и в этом файле определить функцию с таким же именем:

```
function r = plus(p,q)
% @ratio/plus.m
p = ratio(p);
q = ratio(q);
r = ratio(p.a*q.b + p.b*q.a, p.b*q.b);
```

Эта функция, текст которой должен располагаться в файле `@ratio/plus.m`, позволит складывать отношения как обычные вещественные матрицы системы MATLAB. При этом принудительное преобразование формальных параметров

к типу «отношение» гарантирует, что будут правильно вычисляться смешанные выражения:

```
p = ratio([1 2]);
q = ratio(1,3);
pq = p + q;
p1 = p + 1;
q1 = 1 + q;
```

В следующей таблице приводится полный список имен методов для переопределения всех операций системы MATLAB:

Операция	Метод	Описание
a + b	plus(a,b)	Сложение
a - b	minus(a,b)	Вычитание
-a	uminus(a)	Унарный минус
+a	uplus(a)	Унарный плюс
a.*b	times(a,b)	Поэлементное умножение
a*b	mtimes(a,b)	Матричное умножение
a./b	rdivide(a,b)	Правое поэлементное деление
a.\b	ldivide(a,b)	Левое поэлементное деление
a/b	mrdivide(a,b)	Правое матричное деление
a\b	mldivide(a,b)	Левое матричное деление
a.^b	power(a,b)	Поэлементное возведение в степень
a^b	mpower(a,b)	Матричное возведение в степень
a < b	lt(a,b)	Меньше
a > b	gt(a,b)	Больше
a <= b	le(a,b)	Меньше или равно
a >= b	ge(a,b)	Больше или равно
a ~= b	ne(a,b)	Не равно
a == b	eq(a,b)	Равно
a & b	and(a,b)	Логическое И
a | b	or(a,b)	Логическое ИЛИ
~a	not(a,b)	Логическое НЕ
a:d:b или a:b	colon(a,d,b) colon(a,b)	Оператор «двоеточие» – генерация
a.'	transpose(a)	Транспонирование
a'	ctranspose(a)	Комплексно-сопряженное транспонирование

Операция	Метод	Описание
[a b]	horzcat(a,b,…)	Горизонтальная конкатенация
[a; b]	vertcat(a,b,…)	Вертикальная конкатенация
a(s1,s2,…sn)	subsref(a,s)	Ссылка по индексу
a(s1,…,sn) = b	subsasgn(a,s,b)	Присваивание по индексу
b(a)	subsindex(a,b)	Индексирование
Вывод в командной строке	display(a)	Распечатка значения

Среда MATLAB всегда начинает просмотр функций, оперирующих объектом, с папки, содержащей методы класса, – перед любой другой папкой, указанной в списке путей доступа системы MATLAB. Это означает, что при создании нового класса всегда есть возможность переопределить любую функцию для данного класса. Например, для обычных матриц можно создать специализированную функцию `norm()`, вычисляющую норму матрицы каким-либо методом, отличным от стандартного. Для этого нужно в папке `@double` создать файл `norm.m`.

Обычно MATLAB считает объекты всех классов имеющими одинаковый приоритет и вызывает бинарный метод у левого операнда выражения. Существует возможность управлять иерархией приоритетов для разных классов так, что в ситуациях `a + b` и `b + a` будет гарантированно вызываться метод операнда `b`.

Для того чтобы повысить приоритет создаваемому классу по сравнению с другим классом нужно в конструкторе вызвать функцию `superiorto()` и в качестве ее единственного аргумента указать текстовую строку – имя класса, приоритет которого в выражениях должен быть всегда ниже приоритета создаваемого класса.

Аналогично можно понизить приоритет создаваемого класса по сравнению с некоторым другим. Для этого в конструкторе надо вызвать функцию `inferiorto()`.

Нужно не забывать, что объекты базируются на структурах, которые в системе MATLAB на самом деле всегда являются массивами структур. Этим фактом можно воспользоваться для конструирования объектов-массивов. При этом во всех методах такого класса нужно будет пользоваться циклом по элементам базового массива структур – `r(i).a`. Например, можно было бы расширить определение отношения до отношения двух векторов, которое являлось бы массивом структур.

К весьма нетривиальным последствиям может привести переопределение взятия значения по индексу. В самом деле, для какого-то математического объекта операция `p(3)` может означать следующее:

1. Значение функции при `x = 3`.

2. Третью производную.
3. Коэффициент полинома при x3.
4. Третий элемент последовательности.

Разработчик класса может выбрать любой из этих путей в зависимости от своих потребностей. Для того чтобы зафиксировать выбранный способ интерпретации выражения `p(c)`, ему придется переопределить метод *ссылка по индексу* – `subsref(A,S)` (см. показанную выше таблицу операций).

Есть три возможных причины вызова этого метода: `A(i)` – обычный индекс в вещественном массиве; `A{i}`- обращение по индексу в массиве ячеек; `A.val` – обращение к полю структуры. Во всех случаях будет произведен вызов метода `subsref(A,S)`, где `S` – структура с полями `type` и `subs`. Поле `S.type` будет иметь значение текстовой строки `'()'`, `'{}'` или `'.'` в зависимости от того, какой из приведенных выше случаев имеет место. Поле `S.subs` является массивом ячеек, содержащим сами индексы.

Если в выражении встретится `A(1:3,2,:)`, то `S.subs` будет равно

```
{1:3, 2, ':'}
```

Для более сложного примера `A(1,2).name(3:4)` породит один вызов `subsref(A,S)` с `S`, представляющей собой массив `3 x 1`:

```
S(1).type = '()'
S(2).type = '.'
S(3).type = '()'
S(1).subs = '{1,2}'
S(2).subs = 'name'
S(3).subs = '{3:4}'
```

Аналогично в случае присваивания значения по индексу `A(I) = B`, или `A{I} = B`, или `A.I = B` генерируется вызов

```
A = subsasgn(A,S,B)
```

где `S` – та же структура, что и в предыдущем случае.

Наконец, последний переопределяемый метод индексирования – `subsindex()` вызывается независимо для каждого указанного индекса в выражении `X(A)`, где `A` является объектом. Этот метод обязательно должен возвращать целочисленный массив реальных индексов, которые и будут использоваться для извлечения значений из массива `X`. Этот метод вызывается встроенными методами `subsref()`, `subasgn()` и может понадобиться для переопределения этих методов.

Система MATLAB поддерживает одиночное и множественное наследование для классов. Класс может унаследовать все поля и методы родительского класса,

добавить свои поля и методы и переопределить часть методов родительского класса на свои. Наследование осуществляется вызовом функции `class()` в конструкторе класса с указанием имен всех родительских классов:

```
function p = circle(x,y,R)
% @circle\circle.m

center = figure(x,y);
p.r = R;
p = class(p,'circle',center);

end
```

Здесь мы предположили, что уже имеется класс «геометрическая фигура» с единственным атрибутом – координатами центра. Данный пример реализует конструктор класса «окружность», который наследует классу «геометрическая фигура» и добавляет свой атрибут – радиус. Отметим, что созданный объект будет генерировать логическое значение «истина» при передаче его в функцию `isa()` как при проверке, является ли он кругом, так и при проверке, является ли он геометрической фигурой.

Для реализации множественного наследования нужно при вызове функции `class()` в конструкторе указать список всех родителей через запятую:

```
p = class(p,'name', parent1, parent2, ...);
```

Создаваемый класс унаследует все данные и методы всех родителей. Если у нескольких родителей совпадут имена каких-либо методов, то приоритет при вызове у потомка получат методы классов, идущих первыми в списке параметров функции `class()`.

Как уже отмечалось, все *данные у объекта являются защищенными* – они доступны только для методов данного класса и его потомков. Методы класса при этом можно вызывать откуда угодно. MATLAB позволяет создавать также методы, полностью недоступные для внешнего использования. Для этого в папке класса нужно завести подпапку с именем `private` и в ней разместить все М-файлы методов, которые необходимо скрыть от внешнего использования. Они будут доступны только из других методов данного класса и его потомков.

Оглавление

Часть 1. Вычисления и визуализация 3

Глава 1. Числовые массивы в системе MATLAB 3

Рабочее пространство системы MATLAB и ее командное окно..... 3

Вещественные числа и тип данных double 9

Комплексные числа и комплексные функции 16

Формирование одномерных числовых массивов 19

Двумерные массивы чисел: матрицы и векторы 24

Многомерные числовые массивы 29

Вычисления с массивами..... 33

Множественная индексация массивов в системе MATLAB 39

Глава 2. Визуализация результатов вычислений 47

Построение графиков функций..... 47

Оформление графиков и графических окон 52

Специальная графика системы MATLAB 60

Трехмерная графика 65

Дополнительные детали оформления трехмерных графиков 73

Растровые изображения и тип данных uint8 77

Глава 3. Массивы символов, структур, ячеек. Файловые операции 87

Массивы символов и тип данных char 87

Встроенные функции для обработки строк 94

Массивы структур 98

Массивы ячеек..... 102

Чтение и запись произвольных бинарных файлов 109

Чтение и запись произвольных текстовых файлов 119

Глава 4. Краткий обзор встроенных средств решения типовых задач алгебры и анализа 125

Решение систем линейных уравнений 125

Операции линейной алгебры над матрицами. Матричные функции 126

Разреженные матрицы 130

Вычисление спецфункций математической физики 131

Нахождение нулей функций 133

Поиск минимума функции 136

Вычисление определенных интегралов 139

Решение систем обыкновенных дифференциальных уравнений 143

Глава 5. Интерактивный режим работы и его автоматизация с помощью сценариев 148

Сохранение результатов вычислений интерактивного сеанса работы 148

Операторы цикла. Векторизация как альтернатива циклам 153

Анимация и звук в системе MATLAB 157

Сценарии и M-файлы 162

Аналитические вычисления с помощью пакета расширения Symbolic Math Toolbox 166

Справочная подсистема пакета MATLAB 172

Часть 2. Программирование в среде системы MATLAB 176

Глава 6. Программирование функций на M-языке 176

Синтаксис определения и вызова M-функций 176

Конструкции управления 181

Интерактивное взаимодействие M-функций с пользователем 185

Локальные, глобальные и статические переменные 190

Рекурсивные функции. Производительность M-функций 193

M-функции с переменным числом входных параметров и выходных значений 198

Контроль входных параметров и выходных значений M-функции 200

Практические советы по разработке и отладке M-функций 205

Глава 7. Примеры конкретных разработок M-функций 208

Функции, работающие со временем и датами 208

Обработка текстов ... 213
Функции для работы с файлами данных ... 219
Динамическое построение графика функции ... 222
Вращение трехмерных графиков ... 227

Глава 8. Программирование функций на языке C ... 230

Интерфейс MEX-функций с системой MATLAB ... 230
Создание и компиляция DLL-проекта в среде Microsoft Visual C++ ... 234
Вызов функций MATLAB API ... 238
Отладка MEX-функций ... 243
Примеры конкретных разработок MEX-функций ... 247
Вызов функций и команд системы MATLAB из MEX-функций ... 256

Часть 3. Создание законченных приложений ... 260

Глава 9. Законченные приложения на базе графического интерфейса пользователя системы MATLAB ... 260

Графические окна системы MATLAB и элементы управления ... 260
Создание основных элементов управления ... 263
Графический объект axes ... 270
Callback-функции ... 275
Применение утилиты guide для формирования пользовательского интерфейса ... 280
Динамическая перестройка элементов управления ... 282
Использование манипулятора мышь в графических окнах пакета MATLAB ... 290
Создание меню ... 293

Глава 10. Взаимодействие внешних приложений с системой MATLAB 299

Взаимодействие приложений Windows с MATLAB Engine ... 299
Создание и компиляция EXE-проекта в среде Microsoft Visual C++ ... 311
C-библиотеки математических функций системы MATLAB ... 314
Изолированные от matlab.exe приложения Windows ... 318

Приложение ... 322

Создание новых типов данных. Классы и объекты ... 322