*Государственное образовательное учреждение   
высшего профессионального образования*

**Московский технический университет связи и информатики**

***Кафедра информатики***

В.Н.Шакин, Т.И.Семенова, В.В.Фриск

**Базовые средства   
математического пакета Scilab**

Для студентов в качестве учебника по дисциплине   
«Введение в математические пакеты прикладных программ», обучающихся по направлений 11.03.02, 09.03.01, 09.03.02, 10.03.01 подготовки бакалавров

**Москва**

**2019**

**Предисловие**

Современные Математические Пакеты Прикладных Программ (МППП), такие как Maple [1], Mathematica [2], Matlab [3]и некоторые другие, являются сложными программными системами, включающими специализированные системные и языковые средства. Хотя пользовательские интерфейсы и языки программирования у них различны, а внутренние структуры и даже используемые алгоритмы отличаются друг от друга, все они содержат минимально необходимый набор методов решения математических задач и средства для визуализации и отображения полученных результатов. И поскольку принципиально общих свойств значительно больше, чем различий, то после освоения одной компьютерной математической системы переход к другой не является проблемой.

Наиболее известным и популярным среди вышеперечисленных программных средств численных расчетов является математический пакет Matlab [5,6,7]. Он позволяет производить вычисления различной степени сложности, содержит одноименный язык программирования, предоставляет большое количество функций анализа и обработки данных, связанных практически со всеми областями как классической, так и прикладной математики. Однако этот математический пакет является коммерческим, что затрудняет его использование в учебном процессе.

Существуют и свободно распространяемые альтернативы данного пакета, например, пакет Scilab [4]. Немаловажно, что пакет Scilab может быть использован при работе в различных операционных системах: Linux, Windows и MacOS при этом онзанимает значительно меньший объем памяти по сравнению с пакетом Matlab.

Scilab – это математический пакет численных вычислений, являющийся самым полным аналогом пакета Matlab, также предназначенный для выполнения научных и инженерных расчетов. Однако в этом пакете отсутствуют средства аналитических вычислений, в чем он пока уступает пакету Matlab по своим вычислительным возможностям, но Scilab постоянно совершенствуется, и в каждой его следующей версии появляются все новые и новые возможности. Последнюю на настоящий момент версию пакета Scilab 6.02(в том числе русифицированную), описанию которой посвящен данный учебник, можно скачать с официального сайта программы [www.scilab.org](http://www.scilab.org). Для более глубокого изучения возможностей пакета на сайте разработчиков имеется подробная справочная система.

В настоящее время Scilab все шире используется в учебном процессе многих вузов России. Практика показывает, что работа с пакетом Scilab легко осваивается студентами, однако для того, чтобы систематизировать процесс изучения и объяснить реализацию многих возможностей Scilab недостаточно только одной (пока не совсем совершенной) справочной информаций. В последнее время, в связи с возрастающей популярностью Scilab, появляются учебные пособия, предназначенные, как правило, для узкоспециализированного использования Scilab [8, 9, 10, 11], где описание основных возможностей системы, а именно системы численных вычислений и программирования, не выходит за рамки справочной системы. Именно поэтому возникла необходимость в появлении учебника, в котором были бы систематизированы, описаны и проиллюстрированы примерами все базовые средства Scilab.

Учебник состоит из двух разделов. Первый раздел посвящен интерфейсу пользователя и Рабочей среде Scilab, основным объектам математическим пакетом Scilab и средствам программирования.

Описание элементов рабочей среды (параграф 1.1) позволяет начинающему пользователю познакомиться с интерфейсом пользователя Рабочей среды Scilab и назначением основных окон Scilab, а основные объекты Scilab (параграф 1.2) рассматриваются в основном с точки зрения использование их в численных расчётах*.* Поскольку в системе Scilab все объекты являются матрицами различных допустимых типов, которые в основном предназначены для проведения математических расчетов и обработки данных, рассмотрены особенности, как алгебраических, так и поэлементных операций, а также представлены соответствующие наборы встроенных библиотечных математических функций. В соответствии с приведенной иерархией типов объектов, кроме числовых данных описана работа с логическими данными, а также списками, структурами и ячейками данных, Особое внимание отводится описанию и последовательности действий по созданию в Рабочем окне сценариев и встроенных функций пользователя.

Учитывая ориентацию Scilab на работу с матрицами, в параграфе 1.3 представлены различные способы создания векторов и матриц (с помощью различных операций, встроенных функций, объединения уже существующих матриц и другие способы). При реализации доступа к элементам матрицы, наряду со стандартным индексированием векторов и матриц, здесь вводится и описывается векторное (линейное) индексирование, которое может влиять не только на скорость выполнения программного кода, но и его читабельность. Далее рассматриваются операции и функции, позволяющие осуществлять эффективные поэлементные матричные вычисления, а также логическая индексация матриц, и способы создания массивов структур и ячеек.

Одним из самых наглядных способов представления результатов вычислений в Scilab является визуализация данных (параграф 1.4). В пакете Scilab имеет множество команд функций, предназначенных для построения различных типов графиков: стандартные с линейными осями, ступенчатые, трехмерные поверхности, контурные, сетчатые и многие другие. В Scilab графические объекты строятся в графических окнах, где они могут быть отформатированы с использованием команд графического окна или программно. В параграфе 1.4. рассмотрены наиболее часто используемые типы графикови команд, позволяющих их построить, а также и графические окна, иллюстрирующие выполнение этих команд, причем использование каждой команды или описываемой функции иллюстрируется примерами или рисунками.

Параграф 1.5 является одним из самых объемных и информативных с точки зрения описания работы в Scilab. Он содержит описание средств программирования в Scilab. Здесь вначале вводятся некоторые общие понятия идеологии программирования: система программирования, программа, алгоритм, транслятор, интерпретатор, макрос, сценарий, примитив и многие другие. Затем рассматриваются процессы создания, отладки и сохранения программв виде файлов текстовых сценариев и функций с помощью встроенного текстового редактора **SciNotes**ианализируютсяпроблемы видимости переменных при создании сценарий и функций.

Второй раздел данного учебника посвящен решению задач вычислительной математики средствами Scilab: аппроксимации и интерполяции функций, дифференцирования и интегрирования, решения нелинейных и дифференциальных уравнений и оптимизации функций. Эти численные методы широко используются при решении прикладных задач в области связи. Теоретический материал по каждой теме, включающий постановку задачи и описание методов, применяемых для ее решения, изложен очень кратко. Каждый параграф данного раздела содержит описания наиболее часто используемых функций Scilab, а реализация каждой из функции проиллюстрирована примерами.

Учебник содержит несколько приложений, каждое из которых включает таблицы, содержащие справочную информацию по командам и функциям той или иной темы. В тексте учебника предусмотрены ссылки на определенные приложения, с указанием номера таблицы, в которой содержится необходимый справочный материал, поэтому данныйучебник может являться не только учебно-методическим материалом при изучении базовых средств пакета Scilab, но и справочником при работе с этим пакетом и использоваться в учебном процессе для всех уровней студентов: бакалавров, магистров и аспирантов.

**Глава 1 Основы работы   
с математическим пакетом Scilab**

* 1. **Рабочая среда Scilab**

**1.1.1 Графический интерфейспользователя**

После запуска пакета Scilabверсии 6.01на экране дисплея появляется его **Рабочая среда** (интерфейс пользователя) в стандартной конфигурации (рис. 1.1.1-1). При этом система готова к проведению вычислений в  
***Командном окне***Scilab (в дальнейшем ***Командное окно***).

**Рабочая среда** Scilab – это обычный графический интерфейс приложений MS Windows. В **Рабочей среде** стандартной конфигурации размещены несколько его основных компонентов изображенных на рис. 1.1.1-1 с соответствующими выносками, пронумерованными от 1 до 6: ***О*с*новное меню***,  
***Панель инструментов***, ***Командное Окно***, окно ***Обозреватель переменных***, окно ***Журнал команд*** и окно ***Обозреватель файлов***.

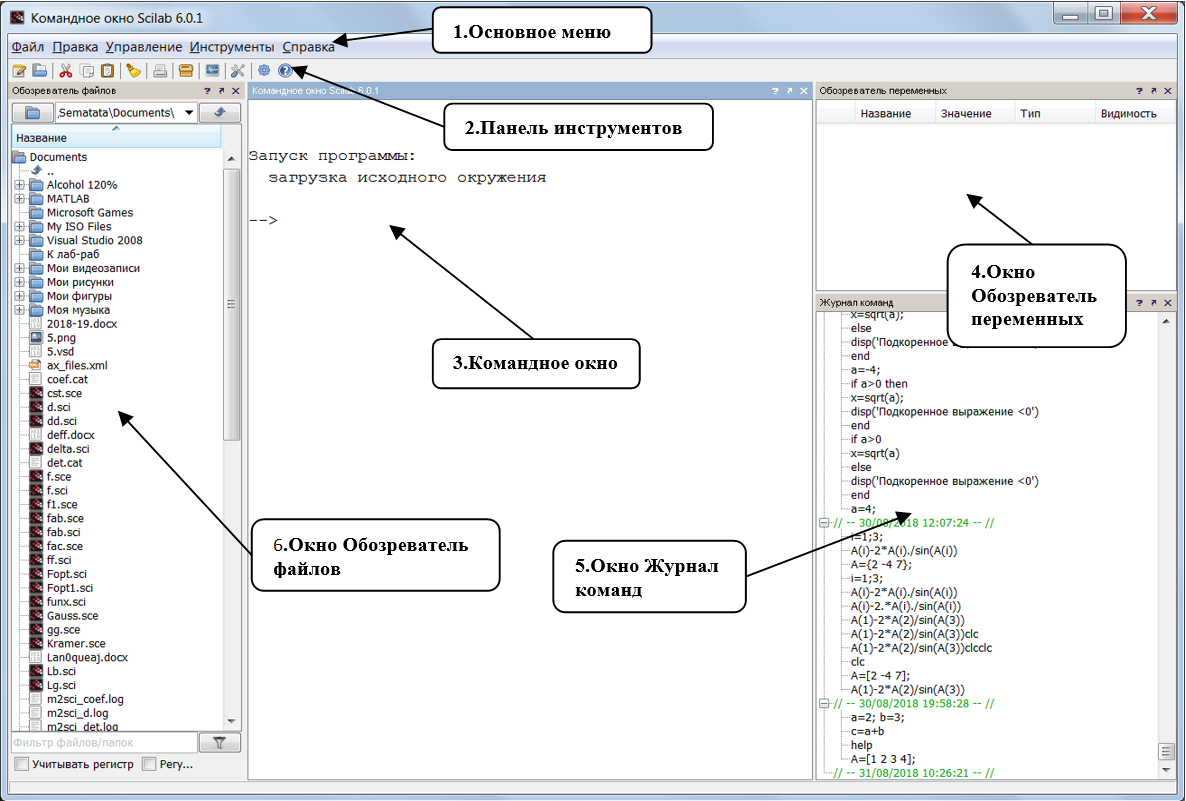


Рис. 1.1.1-1 Стандартная конфигурация **Рабочей среды** Scilab 6.01

* + 1. **Основные элементы Рабочей среды Scilab**

Основными элементами **Рабочей среды** Scilab являются следующие элементы:

1. ***Основное меню*** позволяет, с помощью команд, содержащихся в его элементах, обеспечить доступ ко всем функциональным возможностям Scilab*.* В зависимости от того, какое окно в данный момент активно, набор элементов меню может меняться (заголовок активного окна подсвечивается синим цветом). Более подробно все элементы основного меню будут рассмотрены ниже.
2. ***Панель инструментов*** позволяет организовать выполнение команд: ***ОткрытьSciNotes*, *Открыть файл*, *Вырезать, Скопировать, Вставить, Очистить командное окно, Печать, Управление модулями, Xcos, Настройки Scilab, Примеры*** и ***Справочная система***. Все команды отображаются на панели в виде пиктограмм с соответствующим изображением, подсказывающим их назначение.
3. ***Командное Окно*** предназначено для отображения вводимых команд, результатов их выполнения, а также сообщений об ошибках, то есть для работы пользователя с пакетом Scilab в интерактивном режиме.
4. Окно ***Обозреватель переменных*** отображает содержимое рабочего пространства объектов Scilab, и с помощью команд контекстного меню и ***Редактора переменных*** позволяет выполнять соответствующие действия с объектами этого пространства.
5. Окно ***Журнал команд*** предназначено для хранения, просмотра и повторного вызова ранее введенных командных строк.
6. Окно ***Обозреватель файлов*** предназначено для просмотра и установки путей доступа к файловой системе и выбора текущей папки, в которой хранится нужный файл, поскольку перед работой с конкретным файлом (чтение или запись файла), к нему требуется установить путь доступа. Внизу окна отображается фильтр файлов/папок.

В **Рабочую среду** Scilab стандартной конфигурации вписаны четыре основных окна. Они могут перемещаться вместе с основным окном и вместе с ним изменять свои размеры. Если в процессе сеанса работы конфигурация окна нарушена, то ее можно восстановить. Для этого на панели инструментов следует выбрать элемент ***Настройки Scilab***, далее в списке, находящемся в левой части открывшегося одноименного окна (рис. 1.1.2-1) двумя щелчками мыши открыть ***Общее***, а затем ***Конфигурация рабочего стола***. После чего необходимо ***Сбросить расположение панелей***, выполнить команду ***Apply***(***Применить***) и нажать**OK**. Следует помнить, что после изменении конфигурации рабочего стола требуется произвести перезагрузку Scilab.

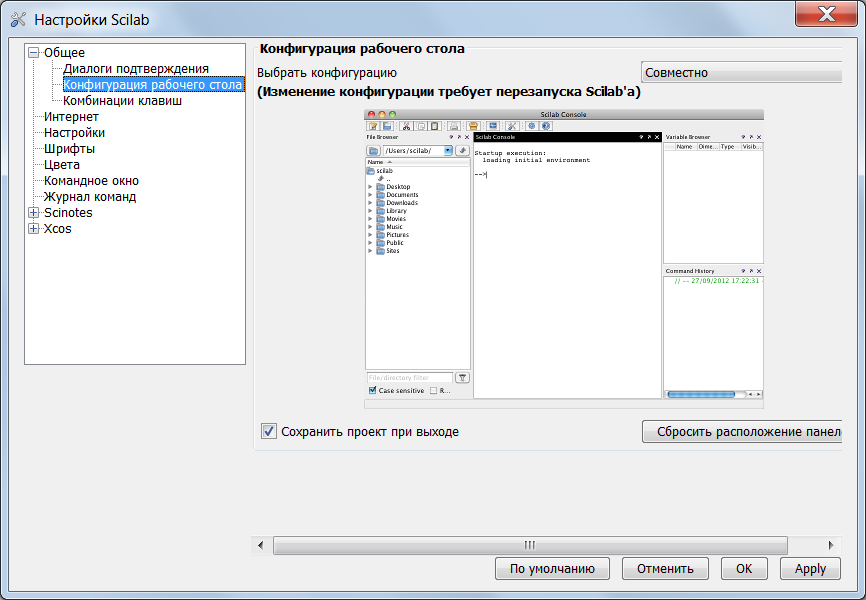


Рис. 1.1.2-1. Окно ***Настройки Scilab***(***Конфигурация рабочего стола)***

* + 1. **Основное меню и его элементы**

Рассмотрим основные компоненты **Рабочей среды** стандартной конфигурации более подробно.

Количество элементов ***Основного меню*** может меняться в зависимости от того, какое окно в данный момент является активным.

Если активно ***Командное окно****,* то ***Основное меню*** состоит из следующих пяти элементов: *Файл, Правка, Управление, Инструменты, Справка*. Каждому из этих элементов соответствует своё всплывающее меню со своим набором команд.

Элемент меню *Файл* содержит набор команд, необходимых для текущей работы с файлами: ***Выполнить****,* ***Открыть файл****,* ***Загрузить окружение****,* ***Сохранить окружение****,* ***Сменить текущий каталог****,* ***Отобразить текущий каталог****,* ***Параметры страницы****,* ***Печать*** и ***Выход****.*

Элемент меню *Правка* содержит команды, которые позволяют осуществлять различные действия: ***Вырезать****,* ***Копировать****,* ***Вставить****,* ***Очистить   
буфер обмена****,* ***Выделить все****,* ***Показать/Скрыть панель инструментов****,* ***Очистить журнал команд****,* ***Очистить командное окно*** и ***Настройки***.

Рассмотрим, например, последовательность действий по изменению размера шрифта и его цвета в ***Командном окне*** с использованием команды *Настройки Scilab*.Выполнение команды вызывает появление окна ***Настройки Scilab***(рис. 1.1.3-1).

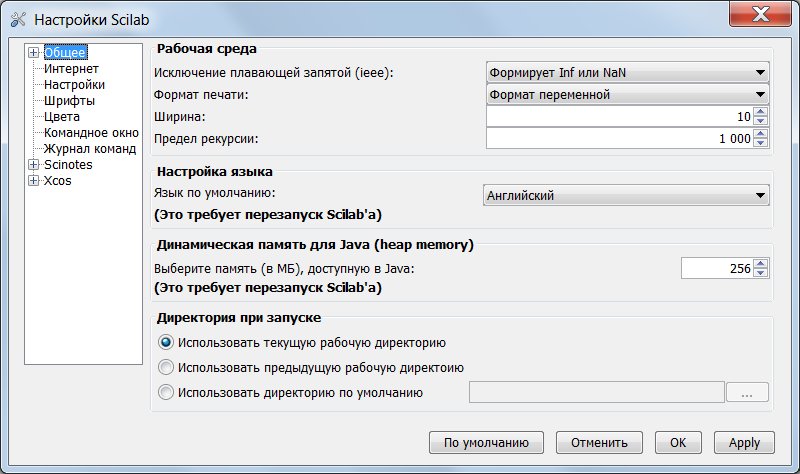


Рис. 1.1.3-1. Окно ***НастройкиScilab****(****Общее****)*

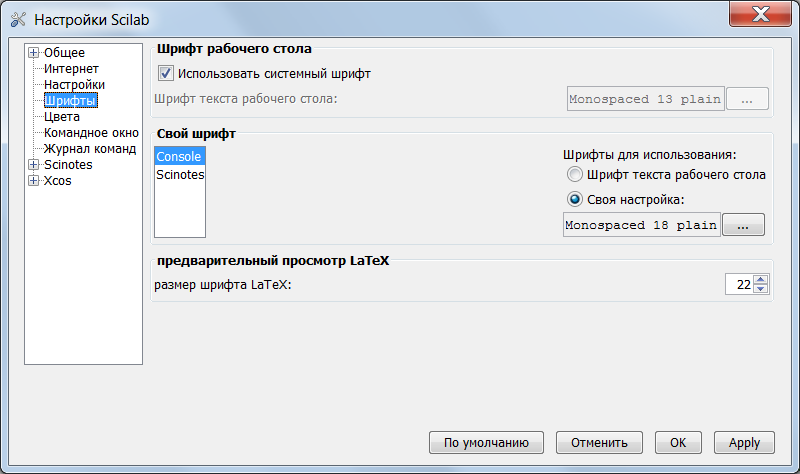


Рис. 1.1.3-2. Окно ***Настройка Scilab***(***Шрифты***)

Для изменения размера шрифта, вменю **Общее** нужно выбрать раздел **Шрифты**,при этом откроется вкладка **Шрифт рабочего стола** (рис.1.1.3-2). Следует отметить, что здесь можно настроить шрифт не только для ***Командного окна*** *(****Console****),* но и для текстового редактора ***SciNotes***. Прежде чем изменить шрифт, следует удалить галочку рядом с элементом окна   
***Использовать системные шрифты***. Далее в ***Командном окне*** в списке ***Свой шрифт*** выбрать ***Console***, а в правой части окна нажать на клавишу с многоточием, где в открывшемся окне (рис. 1.1.3-3) произвести выбор типа, размера и стиля шрифта. Выход из текущего окна осуществляется нажатием клавиши **ОК**. Вернувшись в окно ***Настройки Scilab***надо завершить настройку последовательным нажатием кнопок **Apply**и **ОК**.

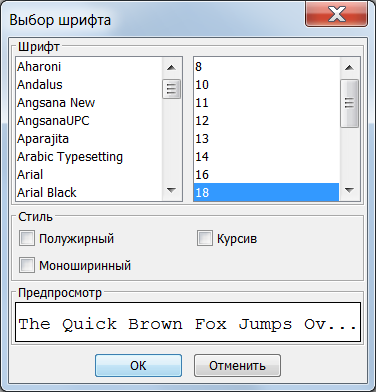


Рис. 1.1.3-3. Выбор типа, размера и стиля шрифта

Чтобы изменить цвет шрифта, в меню **Общее** следует выбрать элемент ***Цвета*** – откроется вкладка ***Цвет рабочего стола*** (рис. 1.1.3-4), где при настройке цвета прежде всего снять галочку, расположенную рядом с элементом *Использоватьсистемные цвета*.

Цвет также можно настроить для команд, текста, фона или курсора. Например, для изменения цвета фона, в поле команды  
***Использовать системные цвета*** надо сняв галочку, нажать на кнопку ***Фон***. В открывшейся цветовой палитре (рис. 1.1.3-5) выбрать цвет фона и для подтверждения выбора нажать **ОК**. Затем, вернувшись в окно ***Настройки***, последовательно нажать на кнопки **Apply**и **ОК**.

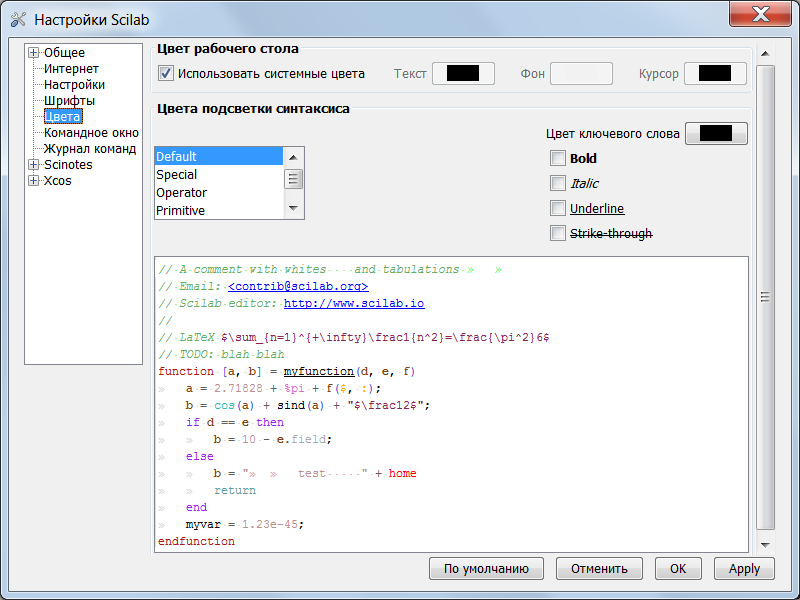


Рис. 1.1.3-4. Окно ***Настройка Scilab***(***Цвета)***

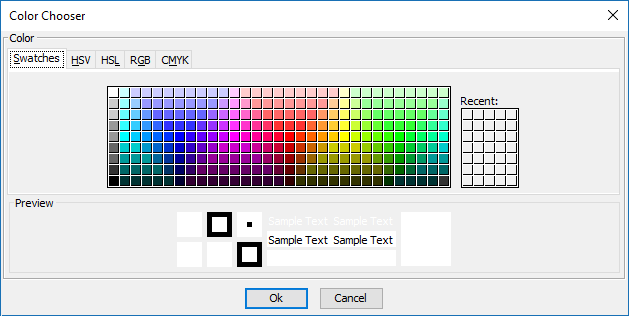


Рис. 1.1.3-5. Настройка цвета фона (окно ***ColorChooser***)

Элемент меню ***Управление*** содержит команды для управления программой в Scilab и содержит команды: ***Возобновить****, З****авершить****,* ***Приостановить***.

При активизации элемента меню ***Инструменты***, отображается набор команд, которые позволяют сделать активным инструменты, необходимые для текущей работы в Scilab. Они активизируют такие средстваScilab, как ***Текстовый редактор SciNotes****,* ***Визуальное моделирование****,****Преобразование файла из Matlab в Scilab****,* ***Управление модулями Atoms****,****Обозреватель переменных****,* ***Журнал команд*** и ***Обозреватель файлов****.*

Отметим, что окно редактора **SciNotes** можно открыть несколькими способами: либо выполнением соответствующей команды меню   
***Инструменты***, либо щелчком по первой кнопке панели инструментов , либо вводом в строке ***Командного окна*** команды: **--> SciNotes.**

Вызов редактора открывает новое окно с именем «**Безымянный  
документ 1**», в котором можно ввести необходимый программный код   
(рис. 1.1.3-6). По завершении работы с редактором **SciNotes** программный код можно сохранить в файле. Работа с текстовым редактором **SciNotes** будет подробно рассмотрена в п. 1.5.

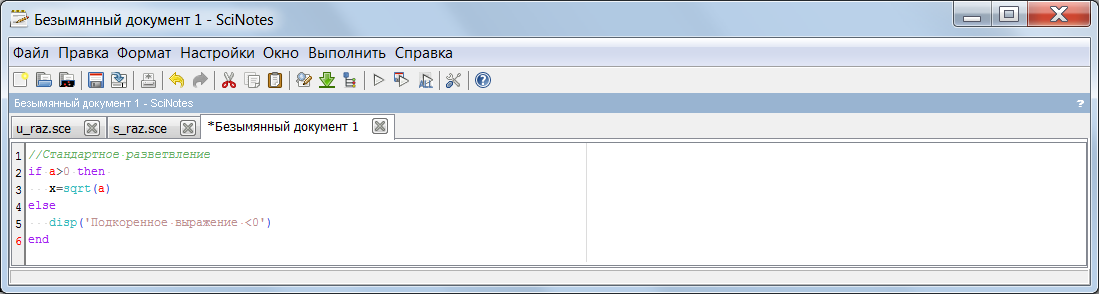


Рис. 1.1.3-6. Окно редактора **SciNotes**

Элемент меню ***Справка***, служит для ознакомления пользователя с возможностями пакета Scilab или получения текущей справки. Этот элемент содержит команды, позволяющие организовать доступ к **Содержаниюсправочной системы Scilab; к Примерам, иллюстрирующим возможности Scilab; к Веб-ресурсам; к ScilabEnterprises–справки о составе Scilab; к справке о версии Scilab.**

Наиболее простой способ получения справки о возможностях пакета Scilab –выполнение в ***Командном окне*** команды **help**, с последующим нажатием <Enter>: **--> help.**

Окно ***Справочная система*** Scilab показано на рис. 1.1.3-7.

Если требуется получить справку о конкретной функции (например, **optim**), то можно, пролистав содержание справки, найти раздел, посвященный оптимизации, где выбрать имя команды **optim**, после чего в правой части окна будет отображена информация по данной функции. Однако более удобным способом получения информации о конкретной функции (если вы знаете ее название) является использование команды **help** с указанием имени интересующей функции, например, **help optim.**

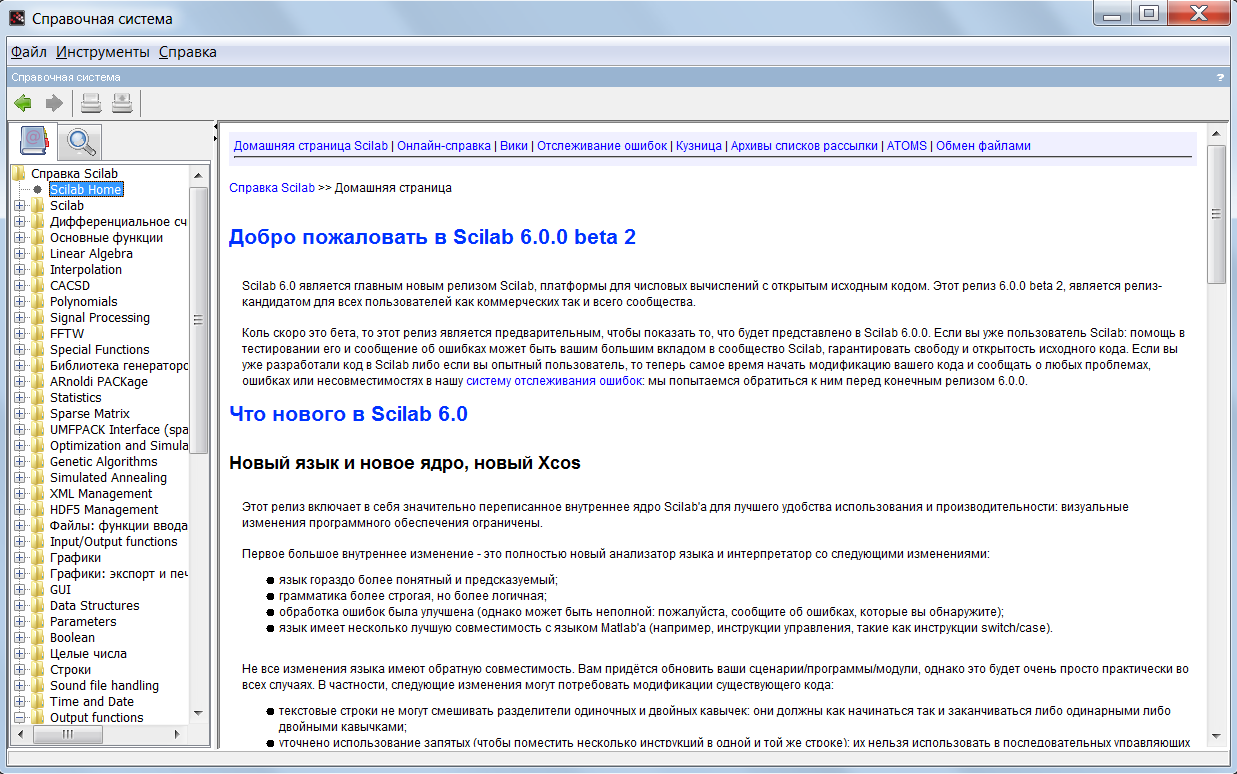


Рис. 1.1.3-7. Окно ***Справочная система***

В этом случае Scilab автоматически отобразит окно справки, содержащее информацию о выбранной функции (рис.1.1.3-8).

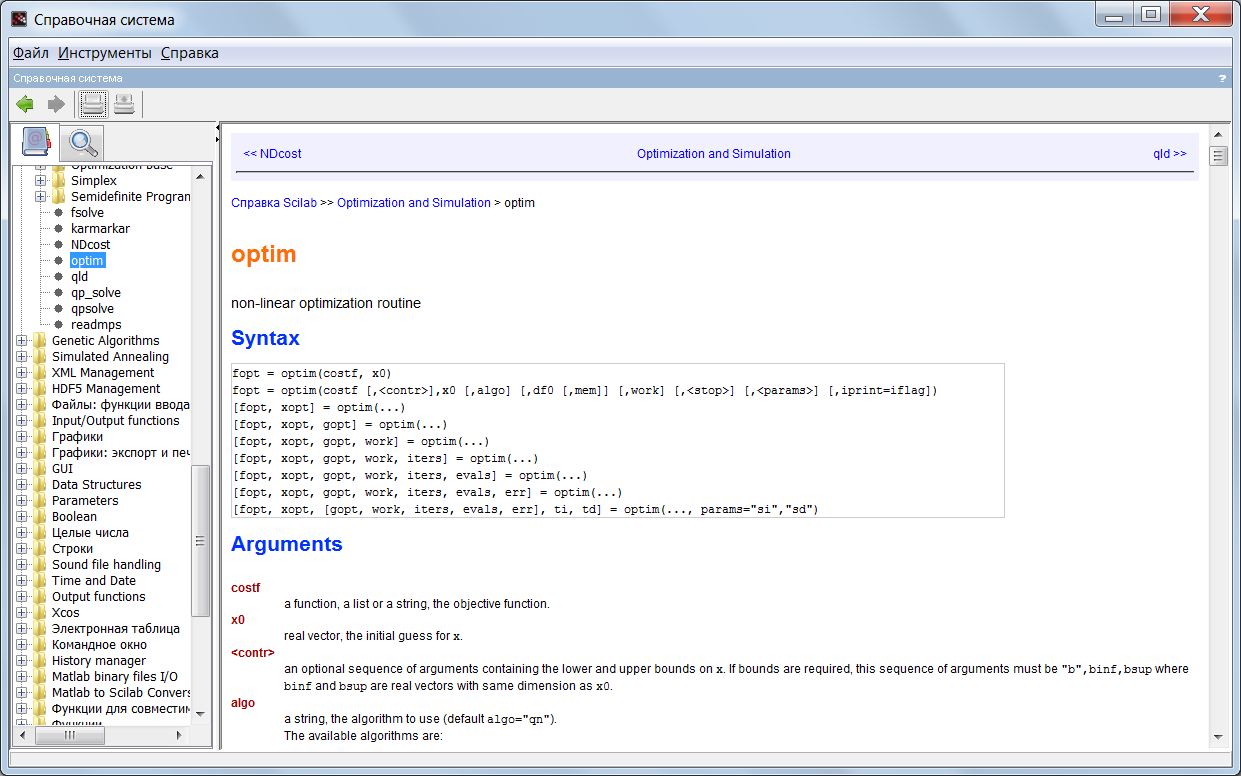


Рис. 1.1.3-8. Справка о выбранной команде ***optim***

* + 1. **Основные окна Рабочей среды**

К основным окнам **Рабочей среды** относятся: ***Командное окно***, окно ***Обозреватель переменных***, окно ***Журнал команд*** и окно ***Обозреватель файлов.***

***Командное окно*** используется для ввода команд с соответствующими аргументами и вывода результатов их выполнения. Работа в этом окне происходит в диалоговом режиме: пользователь вводит команду, которая передается ядру Scilabдля обработки и получения результата. Все команды вводятся в командную строку после появления ***приглашения* -->**. Заканчивается ввод каждой командной строки нажатием клавиши <Enter>.

Сеанс работы с пакетом Scilab в ***Командном окне*** принято называть **сессией**. Рассмотрим несколько простых примеров выполнения команд в   
***Командном окне***(рис. 1.1.4-1).

|  |  |
| --- | --- |
| *--> //* ***Вычисления в* Командном окне**  *-->*  *-->*2 + 3//***Пример1***  ans =  5.  *-->*b= sin(%pi) // **Пример2**  b =  1.225D-16  -->a = %e^2; // **Пример3**  -->p = 2 \* b / %pi... // **Пример4**  > + 4 / b  p =  3.266D+16 |  |

Рис. 1.1.4-1. Примеры простейших вычислений в ***Командном окне***

В **Примере1** (рис. 1.1.4-1) вычисляется результат выражения **2+3**. Scilab по умолчанию создаёт переменную с именем **ans**, и записывает в нее значение результата текущей операции, которое затем выводится в следующей строке.

В **Примере2** вычисляется значение выражения **sin(%pi)** и результат, присваивается переменной **b**, значение которой выводится в следующей строке.

В **Примере3** выражение заканчивает точка с запятой, которая «гасит» вывод результата, но он по-прежнему сохраняется, в этом случае в переменной с именем **a.**

Особенность**Примера4**состоит в том, что в случае ввода длинного выражения его часть переносится на следующую командную строку. В качестве символа переноса в командной строке Scilab используются три и более точки.

В двух примерах были использованы системные переменные **pi** и **e**. Чтобы указать, что эти переменные являются системными (т.е. они имеют значения по умолчанию), перед их именами введен символ **%**.

В результате выполнения рассмотренных выше примеров все атрибуты определенных переменных появились в окне ***Обозреватель переменных***(рис. 1.1.4-1 справа)и их можно использовать в дальнейших вычислениях.

Для очистки ***Командного окна*** служит команда **clc**, которая вводится в строке ***Командного окна***.

Окно ***Обозреватель переменных*** предназначено для просмотра атрибутов переменных, расположенных в ***Рабочей области*** (рис. 1.1.4-2). Под   
***Рабочей областью*** понимается область оперативной памяти компьютера, в которой хранятся данные текущей сессии. В окне ***Обозреватель переменных*** можно уви­деть графическое обозначение типа данных Scilab, имена переменных, их значения, типы и видимость.

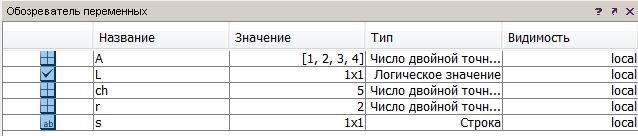


Рис. 1.1.4-2. Окно ***Обозреватель переменных***

При нажатии правой кнопкой мышки на любое место строки, в которой расположена нужная переменная (например, массив **А)**, в контекстном меню отображаются команды (рис. 1.1.4-3), которые можно с ней выполнить:   
**изменить значение; удалить; экспортировать в…; построить графическоеизображение матрицы**.

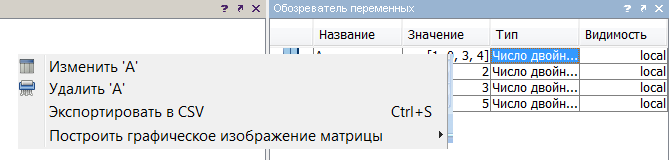


Рис. 1.1.4-3. Команды контекстного меню

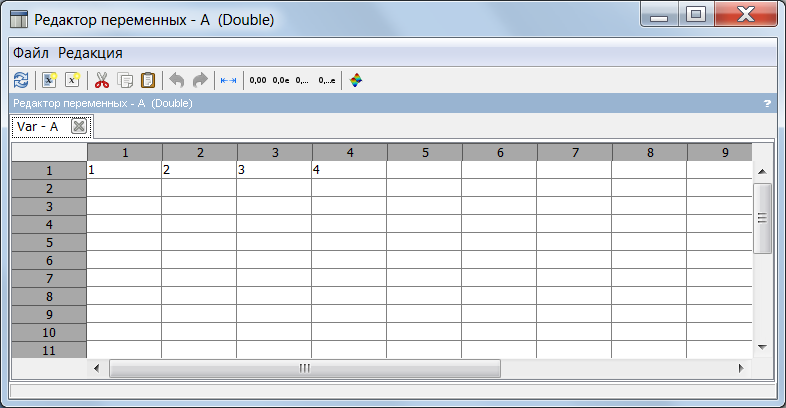


Рис.1.1.4-4. Окно ***Редактора данных***

При выборе команды **Изменить 'А'**, появляется ***Редактор переменных*** (рис.1.1.4-4), в котором можно изменить значения элементов вектора **А**.

Меню этого окна содержит два элемента: ***Файл*** и ***Редакция***. Команды, которые они содержат, приведены на рис.1.1.4-5. Наиболее часто используемые команды выведены на панель инструментов.

|  |  |
| --- | --- |
|  |  |

Рис.1.1.4-5. Команды элементов меню ***Редактора переменных***

Для очистки содержимого окна ***Обозреватель переменных*** предназначена команда **clear**, которая вводится в строке***Командного окна***.

|  |
| --- |
|  |

Рис. 1.1.4-6. Содержимое окна ***Журнал команд***

***Журнал команд*** (рис. 1.1.4-6) можно использовать при вводе команд, однотипных вычислений или просто повторений группы командных строк. Чтобы ввести в текущую строку содержимое ранее введенной командной строки, достаточно нажатием клавиш <**↑**> или <**↓**> подобрать нужную строку. Также перенести конкретную строку в командное окно можно двойным щелчком мыши по этой строке в окне ***Журнал команд*** или просто копированием и вставкой строки или части строки из окна ***Журнала команд*** в текущую строку ***Командного окна***.

Окно ***Обозреватель файлов*** предназначено для установки пути доступа к файлу, сохраненному в памяти компьютера или внешнего носителя, а также для вызова выбранного файла в редактор Scilab (рис. 1.1.4-7).



Рис. 1.1.4-7. Окно ***Обозреватель файлов***

Для установки текущего каталога (каталога в котором сохранен нужный файл), можно воспользоваться кнопкой , которая располагается в первой сроке окна ***Обозреватель файлов***. В результате щелчка по этой кнопке возникает окно ***Выберите папку*** (рис. 1.1.4-8), в котором нужная папка выбирается традиционным способом.

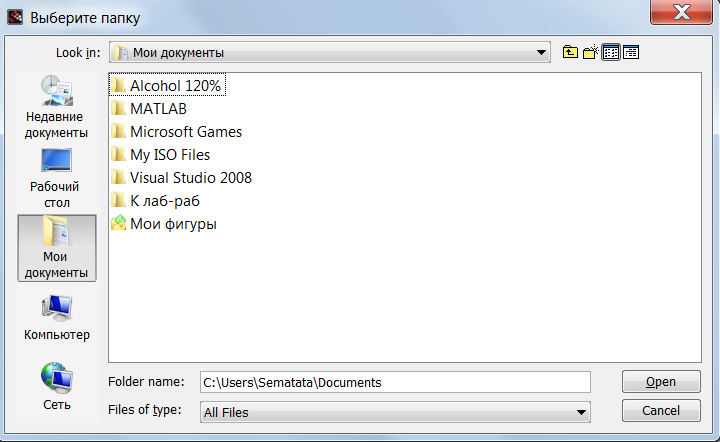


Рис.1.1.4-8. Окно ***Выберите папку***

Выбор папки завершается щелчком по кнопке ***Open* (*Открыть*)**, после чего содержимое окна ***Обозреватель файлов*** обновляется. Открытие файла из окна ***Обозреватель файлов*** производится двойным щелчком по его имени.

* + 1. **Контрольные вопросы**

1. Что является основными элементами ***Рабочей среды*** Scilab?
2. Каково назначение ***Командного окна***?
3. Каково назначение окна ***Журнал команд***?
4. Каково назначение окна ***Обозревателя переменных***?
5. Каково назначение окна ***Обозреватель файлов***?
6. Как установить текущий каталог в окне ***Обозреватель файлов***?
7. Как установить ***Конфигурацию рабочего окна***?
8. Как называется сеанс работы с пакетом Scilab в ***Командном окне***?
9. Каким образом перенести командную строку из окна ***Журнал команд*** в окно ***Командное окно***?
10. Что происходит, если, находясь в командной строке, нажать клавишу <**↑**> или <**↓**>?
11. Каким образом изменить значение переменной в окне ***Обозревателя переменных***?
12. Для чего предназначен редактор **SciNotes**?
13. Как выбрать тип, размер и стиль шрифта в ***Командном окне***?
14. Для чего используется ***Справочная система*** Scilab?
15. Как вызвать окно ***Справочной системы*** с использованием менюи осуществить поиск нужной команды?
16. Какой командой вызывается справка из командного окна?
    1. **Основные объекты системы Scilab**
       1. **Объекты Scilab и их типы**

К основным ***объектам (средствам языка программирования)***Scilab можно отнести:

* данные различного типа, представленные в виде констант, скалярных переменных, матричных переменных и соответствующих операций (арифметических, отношений, логических, строковых, полиноминальных и других).
* команды, операторы и функции;
* предопределенные библиотечные функции системы Scilab;
* функции пользователей (библиотечные функции и функции, встроенные в программный код);
* файлы различных типов данных;
* библиотеки функций;
* графические объекты;
* пакеты расширения.

Для лучшего понимания методологии системы Scilab прежде чем приступить к подробному описанию основных объектов системы Scilab, сделаем важные замечание.

В системе Scilab все данные являются ***матричными объектами***,элементы которых могут быть различными допустимыми типами, которые в основном предназначены для проведения математических расчетов и обработки данных. Структура объектов различного типа заранее предопределена внутри среды системы и скрыта от пользователей так же, как и работа с оперативной памятью при обработке объектов. В дальнейшем эти предопределенные объекты могут послужить для создания более сложных пользовательских объектов. Однако кроме матричных объектов в Scilab имеются и особые типы, такие как ***функции и библиотеки****,* которые предназначены для хранения данных, организации программного кода, а такжехранения данных и программ (функций) в файловой системе (библиотеках).

Условно иерархию типов объектов в Scilabможно представить схемой, показанной на рис. 1.2.1-1, причем каждый из представленных типов может быть матрицей**m×n (1×1, 1×n, m×1, m×n).**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | |  | | |  | | ***Матрицы (matrix)*** | | | | | | | | | ***Функции*** | | | ***Библиотеки*** | | |
|  | |  | | |  | |  | | | | |  | | | |  | | |  | | |
|  |  | | |  |  | |  | |  | | | |  |  |  |  | | |  | | |
| ***логические***  ***boolean*** | | ***числовые***  ***number*** | | | ***строковые***  ***string*** | | | |  | | | | | ***списки***  ***list*** | | | |  | | | |
|  | | | |  | | | | |  | | | | | | |  | | |  | | |
|  |  | | |  |  |  |  |  | | |  | | |  | | |  | |  | |  |
| ***вещественные***  ***double*** | | | ***int8*** | | ***uint8*** | |  | | | | ***t-списки***  ***tlist*** | | | | | ***m-списки***  ***mlist*** | | | | | |
| ***int16*** | | ***int16*** | |
|  |  | | ***int32*** | | ***int16*** | |  | | | |  | | |  | |  |  | |  |  | |
|  |  | | ***int64*** | | ***int32*** | |  | | |  |  | | |  | |  |  | |  |  | |
|  |  | |  | |  | | ***полиномы***  ***polinom*** | | | | ***Рациональные дроби***  ***rational*** | | | | | ***структуры***  ***struct*** | | | ***ячейки***  ***cell*** | | |
|  |  | |  | |  | |  | | | |  | | | | |  | | |  | | |

Рис. 1.2.1-1 Условная иерархия типов основных объектов Scilab

***Обратите внимание, что все скалярные числа в памяти Scilab сохраняются как матрица* 1×1*. Тот же принцип относится и к более сложным типам данных.***

Приведенная на рис.1.2.1-1 иерархия типов объектов позволяет сделать вывод, что, элементами матриц могут быть предопределенные объекты, принимающие действительные или комплексные числовые значения, строковые значения, логические значения, полиномиальные значения, списки, структуры и ячейки и другие.

Прежде чем перейти к подробному рассмотрению основных объектов Scilab целесообразно рассмотреть некоторые системные встроенные функции, такие как: **whos, who, who\_user, typeof, type**и**clear**. Описание этих функций приведены в ***Приложении 1.2, табл. 1.2.1-1****.*

Для работы с объектами в среде Scilab существует множество предопределенных переменных, которые всегда загружаются вместе со средой. Их можно посмотреть, выполнив команду **whos**. В результатечего в  
***Командном окне*** появляется так называемый *длинный список*, в котором перечислены все объекты, объявленные средой во время инициализации и которые были созданы в процессе текущей сессии. Из данного списка пользователь может получить информацию о типе объекта, его размерности и другую информацию. Именно поэтому данный список называется *длинным*. В этом списке присутствуют и объекты с типами, которые отображены на   
рис. 1.2.1-1.

Обратите внимание на функции **type** и **typeof** (***Приложение 1.2,  
табл. 1.2.1-2***), которые предназначены для получения информации о номере (все типы пронумерованы) и названии типов объектов.

***Обратите внимание, что создание любого программного кода, как правило, начинается с определения данных и способа их представления. Следовательно, необходимо понимать, как задать переменным необходимые типы и соответствующие значения***, ***и как их можно использовать.***

* + 1. **Числовые выражения, их вычисления  
       и создание переменных**

***Числовые выражения***

**Числовые (математические) выражения** в Scilab состоят из числовых типов данных (*целочисленных и вещественных с плавающей точкой*) и строятся из следующих объектов: *числовых констант, переменных, встроенных математических библиотечных функций* Scilab, функций пользователя*, арифметических операций над матрицами (алгебраических и поэлементных), круглых скобок и специальных операций.* Перечень операций Scilab, которые представляют собой различные символы или их комбинации, приведен в ***Приложении 1.2, табл. 1.2.2-1***. Несколько примеров их использования, а также соответствующее отражение в окне ***Обозреватель переменных*** информации о созданных с их помощью переменных, представлено на рис. 1.2.2-1.

|  |  |
| --- | --- |
| --> // ***Примеры использования специальных сим***волов  -->A = 1:5; // ***Двоеточие для формирования вектора***  -->x = 2; // ***Точка с запятой для запрета вывода результата***  -->t = '12345'; // ***Апостроф для создания строки символо****в*  -->d = 5; // ***Равно для присваивания значения переменной***  ***-->***k = 0.0018; // ***Десятичная точка для отделения***  --> // ***дробной части числа от целой*** |  |

Рис.1.2.2-1 Примеры использования некоторых специальных операций

К числовым типам в Scilab относятся *целые числа без знака,* целые *числа со знаком*, а также *числа с плавающей точкой двойной точности*. По умолчанию Scilab размещает в оперативной памяти компьютера все числовые значения как значения с плавающей точкой двойной точности – **double**, а формат результатов математических выражений в ***Командном окне*** зависит от установленного формата в **Рабочей среде** или установленного формата функцией **format**.

**Действительные числовые константы** – это предопределенные числа (числовые значения). Числа, например, **11, -7, 12.07** являются *безымянными числовыми константами.* Числа могут быть представлены в целом, вещественном или в экспоненциальном виде:

**0 5 -9 13.06 0.0025167.9D-7 0.99D+40.**

Кроме *действительных чисел* в выражения могут использоваться *комплексные числа*.

**Комплексные числа** – это числа, имеющие действительную и мнимую часть, причем в мнимой части имеется множитель **%i**, *означающий корень квадратный из* -**1 (√-1 = i)**. Например,

**3\*%i; 3+%i\*5; -%i\*8; 0.05\*%e-0.006\*%i**.

Комплексное число расширяет одномерную (линейную) область вещественных чисел в двумерную область, где комплексное число имеет на горизонтальной оси вещественную часть, а на вертикальной оси мнимую часть (рис. 1.2.2-2).

|  |  |
| --- | --- |
| [Graphical representation of complex numbers](https://x-engineer.org/wp-content/uploads/2016/08/complex_number_representation.jpg?11c992&11c992) | **Re** – вещественная ось;  **Im** – мнимая. |

Рис. 1.2.2-2 Графическое представление комплексного числа

Показанный способ задания комплексного числа не всегда удобен и требует ввода знаков **+, \*** и **%i**. Создание комплексного числа можно упростить, если воспользоваться функцией **complex(a,b)**, где в качестве параметров указываются, соответственно, действительная и мнимая части числа.

Для работы с комплексными объектами существует традиционный набор функций, который представлен в ***Приложении 1.2.2, табл. 1.2.2-2*.**

**Системные константы** – это константы, имеющие символические имена (поименованные константы) значения и смысл которых задаются системой при загрузке. Имена всех системных констант начинаются с символа **%** (кроме константы **ans**). Например, системная константа %**pi** представляет число **π=3.1415926…** Системные константы описаны в ***Приложении 1.2****,* ***табл.1.2.2-3****.*

**Переменные** – это объекты, имеющие имена (идентификаторы). Они способны хранить разные по значению и типу данные в оперативной памяти компьютера. В зависимости от типов данных переменные могут быть *числовыми*, *логическими, символьными*, *полиномиальными, структурами и ячейками и т.д*. Ихимена задаются по следующим правилам:

* имена переменных могут состоять из букв латинского и русского алфавитов (верхнего и нижнего регистра) и цифр;
* имена переменных не могут начинаться с цифры, но могут начинаться с символов: **%**, **\_** (подчеркивание), **#**, **!**, **$**, **?**;
* в именах переменных не должны входить пробелы;
* имена переменных не должны совпадать с именами других переменных или функций, то есть должно быть уникальными;
* для обозначения переменных желательно использовать содержательные имена.

Поскольку Scilab является *матричной системой,* все ее переменные являются матрицами. В Scilab принято, как и в математике, двумерный массив называть ***матрицей***, а одномерный – ***вектором***. Вектора, в свою очередь, могут быть ***вектор-строкой*** или ***вектор-столбцом***. Элементы векторов и матриц являются *индексированными переменными* (**п. 1.3.3**).

***Отметим, что индексы у векторов и матриц в Scilab имеют целочисленные значения, которые начинаются с* 1*. Даже скалярные объекты рассматриваются как матрицы размера* 1**×**1*.***

**Матрицы** характеризуются *размерностью, размером матрицы и размером каждого измерения*. Поэтому вектор, являясь одномерным массивом, имеет размерность **1**, а матрица – размерность **2**.

Для вектора *размер* – это число его элементов, а для матрицы *размер* определяется произведением числа ее строк **n** и столбцов **m** как **n×m**. Если число строк равно числу ее столбцов (**n==m**), матрица называется квадратной.

Для вектора, имеющего одно измерение, размер *измерения* совпадает с его *размером,* а для матрицы *размер первого измерения* – это количество элементов в столбце **n**, а второе – количество элементов в строке **m**.

Для работы с матрицами существует большое количество встроенных библиотечных функций, которые будут рассмотрены в дальнейшем, а с другими можно будет ознакомиться мере необходимости по документации Scilab.

В ***Приложении 1.2, табл. 1.2.2-4***приведены функции, используемые при обработке матриц, то есть функции, которые определяют размер матрицы, число измерений и размер каждого измерения.

Рассмотрение арифметических операций начнем с определения операции присваивания, которая является универсальной операцией для задания всех типов объектов Scilab и присваивании соответствующих значений.

**Операция присваивания** используется, как для объявления (задания типа переменной), так и задания (присваивания) значений переменной и обозначается символом «равно»:

*ИмяПеременной = Выражение*.

При выполнении операции присваивания происходит вычисление выражения, расположенного справа от знака присваивания, а затем результат сохраняется в переменной, имя которой указано слева от знака присваивания. При этом тип переменной присваивается типу результата, что сразу после выполнения оператора присваивания находит свое отражение в окне   
***Обозреватель переменных***.

***Обратите внимание, что в Scilab переменные и их типы заранее не декларируются (не объявляются). Тип переменной определяется типом результата вычисленного выражения, значение которого после вычисления присваивается переменной.***

***Следует отметить, что тип одной и той же переменной может меняться в процессе сессии, так как данные в Scilab не имеют строгой типизации. Например, если в переменной хранился текст, то на следующем шаге выполнения в нееможно записать число, а затем и логическое значение. Scilab следит за соответствием типов только при вычислении значений выражений.***

***Арифметические операции и функции***

**Алгебраическиеоперации (-, +, \*, /, \, ^, '**) в системе Scilab выполняются как над скалярными элементами, так и над векторами и матрицами. При этом операнды могут быть, как действительными, так и комплексными.

На рис. 1.2.2-2 приведены примеры использования алгебраических операций и функций над скалярами, векторами и матрицами.

|  |
| --- |
| --> // ***Алгебраические операции над векторами***  --> V1 = [1 4 6 8]; V2 = [2 3 4 7]; // ***Создание векторов* V1 *и* V2**  -->  --> V = V1 + V2 // ***Вычисление суммы векторов***  V =  3. 7. 10. 15.  -->  --> U = V1^2 // ***Возведение в квадрат вектора* V1**  U =  1. 16. 36. 64.  -->  --> // ***Алгебраические операции над матрицами***  --> mA = [1 2 3; -1 2 0; 3 2 4]  mA =  1. 2. 3.  -1. 2. 0.  3. 2. 4.  -->  -->mB = [1 2; -2 1; 0 1];  --> mC = mA \* mB // ***Алгебраическое произведения матриц***  mC =  -3. 7.  -5. 0.  -1. 12.  -->  --> mP = mA^2 // ***Возведение матрицы в квадрат***  mP =  8. 12. 15.  -3. 2. -3.  13. 18. 25.  --> d = det(mP) // ***Определитель квадратной матрицы***  d =  64. |

Рис.1.2.2-2 Примеры алгебраических операций и функций   
над векторами и матрицами

Список алгебраических операций над векторами и матрицами, а также функции **det**и **trace**, позволяющие вычислить определитель квадратной матрицы и след матрицы, приведены в ***Приложении 1.2, табл.1.2.2-5***,причем *совместимость* операндов в приведенных операциях определяется правилами соответствующих алгебраических операций. Матричные алгебраические операции следуют правилам *линейной алгебры*, а требуемый размер и форма операндов относительно друг друга, зависит от конкретной операции.

Рассмотрим пример решения матричного уравнения вида **А∙Х=В** или   
**Х∙А=В**, где **Х**– неизвестная матрица. Если умножить матричное уравнение на матрицу обратную к **А**, то оно примет вид: **А-1А∙Х=А-1В** или **Х∙А∙А-1=В∙А-1**. Так как **А-1А=А∙А-1=Е**, а **Е∙Х=Х∙Е=Х**, то неизвестную матрицу **Х** можно вычислить так: **Х=А-1В** или **Х=В∙А-1**. Понятно, что матричное уравнение имеет единственное решение если **А** и **В** – квадратные матрицы **n-**го порядка, а определитель матрицы **А** не равен нулю. Решив матричные уравнения **А∙Х=В** и   
**Х∙A=B**, выполним проверку. Решения матричных уравнений в Scilab показаны на рис. 1.2.2-3.

|  |
| --- |
| --> // ***Примеры решения матричных уравнений***  --> A = [3 2; 4 3];  --> B = [1 7; 3 5];  -->  --> // ***Решение матричного уравнения* A∙X=B**  --> X = A \ B // ***Первый способ***  X =  -3. 11.  5. -13.  -->  --> X = inv(A) \* B // ***Второй способ, использующий обратную матрицу***  X =  -3. 11.  5. -13.  -->  --> А \* X - B // ***Проверка***  ans =  0. 0.  0. 0.  -->  --> // ***Решение матричного уравнения* X∙A=B**  --> X = B /A // ***Первый способ***  X =  -25. 19.  -11. 9.  -->  -->X = B \* inv(A) // ***Второй способ, использующий обратную матрицу***  X ***=***  -25. 19.  -11. 9.  -->  -->X \* A – B // ***Проверка***  ans =  0. 0.  0. 0. |

Рис. 1.2.2-3 Решение матричных уравнений

**Поэлементные арифметические операции** в системе Scilab, операции *с точкой* **(-**, **+**, **.\***, **./**, **.\**, **.^**, **.'**) – это операции между *элементами массивов*. Они осуществляют вычисления, как над скалярными элементами, так и над элементами *одномерных* и *двумерных массивов*, при этом используются как действительные, так и комплексные операнды.

Операции *с точкой* между элементами *массивов* отличаются от алгебраических матричных операций концептуально, хотя их операнды в оперативной памяти компьютера структурно представлены одинаково. Поэтому для упрощения изложения, в дальнейшем будет использоваться термины вектор и матрица для любых операций в системе Scilab, а концептуальный (содержательный) смысл, использованных данных будет зависеть от решаемой задачи.

Отметим, что поэлементные операции, аргументами которых являются вектора и матрицы, выполняются поэлементно только над матрицами, имеющими одинаковые размерности и размеры по каждому измерению.

Примеры некоторых операций над скалярными данными приведены на рис. 1.2.2-4, а над элементами матриц на рис. 1.2.2-5.



**--> // *Арифметические операции над скалярными данными***

-->

--> a = 2.3; b = -14.654;

-->c = a .\* b// ***Формат вывода результата по умолчанию***

c =

-33.7042

-->r = a ./ b

r =

-0.1569537

-->

-->format(4)// ***Формат вывода результата с округлением***

-->c = a \* b

c =

-34.

-->r1 = a / b

r1 =

-0.2

-->

-->format(16) // ***Формат вывода результата с двойной точностью***

-->c = a .\* b

c =

-33.7042

-->r1 = a / b

r1 =

-0.1569537327692

Рис.1.2.2-4 Примеры арифметических поэлементных операций

над скалярами данными (матрицами размерностью **1х1**)

Список используемых арифметических операций над элементами матриц приведен в ***Приложении 1.2.2, табл.1.2.2-6***.



--> // ***Примеры арифметических поэлементных операций матрицами***

-->

--> // ***Арифметические поэлементные операции над векторами***

-->vX = [1 4 6]

vX =

1. 4. 6.

-->z = (-vX.^2 + vX ./ 4) .\ 4

z =

-5.333333333 -0.266666667 -0.115942029

-->vX1 = -vX, vX1 .\* vX

vX1 =

1. -4. -6.

ans =

-1. -16. -36.

-->

--> // ***Арифметические поэлементные операции над элементами матриц***

--> M1 = [1 2 3; 3 2 1];

--> M2 = 3 .\* M1

M2 =

3. 6. 9.

9. 6. 3.

--> M3 = M2 + M1.^2

M3 =

4. 10. 18.

18. 10. 4.

Рис.1.2.2-5 Примеры арифметических поэлементных операций

над векторами и матрицами

Если при выполнении поэлементных операций требования к размеру или размерности матрицы не выполняются, то матрицы можно преобразовать к необходимому размеру с помощью специальных функций. Например, функция **matrix** позволяет изменить размеры вектора или матрицы, а функция **resize\_matrix** выполняет обрезку или расширение матрицы. В Scilab имеются также функции, которые позволяют проверить число измерений (**ndims**) и размеры по каждому измерению аргументов (**size**). Описание перечисленных выше функций будет рассмотрено в **п. 1.3.2**.

**Поэлементные математические функции**, реализованные в Scilab, используются при проведении различного вида математических расчетов. Список наиболее часто используемых в математических расчетах функций, приведен в ***Приложении 1.2, табл. 1.2.2-7***, а примеры обращения к некоторым из библиотечных математических функций показаны на рис.1.2.2-6.

|  |
| --- |
| --> // ***Примеры использования встроенных математических функций Scilab***  -->x = 10;  -->d = sqrt(x) // ***Квадратный корень***  d =  3.16227766  -->  -->f = abs(-10.65) // ***Абсолютная величина числа*-10.65**  f =  10.65  -->  --> sin(x) // **Синус(10**)  ans =  -0.544021111  -->  --> factorial(3) // ***Факториал* 3**  ans =  6.  -->  --> hh = sin(10 + 10) .\* 10.65 ./ (1 – 3.16227766 \* %i)  hh =  0.8838969928355 + 2.7951277141849i |

Рис. 1.2.2-6. Примеры использования библиотечных функций Scilab

***Преобразование данных к арифметическому типу***

Поскольку во всех языках программирования, в том числе и в Scilab, основные арифметические операции должны иметь аргументы одного и того же типа, то необходимо осуществлять **преобразование аргументов к одному типу**. Например,в случае если аргументы арифметических операций разного арифметического типа, вещественные и целые. Кроме того, в Scilab могут использоваться типы данных, отличных от арифметических типов, например, *логические*, *строковые* и другие. Преобразования одних типов данных к другим допустимым типам числовых данных можно осуществить *явным образом* с помощью функций, список которых приведен в ***Приложении 1.2,табл. 1.2.2-8***.

Здесь необходимо сделать несколько замечаний по поводу преобразования типов в операциях с различными типами аргументов.

Когда бинарная арифметическая операция (например, умножение) применяется между двумя различными операндами различных, но совместных типов, обычно тип одного из двух *неявно преобразуется*, а другой операнд устанавливает свой тип результату.

Так, результат арифметических операций между операндами целочисленного и вещественного типа с плавающей точкой двойной точности всегда преобразуются к целочисленному типу данных, и, когда это необходимо, по умолчанию, округляется, в соответствии с алгоритмом округления. То есть, он округляется до ближайшего целого числа.

***Приоритет выполнения математических операций***

**Приоритет выполнения математических операций** в Scilab аналогичен приоритету, принятому в языках программирования высокого уровня: вычисление функции, возведение в степень, умножение и деление, сложение и вычитание. Для изменения приоритета в математических выражениях, как обычно, используются круглые скобки.

***Замечания по использованию данных типа double***

Как известно, система Scilab по умолчанию осуществляет все вычисления с максимальной возможной точностью – с вещественным типом данных с двойной точностью(**double**). Поэтому необходимо, во-первых, понимать, как представлены числовые данные этого типа в памяти компьютера, а во-вторых, особенности операций с ними.

В оперативной памяти компьютера Scilab представляет данные типа **double** в соответствии с принятыми стандартами. Любое числовое значение, хранящееся как **double**, занимает **64** бита (**8** байтов) и состоит из знака числа, мантиссы и порядка. При этом имеет допустимый диапазон отрицательных числовых значений от -**1.80D+308** до -**2.205D+308**, а положительных – от **2.23D-308** до **1.80D+308**.

Переменную типа **double** в Scilab можно создать с помощью простого оператора присваивания, например, **x=34.723**.

Хотя характеристики представления созданных значений из набора вещественных чисел с плавающей точкой двойной точности известны, однако их можно уточнить, воспользовавшись функцией **number\_properties,**которая возвращает значение параметра в зависимости от заданного запроса  
(рис. 1.2.2-7)



-->// ***Примеры использования функции* number\_properties**

-->

-->b = number\_properties("radix") // ***Основание***

b =

2.

-->

-->eps = number\_properties("eps") // ***Относительная максимальная погрешность***

eps =

1.110D-16

-->

-->ps = number\_properties("huge") // ***Максимальное допустимое положительно***е

-->// ***значение***

ps =

1.80D+308

-->

-->ps = number\_properties("tiny") // ***Минимально допустимое положительное***

-->// ***значение***

ps =

2.23D-308

Рис.1.2.2-7 Примеры использования функции **number\_properties**

В ***Приложении 1.2, табл. 1.2.2-8***представлены функции, связанные с представлением результатов вычислений или выводом значений в командное окно, а также с представлением исключительных ситуаций.

* + 1. **Логические выражения вScilab**

**Логические выражения** в Scilab строятся на основе *математических выражений*, *логических выражений, логических констант* (**%T**, **%F**), *операций отношения* и соответствующих им функций, *логических операций и функций*, а также *круглых скобок*. Результатом логического выражения является значение **T** (**True**), если выражение «Истинно», и значение **F** (**False**) в противном случае – выражение «Ложно».

**Операции отношения и соответствующие им функции** служат для поэлементного сравнения *скалярных величин, векторов* и *матриц*. Список операций отношения приведен в ***Приложении 1.2, табл. 1.2.3-1***.

***Операции отношения выполняют поэлементное сравнение значений векторов или матриц одинакового размера и возвращают значение равное* T*, если имеет место полное совпадение, и значение* F*в противном случае.***

***Операции* <, <=, >, >= *при комплексных операндах используются для сравнения только действительных частей операндов – мнимые отбрасываются. В то же время* операции == и ~= *ведут сравнения с учетом как действительной, так и мнимой части операндов.***

Логические выражения обычно используются в операторах **if, for, while, switch** и служат для изменения последовательности выполнения операторов программы Scilab, а также при *логической индексации матриц*.

При вычислении выражений операции отношения имеют *более низкий* приоритет, чем арифметические операции, но *более высокий*, чем логические.

На рис. 1.2.3-1 приведены примеры вычисления логических выражений.

|  |
| --- |
| --> // ***Примеры логических выражений***  -->  --> // ***Пример1. Логическое умножение вектор*ов**  --> vF = [%T %F %F]; vK = [%T %T %F];  --> vF & vK  ans =  T F F  -->  --> // ***Пример2. Логическое сложение мат****риц*  -->mA = [%F %T %F; %F %T %T;%T %T %F];  -->mB = [%T%T %F; %T %T %T;%T %T %F];  --> IL = mA | mB  IL =  T T F  TTT  TTF  -->  --> // ***Пример3. Умножение элементов вектора на константу***  -->vF = [%T %F %F];  -->vF .\* %F  ans =  0. 0. 0. |

Рис. 1.2.3-1 Примеры логических выражений

***Обратите внимание, что операнды операций отношения и логических операций должны иметь совместимые размеры.***

Список логических операций и функций приведен в ***Приложении 1.2, табл. 1.2.3-2*.**

* + 1. **Строки символов и форматирование данных**

***Строки символов***

Строковая константа представляется в Scilab заключением символов в одинарные или двойные кавычки, а скалярная переменная (матрица **1×1**) присвоением этой переменной значения символьной константы (рис.1.2.4-1).

|  |  |
| --- | --- |
| --> // ***Создание данных строкового типа***  -->  --> // ***Создание символьных пере*менных**  --> a = 'Система ', b = "Scilab"  a =  Система  b =  Scilab  -->  --> // ***Создание символьной матрицы***  --> mSt = ['Мы ' 'изучаем' 'Scilab' ' 6']  mSt =  !МыизучаемScilab6!  -->size(mSt)  ans =   1. 4.   --> length(mSt)  ans =  3. 7. 7. 2. |  |

Рис. 1.2.4-1 Создание данных строкового типа

Для создания матрицы строк используется тот же синтаксис, что и для матриц арифметического типа. В примере, приведенном на рис.1.2.4-1, создана матрица строк размером **1×4 'Мы изучаем Scilab6'**. Для того, чтобы вычислить ее размер, использована, как для обычных матриц, функция **size,** а для вычисления числа символов в каждом элементе матрицы – функция **length**.

В оперативной памяти строка представляет собой последовательность кодов символов, из которых она образуется. Однако, после создания строки, она представляется для пользователя единым целым, поэтому без специальных функций при работе со строками невозможно использовать отдельные символы строки.

Следует помнить, что строки поддерживают лишь операцию   
***конкатенации*** (объединения), которая обозначается символом плюс (**+**) (рис.1.2.4-2).

|  |  |
| --- | --- |
| --> // ***Объединение строк***  -->  -->a = 'Система'; b = "Scilab";  --> c = a +' ' + b  c =  Система Scilab |  |

Рис. 1.2.4-2. Выполнение над строками операции ***конкатенации***

В системе Scilab имеется множество встроенных функций обработки строковых типов данных. Наиболее востребованными функциями, которые используются при работе со строками, являются функции **string** и **strtod**. Функция **string** позволяет преобразовать свой входной параметр из любого типа данных в строку символов, а функция **strtod** позволяет преобразовать свою входную строку цифровых символов в число (рис. 1.2.4-3).

|  |  |
| --- | --- |
| --> // ***Использование строковых функций***  --> x = [1 2 3 4 5]; // ***Вектор* х**  -->  --> // **Пример1. Преобразование х в строку**  -->str = string(x)  str =  !1 2 3 4 5 !  -->  --> // ***Пример2. Определение типа* str**  -->typeof(str)  ans =  string  -->  --> // ***Пример3.* Str - *матрица разме*ром 1х5**  --> size(str)  ans =   1. 5.   -->  --> // ***Пример4. Преобразование* '10' *в число***  --> a = 2;  --> c1 = a + strtod('10')  c1 =  12. |  |

Рис. 1.2.4-3 Примеры использования строковых функций

В **Примере1** вектор чисел **х** преобразован в массив строк **str** с использованием функции **string**,где каждая строка массива состоит из одного символа. Затем, в **Примере2** с использованием функции **typeof** проведена проверка того, что переменная **str** действительно является строкой, а в **Примере3** использована функция **typeof**,выполнение которой показало, что переменная **str** является матрицей строк размером **1×5**. В **Примере4** показано преобразование строковой константы **'10**'в число **10**.

Рассмотрим еще одну функцию, которая используется при работе со строками. Функция **strcat** объединяет свой первый входной аргумент с разделителем, определённым во втором входном аргументе (рис. 1.2.4-4). В приведенном примере функция **strcat** используется для получения новой строки.

|  |  |
| --- | --- |
| --> // ***Использования функций* strcat**  -->  --> strcat(["1" "2" "3" "4" "5"]," + ")  ans =  1 + 2 + 3 + 4 + 5 |  |

Рис. 1.2.4-4 Пример использования функций **strcat**

***Форматирование строковых данных***

При выводе данных в удобном для пользователя формате в Scilab имеется несколько функций, позволяющих выполнить преобразование произвольных типов данных в текстовые данные и производить управление их форматом:

* [**mprintf**](https://help.scilab.org/docs/6.0.1/en_US/mprintf.html) – преобразует, форматирует и отображает данные в  
  командном окне;
* [**mfprintf**](https://help.scilab.org/docs/6.0.1/en_US/mfprintf.html) – преобразует, форматирует и записывает данные в файл;
* [**msprintf**](https://help.scilab.org/docs/6.0.1/en_US/msprintf.html) – преобразует, форматирует и записывает данные в строку.

Форматирование в этих функциях (выравнивание, число значащих цифр и так далее) осуществляется с использованием определенных элементов  
***нотаций*** (системой условных обозначений). Они аналогичны тем, которые используются для форматирования строковых данных в языке C++. Например, формат **%f** преобразует значения с плавающей точкой к соответствующему строковому формату, а формат **%.2f**, представляют две цифр после десятичного знака, формат **%12f** представляет при выводе**12**символов, заполняя по мере необходимости незначащие символы пробелами. Кроме того, имеется возможность комбинировать элементы формата с обычным текстом и специальными непечатными управляющими символов – ***escape***-символами, такими, например, как символ новой строки**\n**.

Строка, содержащая элементы форматирования, может иметь шесть элементов. Справа налево эти элементы представляют: *символ преобразования, подтип, точность, ширина поля, флаги* и *числовой идентификатор*.

В общем виде функции форматирования данных имеют следующий вид:

**mprintf**([*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)*,*[*С*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1-A1An)*писокПеременны*х);

**mfprintf**([*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)*,*[*С*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1-A1An)*писокПеременны*х);

**msprintf**([*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)*,*[*С*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1-A1An)*писокПеременных*)**.**

Работу перечисленных выше функций рассмотрим на примере функции **mprintf**:

*Строка***=mprintf**([*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)*,*[*С*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1-A1An)*писокПеременных*),

где *Строка*– строковая переменная; [*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)– параметр, который является строкой, содержащей символы преобразования (форматирования); [*С*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1-A1An)*писокПеременных–* список переменных, значения которых необходимо отобразить в соответствующем формате*.*

Каждый [*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)является строкой символов, который начинается с символа **%**, и сообщает всё о переменной, значение которой нужно подставить: её тип и модификаторы (ширина, точность, размер):

[*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)**='%***ФлагиШирина.ТочностьРазмерТип***'**.

Ни один из перечисленных элементов не является обязательным, кроме элемента Тип, а часть элементов формата относятся только к некоторым типам данных. Например, ширина, точность и размер – относятся только к числам.

***Обратите внимание на то, что в элементах формата не допускаются пробелы, причем символ преобразования является единственным обязательным полем, наряду с ведущим символом* %*.***

Рассмотрим основные допустимые символы преобразования строки[*Формат*](https://la.mathworks.com/help/matlab/ref/sprintf.html#btf_bfy-1_sep_shared-formatSpec)подробнее.

В соответствии с синтаксисом, формат преобразования начинается с символа **%** и содержит следующие необязательные и обязательные элементы:

* флаги (необязательно);
* поля ширины и точности (опционные);
* спецификатор подтипа (необязательно);
* символ типа (преобразования) (обязательный).

Эти элементы задаются в порядке, показанном в примере на рис. 1.2.4-5

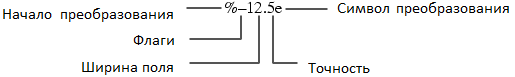


Рис. 1.2.4-5 Пример строки форматирования

***Символы преобразования***, указывающие нотацию (систему условных обозначений) для преобразования выходных данных, приведены в  
***Приложении 1.2, табл. 1.2.4-1****.*

***Ширина и точность поля*** позволяют управлять шириной и точностью выходных данных (***Приложение 1.2, табл. 1.2.4-2***).

|  |
| --- |
|  |

Например, если указать спецификатор формата **%5d**, то будет гарантировано, что вывод числа всегда занимает **5** символьных позиций (если нужно, то больше, но не меньше). Эта возможность очень полезна при печати таблиц, потому что и большие, и маленькие числа займут в строке одинаковое число позиций.

Следует иметь в виду, что результат вывода будет дополнен до необходимой указанной ширины слева. При этом большие числа, которые требуют для печати большее количество символов, будут выведены полностью.

Поле опции ширины должно быть указано таким, чтобы удовлетворять максимальному размеру ожидаемого выводимого числа. Например, если числа могут состоять из 1, 2 или максимум 3 цифр, то формат **%3d** подойдет. Опция ширины будет работать неправильно, если потребуется отобразить слишком большое число, которое не умещается в заданной ширине поля.

Числа с плавающей точкой (например, **3.1415**) содержат точку, тогда как числа целого типа (например, **27**) не имеют такой точки. Для печати чисел с плавающей точкой (**double**) флаги и правила работают точно так же, как и для целых чисел, но имеют несколько новых опций. Самая важная опция указывает, какое количество цифр может появиться после десятичной точки. Это количество цифр называется ***точностью*** *(precision)*числа.

В ***Приложении 1.2,*** *табл. 1.2.4-3* приведены примеры возможных вариантов вывода числа **e=2.718281828**. Почему красным?

Обратите внимание, что если в качестве элементов формата указаны точка и число, то ***precision*** указывает, сколько десятичных знаков должно появиться после десятичной точки.

Если точка для **%f** не указана, то по умолчанию будет приведен формат **%.6f** (**6** цифр после десятичной точки).

Если указано ***precision*** равное **0**, то десятичная точка также исчезает. Для того чтобы ее вернуть, нужно после спецификатора формата **%f** указать ее в виде простого текста.

Можно в одном спецификаторе формата указать одновременно и ширину (***width***), и точность (***precision***). Например, **5.2** означает общую длину **5**, с **2** цифрами после десятичной точки. Самая распространенная ошибка, когда считают, что это означает **5** цифр до точки и **2** цифры после точки.

|  |
| --- |
|  |

***Флаги позволяют*** управлять выравниванием выходных данных с помощью дополнительных ***флагов***.

|  |
| --- |
|  |

***Ширину и точность с флагами*** можно комбинировать, чтобы указать левое выравнивание, дополнение слева нулями и применение знака **+**, - и т. д.

До или после операторов форматирования, перед знаком процента **%** или после преобразования, можно включать дополнительный текст. Причем этот текст может быть, как обычнымтекстом(текст для печати), так и текстом, состоящим из специальных символов, который нельзя вывести как обычный текст.

В ***Приложении1.2,*** *табл.1.2.4-*4 показаны допустимые специальные непечатные символы.

|  |
| --- |
|  |

Рассмотрим несколько примеров форматирования, представленных на рис. 1.2.4-6.

|  |
| --- |
| -->// ***Примеры форматированного вывода***  -->  --> // ***Пример1. Использование сим*волов %s *для вывода текста***  --> t ='Текст';  --> mprintf('%s' ,t)  Текст  -->  *-->*// ***Пример2. Использование символов* %f *для вывода числа***  --> x = 234.2;  --> mprintf('%6.2f', x)  234.20  -->  --> // ***Пример3.Вывод элементов матрицы в различных форматах***  -->A = %pi \* 1000 \* ones(1,4) // **ones(1,4) – *создание матрицы из 1***  A =  3141.5927 3141.5927 3141.5927 3141.5927  -->  --> mprintf('%f\n%9.2f\n%+12.5f\n%12.2f\n' ,A)  ans =  !3141.592654 !  ! !  ! 3141.59 !  ! !  ! +3141.59265 !  ! !  ! 3141.59 ! |

Рис. 1.2.4-6 Примеры использования функции **mprintf**

В **Примере1** для вывода строки (текста) использованы символы **%s**. В   
**Примере2** для вывода значения числовой переменной использован символ **%f**. В **Примере3** создается вектор **А**, состоящий из четырех элементов, а затем каждый элемент вектора выводится в своем числовом формате **%f**. Если формат содержит знак плюс (**+**), число выводится со знаком плюс, если использован формат **%n.mf**, то число при выводе состоит из **n** знаков, из которых **m** – число знаков после запятой. Использование управляющих символов **\n** позволяет вывести каждый элемент вектора с новой строки.

**1.2.5. Списки, структуры и ячейки**

***Списки***

**Списки в Scilab** – это типы структур данных, состоящих из множества элементов, которые могут иметь разные типы. Они могут быть созданы вручную или с помощью одной из следующих функций:

* **list** – создает *простой список*, поля которого могут содержать произвольный тип данных;
* **tlist** – создать *типизированный список*;
* **mlist** – создает *матрично-ориентированный список*.

***Простой список*** – **list** обычно используется, когда данные в нем обезличены. В остальных случаях удобно использовать типизированный список  
(рис. 1.2.5-1).



--> // ***Пример создания простого списка и обращение к его элеме****нтам*

-->

-->lt = list(complex(12, -6), [2312], [12; 34]);

-->lt(1) // ***Обращение к 1-му элементу спис*ка**

12. - 6.i

-->

-->lt(2) // ***Обращение ко 2-му элементу спис*ка**

2312.

-->

-->lt(3) **//*Обращение к 3-му элементу списка***

12.

34.

-->

-->lt(1), lt(3) ($,$) // ***Обращение к 1-му, 3-му элементу списка и до конца***

ans =

12. +6.i

ans =

34.

Рис.1.2.5-1 Пример создания простого списка и обращение к его элементам

Чтобы обратится к элементу списка, достаточно указать его индекс списка, например, к первому, второму и третьему элементу, а затем индекс элемента.

Над простым списком определены операции:

* *Доступ к элементам списка* – **[***x,y,z,...***]=***ИмяСписка***(***Vi***)**,

где *Vi* – вектор индексов; **[***x,y,z,...***]=***L***(:)** извлекает все элементы.

* *Присвоение значения элементам списка* по индексу *iL***(***i***)=***a*.
* *Добавление элемента в конец списка L*($+1)=*e*.
* *Добавление элемента в начало списка L(0)=e*(после этого операция **e** является элементом списка с индексом **1**,  а исходные элементы списка смещаются вправо).
* *Исключение элементов из списка****L (i)=***[***null***](https://help.scilab.org/docs/6.0.1/en_US/null.html)***()*** удаляет ***i***-й элемент списка ***L***.
* *Объединение списковL3*=[lstcat](https://help.scilab.org/docs/6.0.1/en_US/lstcat.html)(*L1,L2*).
* *Определение числа элементов списка либо* *ne=*[*size*](https://help.scilab.org/docs/5.5.2/ru_RU/size.html)*(L)*, *либо* ***ne****=*[***length***](https://help.scilab.org/docs/5.5.2/ru_RU/length.html)*(***L***)*.
* *Итерации со списком* *forj=L,...,end*– цикл с числом итераций ***length(L)***, переменная цикла ***j*** будет равна *L(i)* на ***i***-той итерации.

Примеры некоторых операций над простыми списками и их элементами приведены на рис. 1.2.5-2.

|  |
| --- |
| -->// ***Примеры операции над простыми списками и их элемента****ми*  -->  --> L1 = list(5, ["М" "Т"]); // *Простой список с двумя элеме*нтам  --> size(L1) // Размер 2  ans =  2.  -->  --> // **L1(0) - *не существует! Выдается сообщение: Недопустимый индекс.***  --> L1(1) // ***Доступ к числу 5 с двойной точностью***  ans =  5.  -->  --> L1(2) // ***Доступ к вектору строк***  ans =  !М Т !  -->  --> size(L1(2)) // ***Размер 1, 2***  ans =   1. 2.   --> L1(0) ="фу" // ***Вставка в начало списка***  L1 =  L1(1)  фу  L1(2)  5.  L1(3)  !М Т !  -->  --> // **L1(0) - *не существует, "фу" записано в L1(1)***  --> L1($+1) = "Привет"; // ***Вставка элемента в конец сп***иска  --> L1(2) = "Окно";// ***Переопределение 2-го элемента списка***  --> L1(3) = null(); // ***Удаление 3-го элемента списка***  --> L2 = list("С1", "С2", "С3"); // ***Объявление нового списка***  --> F = lstcat(L2, L1); // ***Объединение двух спис***ков |

Рис. 1.2.5-2. Примеры операций над простыми списками и их элементами

***Типизированные списки*** используются главным образом для определения новых структур данных. Многие функции в Scilab перегружены с помощью типизированных списков. Также некоторые объекты Scilab на самом деле являются типизированными списками, и это скрыто от пользователя.

Отличительной особенностью типизированного списка от простого списка является то, что у него есть *поля*. Полем списка, по сути, является элемент списка, причем, каждому полю присваивается уникальное имя.

Для определения типизированного списка служит функция **tlist**:

**tlist** (*Тип,А1,...,An*),

где: *Тип –* символьная строка или вектор символьных строк; *А1,...,An –*любой фрагмент объекта (**matrix, list, string**). Символьная строка (*Тип)* позволяет пользователю определить новые операции, выполняемые над этими объектами. Типы списков **tlist** и **list** имеют значения типа, соответственно, **16** и **15**.

Стандартные операции с **tlist** соответствуют операциям **list**.

Конкретными реализациями списка **tlist**являются, например,   
*Линейные системы*, *Рациональные матрицы*, *Полиномиальные матрицы* и т.д., которые будут подробно изложены в **п. 2.1**.

|  |
| --- |
| --> // ***Создание и обращение к полям типизированного списка***  -->  --> TL = tlist(["Nlict", "Year", "Month"], [], [])  TL =  TL(1)  !Nlict Year Month !  TL(2)  []  TL(3)  []  -->  --> TL.Year(1) = 2018;  --> TL.Month(1) = "Октябрь";  TL =  TL(1)  !Nlict Year Month !  TL(2)  2018.  TL(3)  Октябрь  -->  -->TL.Month  ans =  Октябрь  -->  -->TL(3)  ans =  Октябрь  -->  -->typeof(TL)  ans =  Nlict |

Рис. 1.2.5-3 Создание типизированного списка

Для работы с типизированными списками использоваться следующие функции:

* **tlist** – создание типизированногосписка;
* **fieldnames** –вывод всех полей типизованного списка;
* **definedfields** –вывод полей, в которых есть данные;
* **setfield** –установка поля для типизованного списка;
* **getfield** –извлечениеданных из поля.

Для примера создадим типизированный список, который будет хранить информацию о некотором двигателе. Пусть у этого списка будет три поля: тип двигателя, маркировка, мощность и размерность мощности (рис.1.2.5-4).



--> // ***Обращения к полям типизированного списка***

-->

--> tt = tlist(['Engine', 'type' ,'power', 'dim'], 'П61' ,5, 'кВт')

tt =

tt(1)

!Enginetype power dim !

tt(2)

П61

tt(3)

5.

tt(4)

кВт

-->

-->tt.type

ans =

П61

Pис.1.2.5-4 Примеры обращения к полям типизированного списка

Типизированный список наследует все приемы работы, присущие обычным **list**-спискам. Он отличается от обычного списка еще и тем, что позволяют создавать пользовательские типы данных. Хотя к элементам типизированного списка можно обращаться по индексам, надо помнить, что это следует делать только на уровне разработчика, а при непосредственном использовании пользователь все же должен использовать имена полей. Поля в типизированном списке при желании можно добавлять.

Для операций доступа, извлечения из списка и присваивания значения элементу в списке имеются две соответствующие функции **getfield** и **setfield**. Они могут использоваться во всех типах списков.

Кроме списков **list** и **tlist** в Scilab есть еще одна разновидность типизированного списка – **mlist**. При обычной работе с этими списками пользователь не почувствует между ними разницы. Разница начинает проявляться при программировании пользовательских структур данных.

Типизированные списки являются надстройками над списками типа **list** и позволяют создавать пользовательские типы данных с уникальным именем в виде строки. К элементам типизированного списка можно обращаться по их уникальным именам.

***Объекты типа структур и ячеек***

В Scilab реализовано два особых типа структур данных, которые созданы скорее для переносимости данных из одной среды в другую.

Первый особый тип данных это **struct** (структура), который был введен для переносимости сценариев между Scilab и Matlab. Внешне структура **struct** очень похожа на типизированный список. В отличие от типизированного списка, который хранит имя в себе, именем структуры является переменная, в которую эту структуру записывают. В остальном структура похожа на обычный список, в котором есть имена полей и присвоенные им значения   
(рис. 1.2.5-7).



*--> //* ***Создание структуры и доступа к ее элементу***

-->

--> s = struct("firstname", "Виктор", "Age", 23)

s =

firstname:[1x1 string]

Age: [1x1 constant]

Рис. 1.2.5-7 Пример создания структуры и доступа к ее элементу

При объявлении структуры на нечетных позициях в аргументах всегда стоят имена полей, а на четных – присваиваемые значения. В отличие от списков, у структур сразу удобный для чтения формат вывода.

Неудобства структуры проявляют себя при программировании. В структуре очень просто создаются новые поля (рис 1.2.5-8), однако это заставляет постоянно держать в памяти их имена.



--> // ***Пример добавления нового поля в структуру***

-->

--> s.address = 'Авиамоторная 8а'

s =

firstname: [1x1 string]

Age: [1x1 constant]

address: [1x1 string]

Pис.1.2.5-8 Пример добавления нового поля в структуру

В структурах нельзя обращаться к полям по индексам. Поэтому, например, если структура хранит в себе данные работников, а работников несколько, то единственный способ их как-то объединить – это записать их в вектор. В Scilab такую операцию удобнее выполнять с помощью **mlist**-списков.

Другой тип структурных данных это **cell** (ячейка). Этот тип данных также частично совместим с Matlab. Эта структура сочетает в себе особенности списка и матрицы. Напомним, что в матрицах можно хранить данные только одного типа данных, но в **cell** это ограничение не действует. Таким образом, **cell** – это матрица, которая способна хранить данные разных типов.

Чтобы создать **cell**-матрицу, например, (**2х3**) нужно использовать одноименную функцию (рис.1.2.5-9).



--> // ***Пример создание и заполнение позиций cell-массива***

--> // ***и обращение к ее элемен*там**

-->

--> c = cell(2,3)

c =

[0x0 constant] [0x0 constant] [0x0 constant]

[0x0 constant] [0x0 constant] [0x0 constant]

-->

--> c{1,1} = 11; c{1,2} = 12; c{1,3} = 13; c{2,1} = 21;

-->

--> c{:,:}

ans =

ans(1)

11.

ans(2)

21.

ans(3)

12.

ans(4)

[]

ans(5)

13.

ans(6)

[]

Рис.1.2.5-9 Пример создания и заполнения позиций **cell**-матрицы,  
и обращение к ее элементам

Индексы в **cell** сами по себе используются исключительно для извлечения ее элементов по правилам матриц.

Таким образом, **cell** удобно использовать, когда необходимо получить объект, обладающий свойствами матриц и **list**-списков. Если же предполагаются более сложные действия над собственными объектами, но не предполагается переносимость, лучше пользоваться типизированными списками Scilab.

Функция **cell,** возвращающая массив ячеек пустых матриц, может иметь следующие форматы:

*с=***cell()**

*с=***cell(***m1***)**

*с=***cell(***m1,m2***)**

*с=***cell(***m1,m2,...,mn***)**

*с=***cell(***х***),**

где *: х*-вектор, содержащий размеры ячейки для создания;

*m1,m2,...,mn*- размеры создаваемой ячейки.

Рассмотрим назначение каждого из перечисленных выше форматов:

**сell(0,0)***–* возвращает **(0,0**) массив ячеек пустых матриц;

**cell*(*m1*)*** *–*возвращает (**m1,m1**) массив ячеек пустых матриц;

**cell*(*m1,m2*)****–*возвращает **(m1,m2)** массив ячеек пустых матриц;

**cell*(*x)***–*возвращает массив ячеек пустых матриц где: первый;

размер ячейки массива **x(1)**, второе измерение **x(2)**;

**cell()** *–*эквивалентно **cell(0)**.

Если **A** является массивом ячеек, то можно получить доступ к содержимому элемента **A**с помощью записи **A{m1,m2,...,mn}**, где номера ячеек заключены в фигурные скобки. Таким образом, выражение **A(1,1) =**[**zeros**](https://help.scilab.org/docs/6.0.1/en_US/zeros.html)**(2,2**)  является недопустимым, а правильный синтаксисвыражения -**A{1,1}=**[**zeros**](https://help.scilab.org/docs/6.0.1/en_US/zeros.html)**(2,2)**.

***Объекты типа полином и рациональное число***

Из алгебры известно, что полиномом называется алгебраическое уравнение вида **Pn(x)=a0+a1x+a2x2+…+anxn**.

В Scilab присутствуют объекты, которые могут быть созданы только с помощью функций, а их внутренняя организация скрыта от пользователя под оболочкой. К таким объектам относится и полином.

В среде Scilab {\displaystyle a\_{n}x^{n}+a\_{n-1}x^{n-1}+...+a\_{1}x+a\_{0}=0,a\_{n}\neq 0,n\geqslant 1.}полином определяется как объект **polynomial**, с которым работают специальные функции, например, функция нахождения корней полинома, а сам полином можно определить с помощью функции **poly**.

С помощью функции **horner**можно вычислить значение полинома от любого значения аргумента. Для этого функции необходимо передать два аргумента: имя полинома и массив значений аргументов.

Еще одним объектом, который является производным от объекта **polinomial**, является **rational**. Объект **rational** (рациональное число) образуется делением одного полинома на другой, при этом этот объект имеет свой внутренний идентификатор и инициируется именно как **rational**. Объект **rational** полностью наследует все функции полинома

Если углубиться в детали, то можно увидеть, что и **rational**, и **polynomial** являются ничем иным, как типизированными списками. Объявить объект **rational** можно также через специальную функцию **rlist**.

Подробно создание и работа с полиномиальными типами данных будут рассмотрены в п. 2.1.

* + 1. **Определение и использование  
       внутренних встроенных функций**

***Функции, используемые в Scilab***

**Функция** – это объект, имеющий уникальное имя, предназначенный для многократного проведения некоторых функциональных вычислений или программной реализации некоторых часто используемых программных фрагментов, и возвращающий, при необходимости, полученные результаты через значения ее выходных параметров.

С точки зрения программирования, функции в системе Scilab могут играть роль как собственно *функций,* так и*подпрограмм-функций*.

Функции в Scilab могут быть встроенными в систему (так называемые *библиотечные функции*, которые являются неотъемлемой частью Scilab), *функциями пакетов расширения* и *функциями пользователя*, создаваемые конкретным пользователем для решения своих задач.

***Функции пользователя*** в свою очередь могут быть, как *встраиваемые* в *программный код (****Командного окна*** *или сценария)*, так и *загружаемые из текстовых файлов системы* или из библиотек пользователя, где находятся в откомпилированном виде.

Система Scilab содержит огромное число библиотечных функций ядра системы, которые при описании в справочной системе Scilab разбиты по категориям: *Операторы и элементарные операции, Типы данных, Линейная алгебра, Элементарная математика, Численное интегрирование и Дифференциальные уравнения, Интерполирование, Оптимизация, Графика* и многие другие.

В таблицах ***Приложений***, соответствующих **п. 1.2.2 - 1.2.5**,описаны и проиллюстрированы примерами системные библиотечные функции.

Самые простые функции пользователя, встроенные в программный код ***Командного окна***, можно создать двумя способами, используя оператор **deff** или оператор **function**. Независимо от способа создания функции, следует помнить, что сначала функцию следует описать с формальными параметрами, а затем обратиться к ней с фактическими входными параметрам.

***Следует помнить, что все входные параметры функций (фактические параметры) перед обращением к ней должны быть обязательно определены (заданы конкретные значения).***

***deff – Описание и вызов однострочной функции***

В соответствии с синтаксисом Scilab описание функции с использованием **deff**должно располагаться в одной строке командного кода, поэтому назовем ее *однострочной* функцией.Начинается описание с ключевого слова **deff**и имеет следующий формат:

**deff('[***y****1****,…,y****n*]=***ИмяФункции*(*x****1,****…,x****k***)**',**'*y****1*=***Выражение****1****,*…,*y****n*=***Выражение****n*')**,

где: *x****1****,…,x****k*** – *СписокВходныхФормальныхПараметров*;   
*y****1****,…,y****n*** – *СписокВыходныхФормальныхПараметров* (для одного выходного параметра квадратные скобки необязательны);

*Выражение****1****,…,Выражение****n*** – список выражений, по которым определяются выходные параметры функции.

***Если описание функции не умещается в одной строке экрана, то ее можно перенести, используя символы переноса – три точка (…).***

Параметры передаются системе в виде строк, которые затем разбиваются на лексемы и из которых Scilab строит однострочную функцию.

***При описании функции* deff *в выражениях для вычисления выходных параметров могут использоваться не только входные параметры функции, но и выходные параметры, вычисленные в предыдущих выражениях этой функции, а также переменные, определенные в* Командном окне*.***

Обращение к функции, описанной с помощью **deff**, производиться из программного кода:

**[***z****1****,…,z****n*]=***ИмяФункции*(*p****1****,…,p****k***)

**[***z****1****,…,z****n*]*=*** *ИмяФункции(p****1****,*…,*p****k***);,

где: *p****1,****…,p****k*** – *СписокВходныхФактическихПараметров* функции;   
*z****1****,…,z****n***– *СписокВыходныхФактическихПараметров* (для одного   
выходного параметра квадратные скобки необязательны).

В первом случае на экран выводятся значения всех выходных параметров (причем вывод начинается с последнего), а во втором – вывод значений параметров гасится точкой с запятой, но в переменных *z****1****,…,z****n*** сохраняются все вычисленные значения выходных параметров.

Если вызов функции осуществляется без указания имен выходных параметров (*ИмяФункции*(*p****1****,…,p****k*))**, то результат отображается в переменной **ans**, значение которой равно значению последнего выходного параметра. Именно поэтому такое обращение к функции рекомендуется применять к функции только с одним выходным параметром. Обращение по имени функции можно использовать не только при выводе результата, но в записи арифметических выражений внутри его описания (рис.1.2.6.1).

|  |
| --- |
| --> // ***Использование* deff *для описания функции***  --> // ***с одним выходным параметр*ом**  -->  --> deff('y = f1(x1, x2)','y = sin(x1) + a \* x2');  -->  --> a = 2; //***Определение переменной* a*, используемой в выражении функции***  --> r = f1(2, 3)//***Обращение к функции* f1**  r =  6.9092974  -->  --> f1(2, 3) // ***Обращение к функции* f1**  ans =  6.9092974  -->  --> d = 1 + f1(2,3) // ***или*d=1+ans**  d =  7.9092974 |
| --> // ***Использовани****е* **deff *для описания функции***  --> // ***с двумя выходными параметрами***  *-->*  --> deff('[y1, y2] = ff1(x1, x2)', 'y1 = sin(x1) + x2; y2 = cos(x2) + y1');  -->  --> [z1, z2] = ff1(2, 3) // ***Обращение к функции* f2 *в командной строке***  z2 =  2.9193049  z1 =  3.9092974  -->  --> z = z1 + z2 // ***Использование результатов выполнения функции***  z =  6.8286024  -->  --> ff1(2, 3) // ***В* ans *сохраняется значение первого выходного*** *параметр****а***  ans =  3.9092974 |

Рис. 1.2.6-1 Использование **deff** для описания однострочной функции

пользователя в ***Командном окне***

***function – Описание и вызов многострочной функции***

Второй способ описания функций в ***Командном окне*** начинается со слова **function**, а заканчивается словом **endfunction** (или **end**). В этом случае описание функции имеет следующий формат:

**Function** [*y****1****,…,y****n***]=*ИмяФункции*(*x****1****,…,x****k***)

*ТелоФункции*

***…***

*y1*=*Выражение1;*

*...*

*yn*=*Выражениеn;*

***…***

**endfunction**

где: *x****1****,…,x****k*** – *СписокВходныхФормальныхПараметров* функции (если функция не имеет входных параметров, тогда после имени функции указываются пустые скобки);

*y****1****,…,y****n*** – *СписокВыходныхФормальныхПараметров*(для одного выходного параметра квадратные скобки необязательны);

*ТелоФункции* – команды, операторы и библиотечные функции, необходимые для вычисления выходных параметров. Если вычисленные результаты должны быть в ходе выполнения функции (или ее отладки) отображены в   
***Командном окне***, то символ точка с запятой в конце операторов отсутствует, иначе вывод результата гасится символом точки с запятой. Ввод каждой строки *ТелаФункции* осуществляется последовательно и завершается нажатием клавиши <Enter>, а ввод всех строк *ТелаФункции* считается незавершенным до тех пор, пока не будет введено ключевое слово **endfunction** (или **end**)**.**

*ИмяФункции* формируется аналогично имени переменной и должно быть в данном сеансе работы уникально. Если имя функции уже было использовано, то выводится сообщение:«**Предупреждени**е: переопределение функции:… » с указанием имени переопределенной функции.

Поскольку в ***Командном окне*** все строки выполняются последовательно, то очевидно, что описание функции должно предшествовать первому обращению к этой функции.

Правила обращения к функциям, описанных с помощью **deff** и **function**, полностью идентичны (Рис.1.2.6-2).

|  |
| --- |
| --> // ***Описание* function *для функции с одним выходным параметром***  --> function y =f2(x1, x2)  > y = sin(x1) + a \* x2;  > end  -->  --> a = 2; // ***Определение переменной* a**, ***используемой в выражении функции***  --> r = f2(2, 3) // ***Обращение к функции*f2*в Командной строке***  r =  6.9092974  -->  --> f2(2, 3) //***Обращение к функции* f2**  ans =  6.9092974  -->  --> d = 1 + f2(2, 3) **// *или*d=1+ans**  d =  7.9092974  --> |
| --> // ***Описание* function для *функции* ff2 *с двумя выходными параметрами***  --> function [y1, y2] = ff2(x1, x2)  > y1 = sin(x1) + x2;  > y2 = cos(x2) + y1;  >end  -->  --> [z1, z2] = ff2(2, 3) // ***Обращение к функции* ff2**  z2 =  2.9193049  z1 =  3.9092974  --> z = z1 + z2 // ***Использование результатов выполнения функции***  z =  6.8286024  -->  --> ff2(2, 3)// ***В* ans *сохраняется значение только первого выходного параметра***  ans =  3.9092974 |

Рис. 1.2.6-2 Использование **function** в ***Командном окне***

Как видно из приведенных примеров (рис.1.2.6-1 и 1.2.6-2), использование для описания в **deff** и **function** позволило получить одинаковые результаты.

В каких же случаях для описания функции пользователя целесообразно использовать **deff**, а в каких **function**?

Описание функции с помощью **deff** используется, как правило, если функция имеет один или два выходных параметра, а выражения для их вычислений достаточно короткие. В противном случае описание функции либо полностью не видно в ***Командном окне***, либо строка должна быть перенесена с использованием символов три точки (**…**), что затрудняет чтение и контроль за правильностью ввода. Громоздкие выражения принято вычислять по частям, но тогдадля вычисления значения одного выходного параметра требуется не один, а несколько операторов, что не соответствует формату описания функции с помощью **deff**. Описание функции с помощью **function** в данном случае предпочтительнее, поскольку оно может занимать несколько строк.

Наряду с рассмотренными выше способами создания *внутренних встраиваемых функций пользователя* в ***Командном окне***, в Scilab имеется возможность создания функций с использованием текстового редактора SciNotes, который позволяет сохранить содержимое его окна в виде **sce-**файла. В этом случае сохраненный файл можно многократно подключать и использовать в других сеансах работы. Подробнее материал о создании и работе со **sce-**файлами, а также о средствах программирования в среде Scilab, будет изложен в **п. 1.5**.

Рассмотрим несколько простейших задач по ТЭЦ.

Заданы: катушка с резистивным сопротивлением **R=10 Ом**, индуктивностью **L=0.05 Гн** подключена к источнику ЭДС **E=120 В** с частотой **f=50 Гц**. Определить комплексный ток **I**в данной последовательной **RL**-цепи (рис. 1.2.6-3).

|  |  |
| --- | --- |
| // Определение комплексного тока в  // последовательной RL-цепи  // Исходные данные  R=10;// Сопротивление в Ом  L=0.05; // Индуктивность в Гн  E=120;// ЭДС в В  F=50;// Частота, Гц  W=2\*%pi\*f;// Угловая частота в рад/с  XL=w\*L;// Индуктивное сопротивление в Ом  Z=R+%i\*XL;// Комплексная сопротивление в Ом  I=E./Z; // ТоК в RL=цепи в А  Imod=abs(I);// Модуль тока в А  // Фазатокавградус  arg\_I=atan(imag(I),real(I))\*180/%pi;  mprintf(  I=3.4608053 - 5.4362202i  Imod=6.4443513  arg\_I=-57.518363 |  |

Рис. 1.2.6-3. Определение комплексного токапоследовательнойRL-цепи

Резистивное сопротивление **R=10 Ом** и катушка с индуктивностью **L=0,05 Гн** параллельно подключены к источнику ЭДС **E=120 В** с частотой f**=50 Гц**. Рис. 1.2.6-3. Определение комплексного токав RL-цепи (рис. 1.2.6-4).

|  |  |
| --- | --- |
| // Определение комплексного тока в  // параллельнойRL-цепи  // Исходные данные  R=10; // Сопротивление, Ом  L=0.05; // Индуктивность, Гн  E=120; // ЭДС, В  f=50; // Частота, Гц  w=2\*%pi\*f; // Угловая частота, рад/с  XL=w\*L;// Индуктивное сопротивление, Ом  BL=1/XL; // Проводимость катушки, См  G=1/R; // Проводимость резистора, См  Y=G-%i\*BL; // Общая комплексная проводимость, См  I=E\*Y//То в RL=цепи, А  Imod=abs(I)// Модультока, А  // Фазатока, градус  arg\_I=atan(imag(I),real(I))\*180/%pi  mprintf(  I =12. -7.6394373i  Imod = 14.225365  arg\_I=-32.481637 |  |

Рис. 1.2.6-4. Определение комплексного токав параллельнойRL-цепи

* + 1. **Контрольные вопросы**

1. Какие объекты Scilab Вам известны?
2. Как называются символы **., : ; () [] {} // % $ ' " ~=** и в каково их назначение в Scilab?
3. Что означает в Scilab «определить данные»?
4. Какой символ служит для переноса части командной строки на следующую строку?
5. Для чего используются системные встроенные функции: **whos, who, who\_user, typeof, type**и **clear**?
6. Как в Scilab представлены комплексные числа?
7. Какие системные константы Вам известны?
8. Из каких элементов строятся числовые арифметические выражения?
9. Каков приоритет действий в арифметических выражениях?
10. Какой символ используется для запрета вывода результата вычислений на экран?
11. Какие типы переменных используются в Scilab?
12. Чем характеризуются матрицы?
13. Как создать вектор-строку?
14. Как создать вектор-столбец?
15. Какая функция служит для определения длины вектора?
16. Каким образом создать вектор с постоянным шагом?
17. Требуется ли при работе с векторами и матрицами предварительное объявление их размера?
18. Какой символ используются для разделения элементов матрицы в строке, а какой для разделения ее строк?
19. Какие команды предназначены для заполнения матрицы случайными числами, распределенными по равномерному или нормальному закону распределения?
20. Формат команд выбора минимального и максимального значения элемента матрицы.
21. Как записать выражение для решения матричных уравнений?
22. Какие поэлементные матричные операции Вам известны?
23. Как происходит преобразование данных к арифметическому типу?
24. Из каких элементов состоят логические выражения?
25. Как представлены в Scilab строки символов?
26. Что такое форматированный вывод данных?
27. Для чего используется функция **mprintf** и каков ее формат?
28. Какие символы преобразования вам известны?
29. Что такое списки, структуры и ячейки в Scilab?
30. Какие типы функций пользователя используются в Scilab?
31. Как описываются и вызываются функции, описанные с использованием операторов **deff**и**function**?

## **Матричные операции создания, доступа, извлечения и модификации. Векторизация и индексирование**

* + 1. **Представление данных в Scilab**

Из предыдущих разделов известно, что математический пакет Scilab построен как программная система, ориентированная на работу с матрицами. Двумерная структура данных способна хранить данные различного типа. Это могут быть числа, символы, логические значения, полиномы, рациональные дроби и другие объекты. Scilab размещает матрицы всех типов данных в   
**Рабочей области** – оперативной памяти компьютера, по столбцам и отображает информацию об активных объектах текущей сессии в окне   
***Обозревателя переменных***.

Из рис. 1.3.1-1, иллюстрирующем матричную структуру представления данных в Scilab, следует, что матрицы могут быть пустыми **[]**; *скалярными*; *векторными*, состоящими из одной строки или одного столбца, называемыми соответственно *вектор-строкой* или *вектор-столбцом*; *квадратными* и *прямоугольными*, а их элементы могут быть объектами практически любыхтипов данных, допустимыхвсистеме *Scilab*.

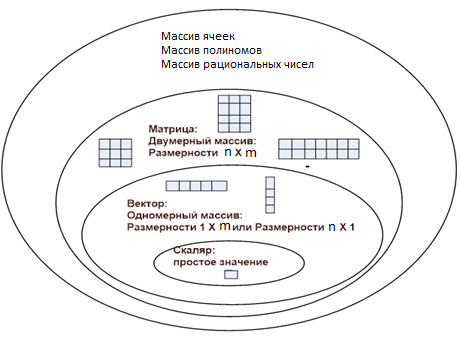


Рис. 1.3.1-1 Представление данных в Scilab

При работе с матрицами в матрично-ориентированных программных системах необходимо понимать, как можно ***создавать*** различные типы матриц и осуществлять ***доступ*** к ее элементам, и знать правила ***матричных операций и функций***.

* + 1. **Создание векторов и матриц**

***Способы создания векторов и матриц***

Под созданием матриц понимают создание матриц определенного размера, структуры и содержания (присвоение ее элементам соответствующих значений заданного типа, в том числе и пустых матриц).

***Напомним еще раз, что прежде чем выполнять операции с матрицами и их элементами, необходимо создать и присвоить элементам матрицы значения допустимого типа.***

В Scilab реализовано несколько различных способов создания векторов и матриц:

* *с помощью операции квадратные скобки* (*операция конструирования -* **[]***)*;
* *с помощью соответствующих встроенных функций*;
* *путем объединения уже существующих матриц*;
* *с помощью операции двоеточие* (**:**).

***Создание векторов и матриц с помощью операции –* []**

*При создании векторов с помощью операции конструирования*, квадратные скобки используют совместно с операцией присваивания (=). При этом значения элементов векторов, заключенные в **[]**, отделяются друг от друга символами:

* ***точкой с запятой***, если нужно создать вектор-столбец;
* ***пробелом*** или ***запятой***, если нужно создать вектор-строку.

На рис. 1.3.2-1. приведено несколько примеров создания векторов с помощью операции **[]**.

|  |
| --- |
| --> // ***Примеры создания векторов с помощью операции* []**  -->  --> // ***Пример1. Создание вектор-строки* av *из 4-х элементов***  --> av = [2 4 -7 5]  av =  2. 4. -7. 5.  -->  --> // ***Пример2. Создание вектор-столбца* bv *из 3-х элементов***  --> bv = [0; 3 ;5]  bv =  0.  3.  5.  -->  --> // ***Пример3. Создание вектор-столбца* сv *из 4-х элементов***  --> // ***и вектор-строки* tv *из 4-х элементов***  --> cv = [4; 6; -3; 2]; length(cv)  ans =  4.  --> tc = cv'  tc =  4. 6. -3. 2. |

Рис. 1.3.2-1. Примеры создания векторов с помощью операции **[]**

В **Примере3** для определения длины вектора использовалась функция **length**, а для транспонирования матрицы – апостроф (').

*При создании матриц с помощью операции* **[]** точно также, как и при создании векторов, необходимо сначала ввести имя матрицы и знак присваивания, а затем в квадратных скобках значения их элементов. При этом значения элементов строк записываются через *пробел* или *запятую*, а строки матрицы разделяются *точкой с запятой*, например, **А=[v1;v2;v3],**  
где **v1, v2, v3** – вектора одинакового размера (рис.1.3.2-2). Если строки матрицы слишком длинные (**Пример3**), то ввод очередной строки матрицы можно перенести в следующую строку экрана, нажав после символа точки с запятой клавишу <Enter>.

В **Примере1** создается матрица **mA1(3,4),** а в **Примере2** пустая матрица **0×0**. Матрица называется пустой, если она имеет число элементов равное нулю, хотя бы в одном измерении. В **Примере3** показан ввод элементов матрицы в две строки.

|  |  |
| --- | --- |
| --> // ***Примеры создания матриц –*[]**  -->  --> // ***Пример1.Матрица 3х4***  --> mA1 = [1 2 3 4;5 6 7 8; 9 10 11 12]  mA1 =  1. 2. 3. 4.  5. 6. 7. 8.  9. 10. 11. 12.  -->  --> // ***Пример2. Матрица 0×0***  --> mA2 = []  mA2 =  []  -->  --> // ***Пример3. Ввод в две строки***  -->B=[1.4444 2.78888 3.565555 -4.36789;  > 6.4444 2.55555 6.32322 -4.8796]  B =  1.4444 2.78888 3.565555 -4.36789  6.4444 2.55555 6.32322 -4.8796 |  |

Рис. 1.3.2-2. Примеры создания матриц с помощью операции **[]**

***Создание матриц с помощью встроенных функций***

Для создания матриц определенного размера, структуры и содержания в Scilabможно использовать встроенные функции [**ones**](https://www.mathworks.com/help/matlab/ref/ones.html), **zeros**, **eye**, [**rand**](https://www.mathworks.com/help/matlab/ref/rand.html), **repmat** и некоторые другие, описанные в ***Приложении 1.3, табл.1.3.2-1***.

Перечисленные выше функции имеют следующие назначения: [**ones**](https://www.mathworks.com/help/matlab/ref/ones.html)–создает ***единичную*** матрицу; **zeros** –имеет те же форматы, что и функция [**ones**](https://www.mathworks.com/help/matlab/ref/ones.html), но создает матрицу, состоящую из нулей; **eye** –создает матрицу ***с единицами*** на главной диагонали; [**rand**](https://www.mathworks.com/help/matlab/ref/rand.html)–создает матрицу чисел двойной точности ***со случайными вещественными*** или ***комплексными элементами***;**repmat** –создает большую матрицу, состоящую из копий другой матрицы, заданной параметром.

Все перечисленные выше функции (кроме функции **repmat**) в зависимости от входных параметров, создают матрицы в соответствии с заданной размерностью (или того же размера), что и уже существующая матрица, указанная в параметрах.

Рассмотрим несколько примеров по созданию матриц с использованием описанных функций (рис. 1.3.2-3).

|  |  |
| --- | --- |
| -->// ***Создание матриц с помощью функций***  -->  -->A = ones(2, 2) // ***Матрицы, состоящей из 1***  A =  1. 1.  1. 1.  -->  -->B = zeros(2, 2)// ***Матрицы, состоящей из 0***  B =  0. 0.  0. 0.  -->  -->C = eye(A) // ***Матрицы в которой элементы***  -->// ***главной диагонали равны 1***  C =  1. 0.  0. 1.  -->  -->r = rand(A) // ***Матрицы из случайных чисел***  r =  0.7560439 0.3303271   * 1. 53811   -->  -->D = repmat(1:3,2,2)  D =  1. 2. 3. 1. 2. 3.  1. 2. 3. 1. 2. 3.  -->  --> f = testmatrix('frk',3) // ***Матрица*Franck**  f =  3. 2. 1.  2. 2. 1.  0. 1. 1. |  |

Рис. 1.3.2-3. Созданиеспециальных матрицы с помощью встроенных функций

***Создание матриц путем объединения существующих***

Новую матрицу можно создать ипутем объединения уже существующих матриц с помощью операции **[]** и соответствующих функций. В этом случае ***операция*** **[]** используется не только как ***конструктор*** матриц, но и как ***операция объединения***. Например, выражение **c=[mA,mB]** предполагает горизонтальное сцепление матриц **mA**и **mB,** а **mC=[mA;mB]** их вертикальное объединение (рис. 1.3.2-4).

|  |
| --- |
| --> // ***Примеры матриц, созданных путем объединения***  -->  --> // ***Пример1***  --> ma = ones(2, 4) \* 3;  --> mb = rand(3, 4);  -->mc = [ma; mb]  mc =  3. 3. 3. 3.  3. 3. 3. 3.  0.2113249 0.3303271 0.8497452 0.068374  0.7560439 0.6653811 0.685731 0.5608486  0.0002211 0.6283918 0.8782165 0.6623569  -->  --> // ***Пример2***  -->md = [5.36, 7.01, [], 9.44]  md =  5.36 7.01 9.44 // ***Пустая матрица не влияет на формирования*md** |

Рис. 1.2.3-4 Примеры создания новых матриц путем   
объединения уже существующих

В **Примере1** создается новая матрица**mc** из матриц **ma** и **mb** в вертикальном направлении, а в **Примере2** создается новая матрица**md** из матриц **ma** и **mb** в горизонтальном направлении.

Кроме операции объединения в Scilab имеется встроенная функция [**ca**t](https://www.mathworks.com/help/matlab/ref/cat.html), которая также позволяет объединить существующие матрицы для формирования новой матрицы. Функция **cat** возвращает матрицу, которая образуется путем сцепления двух или нескольких матриц, перечисленных в параметрах. Эта функция являются ***альтернативой операции* []*для объединения матриц***. С помощью этой функции можно построить новые матрицы вдоль заданного измерения (рис. 1.3.2-5).

|  |  |
| --- | --- |
| -->// ***Сцепление по строкам и столбцам***  -->  -->A = [1 2 3; 6 7 8]; B = [0 8 7; 3 2 1];  -->C = cat(1, A, B) // ***Сцепление по строкам***  C =  1. 2. 3.  6. 7. 8.  0. 8. 7.  3. 2. 1.  --> // ***Сцепление по столбцам***  --> C = cat(2, A, B)  C =  1. 2. 3. 0. 8. 7.  6. 7. 8. 3. 2. 1. |  |

Рис. 1.2.3-5 Сцепление матриц по строкам и столбцам

***Если матрицы объединяются по горизонтали, то они должны иметь одинаковое количество строк. Если матрицы объединяются по вертикали, то матрицы должны иметь одинаковое количество столбцов.***

***Создание матриц с помощью операции двоеточие –* :**

Создание векторов и матриц, элементы которых представляют собой некоторую последовательность значений, можно осуществить с помощью операции ***двоеточие*** *–*(**:**).

Операция двоеточия позволяет генерировать последовательность значений вектора или строки (столбца) матрицы от **n** до **m** (**n:m**). Причем в этой последовательности каждое последующее значение отличается от предыдущего на **1**, даже если конечное значение не является точным значением   
(**Примеры1-6**, рис. 1.3.2-6).

|  |  |
| --- | --- |
| --> // ***Создание векторов и матриц с помощью –* :**  -->  --> // ***Пример1***  --> vA1 = 10:15  vA1 =  10. 11. 12. 13. 14. 15.  -->  --> // ***Пример2***  -->vA2 = -2.5:2.5  vA2 =  -2.5 -1.5 -0.5 0.5 1.5 2.5  -->  --> // ***Пример3***  -->vA3 = 1:6.3  vA3 =   1. 2. 3. 4. 5. 6.   -->  -->//***Пример4***  --> vA4=10:5:50  vA4 =  10. 15. 20. 25. 30. 35. 40. 45. 50.  -->  **--> //*Пример5***  -->vA5 = 3:0.2:3.8  vA5 =  3. 3.2 3.4 3.6 3.8  -->  --> // ***Пример6***  -->vA = 9:-1:1  vA6 =  9. 8. 7. 6. 5. 4. 3. 2. 1.  -->  -->//П***ример7***  -->mA=[10:14;-2:2.5;1:5.3;5:-1:1]  mA =  10. 11. 12. 13. 14.  -2. -1. 0. 1. 2.  1. 2. 3. 4. 5.  5. 4. 3. 2. 1. |  |

Рис. 1.3.2-6 Создание векторов и матриц с помощью операции двоеточие

Если шаг изменения последовательности значений отличается от 1, то необходимо указать значение шага **h** (**n:h:m**). Причем шаг может быть, как положительным, так и отрицательным. В **Примере7** показано создание матрицы, элементы каждой строки которой генерируются в различных диапазонах и с разными шагами. Используя такой способ задания матриц, следует помнить, что в каждой строке матрицы должно быть одинаковое количество элементов. В случае ошибки система выводит сообщение:   
«**несовпадающие размеры строки/столбца**».

***Динамическое изменение размера матрицы***

Матрицы могут динамически изменять свой размер в процессе выполнения про­граммы с помощью функций **matrix** и **resize\_matrix**. Это позволяет адаптировать размер уже созданной матрицы к тем данным, которые ей предстоит хранить (рис. 1.3.2-7).



--> // ***Динамическое изменение размеров матриц***

-->

--> А = [1 2 3; 4 5 6] // ***Исходная мат***р***ица***

А =

1. 2. 3.

4. 5. 6.

--> А(3, 1) = 7 // ***Изменение размера матрицы путем дополнения элемента***

А =

1. 2. 3.

4. 5. 6.

7. 0. 0.

-->

--> А(:, 3) = [] // ***Удаление 3-го столбца***

А =

1. 2.

4. 5.

7. 0.

-->

-->// ***Создание вектора заданного размера из матрицы* А**

--> В = matrix(А, 1, 6)

В =

1. 4. 7. 2. 5. 0.

-->

--> // ***Создание матрицы (3х4) по экспоненциальному закону распределения***

--> M = grand(3, 4, 'exp', 5) // ***с математическим ожиданием 5***

M =

3.1472369 0.7660456 0.472206 0.4688152

7.350552 4.6290769 4.0357003 3.3184658

0.4401028 13.20044 9.4667509 0.5462453

-->

--> M1 = resize\_matrix(M, 2, 2)// ***Изменение размера матрицы М(3х4) на М1(2х2)***

M1 =

3.1472369 0.7660456

7.350552 4.6290769

Рис. 1.3.2-7 Примеры динамического изменения размеров матриц

На рис. 1.3.2-7 создана матрица **А(2,3)**. Далее в эту матрицу добавлен элемент с индексами **(3,1)**. Добавление элемента вызывает в мат­рице **А** создание третьей строки, где первому элементу в ней будет присвоено значение **7**, а значения остальных элементов данной строки будут установлены в **0**. Таким образом, размер матрицы может увеличиваться динамически.

Далее приведен пример динамического уменьшения размера матрицы, где для удаления третьего столбца матрицы **А**, используется обозначение пустой матрицы **[]**.

Полностью изменить размер матрицы можно с использованием функции **matrix**. Эта функция трансформирует исходную матрицу в матрицу другого размера, копируя элементы столбец за столбцом. В следующем примере матрица **А** раз­мером **3х2=6** преобразуется в вектор-строку с **6** элементами.

В последнем примере (рис. 1.3.2-7) с использованием функции**grand** создана матрица **М** размером **(3х4)**, элементы которой распределены по экспоненциальному закону распределения с математическим ожиданием **5**. Размер матрицы изменен с использованием функции **resize\_matrix**. Таким образом матрица **М(3х4)** стала матрицей **М1(2х2)**.

Функция **size** позволяет проверить размер матрицы и возвращает два параметра **n** и **m**, значения которых равны, соответственно, числу строк и столбцов в данной матрице (рис. 1.3.2-8).



--> // ***Динамическое изменение размеров матриц***

-->

--> А = [1 2 3; 4 5 6] // ***Исходная мат***р***ица***

А =

1. 2. 3.

4. 5. 6.

--> А(3, 1) = 7 // ***Изменение размера матрицы путем дополнения элемента***

А =

1. 2. 3.

4. 5. 6.

7. 0. 0.

-->

--> А(:, 3) = [] // ***Удаление 3-го столбца***

А =

1. 2.

4. 5.

7. 0.

-->

-->// ***Создание вектора заданного размера из матрицы* А**

--> В = matrix(А, 1, 6)

В =

1. 4. 7. 2. 5. 0.

-->

--> // ***Создание матрицы (3х4) по экспоненциальному закону распределения***

--> M = grand(3, 4, 'exp', 5) // ***с математическим ожиданием 5***

M =

3.1472369 0.7660456 0.472206 0.4688152

7.350552 4.6290769 4.0357003 3.3184658

0.4401028 13.20044 9.4667509 0.5462453

-->

--> M1 = resize\_matrix(M, 2, 2)// ***Изменение размера матрицы М(3х4) на М1(2х2)***

M1 =

3.1472369 0.7660456

7.350552 4.6290769

Рис. 1.3.2-7 Примеры динамического изменения размеров матриц



--> // ***Динамическое изменение размеров матриц***

-->

--> А = [1 2 3; 4 5 6] // ***Исходная мат***р***ица***

А =

1. 2. 3.

4. 5. 6.

--> А(3, 1) = 7 // ***Изменение размера матрицы путем дополнения элемента***

А =

1. 2. 3.

4. 5. 6.

7. 0. 0.

-->

--> А(:, 3) = [] // ***Удаление 3-го столбца***

А =

1. 2.

4. 5.

7. 0.

-->

-->// ***Создание вектора заданного размера из матрицы* А**

--> В = matrix(А, 1, 6)

В =

1. 4. 7. 2. 5. 0.

-->

--> // ***Создание матрицы (3х4) по экспоненциальному закону распределения***

--> M = grand(3, 4, 'exp', 5) // ***с математическим ожиданием 5***

M =

3.1472369 0.7660456 0.472206 0.4688152

7.350552 4.6290769 4.0357003 3.3184658

0.4401028 13.20044 9.4667509 0.5462453

-->

--> M1 = resize\_matrix(M, 2, 2)// ***Изменение размера матрицы М(3х4) на М1(2х2)***

M1 =

3.1472369 0.7660456

7.350552 4.6290769

Рис. 1.3.2-7 Примеры динамического изменения размеров матриц

Кроме формата **size(А)**, функция **size** может имеет и другой формат:

*nm***=size(***А***,1),**

*nm***=size(***А***,2),**

*nm***=size(***А***,"\*").**

Возвращаемое значение в этом случае определяется вторым параметром функции:

* при  **l** возвращается число строк;
* при  **2**  возвращается число столбцов;
* при **"\*"** возвращается общее число элементов в матрице, равное

числу строк, умноженному на число столбцов.

***Создание массивов ячеек***

**Создать массив ячеек** можно с помощью операции фигурные скобки – **{}** или функции **cell**.

При наличии данных для ввода в массив ячеек, можно создать массив, используя операцию конструирования массива ячеек **{}** (рис. 1.3.2-8).



-->// ***Создание массива ячеек***

-->

-->// ***Пример1***

-->myCell = {1,2,3; 'текст', rand(5,10), {11; 22; 33}}

myCell =

[1x1 constant] [ 1x1 constant] [1x1 constant]

[1x1 string ] [5x10 constant] [3x1 cell ]

-->

-->// ***Пример2***

--> C = {}

C =

{}

-->

-->// ***Пример3***

-->emptyC = cell(3, 4)

emptyC =

[0x0 constant] [0x0 constant] [0x0 constant] [0x0 constant]

[0x0 constant] [0x0 constant] [0x0 constant] [0x0 constant]

[0x0 constant] [0x0 constant] [0x0 constant][0x0 constant]

Рис. 1.3.2-8. Создание массива ячеек с использованием **{}** и функции **cell**

Как и все массивы Scilab, массивы ячеек – прямоугольные, с одинаковым количеством ячеек в каждой строке. Матрица **myCell** (**Пример1**) представляет собой массив ячеек размером **2×3**.

*Операцию конструирования ячеек* можно использовать для создания пустого массива ячеек **{}** – **Пример2**.

Чтобы добавить значения в массив ячеек, вначале нужно создать пустой массив. Для этого в **Примере3** используется функция **cell(3х4)**. В результате чего массив**emptyC** представляет собой массив ячеек размером *3×4*, где каждая ячейка содержит пустой массив**[]** – **Прим*ер3***.

* + 1. **Индексирование и векторизация**

***Понятия индексирования и векторизации***

**Индексирование матриц** является средством **доступа** к подмножеству элементов матриц, которое позволяет ***извлекать*** из матриц подмножества их элементов, ***присваивать извлеченное*** из матриц подмножество новым матрицам и ***модифицировать подмножество элементов*** существующих матриц. В Scilab имеется несколько способов индексирования, которые влияют не только на скорость выполнения программного кода, но и на его читаемость. Таким образом, индексирование является ключевым моментом эффективности использования средств пакета Scilab при реализации матричных операций. Причем индексирование тесно связано с понятием *векторизации* [12].

**Векторизация** означает использование таких конструкций программной системы, которые позволяют отказаться от операторов **for, while**и других, используемых для организации явных циклов программы. Причем векторизация обычно приводит к тому, что программа начинает работать быстрее. Большинство из возможных подходов к векторизации используется и при индексировании матриц Scilab.

Обычно поэлементные операции над матрицами выполняются для всех элементов матриц одного размера (одинаковой размерностью и с одинаковым числом элементов по каждому измерению). В некоторых случаях возможно выполнять операции над матрицами разного размера за счёт ***неявного расширения*** одной из матриц, которое является одним из элементов векторизации.

***Следует помнить, что нумерация элементов матрицы в строках и столбцах начинается с* 1*, а также, что операция* [] *и соответствующие функции всегда создают двумерную матрицу, включая матрицы* 0×0, 0×n, n×0, 1×1 и 1×n**

***Индексирование векторов***

**Индексирование векторов**– это стандартный подход к индексированию одним индексом, который используется во всех языках программирования:

*V1***(***НомерЭлементаСтроки***)**

*V2***(***НомерЭлементаСтолбца***),**

где *V1* **–** вектор строка**,** *V2* **–** вектор столбец и*V2***=***V1*'**.**

Рассмотрим **Примеры1-11** на рис. 1.3.3-1, иллюстрирующие особенности использования индексации в Scilab.



--> // ***Примеры индексирования векторов***

-->

--> vA1 = [40 5 13 4 2 11 7 14] // ***Создание вектора 1×8***

-->

--> // ***Пример1 Доступ к одному элементу вектора***

--> vA1(3) // ***Доступ к третьему элементу вектора***

ans =

13.

-->

--> // ***Пример2 Доступ к нескольким элементам векто***ра

--> vA1([1 5 6]) // ***Доступ к 1, 5 и 6 элементам вектора***

ans =

40. 2. 11.

-->

--> // ***Пример3 Доступ от 3-го до 7-го элемента вектора***

--> vA1(3:7)

ans =

13. 4. 2. 11. 7.

-->

*-->*// ***Пример4*** Доступ, извлечение и создание нового вектора

--> vA2 = vA1([5:8, 1:4])

vA2 =

2. 11. 7. 14. 40. 5. 13. 4.

-->

--> // ***Пример5 Доступ к последнему элементу вектора***

--> vA1($)

ans =

14.

-->

--> //***Пример6 Доступ от 5-го элемента вектора до последнего***

--> **vA1(5:$)**

ans =

2. 11. 7. 14.

-->

--> //***Пример7 Использование* $ *в арифметических опе*рациях**

--> vA1(2:$ - 1) //***Доступ от 2-го до7-го элемента (предпоследнего)***

ans =

5. 13. 4. 11. 7.

-->

--> // ***Пример8 Доступ ко всем нечетным элементам вектора***

**--> vA1(1:2:$)**

**40. 13. 2. 7.**

-->// ***Пример9 Обратный порядок доступа к элементам вектора***

--> vA1($:-1:1)

ans =

14. 7. 11. 2. 4. 13. 5. 40.

*-->*// **Пример10 Модификация элементов существующего вектора**

--> vA1([2 3 4]) = [10 15 20]

vA1 =

40. 10. 15. 20. 2. 11. 7. 14.

-->

--> // ***Пример11Скалярное расширение (присвоение значение 30 2 и 3* элементу)**

--> vA3([2 3])' = 30 // *Замена 2-го и 3-го элементов на значение 30*

vA3 =

0.

30.

30.

Рис. 1.3.3-1. Примеры векторного индексирования

***Стандартное индексирование матриц***

**Стандартное индексирование матриц** реализуется с помощью двух индексов. Матричное стандартное индексирование предполагает использование двух индексов, разделенных запятой – первой для строк, а второй для столбцов, причем не надо забывать, что индексирование начинается с **1**:

**mA**(*НомерЭлементаСтроки***,** *НомерЭлементаСтолбца***).**

Создадим матрицу **mA(4,4)** и проиллюстрируем на ней использование матричной индексации (**Примеры 1-4** на рис. 1.3.3-2).

|  |
| --- |
| --> // ***Примеры матричной индексации***  -->  --> mA = [16 2 3 13; 5 11 10 8; 9 7 6 12; 4 14 15 1]  mA =  16. 2. 3. 13.  5. 11. 10. 8.  9. 7. 6. 12.  4. 14. 15. 1.  -->  --> // ***Пример1 Доступ к одному элементу в строке 2 и столбце 4***  --> mA(2, 4)  ans =  8.  -->  --> // ***Пример2 Доступ к подмножеству элементов (индексы являются векторам***и)  --> mA(2:4, 1:2)  ans =  5. 11.  9. 7.  4. 14.  -->  --> // ***Пример3 Доступ к элементам 3-ей строки, используя операцию :***  -->  --> A(3, :)  ans =  9. 7. 6. 12.  -->  --> // ***Пример4 Доступ к последнему столбцу***  --> mA(:, $)  ans =  13.  8.  12.  1. |

Рис. 1.3.3-2. Примеры стандартной матричной индексации

***Векторное (линейное) индексирование матриц***

Линейное векторное индексирование матриц, это такое индексирование матриц, при котором индексирование ее элементов осуществляется только одним индексом. В этом случае матрицу рассматривают как суммарный вектор, составленный из векторов матрицы, то есть вектор, в котором все элементы матрицы вытянуты в один длинный вектор-столбец, где каждый следующий столбец индексируемой матрицы следует за предыдущим столбцом  
(рис. 1.3.3-3). Так на самом деле матрица хранится в памяти (в нашем случае это последовательность: 1 5 9 2 6 10 37 11 4 8 12). Следовательно, двумерные массивы располагаются в **Рабочей области** данных по столбцам.

|  |  |  |
| --- | --- | --- |
| **Значения элементов** | **Линейные индексы** | **Матрица mA** |
| 1  5  9  2  6  10  3  7  11  4  8  12 | 1  2  3  4  5  6  7  8  9  10  11  12 | |  |  |  |  |  | | --- | --- | --- | --- | --- | | i/j | 1 | 2 | 3 | 4 | | 1 | 1 | 2 | 3 | 4 | | 2 | 5 | 6 | 7 | 8 | | 3 | 9 | 10 | 11 | 12 | |

Рис. 1.3.3-3. Элементы матрицы и их линейные индексы

Индексирование матриц одним индексом называется ***линейным  
векторным индексированием матриц***. Например, выражение **mA(8)** просто извлекает **8**-й элемент неявного вектора-столбца.

Примеры линейного индексирования матриц приведены на рис. 1.3.3-4.



***-->*// *Примерылинейной матричной индексации***

-->

--> mA = [16 2 3 13; 5 11 10 8; 9 7 6 12; 4 14 15 1];

--> // ***Пример1***

--> mA([2 7 16]) // ***Доступ ко 2-му, 7-му и 16-му элементу линейной матрицы***

ans =

5.

7.

1.

--> // ***Пример2 Преобразование индексы матрицы в линейные индексы***

--> idmA = sub2ind(size(mA), [2 3 4], [1 2 4])

idmA =

2. 7. 16.

-->mA(idmA)

ans =

5.

7.

1.

--> // ***Пример3 Преобразование из матричного представление в векторное***

--> A = [2 6 9; 4 2 8; 3 5 1];

--> linearindex = sub2ind(size(A), 3, 2)

linearindex =

6.

-->A(linearindex)

ans =

5.

Рис. 1.3.3-4 Примерылинейнойиндексацииматриц

В **Примере1** показано, что индекс при линейном индексировании матрицы может быть вектором, содержащим более одного линейного индекса, и в этом случае они должны быть перечислены через пробелы или запятые и заключены в квадратные скобки. По заданным линейным индексам производится доступ и извлечение значений элементов матриц.

В **Примере2** проведено преобразование индексов матрицы в линейные индексы. Чтобы получить эквиваленты линейных индексов строки и столбца, была использована функция [**ind2sub**, имеющая три параметра: размер матрицы **(size(A))**; индексы строк **([2 3 4])**и индексы столбцов **([1 2 4]).**](https://www.mathworks.com/help/matlab/ref/ind2sub.html)То есть в этом примере нас интересуют линейные индексы таких элементов матрицы, которые при обычном стандартном индексировании записаны как **mA(2,1), mA(3,2)** и **mA(4,4)**.Чтобы получить линейный индекс одного элемента воспользуемся формулой перерасчета, по которой собственно и происходит расчет в системе Scilab: iV**=(j-1)\*d1+i,**где **i** и **j** *–* номера, соответственно, строки и столбца элемента матрицы, а **d1** *–* размер матрицы. Рассчитаем, например, линейный индекс **1**-го заданного в примере элемента: **i=2, j=1, d1=3**, а **iV=(1–1)\*3+2=2**. Далее показана возможность извлечения элементов по их линейным индексам.

В **Примере3** проведено преобразование индексов одного элемента матрицы **(3х3)** к его линейному эквиваленту и извлечение с его помощью элемента массива.

Таким образом, для доступа к элементам матриц есть выбор между использованием стандартного матричного синтаксиса и использованием доступа, который называется векторным (линейным) индексированием. Предполагая, что **A(i,j)** элемент матрицы **A** размерности **n×m**, индекс этого же элемента **A(k)** при линейной индексации можно рассчитать по формуле **k=(j-1)\*n+i**.

Функции **sub2ind**и **ind2sub** позволяют осуществить преобразования, соответственно, из линейного индексирования элементов матрицы в матричную и из матричной в линейную (***Приложение 1.3, табл. 1.3.3-1***).

В общем виде возможны два случая ***индексирования одной матрицы другой матрицей****.* В первом случае оно основано на *значениях* элементов индексирующего массива, а во втором – индексирование основано на *позициях* элементов индексирующей матрицы, и поэтому называется ***логическим  
индексированием***.

В примерах на рис.1.3.3-2мы фактически уже сталкивались с первым случаем индексирования одного массива другим массивом. Рассмотрим еще несколько примеров (**Примеры1-5**, рис. 1.3.3-5),иллюстрирующих индексирование, основанное на значениях.

Так, в **Примере1** (рис. 1.3.3-5) значения элементов вектора **V2** являются индексами вектора **V1**. В **Примере2** показано, что результат индексирования не зависит от того, является индексирующий вектор столбцом или строкой. В **Примере3** показано, что результат индексирования зависит от того является индексируемый вектор столбцом или строкой. В **Примере4** показано индексирование матрицы **А1** вектором **V3**.

|  |  |
| --- | --- |
| --> // ***Примеры индексирования матрицы***  --> // ***значениями элементов вектора***  -->  --> // ***Пример1***  --> V1 = 5:5:50  V1 =  5. 10. 15. 20. 25. 30. 35. 40. 45.50.  --> V2 = [1 3 6 7 10];  --> V1(V2)  ans =  5. 15. 30. 35. 50.  -->  --> // ***Пример2***  --> V1(V2')  ans =  5. 15. 30. 35. 50.  -->  --> // ***Пример3***  --> V3 = V1';  -->V3(V2)  ans =  5.  15.  30.  35.  50.  -->  --> // ***Пример4***  --> A1 = [1 3 6; 7 9 10];  --> V3 = [1 2 3 4 5];  --> A1(V3)'  ans =  1. 7. 3. 9. 6. |  |

Рис. 1.3.3-5 Индексирование матрицы другой матрицей, основанное  
на значениях индексирующей матрицы (вектора)

Рассмотрим более сложный пример установки некоторого множества элементов матрицы в ноль. Например, требуется найти максимум каждой строки заданной матрицы, а потом все остальные элементы установить в ноль. Решение данной задачи представлено на рис. 1.3.3-6.



--> // ***Установка некоторого множества элементов матрицы в ноль***

-->

-->A = [1 2 3 4; 5 5 6 7; 7 9 8 3] // ***Создание исходной матрицы***

A =

1. 2. 3. 4.

5. 5. 6. 7.

7. 9. 8. 3.

--> // ***Значения y и номера I максимальных элементов в каждой строке***

--> [y, I] = max(A, 'c')

I =

4.

4.

2.

y =

4.

7.

9.

-->

-->B = zeros(A) // ***Создание матрицы* B*c нулевыми элем***ентами

B =

0. 0. 0. 0.

0. 0. 0. 0.

0. 0. 0. 0.

-->

--> B(sub2ind(size(A), 1 : length(I), I')) = y // *Формированиематрицы***B**

B =

0. 0. 0. 4.

0. 0. 0. 7.

0. 9. 0. 0.

Рис. 1.3.3-6 Пример установки некоторых элементов матрицы в ноль

При доступе к элементам матрицы с числовыми индексами вне границ матрицы (присвоении значений таким элементам), Scilab расширяет (увеличивает) размерности этой матрицы для включения этих элементов в матрицу, заполняя остальные получившиеся при расширении элементы нулями. Это можно квалифицировать как добавление элементов к существующей матрице. При этом, если бы была предпринята попытка сослаться на элементы матрицы за пределами массива справа от оператора присваивания, Scilab выдал бы сообщение об ошибке: «**Недопустимый индекс**».

В **Примере1**(рис. 1.3.3-7) показано присвоение значения элементу массива **А** за пределами его границ. Отображение матрицы показывает, что произошло добавление **5**-го столбца, где в **3**-й строке присвоенное значение **7**, а все остальные элементы равны **0**.

Удаление элементов из существующей матрицы можно осуществить путем присвоения пустого множества этим элементам – **[]**. Удалением элементов можно сделать вектор короче, а матрицу – меньше (**Пример2**,   
рис. 1.3.3-7).

|  |
| --- |
| --> // ***Назначение элементов вне границ и удаление элементов матрицы***  -->  -->// ***Исходная матри*ца A(4x4)**  --> A = [16 2 3 13; 5 11 10 8; 9 7 6 12; 4 14 15 1]  A =  16. 2. 3. 13.  5. 11. 10. 8.  9. 7. 6. 12.  4. 14. 15. 1.  -->  -->A(3, 7)  Недопустимый индекс.  --> //  -->A(3, 5) = 7 // **A(3,5)** – *вне границ массива!*  A =  16. 2. 3. 13. 0.  5. 11. 10. 8. 0.  9. 7. 6. 12. 7.  4. 14. 15. 1. 0.  -->  --> // ***Пример2***  -->A(1, :) = [] // ***Удаление 1-й строки***  A =  5. 11. 10. 8. 0.  9. 7. 6. 12. 7.  4. 14. 15. 1. 0. |

Рис. 1.3.3-7. Назначение и удаление элементов вне границ матрицы

Доступ к нескольким непоследовательным элементам матрицы можно

осуществить также с помощью операции двоеточие со значением шага   
**h(m:h:n)**. Так в примере (рис. 1.3.3-8), выражение **B(1:3:16)=-10** означает, что каждому **3**-му элементу матрицы **В** присваивается значение **-10**. Обратите внимание на то, что в этом примере используется линейное индексирование

|  |
| --- |
| --> // ***Доступ к нескольким элементам матрицы***  -->  -->A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4];  -->B = A;  -->B(1:3:16) = -10  B =  -10. 1. 1. -10.  2. 2. -10. 2.  3. -10. 3. 3.  -10. 4. 4. -10. |

Рис. 1.3.3-8. Доступ к нескольким элементам матрицы

Таким образом, обращение (индексирование) к элементам матрицы можно осуществить как с помощью ***операции двоеточие*** (**:**), ссылаясь на все элементы (или часть элементов) в строке (или столбце) матрицы, так и ко всем элементам матрицы (***Приложение 1.3, табл. 1.3.3-2***).

* + 1. **Операции и функции с матрицами**

В **п.1.2.2** уже рассматривалась *операция присваивания* значений отдельным элементам матрицы и всей матрице, а на рис. 1.2.2-2–1.2.2-5, были приведены примеры с использованием этих операций. Поэтому достаточно только напомнить, что операции для числовых данных без точки действуют в соответствии с правилами линейной алгебры, а операции с точкой осуществляют поэлементные операции над элементами массивов.

В Scilab имеется большое число функций, которые вычисляют различные характеристики матриц. Некоторые из них приведены в   
***Приложении 1.3, табл. 1.3.4-1****.*

***Напомним, что функции, определяющие структуру матриц приведены в   
Приложении 1.2, табл. 1.2.2-4.***

Немаловажную роль в Scilab играют так называемые ***пустые матрицы***. Базой для пустых матриц является то, что любая операция, определенная для матрицы **m×n**, должна быть разрешена даже, если **m** или **n** равны нулю. Размер результата этой операции соответствует результату, получаемому при работе с непустыми значениями, но все значения равны **0**.

Например, горизонтальная конкатенация (объединение) **C=[A B]** требует, чтобы матрицы **A** и **B** имели одинаковое количество строк. То есть, что если **A** –матрица размером **m×n,** а **B -** матрица размером **m×p**, то матрица **C** будет размером **m×(n+p)**. Это будет по-прежнему верно, если **m**, **n** или**p** равно нулю.

На рис.1.3.4-1 приведены некоторые общие операции, которые возвращают ненулевые значения в пустом массиве.

|  |
| --- |
| --> // ***Доступ к нескольким элементам матрицы***  -->  -->A = [1 1 1 1; 2 2 2 2; 3 3 3 3; 4 4 4 4];  -->B = A;  -->B(1:3:16) = -10  B =  -10. 1. 1. -10.  2. 2. -10. 2.  3. -10. 3. 3.  -10. 4. 4. -10. |

Рис. 1.3.3-8. Доступ к нескольким элементам матрицы

Все ***поэлементные операции*** над пустыми матрицами считаются действительными до тех пор, пока согласованы размеры операндов, или непустой операнд является скалярным. Поэлементные операции над пустыми матрицами всегда возвращают пустую матрицу (рис.1.3.4-2)

Любое вещественное или комплексное число представлено в Scilab как матрица **1×1**, называемая скалярным значением, однако к нему также применимы некоторые функции над матрицами (рис. 1.3.4-2).

|  |
| --- |
| --> // ***Проверка характеристик матрицы*1х1**  -->  --> // **А** – ***матрица*1х**1  -->A = 5;  -->ndims(A) // ***Проверка количества измерений в*А**  ans =  2.  -->size(A) // ***Проверка значений строк и столбцов***  ans =  1. 1.  -->isscalar(A) // ***Проверка, имеет ли* A*скалярное значение***  ans =  T |

Рис. 1.3.4-2. Проверка характеристик матрицы **1х1**

Матрицы, у которых одно измерение равно единице, а другое больше единицы, как известно, называются векторами. Пример числового вектора, с элементами различных числовых типов приведен на рис.1.3.4-3.

|  |
| --- |
| --> // ***Пример числового век***тора  -->  --> A = [5.73, 2 - 4\*%i, 9/7, 25\*%e, .046, sqrt(32), %i\*8];  --> size(A) // ***Определение числа строк и столбцов***  ans =  1. 7. |

Рис. 1.3.4-3. Пример числового вектора

Вектор может быть построен из других векторов, если, конечно, правильно согласованы их размеры. Все компоненты вектора строки должны быть скалярами или другими векторами строк (рис.1.3.4-4). Аналогично, все компоненты вектора столбца должны быть скалярами или другими векторами столбцов.

|  |
| --- |
| --> // ***Пример построения вектора из других векторов***  -->  --> A = [29 43 77 9 21];  --> B = [0 46 11];  --> C = [A 5 1 B]  C =  29. 43. 77. 9. 21. 5. 1. 0. 46. 11.  -->  --> A = [5.36; 7.01; []; 9.44] // *Объединение с пустым вектором*  A =  5.36  7.01  9.44  --> isvector(A) // ***Функция, определяющая, является ли переменная вектором***  ans =  T |

Рис.1.3.4-4 Пример построения векторов из других векторов

Из примеров, приведенных на рис 1.3.4-4, следует, что конкатенация пустой матрицы с вектором не влияет на результирующий вектор. В этом случае пустая матрица просто игнорируется, а проверить, является ли переменная вектором можно с использованием функции [**isvector**](https://www.mathworks.com/help/matlab/ref/isvector.html).

* + 1. **Логическая индексация**

Естественным продолжением операций над матрицами и их элементами является векторизация операций сравнения, поиска, принятия решений и логических операций (п. 1.2.3). Напомним, что операции сравнения и логические операции в Scilab над массивами возвращают в качестве результата массив логического типа со значениями элементов **%F**и **%T**.

***Логическое индексирование матрицы***

Логическое индексирование матрицы **А** логической матрицей **L**, обозначаемое как **А(L)**, возвращает такие элементы массива **А**, которые соответствуют позиции в массиве индексирования **L** и имеют значения **%T** (True).   
При этой операции каждый элемент со значением **%T** в массиве индексации **L**рассматривается как позиционный индекс в массиве **A**, к которому осуществляется доступ.

Логические массивы позволяют во многих случаях обходиться без операторов цикла (п. 1.5.4), поэтому хороший уровень программирования в среде Scilab требует умения применять логическое индексирование.

Таким образом, если элементы логического массива **L**, являясь индексами массива **A**, то они определяют позиции элементов массива **A** и эта индексация соответствует их позициямв массиве **L**.

Scilab содержит специальные системные значения для определения переполнения, потери значимости и неопределенности операторов, таких как **Inf**и **NaN**. Чтобы помочь выполнять логические тесты для этих специальных значений существуют логические операторы **isinf** и **isnan**, поскольку часто элементы массива, имеющие значения **NaN**, нужно исключить из вычислений.

С помощью операций **&** и **|** Scilab позволяет выполнить соответствующие логические операции «**и»** и «**или»,** причем над элементами всего вектора.

Рассмотрим несколько примеров на использования логической индексации векторов (рис. 1.3.5-1).

|  |
| --- |
| --> // ***Примеры логической индексации***  -->  --> vD = [-0.2, 1.0, 1.5, 3.0, -1.0, 4.2, 3.14];  --> // ***Пример1***  -->iV1 = vD>= 0  iV1 =  FTTTFTT  -->  --> // ***Пример2***  -->vP1 = vD(iV1 == %T) // ***или*vP1 = vD(vD>= 0)**  vP1 =  1. 1.5 3. 4.2 3.14  -->  --> // ***Пример3***  --> vP2 = vD(iV1 ~= %T) // **или vP1 = vD(vD<= 0)**  vP2 =  -0.2 -1.  -->  --> //***Пример4***  -->vP3 = vD((vD>= 0) & (vD< 3))  vP3 =  4.2 3.14  -->  --> // ***Пример5***  --> x = [2 -1 0 3 %nan 2 %nan 11 4 %inf]  x =   1. -1. 0. 3. Nan 2. Nan 11. 4. Inf   -->  --> // ***Пример6***  -->xvalid=x(~isnan(x))  xvalid =  2. -1. 0. 3. 2. 11. 4. Inf |

Рис. 1.3.5-1 Примеры логической индексации векторов

В**Примере1** формируется матрица логических индексов **iV1**, соответствующих условию **vD>=0**.В**Примере2** возвращаются элементы матрицы **iP1**, соответствующие позиции в матрице индексирования **ivD** и имеющие значения **%T.** Эти примеры можно проиллюстрировать на рис. 1.3.5-2.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  |  |  |  |  |  |  | | -0.2 |  | F |  | -0.2 |  |  | | 1.0 |  | T |  | 1.0 |  | 1.0 | | 1.5 |  | T |  | 1.5 |  | 1.5 | | .0 | 🡺 | T | 🡺 | 3.0 | 🡺 | 3.0 | | -1.0 | **vD >= 0** | F | **vD(ivD)** | -1.0 | **vP1 = vD(iV1)** | 4.2 | | 4.2 |  | T |  | 4.2 |  | 3.4 | | 3.4 |  | T |  | 3.4 |  | VP1 |   vD iV1 vD |

Рис. 1.3.5-2 Иллюстрация примеров из рис. 1.3.5-2

В**Примере3** возвращаются элементы матрицы **vP2**, соответствующие позиции в матрице индексирования **vD**, значения которых **%F**. В**Примере4** возвращаются элементы матрицы **vP3**, соответствующие условию, что элементы матрицы **vD**положительны и больше **3**. В **Примере5** формируется матрица **х**, которая используется в **Примере6**, для формирования матрицы **xvalid** путем исключения из матрицы **х** элементов, имеющих значение **NaN**.

Рассмотрим пример создания матрицы **А** размерностью **3×3** и вычисление логической матрицы **L**, удовлетворяющей условию **A>0.5**. Далее используя полученную матрицу **L** для индексирования матрицы **A**, присвоим соответствующим элементам матрицы **A**, значение равное **0**. Решение данного примера приведено на рис.1.3.5-2.



--> // ***Пример логического индексирования массива*А**

-->A = rand(3, 3)

A =

0.7560439 0.6653811 0.685731

0.0002211 0.6283918 0.8782165

0.3303271 0.8497452 0.068374

--> L = A > 0.5

L =

T T T

F T T

F T F

-->A(L) = 0

A =

0. 0. 0.

0.0002211 0. 0.

0.3303271 0. 0.068374

**// *Более короткое реше*ниеA (A> 0.5) = 0;**

Рис. 1.3.5-2 Пример логического индексирования массива **А**

В ***Приложении1.3, табл. 1.3.5-1***рассмотрены дополнительные примеры логического индексирования матриц, где **L** является матрицей логических значений.

***Использование функции find***

В дополнение к примерам, приведенным в ***Приложении 1.3.5, табл. 1.3.5-*1,** необходимо знать следующее: чтобы найти такие элементы матрицы, которые удовлетворяют условию, можно использовать функцию **find** в сочетании с логическим выражением. Например, **find(X<5)** возвращает линейные индексы для элементов **X** , которые имеют значения менее **5**.

Чтобы непосредственно найти элементы массива **X**, которые удовлетворяют условию **X<5**, можно использовать конструкцию **X(X<5)**, и при этом следует избегать таких конструкций как **X(find(X<5))**, которые необоснованно используют функцию **find** с логической матрицей.

В ***Приложении 1.3, табл.  1.3.5-*2**приведено описание функции **find, gsort, vectorfind.**

***Обратите внимание, что если после ввода элементов матриц или вектороввыявлены ошибки, то для исправления отдельных значений элементов можно воспользоваться окном Редактора данных.***

В большинстве случаев при логическом индексировании матрицы должны иметь одинаковое количество элементов, как индексируемых, так и индексирующих, но это не является жестким требованием. Индексирующие матрицы могут иметь и меньшие (но не большие) размеры (рис. 1.3.5-3).



--> // ***Пример на логическое индексирование, при котором индексирующая***

--> // ***матрица имеет размер меньше чем размер индексируемой матрицы***

-->

--> A = [1 2 3; 4 5 6; 7 8 9]

A =

1 2 3

4 5 6

7 8 9

--> B = ([%f %t %f; %t %f %t])

B =

F T F

T F T

-->

--> isequal(prod(size(A)), prod(size(B)))

ans =

F

-->

--> A(B)

ans =

4

7

8

-->

--> C = [B(:); %f; %t; %f];

--> isequal(size(A), size(C))

ans =

1

-->A(C)

ans =

4

7

8

Рис. 1.3.5-3 Пример на логическое индексирование, при котором

индексирующая и индексируемые матрицы имеют разные размеры

***Обратите внимание, что при логическом индексировании всегда возвращается вектор- столбец, а также, что Scilabобрабатывает отсутствующие элементы индексирующих матриц так, как если бы они присутствовали и были равными %*f**.

* + 1. **Создание и использование массивов   
       структури массивов ячеек**

***Тип данных массивовструктур***

В Scilab для формирования массивов структур существуют два основных варианта. Первый заключается в использовании функции **struct**, аргументами которой выступают названия полей структуры и соответствующие значения. Названия полей должны быть строковыми переменными, а значения в этих полях могут иметь любой допустимый тип.

Второй вариант создания структуры состоит в использовании оператора присваивания и точечной нотации:

*Переменная****=****ИмяСтруктуры.ИмяПоля,*

возвращает значение *ИмяПоля* структуры с указанным вслед за ним именем *ИмяСтруктуры*. Если такая структура уже существует, то значение будет выведено на экран, а если отсутствует, то она создается автоматически. При этом системная функция **whos** будет выводить только информацию о самой структуре, но не выводит, ни *ИмяПоля*, ни содержимое этих полей. Для просмотра этой информации можно или обращаться специально к отдельным полям структуры, или использовать редактор переменных **Variables**. Структура в памяти занимает больше места, чем суммарный размер её полей, поскольку в памяти хранятся имена полей, а каждое имя поля – это переменная типа **char**, и каждый символ в этом имени занимает по два байта.

В Scilab имеется возможность создавать также *матрицы структур*. Для создания таких матриц после имени структуры перед точкой в круглых скобках указывается индекс. Кстати, массив из двух структур занимает в памяти меньше места, чем две структуры по отдельности. Структуры очень часто используются во встроенных и внешних функциях. Этот тип данных чрезвычайно удобен для генерирования отчетов и упорядочивания большого количества схожей информации.

***Массивы ячеек и доступ к элементам массива ячеек***

Если связанные между собой данные имеют разные типы, то их можно сохранить путем объединения в массив ячеек, где ячейка содержит часть данных.  Ссылка на элементы массива ячеек производится с использованием индексирования. При этом для индексирования массива ячеек используются круглые скобки **()**, а для индексирования содержимого ячеек, используются фигурные скобки **{}**.

Массивы ячеек удобны для структурирования разнородной информации. Для доступа к данным, хранящимся в структуре, необходимо знать названия полей структуры, тогда как для доступа к данным массива ячеек нужны только индексы, указывающие на конкретную ячейку. Самый простой вариант создания ячейки состоит в использовании фигурных скобок {}. Элементы одной строки массива отделяются друг от друга запятой, а столбцы точкой с запятой. Массивы ячеек в этом отношении аналогичны матрицам, но без ограничений на типы данных, находящихся внутри ячеек. В ячейке может находиться структура или массив ячеек. Для удобного просмотра содержимого массивов следует также использовать окно Редактор переменных. Кроме того, тип данных **cell** удобен для организации массивов данных, когда заранее не известен тип этих данных или известно, что данные разнородны.

Таким образом, вектор ячеек является универсальным контейнером – его элементы могут содержать любые типы и структуры данных, с которыми работает Scilab – векторы чисел любой размерности, строки, векторы структур и другие (вложенные) векторы ячеек.

Векторы ячеек могут быть созданы либо путём последовательного присваивания значений отдельным элементам массива, либо созданы целиком при помощи специальной функции **cell**. Однако в любом случае важно различать ячейку (элемент вектора ячеек) и её содержимое. Ячейка – это содержимое плюс некоторая оболочка (служебная структура данных) вокруг этого содержимого, позволяющая хранить в ячейке произвольные типы данных любого размера.

Рассмотрим пример создания вектора ячеек для хранения разных типов данных (рис. 1.3.6-1).



--> // ***Создание векторов ячеек для хранения данных разных типов***

--> book = struct('title', 'Онегин', 'author', 'Пушкин', 'year', 2000)

book =

title: [1x1 string]

author: [1x1 string]

year: [1x1 constant]

-->

--> Vcell(1) = {book}, Vcell(2) = {'Пушкин'}, Vcell(3) = {2000}

Vcell =

[1x1 struct]

Vcell =

[1x1 struct]

[1x1 string]

Vcell =

[1x1 struct ]

[1x1 string ]

[1x1 constant]

-->  
--> Vcell{1}

ans =

title: [1x1 string]

author: [1x1 string]

year: [1x1 constant]

-->

--> Vcell(1)   
[1x1 struct]

-->

--> VcellArray = cell(2, 2)

VcellArray =

[0x0 constant] [0x0 constant]

[0x0 constant] [0x0 constant]

Рис. 1.3.6-1 Создание векторов ячеек для хранения данных разных типов

На рис. 1.3.6-1 задан вектор ячеек **Vcell** с тремя элементами. Первый элемент соответствует структуре, второй – строке, а третий – числу. В этом и заключается особенность организации данных с помощью ячеек: у каждого элемента свой тип данных.

Для обращения к содержимому той или иной ячейки используются фигурные скобки, внутри которых ставится индекс элемента, с которым предполагается работать. Если же используются круглые скобки, то будет возвращена структура данных вместо отдельных значений.

Для того чтобы задать вектор или матрицу ячеек с пустыми (неопределенными) значениями, используется функция **cell**(**VcellArray=cell(2,2)**). Данную инициализацию целесообразно выполнять, когда нужно определить большой вектор или матрицу ячеек и в цикле задавать их значения. В этом случае Scilab сразу создает массивы нужных размеров, в результате чего повышается скорость выполнения программ.

На рис. 1.3.6-2 показано, как извлечь несколько элементов из массива ячеек – список, разделенный запятыми, учитывая, что массив ячеек **4×6**.



--> // ***Извлечение несколько элементов из массива ячеек***

--> C = cell(4, 6);

--> for k = 1:24

> C{k} = k \* 2;

> end

-->

C =

[2] [10] [18] [26] [34] [42]

[4] [12] [20] [28] [36] [44]

[6] [14] [22] [30] [38] [46]

[8] [16] [24] [32] [40] [48]

-->

-->С{:, 5}

ans =

34

ans =

36

ans =

38

ans =

40

-->// ***Это то же самое, что вводить* С{1,5}, C{2,5}, С{3,5}, C{4,5}**

Рис. 1.3.6-2 Извлечение несколько элементов из массива ячеек

Рассмотрим еще один пример (рис. 1.3.6-3), в котором показано, как считывать и записывать данные массива ячеек. Для этого создадим массив ячеек **2×3** текстовых и числовых данных.



--> // ***Считывание и запись данных в массив ячеек и из массива ячее***к

-->

--> C = {'one', 'two', 'three'; 1, 2, 3}

C =

[1x1 string ] [1x1 string ] [1x1 string ]

[1x1 constant] [1x1 constant] [1x1 constant]

-->

--> upLeft = C(1:2, 1:2)

upLeft =

[1x1 string ] [1x1 string ]

[1x1 constant] [1x1 constant]

-->

--> // ***Замена значений элементам первой строки***

-->C(1,1:3) = {'first','second','third'}

C =

[1x1 string ] [1x1 string ] [1x1 string ]

[1x1 constant] [1x1 constant] [1x1 constant]

--> nicC = C(2,1:3)

nicC =

[1x1 constant] [1x1 constant] [1x1 constant]

-->

--> // ***Преобразование массива ячеек в вектор***

-->nicV = cell2mat(nicC)

1. 2. 3.

Рис. 1.3.6-3 Считывание и запись данных в массив ячеек   
и из массива ячеек

Существует два способа ссылки на элементы массива ячеек. Можно заключить индексы в круглые скобки **()** и обращаться к массиву ячеек, например, для того, чтобы определить подмножество массива. Также можно заключить индексы в фигурные скобки **{}**, и сослаться на элементы данных (текст, числа или другие данные) в отдельных ячейках.

Индексы массива ячеек в круглых скобках относятся к наборам ячеек. Например, круглые скобки используются, чтобы создать массив ячейки ***2×2***, который является подмножеством **C**.

Обновление набора ячеек выполняется их заменой с тем же числом ячеек. Например, чтобы заменить ячейки в первой строке **C** используется эквивалент массив ячеек размера (**1×3**). Чтобы заменить содержимое ячейки, аналогично можно индексировать и с использованием фигурных скобок   
(рис. 1.3.6-4).



--> // ***Индексирование с использованием фигурных скобок***

-->

--> last = C{2, 3}

last =

3.

--> C{2, 3} = 300;

--> C{2, 3}

ans =

300.

--> las = C{1:2, 1:2}

las =

las(1)

first

las(2)

1.

las(3)

second

las(4)

2.

-->las(1)

ans =

first

Рис. 1.3.6-4 Индексирование с использованием фигурных скобок

Если элементы в массиве содержат числовые данные, можно преобразовать ячейки в числовом массиве, используя функцию **cell2mat**.

Доступ к содержимому ячеек (число, текст или другие данные в ячейках) производится путем индексации с использованием фигурных скобок. Например, для доступа к содержимому последней ячейки ***C***, используйте фигурные скобки.

Можно получить доступ к содержимому нескольких ячеек путем индексации с фигурными скобками.

* + 1. **Контрольные вопросы**

1. Какие способы создания векторов и матриц реализованы в Scilab?
2. Каким образом создается вектор с постоянным шагом?
3. Можно ли создать матрицу, элементы строки которой изменяются с постоянным шагом?
4. С какого номера начинается индексация вектора?
5. Какая функция позволяет определить длину вектора?
6. Какие функции предназначены для определения числа строк, числа столбцов матрицы и общего количества ее элементов?
7. Что такое «пустая матрица» и как ее создать?
8. Какую матрицу создает операция **ones**?
9. Каким образом в Scilab могут быть объединены две матрицы?
10. С использованием каких функций матрицы могут динамически изменять свой размер в процессе выполнения про­граммы?
11. Для чего предназначены в Scilabфункции **matrix** и **resize\_matrix**?
12. Назначение и формат функции **cat**?
13. Каким образом удалить из матрицы определенный столбец?
14. Что такое векторизация?
15. Что такое стандартное индексирование векторов и матриц?
16. Что такое векторное индексирование матриц?
17. Что такое логическое индексирование матриц?
18. Что такое поэлементные действия (операции с точкой) и где они используются?
19. Каким образом можно транспонировать вектор или матрицу?
20. Требуется ли при работе с векторами и матрицами предварительное объявление их размера?
21. Какой символ используются для разделения элементов матрицы в строке, а какой для разделения ее столбцов?
22. Какие команды предназначены для заполнения матрицы случайными числами, распределенными по равномерному или нормальному закону распределения?
23. Какой командой можно осуществить выбор минимального и максимального значения элемента матрицы.
24. В каких случаях матрицы могут динамически изменять свой размер?
25. Как создается массив ячеек?
26. Что представляют собой операции доступа извлечения и модификации матриц?
27. Что такое индексирование и векторизация?
28. Что такое логическая индексация?
29. Когда можно использовать функцию **find**?

## **Средства визуализация в системе Scilab**

Пользователь Scilab имеет целый ряд возможностей для визуализации данных:

* высокоуровневые графические функции (**plot, surf, mesh** и многие другие);
* интерактивную среду, компоненты которой позволяет изменять свойства элементов графика;
* специализированные функции и средства для графического отображения характеристик исследуемых объектов и результатов вычислений;
* низкоуровневые средства построения графиков и простейших геометрических фигур;
* низкоуровневые средства построения графических интеофейсов пользователей.

Высокоуровневая графика позволяет пользователю получать результаты в графическом виде, прикладывая минимум усилий. Основную работу, связанную с построением графика, масштабированием осей, подбором цветов и т.д. берет на себя среда Scilab.

В некоторых случаях этих средств оказывается недостаточно.Например, если создаваемое приложение должно осуществить вывод графических результатов в готовом виде, не предполагающем их дальнейшую правку, или в ходе своей работы управлять элементами графиков: удалять поверхности, изменять цвет и толщину линий, добавлять стрелки и поясняющие надписи и т.д. В этих случаях использование дескрипторной (низкоуровневой) графики дает возможность полного контроля над элементами высокоуровневой графики.

Кроме того, дескрипторная графика будет полезна при создании собственных приложений. Ее понимание обязательно для эффективного написания приложений с графическим интерфейсом пользователя. Кроме того, большинство высокоуровневых графических функций допускают обращение к ним с использованием низкоуровневых свойств того графического объекта, который они создают.

В последующих параграфах рассматриваются высокоуровневые и низкоуровневые графические средства, а также средства для создания графических интерфейсов пользователей (GUI).

### **1.4.1. Высокоуровневые графические средства. Средства отображение графиков функций и простейших геометрических фигур**

*Основные понятия высокоуровневой графики*

***Графическое изображение*** в высокоуровневые графики – это чаще всего графическое представление графика функций или графическое изображение простейших геометрических фигур.

В среде Scilab все графики строятся по точкам, при этом каждые две соседние точки соединяются друг с другом отрезком. Чем меньше расстояние между точками, тем меньше искажение графика за счет замены его реального образа кусочно-линейной функцией. Поэтому главным ограничивающим фактором в желании получить как можно более гладкоепредставление графического изображения, является то, что координаты точек нужно хранить в оперативной памяти.

Внешний вид графика определяют следующие факторы:

* Вид математической функции в системе координат:
* *одной переменной* – графики функции с одной переменной в заданной системе координат:
* в прямоугольной системе координат;
* в полярной системе координат;
* в комплексной плоскости (например, годографы);
* в векторной плоскости;
* *двух переменных –* графики функции с двумя переменными в заданной системе координат*:*
* в пространственной прямоугольной системе координат;
* в сферической системе координат;
* в цилиндрической системе координат;
* в векторном пространстве.
* Способ представленияв виде:
* *точечного графика*;
* *линейного графика* (с учетом всевозможных представлений линии);
* *полигонального графика*;
* *градиентного графика*;
* *контурного графика;*
* *гистограммы*.

*Графические окна*

Графические изображения (объекты) в Scilab отображаются в специальных***Графических окнах***, причем в одном графическом окне (в одной системе координат) могут быть построены несколько графических объектов, ана экранедисплея могут быть размещены одновременно несколько графических окон, причем разных типов.

Графические окнаимеют свою нумерацию. Чтобы создать новое графическое окно с конкретным номером достаточно выполнить функцию**scf:**

*h* **= scf(***id***)**;

*h* **= scf()**,

где: *id* – необязательный целочисленный параметр, задающий номер окна,а*h* –дескриптор создаваемого окна, который связан с набором свойств этого окна и их значениями.

Если параметр, указывающий номер создаваемого окна, отсутствует, то система присваивает окну последний незанятый порядковый номер, начиная с нуля. В ответ функция **scf** возвращает дескриптор окна, а также набор его свойств, причем большая часть значений свойств заранее предопределена (т.е. заданы по умолчанию), кроме того первое обращение к одной из графических команд автоматически вызывает появление графического окна, которому присваивается номер ноль (**id=0**) рис.1.4.1-1).



Рис. 1.4.1-1 Графическое окно с номером **0**

Элементы меню, расположенного в верхней части графического окна, предоставляют большой набор команд, предназначенных для отображения и оформления графиков, позволяющих в интерактивном режиме и без использования программного кода придать графику желаемый вид, а кнопки панели дублируют наиболее часто используемые пункты меню, ускоряя тем самым процесс оформления графика.Каждый элемент меню содержит набор команд, соответствующих его названию:

***Файл***:*Новое графическое окно, Загрузить…, Сохранить…, Экспортировать, Копировать в буфер, Параметры страницы, Печать, Закрыть*;

***Инструменты***:*Показать/скрыть панель инструментов, Увеличить область, Исходный вид,* ***2D/3D****вращение*;

***Правка***: *Установить текущим графическим окном, Очистить графическое окно…, Свойства графического окна…, Свойства осей, Разрешить выбор объектов, Включить/выключить управление подсказками данных, Начать/остановить изменение данных кривых*;

***Справка***: *Содержание, О Scilab*.

Наиболее часто используемые команды вынесены на панель инструментов в виде кнопок.

Переход между***Командным окном***и***Графическим окном***осуществляется с помощью комбинации клавиш<Alt+Tab>или с помощью мыши.

Закрыть графическое окно можно традиционным способом – щелчком по крестику, расположенному в правом верхнем углу окна, или выполнением команды**xdel(***id***)**, в скобках которой указывается *id* – номер закрываемого графического окна.

Перечень всех открытых графических окон можно посмотреть, выполнив команду**winsid**, которая выводит в следующей строке номера открытых графических окон (рис.1.4.1-2).

|  |
| --- |
| -->//***Примеры использования команд*scf**, **xdel*и*winsid**  *-->*scf(2); scf(3); scf(5);//***Открытие графических окон* 2,3,5**  -->winsid//***Вывод списка открытых графических окон***  ans =  2. 3. 5.  -->  -->xdel(3);//***Удаление графического окна с номером* 3**  -->winsid  ans =  2. 5.  -->scf(); |

Рис. 1.4.1-2 Примеры использования команд**scf**, **xdel**и**winsid**

После каждого вызова функции**scf** появляется графическое окно с соответствующем номером, а в***Командном*** окнеотображаются свойствасозданного окна. Причем основная область графического окна называется рабочей областью, и на ней можно отображать любые графические объекты.

***Следует отметить, что если за время сеанса не было создано ни одного графического окна, с помощью функции* scf*, то любая графическая функция создает ее автоматически.***

*Построение графиков функций от одной переменной*

Начнем знакомство с графической системой Scilab построением графика функции одной переменной. Самым простым способом построения графика функции**y=f(x)**являетсяформирование двух векторов одинаковой длины (век­тора значений аргументов**x**и вектора соответствующих им значений функции **у**) с последующим выполнением функции**plot(x,y)**.Функция**plot(x,y)**открывает графическое окно и отображает в нем график функции**y(x)**.

Рассмотрим, например, построение графика функциина отрезке**[-1.5;1.5].** В результате выполнения функции**plot(x,y)**появляется   
**Графическое окно 0**с графиком функции (рис.1.4.1-3).

|  |
| --- |
| -->// ***Построение графика функции***  -->  -->x = -1.5 : 0.01 : 1.5;  -->y = exp(-x.^2);  --> plot(x, y) |
|  |

Рис. 1.4.1-3. Построение графика функции 

***Обратите внимание, что элемент меню графического окна* File *имеет команду*Копировать в буфер обмена*, которая позволяет копировать окно в буфер обмена, и, которое впоследствии можно вставить в* Word *или другой текстовый редактор.***

Можно построить в одном графическом окне графики нескольких графиков функций. На рис.1.4.1-4 приведен пример построения в одном окне трех графиков.

|  |
| --- |
| -->//***Построение в одном окнеграфиков 3-х функций***  -->  -->x=0:0.01:%pi;  --> y=[sin(x)',sin(2\*x)',sin(4\*x)'];  --> plot(x',y) |
|  |

Рис. 1.4.1-4. Построение в одном графическом окне графиков 3-х функций

При построении нескольких графиков в одном окне каждый график автоматически отмечается своим цветом (в нашем случае красным, синим и зеленым). Однако при печати в черно-белом варианте этот эффект теряется. В Scilab при выводе графика можно заменить принятый по умолчанию цвет и тип точек,с помощью которых строится каждый график (рис. 1.4.1-5).

Для быстрой настройки вида линий на графике можно использовать свойство**Line\_style**. **Line\_style**– это необязательный аргумент, который может быть использован внутри команды построения графика **plot** для настройки вида каждой новой линии. Он должен быть указан в виде соединённых строк, содержащих информацию о *цвете, стиле линий* или *маркеров*. Он очень полезен для быстрого определения этих основных свойств линий.

Например, чтобы определить красную штрихпунктирную линию с ромбовидными маркерами, строка должна быть в виде **'r-.diam'**. Причем, полное написание значения каждого свойства не обязательно, однако строка, которая является связкой (в любом порядке) этих трёх типов свойств, должна оставаться однозначной. Кроме того, определение строки не чувствительно к регистру.

Элементы  **Line\_style**, которые можно использовать в[**plot**](https://help.scilab.org/docs/6.0.2/ru_RU/plot.html)для задания типа и цвета линий показаны на рис.1.4.1-5 и рис.1.4.1-6 соответственно.

|  |  |
| --- | --- |
| Определитель | Стиль линии |
| **-** | Сплошная линия (по умолчанию) |
| **--** | Штриховая линия |
| **:** | Пунктирная линия |
| **-.** | Штрихпунктирная линия |

Рис. 1.4.1-5. Символы, указывающие на тип линии

|  |  |
| --- | --- |
| **Определитель** | **Цвет** |
| **r** | Красный |
| **g** | Зелёный |
| **b** | Синий |
| **c** | Голубой |
| **m** | Пурпурный |
| **y** | Жёлтый |
| **k** | Чёрный |
| **w** | Белый |

Рис. 1.4.1-6. Символы, указывающие на цвет линии

Если не указан цвет (ни с помощью **Line\_style**, ни с помощью   
[**ГлобальногоСвойства**](https://help.scilab.org/docs/6.0.2/ru_RU/GlobalProperty.html)**)**, то будет использоваться таблица цветов, устанавливаемых по умолчанию. Во время отображения множества линий, команда **plot** автоматически перебирает в цикле эту таблицу.

**Тип\_маркера** – строка, определяющая тип маркера. Заметьте, что, если указывается маркер без стиля линии, то будут нарисованы только маркеры. Это свойство связано со свойствами объекта **Mark\_style (Cтиль\_маркера)** и **Mark\_mode (Режим маркера)**(рис.1.4.1-7).

|  |  |
| --- | --- |
| **Определитель** | **Тип маркера** |
| **+** | Знак "плюс" |
| **o** | Кружок |
| **\*** | Звёздочка |
| **.** | Точка |
| **x** | Крестик |
| **'square' ('s')** | Квадрат |
| **'diamond'('d')** | Ромб |
| **^** | Треугольник, указывающий вверх |
| **v** | Треугольник, указывающий вниз |
| **>** | Треугольник, указывающий вправо |
| **<** | Треугольник, указывающий влево |
| **'pentagram' ('p')** | Пятиконечная звезда (пентаграмма) |
|  | Нет маркера (по умолчанию) |

Рис. 1.4.1-7. Символы, указывающие тип маркера

Рассмотрим пример на (рис.1.4.1-8).

|  |  |
| --- | --- |
| *-->// Построить штрихпунктирную линию*  *-->// с треугольниками, указывающими --> // вправо, центрованными на каждой --> // точке*  *-->*[clf](https://help.scilab.org/docs/6.0.2/ru_RU/clf.html)();  *-->*x=1:0.1:10;*// Инициализация*  *-->*[plot](https://help.scilab.org/docs/6.0.2/ru_RU/plot.html)(x, [sin](https://help.scilab.org/docs/6.0.2/ru_RU/sin.html)(x), 'r-.>') | https://help.scilab.org/docs/6.0.2/ru_RU/LineSpec_1.png |

Рис. 1.4.1-8. Пример построения различных линий и маркеров

Символы, указывающие на цвет и тип точки, заключаются в апострофы и указываются в функции**plot**после имени функции.

В общем случае, число аргументовфункции**plot**не ограничивается двумя, посколькуфункция в общем видеимеет следующий формат:

**plot(***x1,y1,'c1', x2,y2,'c2',...***)**,

где*xn,yn -* каждая очередная пара векторов предназначена для построения нового графика, а*'сn'–*символы, указывающие на цвет и тип точки. Вэтомслучае происходит построение нескольких графиковв одном графическом окне.

Выполним теперь тот же пример, что и на рис.1.4.1-2, добавив в него при построении каждого графика символы для отображения цвета и типа точки и другой формат функции**plot**(рис.1.4.1-9).

|  |  |
| --- | --- |
| -->//***Использование инструкцийдля установки цвета и типа линии***  -->  -->x =0:0.1:%pi;  -->y1 = sin(x);// ***Тип линии -, черный***  -->y2 = sin(2\*x); // ***Тип линии -, красный***  -->y3 = sin(4\*x); // ***Тип линии – с точкой, синий***  -->plot(x, y1,'-k',x, y2, '-r', x,y3,'.-b'); | |
|  |  |

Рис. 1.4.1-9. Использованием инструкций для цвета и символа при   
построении в одном графическом окне трех графиков

Различие между приведенными выше примерами состоит не только в количестве параметров функции**plot**, но еще и в том, что в примере, приведенном на рис.1.4.1-4, формируется матрица**y**, содержащая значения функций в виде столбцов, а в примере, приведенном на рис.1.4.1-9, формируются три вектора значений функций**(y1,y2,y3)**.

График, выведенный в графическое окно Scilab, может быть снабжен***заголовком***, ***именами осей***, ***текстом, сеткой***и другой дополнительной информацией, которая устанавливается вспомогательными функциями**xtitle**, **xgrid, legend**и др.Параметрами этих функций являются значения текстовых строк. Например, формат команды**xtitle**, добавляющей к графику заголовок и подписи осей, может выглядеть следующим образом:

**xtitle('***title***','***xstr***','***ystr***'),**

где:**'***title***'**– заголовок графика;

**'***xstr***'**– подпись оси**х**;

**'***ystr***'**– подпись оси**y**.

Функция**xgrid**позволяет отобразить в***Графическом окне***координатную сетку, а функция **legend**используется,как правило, когдана одной координатной плоскости надо отобразить графики нескольких функций (как, например, на рис. 1.4.1-9).Очевидно, что в этом случае возникает необходимость использованиясвоих обозначений (цвета и символа) для каждого из графиков. Эти принятые обозначения выводятся на экранфункцией**legend**, которая имеет следующие форматы:

**legend(***leg1***,** *leg2***, ...,** *legn***),**

**legend(***leg1***,** *leg2***, ...,** legn**,** *pos***),**

где**:** *leg****1***– имя первого графика;

***leg2***–имя второго графика;

*leg***n**–имя**n**-го графика;

*pos*–необязательный числовой параметр, определяющий место расположениялегенды в графическом окне (рис.1.4.1-10). По умолчанию числовой параметр*pos*равен**1**, что означает расположение легенды в правом верхнем углу.

|  |  |
| --- | --- |
| **Значение аргумента** | **Размещение легенды** |
| **-1** | В правом верхнем углу над областью графика |
| **0** | Выбирается автоматически, чтобы не перекрывались кривые |
| **1** | В правом верхнем углу (и по умолчанию) |
| **2** | В левом верхнем углу области графика |
| **3** | В левом нижнем углу области графика |
| **4** | В правом нижнем углу области графика |

Рис. 1.4.1-10 Значение аргумента **pos** для указания места

размещения легенды

Безусловно, каждый график удобнее выводить с использованием своей функции**plot**,однако в этом случае происходит создание нового графического окна. Для того чтобы этого избежать,после первого выведенного графика достаточно выполнить функцию**mtlb\_hold('on')**.Эта функция позволяет расположить все выводимыевдальнейшемграфики в одном и том же окне.

Рассмотрим пример, в котором при построении трех графиков в одном***Графическом*** окнеиспользованыи прокомментированы всеописанные выше инструкции (рис.1.4.1-11).

|  |  |
| --- | --- |
| --> //*Построение нескольких графиков в одном окне,* --> // *дополняя их заголовком и легендой*  -->  -->x = 0 : 0.01 : %pi \* 2;  -->y1 = sin(x);  --> y2 = sin(2\*x);  -->y3 = sin(4 \* x);  -->  -->plot(x, y1, '-k') //*Сплошная кривая черного цвета*  -->xgrid() //*Нанесение сетки*  -->//*Построение всех последующих графиков в одном окне*  -->mtlb\_hold('on')  -->  -->plot(x, y2, 'b--') //*Штриховая линия синего цвета*  -->plot(x, y3, 'r.-') //*Штрих-пунктирная линия красного цвета*  -->xtitle('Построение графиков трех функций', 'x', 'y') // *Заголовок*  --> legend('y1(x)', 'y2(x)', 'y3(x)', 1) // *Легенда* | |
| 2 |  |

Рис. 1.4.2-11. Использование инструкцийпри построении графиков

При построении нескольких графиков в одном графическом окнес помощью**plot**следует помнить, что в заданном диапазоне аргумент**х**сами отображаемыефункциидолжны иметь соизмеримые значения, иначе совместное изображение графиков не позволит оценить результаты расчета, поскольку нарушается их «видимость».

Функция**plot2d**лишена этого недостатка, посколькустроит график по формату, который уже определен пользователем при построении предыдущего графика.

На рис.1.4.1-12 приведен пример использования функции**plot2d**для отображения в одном графическом окне графиков двух функций. Заметим, что совместное изображение графиков, для того, чтобы в окне были достаточно хорошо видны оба графика,потребовало изменения масштаба по оси **y**.

|  |  |
| --- | --- |
| -->//***Построение 1-го графика*y=sin(x)**  -->  -->scf(0);  -->deff('y=f(x)','y=sin(x)');  -->x = -10:0.5:10;  -->y = f(x);  -->plot(x, y, '--')  -->//***Добавление графика функции* x^2**  -->  -->scf(0);  -->plot2d(x,x.^2) |  |
|  |

Рис.1.4.1-12 Построение в одном окне двух графиков

Добавление к имени функции**plot2d** цифр **2**, **3**или **4**, приводит к изменению шаблона графика:

* **plot2d2**– функция, предназначенная для построения графика в виде ступенчатой функции;
* **plot2d3**– функция, предназначенная для построения графика в виде вертикальных полосок;
* **plot2d4**– функция, предназначенная для построения графика с указанием направления.

Примеры использования функций **plot2d2, plot2d3**и**plot2d4**приведены на рис.1.4.1-13 (**а**, **b** и **с**).

|  |  |
| --- | --- |
| -->//***Примеры использования функций***  -->//**plot2d2, plot2d3 *и*plot2d4**  -->  -->deff('y = f(x)','y = sin(x)');  -->x=[1:0.1:3\*%pi];  -->y = f(x);  -->  -->// ***Ступенчатый график***  -->plot2d2(x,y)  -->  -->// ***График с вертикальными***  -->// ***полосами***  -->scf(1);  -->plot2d3(x,y)  -->  -->//***График с указанием направления***  -->scf(2);  -->plot2d4(x,y) | a) Ступенчатый график    b) График в виде вертикальных полос    c) График с указанием направления |

Рис. 1.4.1-13 Примеры использования функций

1. **plot2d2**b), **plot2d3,** c) **plot2d4**

Для построения гистограмм на плоскости в Scilab предназначена функция**histplot**.На рис.1.4.2-14 приведен пример, отображающий данные в 20-ти отрезках, имеющие нормальное распределение.

|  |  |
| --- | --- |
| -->// ***Построения гистограммы***  -->  -->d = rand(1, 10000, 'normal');  -->histplot(20,d); |  |

Рис. 1.4.1-14 Пример построения гистограммы функцией**histplot**

Чтобы воспроизвести изображение в двумерном пространстве в виде векторных полей можно использовать функцию**champ**,самый простой формат которой имеет следующий вид:

**сhamp(***x,y,fx,fy***)**,

где: *x, y*– векторы, определяющие сетку координат;

*fx*– матрица, описывающая*x*–компоненту каждого поля вектора;

*fy*– матрица, описывающая*y*–компонентой в точке (**x(i),y(i)**).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| -->// ***Пример использования функции*champ**  -->  -->champ(-5:5,-5:5, rand(11,11), rand(11,11))   |  |  | | --- | --- | |  |  | |  |  | |

Рис. 1.4.1-15. Пример использования функции**champ**

С помощью функции**сhamp**векторныеполя изображаются в виде стрелок, а их длина указывает интенсивность векторного поля(рис. 1.4.2-15).

Все сведения об используемых в п. 1.4.2 функциях приведены в  
***Приложении 1.4. табл. 1.4.2-1***

*Построение графиков функций от двух переменных*

***Трехмерные поверхности***описываются функцией двух переменных**z(x,у)**.Для построения одних трехмерных графиков необходимосформироватьдвематрицы, например,**xМ** и **yМ,** а для другихдва вектора, например, **XV**и**YV**.

Сформированные в виде матриц данные используются функциями:

* **mesh(***xM,xM,zM***)**– для построения сетчатых графиков;
* **surf(***xM,yM,zM***)**– для построения графиков со сплошной поверхностями.

Сформированные в виде векторов данные используются функциями:

• **contour(***xV,xV,zM***)** – для построения графиков с контурными

линиями;

• **plot3d(***xV,yV,zM***)** – для построения графиков точек,   
соединенных отрезкамипрямых и других линий.

Рассмотрим примеры использования перечисленных выше функций, для чего сформируем матрицу**zM(x,y)**с использованием встроенной внутренней функции**f(x,y)**(рис. 1.4.1-16).Результаты построения соответствующих графиков приведены на рис. 1.4.3-3 – 1.4.3-5

|  |
| --- |
| --> // ***Построение различных видов графиков функции двух переменных***  --> // zM = xM.^2 + 2 \* yM.^2  -->  -->// ***Формирование двух матриц*xM*и*yM**  --> [xM, yM] = meshgrid(-4 : 4, -4 : 4)  xM =  0. 1. 2. 3.  0. 1. 2. 3.  0. 1. 2. 3.  0. 1. 2. 3.  -->  yM =  -3. -3. -3. -3.  -2. -2. -2. -2.  -1. -1. -1. -1.  0. 0. 0. 0.  -->  --> // ***Формирование матрицы*zM**  -->deff('k = f(x, y)', 'k = x.^2 + 2 \* y.^2');  -->zM = f(xM, yM)  zM =  48. 41. 36. 33. 32. 33. 36. 41. 48.  34. 27. 22. 19. 18. 19. 22. 27. 34.  24. 17. 12. 9. 8. 9. 12. 17. 24.  18. 11. 6. 3. 2. 3. 6. 11. 18.  16. 9. 4. 1. 0. 1. 4. 9. 16.  18. 11. 6. 3. 2. 3. 6. 11. 18.  24. 17. 12. 9. 8. 9. 12. 17. 24.  34. 27. 22. 19. 18. 19. 22. 27. 34.  48. 41. 36. 33. 32. 33. 36. 41. 48.  -->  --> // ***Формирование векторов*xV*и*yV**  -->x2 = -4 : 4; y2 = -4 : 4;  -->  --> // ***Построение двумерных графиков***  -->scf(1); mesh(xM,yM,zM) // ***Сетчатыйграфик(окно 1)***  ***-->***scf(2);plot3d(xV, yV, zM***)***//***График точек, соединенных отрезками***  -->//***прямых(окно 2)***  -->scf(3);surf(xM,yM,zM) // ***График сплошной поверхности (окно 3***)  -->scf(4); contour(xV, yV, zM, 7) // ***График контурных линий (окно 4)*** |

Рис. 1.4.1-16. Построение различных видов графиков

Результатом выполнения функции ***mesh***является построение в окне 1***графика поверхности в виде сетки***(рис.1.4.1-17).

|  |
| --- |
|  |

Рис. 1.4.1-17 Результат выполнения в окне 1функции**mesh(xM,yM,zM)**

В результате выполнения функции**plot3d**(рис.1.4.3-17) происходит построение в окне**2*графика поверхности***, где точки соединены отрезками прямой, а соответствующая заливка квадратов делает изображение фигуры объемной(рис. 1.4.3-18). Функции**plot3d**имеет формат:

**plot3d(***xV,yV,zM***),**

где*xV*и *yV*–вектора целых чисел, указывающие диапазон изменения параметров функции;*zM*–матрица действительных чисел значений функции, описывающей поверхность **z(x,y)**.

|  |
| --- |
|  |

Рис. 1.4.3-18. Результат выполнения в окне 2функции**plot3d(xV,yV,zM)**

Функция **surf**выполняет построение в окне 3 графика***сплошной поверхности***(рис. 1.4.3-19) и имеет следующий формат:

**surf(***xM,yM,zM***),**

где*xM*и *yM*–массивы целых чисел, указывающие диапазон изменения параметров функции;*zM*–матрица действительных чисел значений функции, описывающей поверхность ***z*(*x,y*)**.

|  |
| --- |
|  |

Рис. 1.4.1-20 Результат выполненияв окне 3функции **surf(xM,yM,zM)**

Графикфункции двух переменных может быть представлен в виде   
***контурных линий***(контурные линии – это линии, в каждой точке которых значение функции одинаково). Для этого в Scilab используется функция contour, имеющая формат:

**contour(***xV,yV,zM,n***),**

где*xV*и*yV*–массивыцелых чисел, указывающие диапазон изменения параметров функции;*zM*–матрица действительных чисел значений функции, описывающей поверхность **z(x,y)**;*n*–целочисленныйпараметр, который устанавливает количество контурных линий.

Функция**contour**позволяет не только построитьв окне 4 график контурных линий, но и нанести в отдельных точках контурных линий значения функции (рис. 1.4.1-21). В этом примере получен график из 7-ми контурных линий, на каждой из которых отмечены значения функции.

|  |
| --- |
|  |

Рис. 1.4.1-21 Результат выполнения в окне 4 функции**contour(xV,yV,zM,7)**

С использованием функции**mtlb\_hold('on'**)(эта функция уже была применена раньше при построении графиков функции от одной переменной) в одном графическом окне можно построить несколько графиков функций от двух переменных. Например, построим в одном окне графики двух функций (рис.1.4.1-22).

|  |
| --- |
|  |
|  |

Рис.1.4.1-22 Построение двух графиков в одном графическом окне

С использованием функции**subplot**в одном графическом окне можно расположить несколько графиков. В этом случае перед построением каждого графика выполняется функция**subplot(n,m,k)**,в которой указывается область построения текущего графика.

|  |
| --- |
|  |

Рис.1.4.1-23. Построения в одном окне 4-х графиков

На рис.1.4.1-23 приведены командные строки, позволяющие в одном окне произвести построение четырех графиков. Для этого графическое окно разбито на четыре области (две строки**n=2**и два столбца**m=2**). Последний параметр (**k**)указывает номер области построения текущего графика. Нумерация областей осуществляется по строкам, начиная с 1. Результат выполнения функций **subplot**приведен на рис. 1.4.3-24.

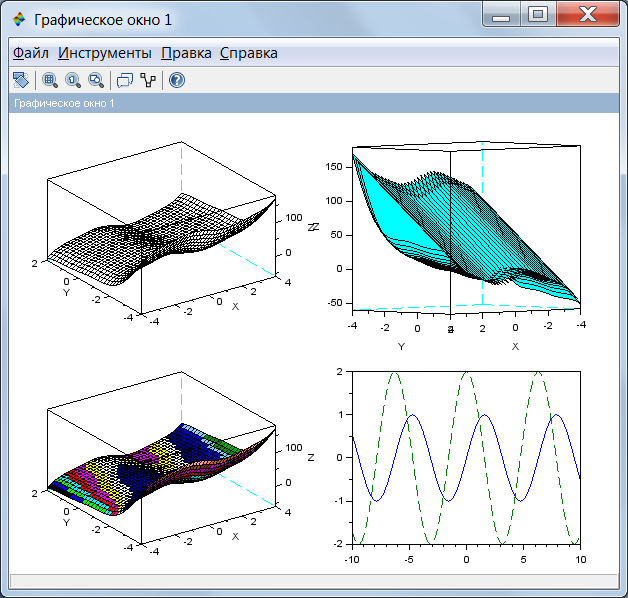


Рис. 1.4.1-24. Построение 4-х графиков в одном графическом окне

Для построения графика в полярных координатах применяется функция**polarplot**, имеющая в простейшем случае два обязательных параметра:

**polarplot(***phi,ro***),**

где:*phi*–диапазон значений угла;*ro*–функция от полярного угла.

На рис.1.4.1-25 приведен пример построения в полярных координатах графика функции**ro=4\*cos(3\*fi)**, где **fi**(полярный угол) изменяется на интервале**[0;2π]**с шагом**0.01**.

|  |  |
| --- | --- |
| --> // РИС14125: ***Построение графика в полярной системе координат***  -->  --> fi=0:0.01:2\*%pi;  -->polarplot(0:0.01:2\*%pi,4\*cos(3\*fi)) | |
|  |  |

Рис.1.4.1-25 Построение графика функции в полярнойсистеме координат

|  |  |
| --- | --- |
| -->// РИС14126:  -->// ***остроение 2D-гистограммы***  -->  -->hist3d(10\*rand(10,10)); |  |

Рис.1.4.1-26 Построение 2D-гистограммы в виде 2D-графика

с помощью функции**hist3d**

Подробную информацию о рассмотренных выше графиках и других возможностях визуализации вычислений можно получить в справочной системе пакета Scilab. Если вас интересует описание и примеры использования функции, название которой вы уже знаете, то достаточно в командной строке ввести, например,**helphist3d**,или ввести имя функции в окне**Справочная система**.

В***Приложении 1.4, табл. 1.4.1-2*** представлены описания всех графических функций, используемых в этом разделе. Более полную справочную информацию о графических средствах и функциях Scilabможно получить на сайте**scilab.org**[13].

*Глобальное свойство графических объектов*

Свойство графических объектов – ***GlobalProperty*** (*ГлобальноеСвойство*) – это необязательный аргумент, который может использоваться в графических функциях и позволяет осуществить глобальную настройку построения всех новых линий и поверхностей. Он может быть указан в**plot** или **surf** виде следующих пар:

'*ИмяСвойства*1',*ЗначениеСвойст*ва1,…,'*ИмяСвойства*n',*ЗначениеСвойства*n,

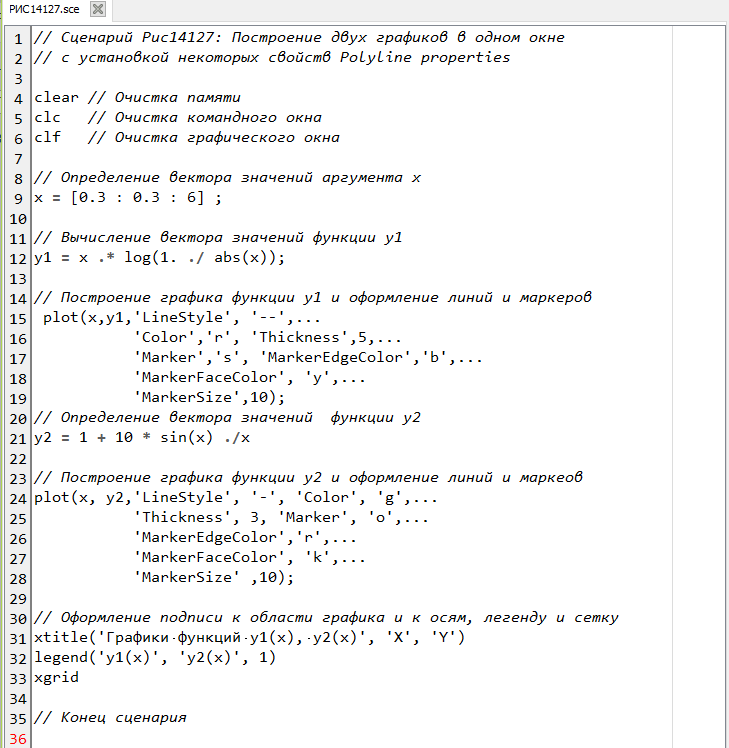
где:*ИмяСвойства***n** должно быть строкой, не чувствительной к регистру и определяющей то свойство, которое необходимо установить, а   
*ЗначениеСвойства***n** может быть вещественным, целочисленным или строкой (скаляром или матрицей) в зависимости от типа используемого свойства. Например, чтобы определить красную (цвет) штрихпунктирную линию с длинными штрихами (стиль линии) с маркерами в виде ромба (маркер), последовательность должна быть следующей:

**'Color','red', 'LineStyle','-.', 'Marker','diamond'**.

В *Приложении 1.4.1, табл.1.4.1-3* приведен полный список   
*ИмёнСвойств*и их возможные значения – ЗначенияСвойства, которые можно установить при использовании [**plot**](https://help.scilab.org/docs/6.0.2/ru_RU/plot.html) или [**surf**](https://help.scilab.org/docs/6.0.2/ru_RU/surf.html)**.**

***Иногда можно использовать два ИмениСвойства, соответствующих одному свойству: первое из них используется по умолчанию в Matlab, а второе имя по умолчанию используется в Scilab (например, Color или Foreground для линии).***

На рис. 1.4.1-27 рассмотрен пример, в котором для построения двух графиков функций в одном графическом окне учтены и прокомментированы все описанные выше свойства графических объектов. Обратите внимание, что перед построением графика проведена очистка окна***Обозревателя переменных***, ***Командного окна***и***Графических окон***, а установка свойств графика (тип и ширина линии, тип точки (маркер) размер шрифта и др.) выполнена с использованием параметров функции **plot**, что является альтернативой средств графического редактора.





**--> exec("РИС14127.sce", 0)**



Рис.1.4.1-27. Построение и отображение графиков двух функций   
с использованием графических средств Scilab

*Примеры построения содержательных графиков*

Графические средства Scilabиспользуются для изучения многих дисциплин [14]. Приведем несколько примеров, иллюстрирующих построение графиков с использованием описанных выше функций, при изучении дисциплины «Теория электрических цепей» [15, 16, 17].

***Построить два периода графика синусоидального напряжения***

**B**,***с амплитудой*Um=5 В**, ***угловой частотой*ω=4рад/с*и сначальной фазой***φ=300(рис.1.4.1-28).



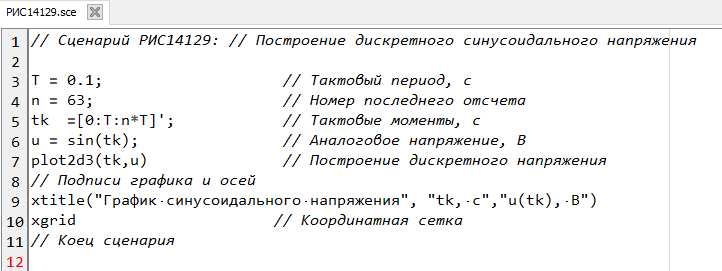


**--> exec("РИС14128.sce", 0)**

|  |
| --- |
|  |

Рис.1.4.1-28. Построение графика синусоидального напряжения

***Построить график дискредитированного синусоидального напряжения*B.*тактовый период примем равным*T=0.1с, n=[0…63]**(рис.1.4.1-29).

****



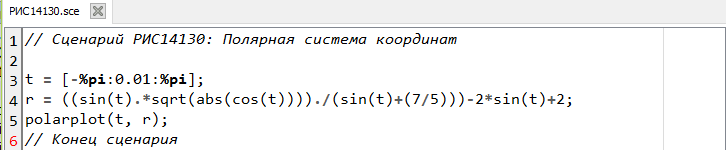
**--> exec("РИС14129.sce", 0)**

|  |
| --- |
|  |

Рис.1.4.1-29. Построить график дискредитированного синусоидального напряжения

***Построить в полярной системе координат график функци***и

-2sin(t)+2 (рис.1.4.1-30).



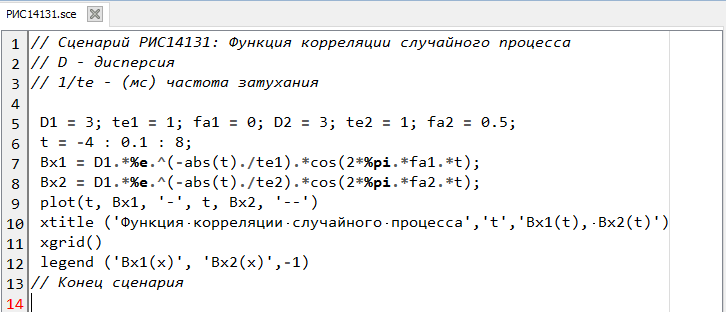
**--> exec("РИС14128.sce", 0)**

|  |
| --- |
|  |

Рис. 1.4.1-30 Построение графика функции в полярной

системе координат

Построить график функции корреляции случайного процесса  
(рис.1.4.1-31).





**--> exec("РИС14131.sce", 0)**

|  |
| --- |
|  |

Рис. 1.4.1-31 Построение графика функции корреляции случайного процесса

|  |
| --- |
| **--> exec("РИС14132.sce", 0)** |

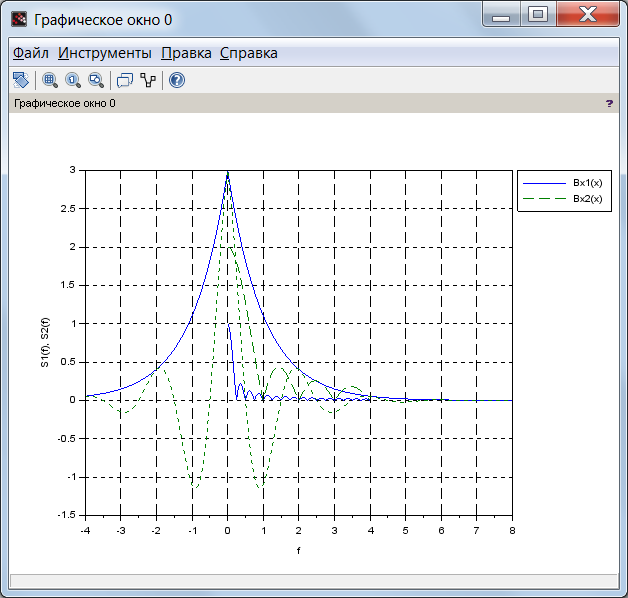


Рис. 1.4.1-32 Построение графика функции амплитудного

спектра частотпрямоугольного импульса

***Построить график сигнала амплитудной модуляции с одним тоном***(рис.1.4.1-33).

|  |
| --- |
| **-->**exec("РИС14133.sce", 0) |

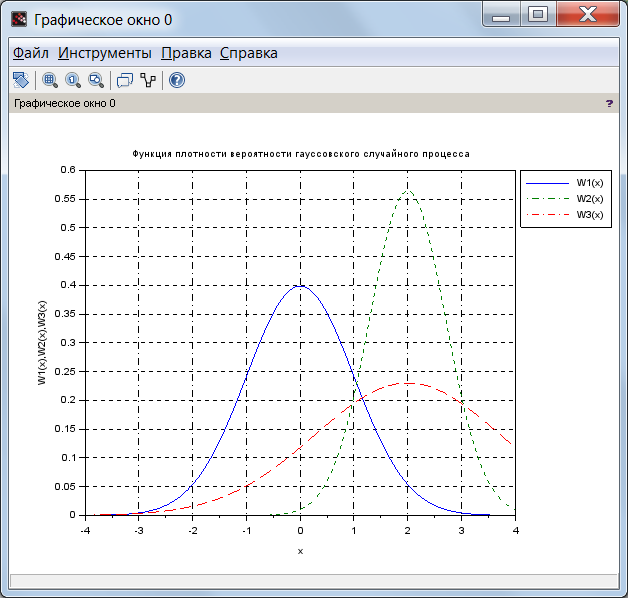


Рис. 1.4.1-33 Построение графика функции плотности

вероятности случайного процесса

*Построение простых* [*геометрическихформ*](https://help.scilab.org/docs/6.0.2/ru_RU/section_74e647d5238fd5f5b4d8fd151d76e61e.html)

Кроме функций, представленных как аналитически, так и таблично иногда необходимо нарисовать в графическом окне простейшие геометрические фигуры. Это легко можно реализовать функцией plot.

В качестве содержательного примера рассмотрим следующую задачу:

Построить в графическом окне графическую фигуру ромб, изображенную на рис. 1.4.1-34. Затем задать координаты нескольких точек, а затем отметить их попадания в заданную область красной звездочкой и непопадания их в заданную область черной точкой.



Рис. 1.4.1-34.Ромб в системе координат x-y

Проведем формализацию поставленной задачи:

Для этого запишем уравнение прямых – **AB, BC, CD, DA:**

**AB: x-y=-1; BC: x+y=1; CD: x-y=1; DA: x+y=-1.**

Для того, чтобы точка**M**с координатами**(x,y)**принадлежала замкнутой заштрихованной области, необходимо выполнить следующие условия:

**.**

На Рис. 1.4.1-35 приведено решение поставленной задачи с помощью функции**Fromb**и ее решение.

|  |
| --- |
| exec("РИС14135.sce", 0) |
|  |

Рис. 1.4.1-35. Пример построение фигуры ромб

В Scilabимеются высокоуровневые средства для построения простейших геометрических фигур:

* **arg\_** [**properties**](https://help.scilab.org/docs/6.0.2/ru_RU/arc_properties.html)— описание свойств объекта дуги;
* [**rectangle properties**](https://help.scilab.org/docs/6.0.2/ru_RU/rectangle_properties.html)—описание прямоугольника свойство сущности;
* [**segs\_properties**](https://help.scilab.org/docs/6.0.2/ru_RU/segs_properties.html)— описание свойств объекта сегментов;
* [**xarc**](https://help.scilab.org/docs/6.0.2/ru_RU/xarc.html)—рисование части эллипса;
* [**xarcs**](https://help.scilab.org/docs/6.0.2/ru_RU/xarcs.html)—рисованиечасти набора эллипсов;
* [**xarrows**](https://help.scilab.org/docs/6.0.2/ru_RU/xarrows.html) — рисованиенабор стрелок;
* [**xfarc**](https://help.scilab.org/docs/6.0.2/ru_RU/xfarc.html)—заливка части эллипса;
* [**xfarcs**](https://help.scilab.org/docs/6.0.2/ru_RU/xfarcs.html)—заливка частей набора эллипсов;
* [**xfrect**](https://help.scilab.org/docs/6.0.2/ru_RU/xfrect.html)— заливка прямоугольника;
* [**xrect**](https://help.scilab.org/docs/6.0.2/ru_RU/xrect.html)—рисованиепрямоугольник;
* [**xrects**](https://help.scilab.org/docs/6.0.2/ru_RU/xrects.html)—рисованиеили заливки набора прямоугольников;
* [**xsegs**](https://help.scilab.org/docs/6.0.2/ru_RU/xsegs.html)—рисованиенесвязанных сегментов.

### **1.4.2. Низкоуровневая графические средства и основные графические объектов**

*Иерархия графических объектов*

В Scilab графические окна и их содержимое представлены иерархическими объектами. Верхний уровень иерархии– это**figure**(Графическое окно).

  Каждое графическое окно (**figure)** определяет по меньшей мере одного потомка – объект**axes**(Оси).  В свою очередь каждый объект axes содержит набор наследуемыхобъектов двух типов. Первый тип представляет объекты, которые являются основными графическими объектами для отображения **2D** и **3D**графиков и различных геометрических фигур, например, **polylines**(Ломанные линии),**rectangles** (Прямоугольники),**arcs** (Дуги), **segs** (Сегменты),**compound**(Склеивание, Соединение) и другие [13]. Второй тип представляет объекты для построения графического интерфейса пользователя (GUI), к которым в первую очередь относится объект **uicontrol**, позволяющий создавать графические интерактивные компоненты такие как ползунки, числовые поля, изменяемые кнопкой мыши, редактируемые таблицы, редактируемые текстовые поля в графических элементах, флажки, группы кнопок, выпадающие списки, выпадающие меню кнопки и др. Все эти объекты являются прямыми потомками объекта**figure**, в которой они определены. Их свойства могут быть установлены и настроены через их указатели, как и для других графических объектов.

Главное в низкоуровневом графическом режиме заключается в том, чтобы сделать обмен свойствами графических объектов проще и эффективно. Этот графический режим имеет набор графических функций низкого уровня, используемых для управления свойствами объектов, такими, как данные, координаты и масштабирование, цвет и вид без необходимости повторять команды инициализации графики.

Все графические объекты, в том числе и figure, связаны между собой переменными Scilab, имеющими тип **handle**(Дескриптор). Дескриптор – это уникальный идентификатор (указатель), который связан с каждым экземпляром созданного графического объекта. Используя этот дескриптор, можно добраться до свойств конкретного объекта используя функции **set** и **get**.

Как известно все графики в Scilab строятся в так называемых *графических окнах –***Figure**, каждое из которых имеет идентификационный номер, который является *дескриптором графического окна* (**handle**).

Рисование – это сложный процесс, который заключается в создании различных объектов в терминах объектно-ориентированного программирования. Например, сначала создается графическое окно – объект**Figure**, которое затем порождает потомков – объекты***Children***в виде объектов**Axes**(оси, которые отображаются на экране). У осей тоже есть потомки – линии графика функций, которые к ним привязываются.

Таким образом, каждое графическое окно в Scilab – это отдельный графический объект, обладающие большим набором свойств (Figureproperties), которые отвечают за его внешний вид, а также содержащий множество подчиненных объектов, наследующие его свойства. При построении графиков происходит неявное обращение к рабочей области графического окна, когда, например, отображаются оси, наносится сетка, помечаются точки или эти точки соединяются отрезками прямых линий и т.д. Для построения графиков в системе есть множество предопределенных высокоуровневых функций, которые выполняют всю работу в графическом окне (п. 1.4.1).

На втором уровне иерархии находятся графические объекты**Axes**(Оси), которые обладают свойствами (**Axes properties**), позволяющими изменять координаты и тип осей, а также значения параметров по умолчанию для создания объектов следующего уровня.

У каждого объекта**Axes** два потомка: объекты**Compound** (Составной   
объект) и **Legend** (Пояснения-Легенда).

Объект**Legend**представляет собой список графических объектов третьего уровня иерархии. Эти объекты обладают свойства(**Legendentityproperties**), которые указывают на тип линии, символы маркеров и их цвета.

Объект**Compound** (Составной объект) определяет взаимозависимости различных графических объектов и их глобальное свойство – отображение (видимость), в том числе объектов следующего уровня**Polyline** (Полилинии) – это ломанные линии отображающие графики, окружности, квадраты и др., которые устанавливают**2D** и **3D** свойства(**Polyline properties**) рисования графиков, фигур и поверхностей.

Например, одно из свойств**Polyline properties** – **Visible**, устанавливающее видимость графика, принимает значение **on** или **off.** А, например, другое свойство – **Thickness**, указывающее ширину линии в пикселях, принимает значение положительного числа.

*Редактор объектов*

Свойства всех объектов можно установить, как программно с помощью соответствующих функций, так и с помощью**GraphicEditor**(Графического Редактора). Сам**GraphicEditor**также может быть вызван и программно и из графического окна. На рис 1.4.2-1 показаны две панели окна**GraphicEditor**. В одной панели –**ObjectsBrowse**(Браузер объектов) отображается иерархия объектов выбранной**Figure(1)**,в другой –**ObjectProperties**(**Свойства Объектов**) – значения соответствующих свойств.

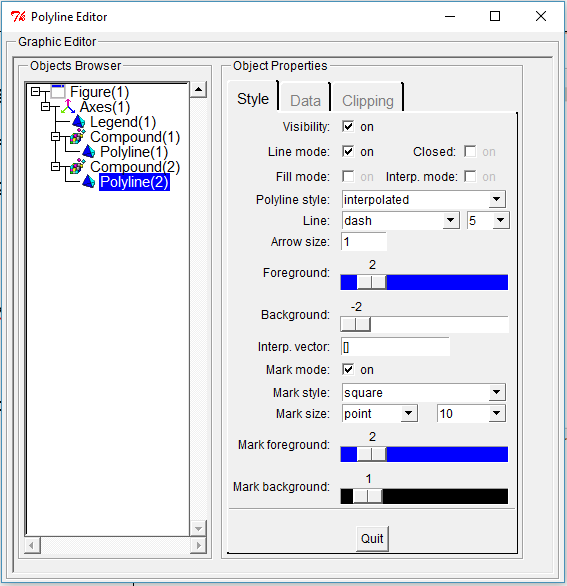


Рис. 1.4.2-1 Окно графического редактора

В панели **ObjectsBrowse**отображаются те объекты, значения свойств которых можно редактировать: **Figure, Axes, Compound**и другие.

Например, выбрав объект **Figure**, можно поменять название графика, его размещение на экране (**X position=0**, **Y position=0** – левый верхний угол); размеры графика, сжав его по любой из осей; задать цвет заднего плана, поменять цветовую гамму.

Выбрав, например, редактирование осей **Аxes,** можно редактировать каждую из осей **X, Y, Z**: задать название оси; цвет (при этом цвет меняется у названия оси, сетки и разметки оси); задать размещение названия для оси **X** (наверху, внизу или посередине) и для оси **Y** (слева, справа или посередине). Кроме того, можно задать цвет метки, угол поворота, ее размер, фон, расстояние между линиями сетки, установить насечку между линиями и другое.

При последовательном выборе Figure**,** Axes **и** Compound, активизируются соответствующий объект **Polyline**, в котором можно менять вид и размеры маркеров, а также аппроксимировать наш график различными кривыми.

*Графические объекты и их свойства*

Для создания графического окна можно использовать функцию**figure:**

*h***=figure**(*id*);

*h****=*figure***("*Свойство**1***","Значение***1***",..., "Свойство***n*"*** *,"*Значение**n*"****),*

где: *id*–необязательный целочисленный параметр, задающий номер окна;

*h* – переменная типа**handle**, которая связана с набором свойств этогоокна и их значениями;

*Свойство***n**–название n-го свойства; *Значение****n***– значение**n**-госвойства и т.д.

Объекты всех уровней имеют определенные свойства. Так, например,

свойство**BackgroundColor**– определяет цвет фона**figure**и может принимать соответствующие*Значение* в виде вектора **1×3** , элементы которого определяют цвет: красный, зеленый и синий, т. е.**[R,G,B]** ;**Figure\_name**– строка, позволяющая задать заголовок окна;**Position**– определяет расположение**figure** на экране и может принимать соответствующее*Значение*в виде вектора**1×4**, элементы которого определяют геометрический аспект фигуры   
[**XY ширина высота**], где координаты **x**и**y**определяют расположение верхнего левого угла;

Например, оператор

**f=figure('Figure\_name','МТУСИ', 'position',[10 100 300 200]);**

создает графическое окно сименем МТУСИ, показанное на рис.1.4.2-2.

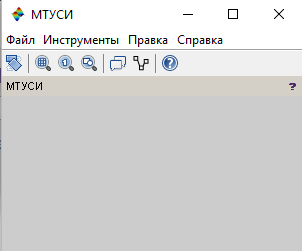


Рис. 1.4.2-2 Графическое окно с именем МТУСИ

Причем *Указатель* (Дескриптор) графического окназаписывается впеременную**f** тип **handle**. Размер и положениеокна на экране компьютера можно задать с помощью параметра:

**'position', [***x y dx dy***],**

где: *x, y* – положение верхнего левого угла окна (по горизонтали и вертикалисоответственно относительно верхнего левого угла экрана);

*dx* – размер окна по горизонтали ширина окна в пикселях;

*dy*– размер окна по вертикали высота окна в пикселях.

Параметры объектов можно задавать одним из двух способов:непосредственно при создании графического окна (рис. 1.4.2-2) или после создания графического окна с помощью функции**set**, которая имеет следующие форматы:

**set(***'Свойство', 'Значение****'*);**

**set(***h****,*'***Свойство***');**

**set(***h,'Свойство','Значение'***);**

**(***handlePath*, *'Свойство1','Значение1',…,'Свойствоn','Значениеn'***);**

*h.Свойство****=****'Значение'*;

где: *handlePath*– символьная строка, содержащая путь, указывающий на графический объект. Этот путь состоит из свойства графической объектов всех его родителей.

Функция**set**используется для изменения значения указанного свойства графичесих объектов или объектов GUI. В этом случае точечная нотация при обращении к свойству эквивалентна использования функции**set**. Например,

**set(h,"background",3)**эквивалентно **h.background=3.**

Имена свойств – это символьные строки, а тип набора значений этих свойств зависит от типа соответствующего свойства.

Просмотреть списка всех существующих свойств  можно в документации Scilabна страницах [**graphics\_entities**](https://help.scilab.org/docs/6.0.2/ru_RU/graphics_entities.html)**для** графических объектови [**uicontrol**](https://help.scilab.org/docs/6.0.2/ru_RU/uicontrol.html) для объектов пользовательского интерфейса.

 Формат функции**set**может быть следующими:

**set**('*current\_entity*',*h*),

который устанавливает новый объект с дескриптором *h*как текущий;

**set**('*current\_figure',fig*),

который устанавливает новый графический рисунок как текущий (этот формат эквивалентен функции[**scf**](https://help.scilab.org/docs/6.0.2/ru_RU/scf.html));

**set**('*current\_axes',axes*),

который  задает новый объект оси как текущий (этот формат эквивалентен функции[**sca**)](https://help.scilab.org/docs/6.0.2/ru_RU/sca.html).

При работе с графическими объектами можно также использовать следующие функции:

* [**scf**](https://help.scilab.org/docs/6.0.2/ru_RU/scf.html) –установить текущий рисунок(окно);
* [**delete**](https://help.scilab.org/docs/6.0.2/ru_RU/delete.html) – удаление графического объекта и его дочерних элементов;
* [**copy**](https://help.scilab.org/docs/6.0.2/ru_RU/copy.html)– копирование графического объекта;
* [**move**](https://help.scilab.org/docs/6.0.2/ru_RU/move.html)–перемещение, перевод, графического объекта и его дочерних

элементов;

* [**gdf**](https://help.scilab.org/docs/6.0.2/ru_RU/gdf.html) –возвращение дескриптора по умолчанию фигура;
* [**gca**](https://help.scilab.org/docs/6.0.2/ru_RU/gca.html) –возвращение дескриптора текущих осей;
* [**gce**](https://help.scilab.org/docs/6.0.2/ru_RU/gce.html) –получение дескриптора текущего объекта;
* [**get**](https://help.scilab.org/docs/6.0.2/ru_RU/get.html) –получение значения свойства графических объектов или объект

пользовательского интерфейса;

* [**set**](https://help.scilab.org/docs/6.0.2/ru_RU/set.html) –установка значения свойства графических объектов или

объектов пользовательского интерфейса.

**1.4.3. Построение графических интерфейсов  
пользователя (GUI)**

***Графический интерфейс пользователя***

В Scilab, так же, как и в других программных системах, можно создаватьWindows-приложения, которые состоят из различных оконных интерфейсов и их компонентов для реализации ввода/вывода данных или любой другой графической информации. Средством для построения таких оконных интерфейсов в Scilabявляется ***графический интерфейс пользователя*** (**GUI**)

Простейшие предопределенные оконные интерфейсы (GUI) будут кратко описаны в п. 1.5.4. Подробно с предопределенными GUI[**waitbar**](https://help.scilab.org/docs/6.0.2/ru_RU/waitbar.html)**,** [**x\_choices**](https://help.scilab.org/docs/6.0.2/ru_RU/x_choices.html)**,** [**x\_choose**](https://help.scilab.org/docs/6.0.2/ru_RU/x_choose.html)**,** [**x\_dialog**](https://help.scilab.org/docs/6.0.2/ru_RU/x_dialog.html)**,** [**x\_matrix**](https://help.scilab.org/docs/6.0.2/ru_RU/x_matrix.html)**,** [**x\_mdialog**](https://help.scilab.org/docs/6.0.2/ru_RU/x_mdialog.html)можно ознакомится настраницеScilab по GUI.

Далее рассмотрим средства Scilab для построения графических интерфейсов, разрабатываемых пользователем для решения своих задач.

***Построение графического интерфейса пользователя***

Функция Scilab**uicontrol** позволяет отображать в графическом окне интерактивные компоненты **(Style) GUI**такие как ползунки, числовые поля, изменяемые кнопкой мыши, редактируемые таблицы, редактируемые текстовые поля в графических элементах, флажки, группы кнопок, выпадающие списки, выпадающие меню кнопки и др. Эти объекты являются прямыми потомками графического окна, в которой они определены. Их свойства (допустимые значения) могут быть установлены и настроены как в функции **uicontrol**, так и в функции **set** через их указатели также, а также с помощью точечной нотации, как и для других графических объектов.

Функция **uicontrol** имеет следующие форматы:

*h****=*uicontrol***("Компонент***1***","Значение***1***",...,"Свойство***n*"****,"*Значение**n*"****);*

*h****=*uicontrol***(h,"Компонент***1***","Значение***1***",...,"Свойство***n*"****,"*Значение**n*"****);*

где: *Компонент***n**– название **n**-го компонента**GUI**;

*Значение***n**– значение **n**-го компонента**GUI**;

*h* – переменная типа**handle**, которая связана с

соответствующей компонентой и ее набором свойств;

Эта функция устанавливает свойства различных компоненты**(Style)**и их значенияграфического окна. Естественно предварительно должно быть задано свойство Style, значениями которого могут быть допустимые имена компонет. Если дескриптор задан, как первый параметр (*h*), то **uicontrol**создает новый элементыGUI, определяемый дескриптором –*h*. Если дескриптор не дается, то**uicontrol** создается на текущем рисунке (который может быть получен с помощью вызова к функции **gcf**).

Затем при создании элемента управления, устанавливаются свойства, заданные в качестве соответствующих значений. Это эквивалентно созданию с помощью**uicontrol** какого-либо компонента, а затем установка его свойств с помощью функции**set**.

Таким образом, Scilab и все графические объекты (компоненты) связывают между собой через механизм свойства. Таким образом, чтобы создать адаптированный**uicontrol**, нужно знать допустимые значения используемых свойств, причем всем неиспользуемым свойствам присваиваются значения по умолчанию.

Описание всех доступных компонентов и их свойств приведены на странице справки по **uicontrol**и **uicontrol\_properties**[]. *В Приложении 1.4.3,  
табл. 1.4.3-1* перечислены наиболее используемые компоненты **GUI** и их свойства.

Рассмотрим в качестве примера, создание в графическом окне с именем МТУСИ компоненты**Командная кнопка** (**PushButton)**. По умолчанию эта кнопка не снабжается никакой надписью, имеет серый цвет и располагается в левом нижнем углу фигуры**figure**.

Модифицируем код создания кнопки, задав дополнительно значения следующих свойств:

* месторасположение и заголовок окна;
* надпись на кнопке;
* месторасположение кнопки.

Текст программного кода приведен на рис. 1.4.3-1, а справа на этом же рисунке можно увидеть графическое окно, которое было получено в результате работы соответствующего программного кода.

|  |  |  |
| --- | --- | --- |
| Этапы | Программный код | Графическое окно |
| Создание  **figure** с  именем МТУСИ и **pushbutton**с именем  Кнопка1 | *// Создать окно*  f=figure();  *// Определить положение окна*  set(f,'position', [0,0,220,100])  *//Определитьимяокна*  set(f,'figure\_name','**МТУСИ**');  *// Создать кнопкуc*  *// надписью –* **Кнопка1***и определить*  *// еепозициюна*figure**МТУСИ**  B1=uicontrol('style','pushbutton',…  'string','Кнопка1',…  'position',[50,50,100,20]); |  |
| Создание  **listbox** | L1=uicontrol(f,'style','listbox',  'position', [50 20 100 20]); |  |
| Определение  **Callback**  при нажатии на кнопке | B1=uicontrol('Style', 'Pushbutton',  'String', 'Кнопка1',  'Callback', 'Fun1'); | Переход на выполнение  функции Fun1 |
| Выполнение функции Fun1 | Function Fun1()  set(h,'string',sprintf("%s\n",…  'Привет!))  endfunction |  |

Рис. 1.4.3-1. Этапы создание простейшего GUI

При щелчке на кнопку вокруг ее надписи появляется пунктирный прямоугольник, свидетельствующий о том, что кнопка «находится в активном состоянии». Щелчок за пределами поверхности кнопки выведет ее из активного состояния, и пунктирная рамка пропадет. Основное назначение командной кнопки – является вызов функции, реагирующей на щелчок по кнопке.

Щелчок по кнопке генерирует событие**Callback**,которое указывается как имя свойства функции**uicontrol**. Значением свойства**Callback**является строка с именем функции – **Fun1**, вызываемой при щелчке по кнопке.

Рассмотрим пример создания графического интерфейса для исследования функции **f(x)=x2** на отрезке **[1;2]**, для чего выведем на экран с шагом **h=1** ее значения, а также значения ее первой (**f1(x)**) и второй (**f2(x)**) производных.

На рис. 1.4.3-2 представлен сценарий, в котором создается графический интерфейс, предназначенный для решения следующей задачи:

Создать приложение, которое при задании исходных данных из графического окна GUI и выполнения функции по вычислению значений функций и ее первой и второй производных, выводит в тоже самое окно. Причем вGUIнеобходимо предусмотреть заголовки полей ввода исходных данных, общие и частные заголовки для полей вывода результатов и две кнопки, при нажатии на которые происходит, в соответствии с их назначением, выполнение указанных функций или очистка числовых полей.

*// Сценарий РИС14302: Создание графического интерфейса пользователя*

*// для ввода исходных данных и*

*// вывода результатов работы созданного приложения*

clear*// очистка памяти*

clc*// очистка командного окна*

*// Создание итерфейса пользователя*

*// Создать графическое окно figure*

t**=**figure('position',[20,40,1110,290],...

'figure\_name','Пример создания интерфейса формы в Scilab');

*// Текстзаданияварианта*

zadanie**=**uicontrol('Style','text','Position',[20,220,890,50],...

'FontSize',28,...

'BackgroundColor',[111],...

'HorizontalAlignment','center',...

'String',' Таблицазначений x, f(x), f1(x) и f2(x) ');

*// СоздатьобъектКнопка: Вычислить*

bt1**=**uicontrol('style','pushbutton','string','Вычислить',...

'position',[20,120,90,40],'CallBack','Y');

*// Создать объект Кнопка: Очистить*

bt2**=**uicontrol('style','pushbutton','string','Очистить',...

'position',[20,20,90,40],'CallBack','C');

*// Создать объек Текстовое поле: X*

text1**=**uicontrol(t,'style','text','String','',...

'position',[140,180,150,20],...

'String',' X ');

h1**=**uicontrol(t,'style','listbox','position',[14020150160]);

*// Создать Текстовое поле: f(x)*

text2**=**uicontrol(t,'style','text','string','',...

'position',[310,180,150,20],...

'String',' f(x) ');

*// Создать объект listbox: f(x)*

h2**=**uicontrol(t,'style','listbox','position',[31020150160]);

*// Создать Текстовое поле: f1(x))*

text3**=**uicontrol(t,'style','text','string','',...

'position',[480,180,200,20],...

'String',' f1(x) ');

*// Создать объект listbox: f2+x)*

h3**=**uicontrol(t,'style','listbox','position',[48020200160]);

*// Создать Текстовое поле: f2(x))*

text4**=**uicontrol(t,'style','text','string','',...

'position',[700,180,200,20],...

'String',' f2(x) ');

*// Создать объект listbox: f2(x))*

h4**=**uicontrol(t,'style','listbox','position',[70020200160]);

*// Текстовоеполе, Вводисходныхданных*

text5**=**uicontrol(t,'style','text','string','',...

'position',[920,240,150,20],...

'String',' Ввод исходных данных');

*// Полеввода a*

texta**=**uicontrol(t,'style','text','string',...

'a','position',[920,170,40,20]);

edit\_1**=**uicontrol(t,'style','edit','string','1',...

'position',[950,170,150,20]);

sa1**=**get(edit\_1,'string');*// Считывание a и преобразование в строку*

sa1**=**evstr(sa1);*// Преобразование a к типу double*

*// Полеввода b*

textb**=**uicontrol(t,'style','text','string','b',...

'position',[920,140,40,20]);

edit\_2**=**uicontrol(t,'style','edit','string','5',...

'position',[950,140,150,20]);

sb1**=**get(edit\_2,'string');*// считывание b и преобразование в строку*

sb1**=**evstr(sb1);*// преобразование b к типу double*

*// Полеввода h*

texth**=**uicontrol(t,'style','text','string',...

'h','position',[920,110,40,20]);

edit\_3**=**uicontrol(t,'style','edit','string','1',...

'position',[950,110,150,20]);

*// Поле ввода исходной фукции f*

textf**=**uicontrol(t,'style','text','string','f',...

'position',[920,80,40,20]);

edit\_f**=**uicontrol(t,'style','edit','string','x^3',...

'position',[950,80,150,20]);

*// Поле ввода первой производной f1 функции f*

textf1**=**uicontrol(t,'style','text','string','f1',...

'position',[920,50,40,20]);

edit\_f1**=**uicontrol(t,'style','edit','string','2\*x^2',...

'position',[950,50,150,20]);

*// Поле ввода второй производной f2 функции f*

textf2**=**uicontrol(t,'style','text','string','f2',...

'position',[920,20,40,20]);

edit\_f2**=**uicontrol(t,'style','edit','string','4\*x',...

'position',[950,20,150,20]);

**functionC**()

*// ФункцияСочищаетвсе listbox*

set(h1,'string'," ");*// очистка listbox*

set(h2,'string'," ");*// очистка listbox*

set(h3,'string'," ");*// очистка listbox*

set(h4,'string'," ");*// очистка listbox*

**endfunction**

**functionY**()

*// Функция Y вычисляет и выводит в графическое окно значения x, f, f1, f2*

sa1**=**get(edit\_1,'string')*// считывание a и преобразование в строку*

aa**=**strtod(sa1)*// преобразование a к типу double*

sb1**=**get(edit\_2,'string')*// считывание b и преобразование в строку*

bb**=**strtod(sb1)*// преобразование b к типу double*

sh1**=**get(edit\_3,'string');*// считывание h и преобразование в строку*

hh**=**strtod(sh1);*// преобразование h к типу double*

sf**=**get(edit\_f,'string')*// считывание f и преобразование в строку*

sf1**=**get(edit\_f1,'string')*// считывание f1 и преобразование в строку*

sf2**=**get(edit\_f2,'string')*// считывание f2 и преобразование в строку*

n**=**int((bb**-**aa)**/**hh)**+**1

*//n=5; x=1; aa=1; bb=5, hh=1 // Значения по умолчанию*

x**=**aa;

fori**=**1:n

arg(i)**=**x

fx(i)**=**evstr(sf);f1x(i)**=**evstr(sf1);f2x(i)**=**evstr(sf2);

x**=**x**+**hh

end

*//Вывод полученных данных*

set(h1,'string',sprintf("%1.2f\n",arg))*// Выводв 1 listbox*

set(h2,'string',sprintf("%4.8f\n",fx))*// Выводво 2 listbox*

set(h3,'string',sprintf("%4.8f\n",f1x))*// Выводв 3 listbox*

set(h4,'string',sprintf("%4.8f\n",f2x))*// Выводв 4 listbox*

**endfunction**

*// Конец сценария*

Рис. 1.4.3-2. GUI для вывода таблицы значений функции   
и ее производных и результат работы созданного приложения

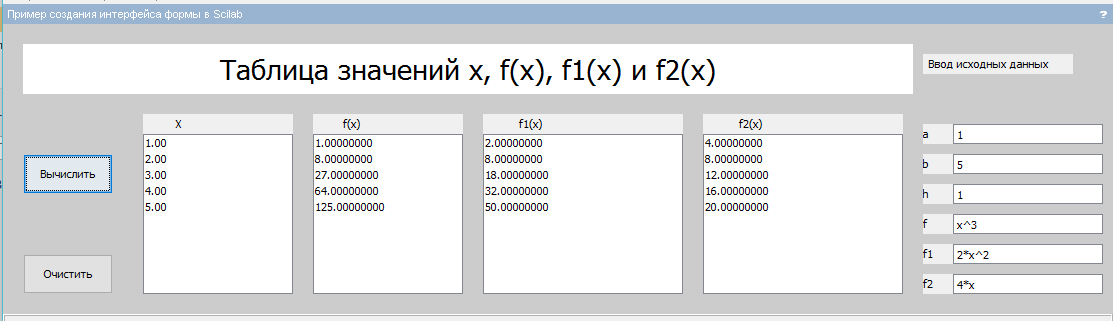


Рис. 1.4.3-2.GUI для вывода таблицы значений функции   
и ее производных и результат работы созданного приложения

1.4.5 Контрольные вопросы

1. Какие способы представления графиков используются в Scilab?
2. Что такое графическое окно Scilab?
3. Что такое дескриптор графического окна?
4. Какое назначение имеет функция**scf(n)**?
5. Какой номер у первого графического окна?
6. Можно ли создать графическое окно с определенным номером?
7. Какое назначение имеют элементы меню графического окна:***Файл***, ***Инструменты***, ***Правка***, ***Справка***?
8. Каким образом можно закрыть графическое окно?
9. Какое назначение имеет функция **xdel(id*)***?
10. Какое назначение имеет команда **winsid**?
11. Как выполнить настройку элементов графики (оси, толщина линий, шрифты и т.д.)?
12. Какие графические объекты имеются в Scilab?
13. Какова иерархия графических объектов?
14. Что представляет собой графический объект**Legend**?
15. Что определяют свойства графических объектов?
16. Что такое графический редактор?
17. Какие инструкций для установки цвета и типа линии известны?
18. Для чего используются функции **plot2d2, plot2d3**и**plot2d4**?
19. Можно ли с использованием одной функции**plot**построить несколько графиков?
20. Можно ли в одном графическом окне построить несколько графиков одного типа?
21. Можно ли в одном графическом окне построить несколько графиков разных типов?
22. Каким образом в операторе указывается цвет и символ отображения графика?
23. Что такое пояснительная информация графического окна?
24. Назначение и формат функций **xtitle**и **legend**?
25. Каково назначение функции**mtlb\_hold**?
26. Каково назначение функции **meshgrid**?
27. Какие типы графиков позволяют строить встроенные функции: **plot**,**contour**,**surf**и**plot3**?
28. Что такое контурные линии, и на каком графике их можно увидеть?
29. Что такое глобальное свойство графических объектов?
30. Для чего используется Редактор объектов?
31. Какая функция используется для создания графического окна?
32. Что такое дескриптор?
33. Для чего используются функции [**get**](https://help.scilab.org/docs/6.0.2/ru_RU/get.html) и **set?**
34. Какие предопределенные оконные интерфейсы известны?
35. Для чего используется функция **uicontrol?**

**1.5. Средства программирования Scilab**

**1.5.1. Общие понятия идеологии   
программирования**

***Программирование*** – процесс ***создания компьютерных программ с помощью языков программирования.***

В узком смысле программирование рассматривается как кодирование – реализация одного или нескольких взаимосвязанных алгоритмов на языке программирования.

В широком смысле программирование – это процесс создания компьютерных программ, то есть разработка ***Программного Обеспечения*** (ПО), в нашем случае разработка ***Прикладных Программ*** (ПП). Причем один из основных этапов разработки ПП связан с написанием исходного кода проекта на языке программирования.

Рассмотрим средства разработки ПП, которые помогают разработчикам в решении их профессиональных задач.

Прежде всего, это ***Система программирования*** *–* это программная система для разработки ПП средствами конкретного языка программирования.

Современные системы программирования обычно предоставляют пользователям мощные и удобные средства разработки ПП. В них входят: средства создания и редактирования исходных текстов программ (исходного программного кода); трансляторы; [библиотеки стандартных](http://www.tomsk.ru/Books/informatica/theory/chapter6/1_6_18.html) процедур; [отладочные средства](http://www.tomsk.ru/Books/informatica/theory/chapter8/1_8_6.html), помогающие находить и устранять ошибки в программе и другие.

Создать ПП – означает ***реализовать алгоритм*** решения поставленной задачи, или иначе, ***представить алгоритм*** в виде понятных компьютеру указаний того, что и в какой последовательности необходимо делать. К сожалению, компьютеры не умеют понимать словесное описание алгоритма, поэтому необходимо превратить его в абсолютно точный набор инструкций (команд), которые однозначно будут интерпретироваться компьютером. Для этого и были созданы языки программирования. Причем процесс создания ПП прошел путь от программирования в инструкциях процессора (машинных командах), через программирование на низкоуровневых языках (языках ассемблера) до программирования на языках высокого уровня и специализированных программных систем.

Уровень в данном случае – это уровень машинного восприятия. Так, языки низкого уровня (ассемблер) по возможности приближены к компьютеру, что делает соответствующие программы особенно эффективными с точки зрения их быстродействия. Однако существенная проблема использования таких языков заключается в том, что разработчик ПП – прежде всего человек, и его способы восприятия информации весьма далеки от машинных, что чрезвычайно затрудняет написание элементов ПП на ассемблере. Подавляющее большинство ПП реализованы на языке программирования высокого уровня.

С языками программирования тесно связаны понятия *синтаксиса* и *семантики*:

* ***синтаксис*** – это набор правил построения фраз (конструкций) алгоритмического языка, позволяющий создавать осмысленные предложения на этом языке;
* ***семантика*** – это система правил истолкования отдельных языковых конструкций, которая и определяет смысловое значение предложений алгоритмического языка.

Язык программирования – это искусственный (формальный) язык, предназначенный ***для записи алгоритмов***. Язык программирования задается своим описанием и реализуется в виде специальных программ: ***трансляторов*** (***компиляторов*** или ***интерпретаторов*)**.

Под ***транслятором*** обычно понимают специальную систему программ, которая переводит исходный программный код, написанный на языке программирования, в последовательность машинных команд – команд процессора.

Трансляторы универсальных языков программирования высокого уровня, таких как C++, C#, VB и другие, обычно являются ***компиляторами***. Этим подчёркивается общепринятый для подобных языков режим трансляции, при котором вначале осуществляется перевод программного кода в двоичное представление, в так называемый хранимый в памяти компьютера объектный модуль, а лишь затем этот модуль передаётся на исполнение.

Другой способ трансляции, называемый ***интерпретацией***, который состоит в совмещении перевода и исполнения программного кода (в этом случае объектный модуль не хранится в памяти компьютера и его, соответственно, нельзя повторно использовать). Другими словами, интерпретация означает, что каждая инструкция программы распознается (транслируется в машинные команды) и тут же исполняется, а программные модули существуют лишь в виде исходных текстовых файлов. В этом случае для выполнения программного кода необходима среда системы программирования.

Уже в самом начале развития методов и средств разработки ПП стал применяться простой и эффективный приём выделения часто используемых алгоритмов в самостоятельные программные элементы – модули и процедуры, которые в свою очередь стали объединять в так называемые библиотеки процедур. Примером могут служить процедуры вычисления элементарных функций (синус, косинус и др.), а также процедуры обмена с внешними устройствами компьютера. Однажды созданные и откомпилированные, они в дальнейшем могут применяться разработчиками ПП в своих задачах путём подсоединения их к разработанному программному коду основного алгоритма. В более широком плане эта идея нашла своё выражение в технологии модульного и процедурного программирования, которая заложена во всех современных системах программирования. Важно, что для создания загрузочного модуля, состоящего из множества оттранслированных программ, в том числе из библиотек, требуются специальные средства. Для этого в состав системы программирования входит программный модуль, называемый ***редактором связей***, обеспечивающий поиск необходимых процедур в библиотеках и присоединяющий их к основному программному коду пользователя.

Поскольку в основе любой среды системы программирования лежат различные технологии, в том числе технологии ***структурного***, ***процедурного, визуального и объектно-ориентированного программирования***, то в первую очередь необходимо следовать правилам этих технологий. Эти технологии – всего лишь инструменты, которыми можно пользоваться при разработке ПП [18].

Совокупность идей и понятий, определяющих стиль написания программ – это ***парадигма программирования***. Парадигма, в первую очередь, определяется ***базовой программной единицей*** и самим принципом достижения модульности программы.

Она определяет и то, в каких терминах описывается логика программы. Так, в традиционном [императивном программировании](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) программа описывается как последовательность действий, а в [объектно-ориентированном программировании](http://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5) (ООП) программу принято рассматривать как набор взаимодействующих объектов. ООП есть, по сути, императивное программирование, дополненное принципом модульности (инкапсуляции данных и методов в объекте) и наследованием (принципом повторного использования разработанного функционала) [18]. Сами же методы (процедуры) разрабатываются на принципах [императивного программирования](http://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5).

***Процедурное программирование*** – это [парадигма программирования](http://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), основанная на концепции описания и вызова *процедур*. Процедуры также известны как [*подпрограммы*](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0)*, методы* или *функции*. Процедурное программирование заключается, прежде всего, в выделении из общего программного кода его некоторых функциональных фрагментов в отдельные блоки (называемые процедурами), которое не только упрощает понимание алгоритма программы в целом, но и, как правило, сокращает код программы за счет удаления из нее многократно повторяющихся фрагментов. В результате разделения программного кода на понятные специализированные функциональные блоки код программы становится значительно проще в понимании, отладке и обслуживании. В ходе выполнения такой программы любая процедура может быть вызвана произвольное число раз.

***Структурное программирование*** – это методология разработки [ПП](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), в основе которой лежит представление программного кода в виде базовых *программных* [конструкций](http://ru.wikipedia.org/wiki/%D0%91%D0%BB%D0%BE%D0%BA_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29). В соответствии с этой методологией [19]:

1. Программный код представляет собой структуру, построенную из трёх типов базовых конструкций: *последовательность* – однократное выполнение операций в том порядке, в котором они записаны в тексте программы; *разветвление* – однократное выполнение одной из двух или более операций, в зависимости от выполнения некоторого заданного условия; *цикл* – многократное исполнение одной и той же операции до тех пор, пока выполняется некоторое заданное условие. В программе базовые конструкции могут быть вложены друг в друга произвольным образом, но никаких других средств управления последовательностью выполнения операций не предусматривается.
2. Повторяющиеся фрагменты программы (либо не повторяющиеся, но представляющие собой логически целостные функциональные блоки) могут оформляться в виде [процедур](http://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0) ([подпрограмм](http://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) или [функций](http://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29)). В этом случае в тексте основной программы, вместо помещённого в подпрограмму фрагмента, вставляется инструкция вызова процедуры. При выполнении такой инструкции выполняется вызванная процедура, после чего исполнение программы продолжается с инструкции, следующей за командой вызова процедуры.
3. Разработка ПП ведётся пошагово, методом «сверху вниз».

***Технология нисходящего проектирования*** с ***пошаговой детализацией*** является неотъемлемой частью создания хорошо структурированных программ. При написании программы с использованием этой технологии вся задача рассматривается как единственное предложение (вершина), выражающее общее назначение программы. Вершина разделяется на ряд более мелких задач (функций) в том порядке, в котором эти задачи должны выполнятся. В результате получим первую детализацию. Далее каждая из подзадач разбивается на подзадачи, принадлежащие второму уровню детализации. Этот процесс называется – ***функциональной декомпозиций.*** Программист завершает процесс нисходящей разработки с пошаговой детализацией, когда алгоритм настолько детализирован, что его можно преобразовать в программу.

При декомпозиции используются только три указанные выше базовые управляющих конструкции (*последовательность, разветвление, цикл*), что позволяет говорить о ***структурной декомпозиции*** или ***структурном проектировании*** программ.

Таким образом, пошаговая реализация программы – это тактика её разработки, а нисходящее проектирование – это стратегия программирования.

Для изучения базовых средств программирования будем использовать простейшие алгоритмы, реализованные в Scilab, как функции пользователя (**sce**-функции).

 Подход к решению задач с ***ориентацией на разработку функций*** включает в себя следующие основные этапы, часть из которых осуществляются до использования компьютера:

1. **Постановка задачи**. Этап включает в себя: сбор информации о задаче; определение конечных целей решения задачи; определение формы выдачи результатов; описание данных.
2. **Анализ и исследование задачи**. На этом этапе анализируются существующие аналогичные задачи; разрабатывается математическая модель задачи; осуществляется формализация; определяются структуры данных.
3. **Разработка алгоритма**. Этап заключается в выборе формы записи алгоритма и в последующем процессе разработки алгоритма и его описания.
4. **Программирование (Разработка программного кода)**. На этом этапе вначале осуществляется выбор и уточнение способов организации данных, а затем разрабатывается программный код, описывающий разработанный алгоритм.
5. **Отладка и тестирование**. При тестировании и отладке выявляют синтаксические, семантические (смысловые) и логические ошибки, допущенные при разработке алгоритма и программировании. Анализ результатов тестирования позволяет устранить все выявленные семантические и логические ошибки.
6. **Анализ результатов решения задачи**. На этом этапе осуществляется выполнение прикладной программы при реальных исходных данных. В результате анализа результатов возможно уточнение математической модели и повторение этапов 2-5.

***Процесс алгоритмизации*** заключается в опи­сании необходимой последовательности действий, с помощью которой можно однозначно реализовать выбранный метод решения задачи. На практике только очень простые задачи представляются в виде известной последовательности арифметических или логических действий. Для боль­шинства задач перед написанием программы требуется разработать соответствующую последовательность действий, приводящую к решению задачи, то есть алгоритм ее решения. Причем, при разработке алгоритма сложной задачи це­лесообразно провести декомпозицию вычислительного процесса, составить укрупненную схему алгоритма с це­лью выявления типовых участков алгоритма и использования для их реализации стандартных или ранее разработанных алгоритмов (***процедурное программирование***). Заметим, что время, потраченное на разработку вначале укрупненного, а затем детального алгоритма, полно­стью окупается при программировании и отладке программы.

***Алгоритм*** формально можно определить, как точное предписание (действие, группу действий), определяющее ***процесс преобразования исходных данных в результат***. Из определения алгоритма вытекают и его основ­ные свойства:

* ***детерминированность*** – однозначность получения результата при одних и тех же исходных данных;
* ***результативность*** – обязательность получения искомого ре­зультата за конечное число шагов;
* ***массовость*** – возможность получения результата при различ­ных исходных данных рассматриваемого класса задач;
* ***дискретность*** – возможность разбиения алгоритма на отдель­ные элементарные действия, позволяющие рассматривать ал­горитм с различным уровнем детализации.

Уже упоминалось, алгоритмы реализуются при разработке программного кода на основе базовых алгоритмических структур, которые подробно будут рассмотрены в **п. 1.5.4.**

В **п. 1.2-1.4** был рассмотрен интерактивный режим работы системы Scilab в ***Командном окне*** и средства (объекты) Scilabэтого режима – команды, операторы и функ­ции, которые являются объектами ***входного языка*** общения с системой Scilabв команд­ном режиме. Большинство объектов ***входного языка общения*** с системой Scilabв команд­ном режиме одновременно являются и объектами ***языка программирования*** системы Scilab.

Различие между командами, выполняемыми при вводе с клавиатуры в ***Командном окне***, и программными операторами, выполняемыми при вызове процедур, является условным. Эта условность связана с выбором между ***процедурной*** и ***функциональной*** технологиями программирования. Предполагается, что модули, процедуры и функции во всех языках программирования высокого уровня имеют возможность храниться в библиотечных файлах. Кроме того, важно, что в современных системах программирования программный код может менять структуру заданного алгоритма вычислений в зависи­мости от входных данных и данных, полученных в ходе вычислений.

С позиций разработчика ПП язык программирования системы Scilab является типич­ным ***проблемно-ориентированным языком программирования высокого уровня интерпретирующего типа***. Точнее говоря, он является ***языком сверхвысокого уровня***, содержащим сложные операторы и функции, реализация которых на обычных универсальных языках программирования (например, VisualBasic, С# или Си++) потребовала бы дополнительных усилий и вре­мени. Такой высокий уровень языка программирования Scilab определяется и тем, что ***базовыми структурами данных являются матрицы,*** а ***базовыми программными единицами*** – ***функции****.* Причем над матрицами реализованы как алгебраические, так и поэлементные операции***.***

Практически все программные структуры Scilab, операторы и функции являются средствами языка программирования. По числу математических и графических средств Scilab значительно превосходят VB, C++, JAVA и многие другие, а кроме того система содержит и все необходимые типовые средства программирования, такие как условные выражения, циклы, операторы ввода/вывода и др.

В **п.п.1.5.2, 1.5.3** будут рассмотрены ***базовые программные единицы*** системы программирования Scilab, правила их описания и вызова, а в   
**п.1.5.4** – ***алгоритмические операторы*** Scilab и ***базовые программные структуры***.

**1.5.2*.* Функциональные структуры –   
сценарий, функция и средства их создания**

***Уровни функциональных программных структур***

Использование системы Scilabтолько в режиме непосредственного расчета в ***Командном окне*** явно недостаточно для решения сложных задач. Поскольку программы, реализующие сложные алгоритмы, создаются, как правило, в соответствии с принципом процедурного программирования, то очевидно необходимо иметь системные и программные средства, позволяющие создать функциональные программные структуры, и средства, обеспечивающие их хранение в текстовых и откомпилированных библиотечных файлах, их загрузку в оперативную память и выполнения.

В соответствии с концепцией разработчиков в системе Scilab имеется три уровня функциональных структур, из которых можно построить различные приложения. Это – ***Сценарий***, ***Макрос*** и ***Примитив*** [13].

***Сценарий*** – это фрагмент программного кода, который можно сохранить в текстовом **sce**-файле, а затем загрузить с помощью функции **exec** и выполнить сразу после загрузки.

Сценарии в Scilab предназначены для выполнения различного рода задач. Они могут осуществлять подготовку данных для предстоящих вычислений (инициализировать переменные и осуществлять их логическую проверку); выполнять вспомогательные действия (подгружать различные библиотеки и т.п.); проводить технические расчеты, а также осуществлять ввод и вывод данных расчетов в виде таблиц, инструкций и графиков.

Сценарий – это заранее определенная последовательность инструкций Scilab, практически он является подобием сессии ***Командного окна***, и поэтому все инструкции, загруженные из файла, могут быть выполнены точно так же, как если бы вы их вводили с клавиатуры в ***Командном окне***.

В начало текста сценария рекомендуется вставлять комментарий о его функциональном назначении и комментировать все его ключевые шаги. Также рекомендуется команду **clear**, предназначенную для очистки содержимого окна ***Обозреватель данных*** от данных, используемых в предыдущей сессии.

Особенности сценария состоят в том, что он:

* хранится в **sce**-файлах с уникальным именем;
* не имеет входных и выходных параметров;
* работает с переменными из ***Рабочей области*** (за исключением переменных, описанных во внутренних функциях этого сценария);
* содержимое в процессе выполнения не компилируется;
* может содержать *внутренние встроенные в сценарий функции* пользователя;
* строки нумеруются автоматически.

***Встроенные в сценарий функция*** так же, как и сценарий, может содержать команды, операторы и другие функции (вложенные), имеет функциональное содержание, более сложную программную структуру и имеет следующие особенности:

* начинается структура функции с ***заголовка***, а именно, с ключевого слова **function**, а завершается ключевым словом **endfunction** или **end**;
* второй строкой должен быть комментарий о ее функциональном назначении;
* все ключевые шаги всегда необходимо комментировать;
* может иметь входные и выходные параметры, назначение которые также необходимо комментировать;
* может содержать вложенные функции;
* все переменные, описанные в теле функции, являются локальными, то есть действуют только в пределах тела функции (за исключением вложенных функций);
* в простейшем варианте, как описания функций, так и обращения к ним, могут быть частью сценария (функции могут быть встроенные в сценарий), который может состоять только из описания функций).
* описания внутренних встроенных функций могут находиться в любых строках сценария.

***Макрос*** – это внешняя *функция*, реализованная средствами языка Scilab, которую можно сохранить в текстовом **sci**-файле, имеющая такое же имя, что **sci**-файле, и, загрузив которую в оперативную, память можно выполнить по имени функции.

***Примитив*** – это внешняя откомпилированная функция, которая может быть реализована средствами Scilab, C++, Фортран или любого другого языка программирования, причем вызов такой функции осуществляется через специальный *шлюз* Scilab. В Scilab шлюз – это функция C++ или Fortran, которая позволяет пользователю соединять интерпретатор с набором библиотечных функций, чтением входных параметров, заданных пользователем, и записью выходных аргументов, затребованных пользователем.

Далее в этом разделе будут рассмотрены правила описания простейших сценариев и встроенных в них функций, являющихся частью сценария, а также запись их в **sce-**файлы, загрузку их из файла в оперативную память и выполнение загруженных сценариев. Создание же текстовых и откомпилированных библиотек будет рассмотрено в **п.1.5.6.**

Инструментом создания и отладки сценариев и встроенных в сценарии функций является редактор **SciNotes**. Этот редактор является одним из основных компонентов системы Scilab, и вместе с файловой системой, обеспечивающей хранение сценариев, макросов и примитивов, а также алгоритмическими средствами, и является *системой программирования* Scilab.

***Обратите внимание, что sce-файлы – это текстовые файлы, в которых могут храниться сценарии, созданные с помощью встроенного редактора* SciNotes*или любого другого текстового редактора.***

Прежде чем перейти к правилам описания и вызова сценариев и функций, а также записи их в **sce**-файлы, рассмотрим элементы редактора **SciNotes** и их использование при создании и выполнении сценариев и функций.

***Средства Редактора SciNotes***

Для работы с редактором его необходимо открыть (активировать). Это можно сделать элементом **SciNotes**инструментальной панели Scilab, в результате чего открывается окно редактора **SciNotes (**рис. 1.5.2-1).

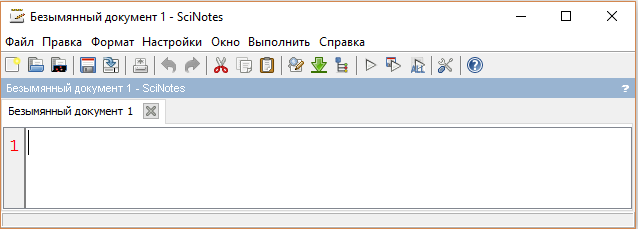


Рис. 1.5.2-1 Окно редактора **SciNotes** для создания сценария

Меню редактора **SciNotes** содержит следующие элементы:

* ***Файл*** – включает команды, которые позволяют создавать новый документ, открывать существующий или последний сохраненный документ, и другие.
* ***Правка*** – включает команды, которые позволяют работать с текстом в окне редактора: отменить действие, повторить, вырезать, копировать, вставить и другие.
* ***Формат –*** включает команды, которые позволяют осуществить форматирование документа: сделать или удалить отступ, удалить пробелы в конце строки, сменить регистр, добавить или удалить комментарий, заменить одинарные кавычки на двойные.
* ***Настройка –*** включает команды, которые позволяют выполнить текущую кодировку файла, настройку шрифтов и цвета.
* ***Окно –*** включает инструменты копирования или отделения вкладки в новое окно, включение или отключение разделителя на две колонки и другое.
* ***Выполнить –*** включает команды, которые позволяют выполнить созданный файл и сохранить в текущем каталоге, а также выполнять выделенную часть файла.
* ***Справка*** – о редакторе**SciNotes**.

Элементы панели инструментов дублируют часто используемые команды меню **SciNotes**, и позволяют, например, *открывать, сохранять, редактировать, запускать* и осуществлять *отладку* созданного документа и многоедругое.

Для редактирования содержимого ранее созданного и записанного в **sce**-файл сценария и, необходимо открыть этот файл либо выполнив команду ***Открыть*** элемента меню **Файл**, либо щелкнув мышкой на соответствующий элемент панели инструментов , либо двойным щелчком мышки на ***Имя файла*** в окне **Обозреватель файлов**. При использовании команды   
***Открыть*** открывается окно **Open** (рис.1.5.2-2), в строке Filename которого необходимо ввести имя открываемого файла и щелкнуть по кнопке **Open** (или вызвать его двойным щелчком в поле текущей директории).

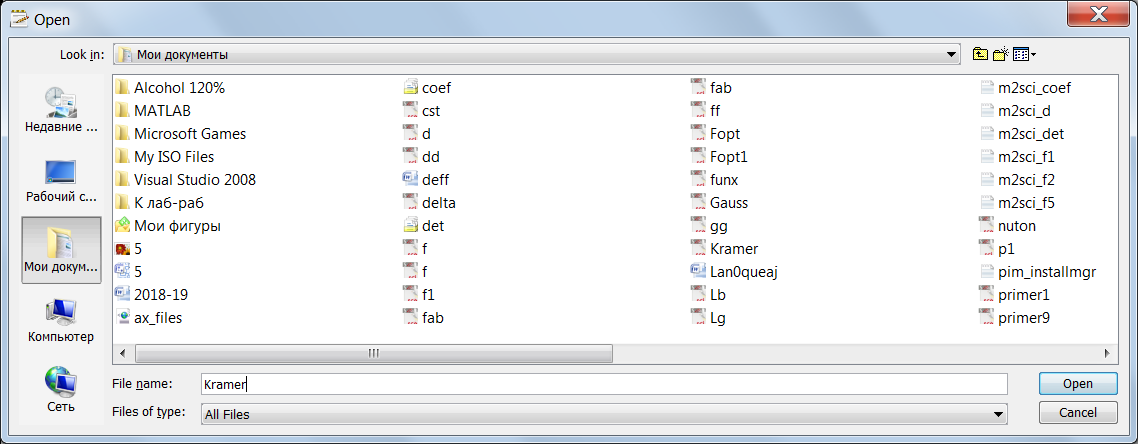


Рис. 1.5.2-2 Окно **Open (Открыть)**

Запись созданного сценария в **sce**-файл осуществляется либо командой ***Сохранить как…***элемента меню **Файл**, либо выбором соответствующего элемента панели инструментов . В открывшемся одноименном окне требуется выбрать путь сохранения программного кода и ввести его имя, а затем щелкнуть по кнопке **Save** (рис. 1.5.2-3). Следует помнить, что по умолчанию все файлы записываются с расширение \*.**sce**.

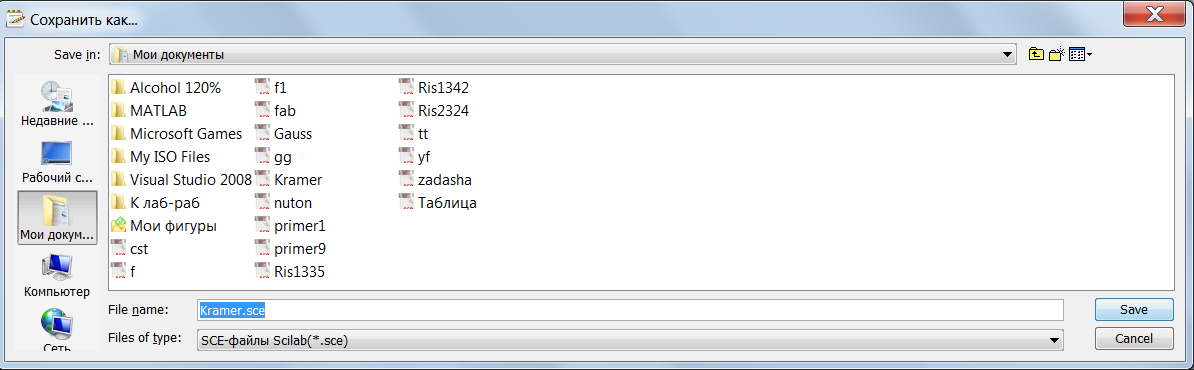


Рис. 1.5.2-3 Окно **Сохранить как…**

Для сохранения программного кода после его редактирования необходимо выполнить команду ***Сохранить*** элемента меню **Файл** или щелкнуть по соответствующему элементу панели инструментов .

Отметим, что одновременно в редакторе могут быть загружено несколько сценариев, что отражается на вкладках, расположенных в строке над полем ввода текста, где перечислены имена загруженных и создаваемых сценариев. Переход между вкладками с соответствующим отображением программного кода в окне ***Редактора*** можно произвести щелчком по вкладке с именем нужного сценария.

***Следует помнить, что в первых строках любого создаваемого sce-сценария, нужно вводить комментарии о назначении программного кода.***

***Создание, сохранение и выполнение сценариев***

Для создания нового сценария достаточно открыть **SciNotes**и ввести в команды, например, необходимые для построения простейшего графика   
(рис. 1.5.2-4.).

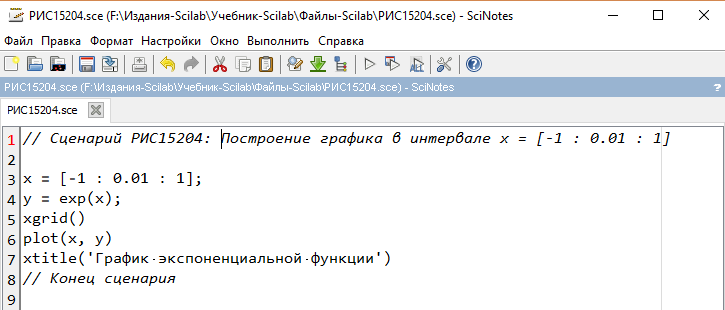


Рис. 1.5.2-4 Сценарий **РИС1524**, содержащий команды для построения графика

Для сохранения только что созданного сценария следует щелкнуть по элементу инструментальной панели **Редактора *Сохранить как…***, затем в соответствующей строке окна **Сохранить** ввести имя файла с расширением\***.sce**, и щелкнуть по кнопке **Save**. Имя файла появится в окне текущего каталога, а в строке активных файлов вместо имени **Безымянный1**, появится введенное *имя*. Например, на рис. 1.5.2-4 показан сценарий **РИС1524**, созданный в редакторе **SciNotes.**

Загрузка и запуск сценария, сохраненного в **sce**-файле, осуществляется из ***Командного окна*** по его имени с указанием пути к этому файлу. Желательно предварительно в окне ***Обозреватель файлов*** установить путь к ***папке***, где находится сохраненный **sce**-файл с соответствующим сценарием, то есть сделать эту папку текущей.

Загрузка содержимого **sce-**файла в оперативную память компьютера осуществляется командой **exec**, которая может иметь следующие форматы:

**exec(*'****Путь доступа к файлу****\****ИмяФайла****.*sce',** *РежимОтображения***);**

**exec(*'****ИмяФайла****.*sce',** *РежимОтображения***);**

В первом формате указывается полный путь к загружаемому **sce**-файлу. Этот формат используется, если папка, где находится загружаемый сценарий, не является текущей. Во втором случае указывается только имя файла, при условии, что ***папка***, где находится загружаемый сценарий, является текущей.

Второй парамер функции **exec** – *РежимОтображения* является необязательным, и может принимать значения **0, 1, -1, 2, 3, 4**. Если он равен **0** или **1**, то результаты работы инструкций Scilab выводятся в ***Командном окне***, а если он равен **-1**, то в ***Командное окно*** ничего не выводится. Этот параметр позволяет управлять отображением информации в ***Командном окне*** в ходе выполнения инструкций Scilab. Вывод строк сценария в ***Командное окно*** и результаты работы инструкций Scilab можно также включать и отключать с помощью встроенной функции Scilab**mode**.

Следует обратить внимание, что при отсутствии параметра   
*РежимОтображения* команда **exec**может заканчиваться символом точка с запятой. Если этот символ отсутствует, то строки программного кода, выполняемого сценария, выводится в ***Командное окно***, а наличие точки с запятой гасит вывод кода сценария.

***Следует помнить, что если выполнение сценария требует задания начальных значений для используемых переменных, то это необходимо сделать перед запуском файла сценария на выполнение.***

Чтобы выполнить сценарий – **РИС1524.sce**, который предназначен для построения графика, следует предварительно загрузить файл с именем **РИС1524.sce**(рис. 1.5.2-5).

|  |
| --- |
| --> // ***Подключение и выполнениесценария*РИС15204.sce**  -->  -->clear  -->exec ('РИС15204.sce', 0); |
|  |

Рис. 1.5.2-5 Результат выполнения сценария с именем **РИС1524.sce**

***Сценарии и встроенные функции***

Общая структура сценария со встроенными в него внутренними функциями может быть следующей:

**//*Комментарий, описывающий назначение сценария***

**...**

*ТелоСценария*

**...**

*Описание встроенных внутренних функций*

**...**

**// *конец сценария***

В отличие от сценариев, функции являются типичными объектами языка программирования высокого уровня, и имеют свою определенную структуру.

Общая структура описания внутренних встроенных функции с **n**входными и**k**выходными параметрами имеет следующий формат:

**function [***y****1,…,yk*] =** *ИмяФункции***(***x****1,…,****x****n*)**

**//*Комментарии, описывающие назначение функции и ее параметров***

***...***

*ТелоФункции*

***...***

*y1* = *Выражение****1;***

***...***

*yk* = *Выражение****k;***

***...***

**endfunction**

где: ***x1,…,xk*** – список входных формальных параметров функции (если функция не имеет входных параметров, тогда после имени функции указываются пустые скобки); ***y1,…,yn*** – список выходных формальных параметров (для одного выходного параметра квадратные скобки необязательны); *ТелоФункции* – содержит команды, операторы и библиотечные функции, необходимые для вычисления выходных параметров. Если вычисленные результаты должны быть в ходе выполнения функции (или ее отладки) отображены в ***Командном окне***, то символ точка с запятой в конце операторов отсутствует, иначе вывод результата гасится символом точки с запятой.

Первая строка описания любой функции начинается с ключевого слова **function**, вслед за которым в квадратных скобках через запятую указываются *имена выходных параметров*, далее после знака присвоения (**=**) указывается уникальное *ИмяФункции*, а затем в круглых скобках –   
*СписокВходныхФормальныхПараметров* функции. Заканчивается описание функции командой **endfunction** или **end**.

***Следует напомнить, что функция может не иметь входных параметров, тогда после имени функции указываются пустые скобки.***

Обращение к функции может иметь следующие форматы:

**[***var****1****,var****2,****…****,****var****k*]=***ИмяФункции****(****СписокВходныхФактическихПараметров****);***

*var***=***ИмяФункции****(****СписокВходныхФактическихПараметров****)***

*ИмяФункции****(****СписокВходныхФактическихПараметров****)***

*Первый* формат предусматривает необходимость вычисления в теле функции **k**значений, которые в результате выполнения функции присваиваются **k**выходным параметрам и выводятся на экран (или только сохраняются под их именами в случае, если после обращения к функции стоит точка с запятой, которая гасит их вывод).

*Второй* формат используется, если функция имеет только один выходной параметр. Формат практически идентичен первому, но здесь имя выходного параметра можно не заключать в квадратные скобки.

*Третий* формат обращения к функции используется в двух случаях. В первом случае функция имеет один выходной параметр, но он не используется в выражениях (в этом случае значение выходного параметра записывается в переменную **ans**). Во втором случае функция не имеет выходных параметров, а, например, предназначена для вывода результатов расчета (построение графика, таблицы значений и т.п.).

По умолчанию все переменные, описанные в теле функции, являются   
***локальными***.

Простейшим случаем использования внутренних встроенных функций является сценарий, состоящий из одной функции. Рассмотрим пример создания сценария с описанной внутри него одной функцией **fun(x)=ax2+x/2** и одним входным параметром, сохранения этого сценария в **sce**-файле, загрузки и обращения к нему из ***Командного окна***.

Для этого необходимо выполнить следующие шаги:

1. ***Загрузить*** текстовый редактор **SciNotes**.
2. ***Описать*** сценарий со встроенной функцией **fun(x)**в редакторе.
3. ***Сохранить*** сценарий в **sce**-файл под именем **РИС1526**.
4. ***Загрузить*** сценарий в ***Командном окне*** командой **exec**.
5. ***Выполнить*** (***вызвать***) функцию **fun(x)**.

На рис. 1.5.2-6 показаны окно редактора **SciNotes**после написания сценария, содержащего функцию **fun(x)**, записанного в файл **РИС1526.sce.** Ниже приведено содержимое ***Командного окна***, из которого происходит загрузка файла*-*сценария **РИС1526**, двукратное обращение к встроенной функции **fun(x)**и в котором отображаются результаты ее вы выполнения.

При выполнении команды **fun(x)**, функция принимает один фактический входной параметр **x=5** (переменная **x**должна быть предварительно определена), вычисляет с использованием встроенной в сценарий функции значение по соответствующей формуле и возвращает его через выходной параметр **ans**. Второе обращение в функции показывает возможность использования функции **fun**в операторе присваивания. Здесь в качестве параметра **x**использовано числовое значение, а результат выполнения присваивается переменной **y**.

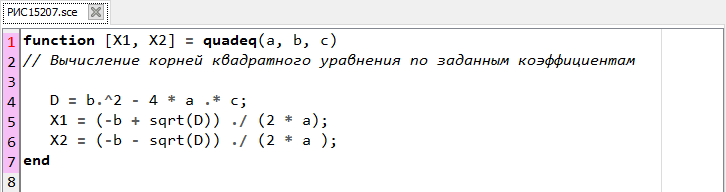
|  |
| --- |
|  |
| --> // ***Загрузка файла* РИС15206.sce *и выполнение функции* fun**  -->exec('РИС15206.sce',0);  -->  -->clear  -->x = 5; // ***Определение значения параметра* х**  -->fun(х) // ***Вычисление функции fun(x) при* х = 5**  ans =  27.5  -->  -->y = fun(4) // ***Использование функции в операторе присваивания***  y =  18. |

Рис. 1.5.2-6 Окно **SciNotes**и **Командное окно**

Тело функции может включать в себя любые выражения Scilab, функции ввода/вывода, операторы управления, комментарии, пустые строки и вложенные функции и многое другое.

***Обратите внимание, что любые переменные, которые определены в функциях, хранятся в своих локальных Рабочих областях данных каждой функции, то есть отдельно от Рабочей области Обозревателя переменных, в которой хранятся только переменные, определенные в Командном окне и сценариях.***

В следующем примере, приведенном на рис. 1.5.2-7, представлено описание сценария, включающего одну встроенную функцию с несколькими выходными параметрами. Заметим, что список выходных параметры в заголовке функции заключен в квадратные скобки, а сами параметры отделены друг от друга запятыми. Загрузим файл-сценарий **РИС1527***.***sce**и вызовем функцию **quadeq**, которая должна по заданным коэффициентам квадратного уравнения вычислить его корни.

****

|  |
| --- |
| --> // ***Загрузка файла* РИС15207.sce *и выполнение функции* quadeq**  -->exec('РИС1527.sce', 0);  -->  -->clear  --> // ***Матрицы с заданными коэффициентами трех уравнений***  -->A = [1 3 5]; B = [2 4 6]; C = [1 2 4];  --> [X1, X2] = quadeq (A, B, C) // ***Обращение к функции* quadeq**  X2 =  -1. -0.6666667 - 0.4714045i -0.6 - 0.663325i  X1 =  -1. -0.6666667 + 0.4714045i -0.6 + 0.663325i |

Рис. 1.5.2-7 Содержимое окна ***SciNotes***и ***Командного окна*** после загрузки сценария **РИС1527** и выполнения функции **quadeq**

В двух рассмотренных выше примерах сценарии **РИС1526** и **РИС1527** были созданы в редакторе **SciNotes**,загружены и вызваны на выполнение из ***Командного окна***, куда и были выведены результаты.

Чтобы запустить на выполнение сценарий из окна редактора **SciNotes**, достаточно нажать кнопку  (**Выполнить**) или выбрать команду **Выполнить** в одноименном элементе меню. В результате в строке ***Командного окна*** появится команда ***exec***c указанием полного пути к файлу, а курсор устанавливается в начале следующей строки.

***Следует помнить, что переменные, созданные внутри функции, являются локальными и действуют только в пределах этой функции. Переменные, созданные в пространстве вне функций, являются глобальными и доступны во всех функциях данного файла и текущей рабочей сессии.***

В одном файле, созданном в редакторе, который является, по сути дела, всегда сценарием, может быть описано сразу несколько функций. Эти функции могут общаться между собой посредством имен функций, а также ***входных*** и ***выходных параметров***.

Общая структура сценария, содержащего несколько встроенных внутренних функций сценария, может быть следующей:

// ***Имя Сценария и его назначение***

*Тело сценария*

***...***

**function***f1*

*ТелоФункцииf1*

**end**

…

**function***fm*

*ТелоФункцииfm*

**end**

**// *end сценария***

Эта структура и некоторые другие вопросы, касающиеся области видимости данных при использовании нескольких функций, будут рассмотрены подробно в **п.1.5.3**.

**1.5.3. Общая структура функций и сценариев.  
Области видимости переменных**

***Общая структура кода сценария***

В предыдущем разделе были даны правила описания сценариев и внутренних встроенных в них функций, а также приведены соответствующие примеры их описания, где в каждом **sce-**файле содержалось по одной функции. Однако в каждом сохраненном **sce-**файле могут находиться описания более чем одной независимой функции, причем в каждую из них может быть вложена другая функция (рис 1.5.3-1).

|  |
| --- |
|  |
| --> // ***Загрузка и выполнение вложенных функцией***  -->exec('РИС15301.sce', 0);  -->clear  --> x = 2;  --> r = mvf(x)  r =  2. |

Рис. 1.5.3-1 Пример реализации вложенных функцией

Как уже отмечалось, в сценарии программные коды ***независимых функций*** должны располагаться одна за другой, когда как ***вложенные функции*** располагаются внутри функций.

На рис 1.5.3-2 показано сначала обращение из ***Командного окна*** к  
 функции **vstfun**, использующей функции **vfun1** и **vfun2**, расположенные в сценарии, а затем отдельно к функции **vfun2**. Этот пример показывает, что все остальные функции, при подключении сценария командой **exec** становятся доступными для использования в ***Командном окне***.

|  |
| --- |
| --> // ***Загрузка сценария РИС15302 и обращения к* vstfun, vfun1 и vfun2**  -->exec('РИС15302.sce', 0);  -->  -->clear  -->y = vstfun(2) //***Обращение к* vstfun2(2), *а из нее к*vfun1(2) *и*vfun2(2)**  y =  2.  -->  -->f=vfun1(2), f = vfun2(2) //***Независимое обращение к* vfun1(2) *и v*fun2(2)**  f =  0.  f =  2. |

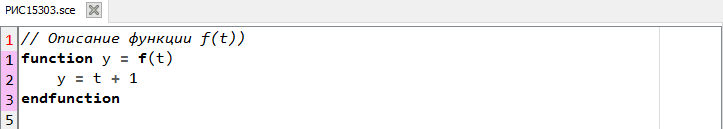
Рис. 1.5.3-2 Пример обращения к функциям сценария

***Следует помнить, что перед обращением к любому сценарию, сохраненному в sce-файле, этот файл должен быть загружен командой exec.***

***Имя функции как тип переменной***

Одно из достоинств языка Scilab является то, что функции являются типом переменных. Это означает, что мы можем хранить функции в переменных и использовать переменные как функции. В компилируемых языках эта возможность часто называется ***«указатель на функцию»***.

В следующем примере (рис 1.5.3-3) определим функцию **f**. Затем, установим содержимое переменной **fp** равным функции **f**. Наконец, мы можем использовать функцию **fp** как обычную функцию.





--> // ***Загрузка сценария* РИС15303 *и обращения к функции* f**

--> exec('РИС15303.sce', 0);

-->

-->clear

--> fp = f

fp =

[y] = fp(t)

-->

--> fp(1)

ans =

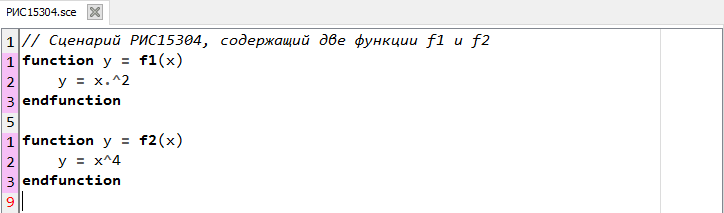
2.

Рис. 1.5.3-3 Пример использования указателя на функцию

Эта возможность позволяет использовать широко распространённый инструмент языков программирования, известный как ***функция обратного вызова*** (**callback**).

Функция обратного вызова – это функция, которая передается другой функции в качестве параметра. При этом передается указатель (ссылка) на эту функцию. В свою очередь другая функция, вызывает переданную через параметр функцию.

Поскольку функции являются переменными можно устанавливать значения этих переменных несколько раз. Для того, чтобы предупредить пользователей от нежелательного переопределения функций, при переопределении может появляться сообщение-предупреждение, как показано в примере на рис 1.3.5-4.





--> // ***Загрузка сценария* РИС15304 *и переопределение функций***

--> exec('РИС15304.sce', 0);

-->

-->clear

--> f = f1

f =

[y] = f(x)

-->

--> f = f2

*Предупреждение: переопределение функции: f.*

*Используйте funcprot(0) чтобы не выводить это сообщение*

--> f =

--> [y] = f(x)

Рис. 1.5.3-4. Получение предупреждения о переопределении функции

В данном случае нет причин защищать себя от присвоения переменной **f** нового значения, в Scilab имеется простой способ отключить на время предупреждение. Функция **funcprot** позволяет заблокировать режим защиты функций:

**pr=funcprot()** – получить текущий режим защиты функций;

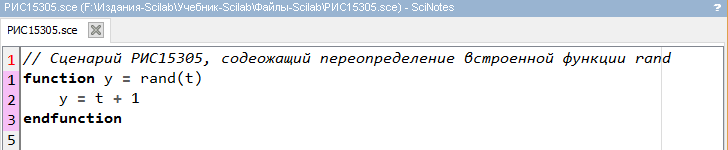
**funcprot(0)** – нет сообщений, когда функция переопределена;

**funcprot(1)**– выдает предупреждение о переопределении функции

(по умолчанию);

**funcprot(2)** – выдает ошибку, когда функция переопределена.

Scilab позволяет переопределить любые функции (даже библиотечные), но это вызывает ошибку и выводится соответствующее сообщение. В следующем примере (рис. 1.5.3-5) функция **rand** определена как обычная функция, проверим, можем ли мы вызвать её как любую другую функцию, определённую пользователем.





--> // **Загрузка сценария РИС15305 и переопределение функций rand**

--> exec('РИС15305.sce', 0);

-->clear

--> rand()

*Предупреждение: переопределение функции: rand.*

*Используйте funcprot(0) чтобы не выводить это сообщение*

-->

-->funcprot(0)

-->y = rand(1)

y =

2.

Рис. 1.5.3-5 Переопределение встроенный библиотечной функции **rand**,   
описанной в сценарии **РИС15305**

Появилось сообщение о том, что функцию **rand** переопределили. То есть функция **rand** уже существует в Scilab, что может быть легко проверено командой **help rand**. Действительно, встроенная функция **rand** позволяет генерировать случайные числа, и мы, конечно же, не хотим ее терять и переопределить. В данном случае ошибка очевидна, но на практике ситуации могут быть гораздо более сложными, поэтому переопределять функции надо очень осторожно.

***Видимость переменных***

Как известно, переменные, созданные в процессе выполнения сессии, хранятся в области ***Обозревателя переменных***, кроме тех, которые описаны в функциях. То есть переменные, описанные внутри функций, хранятся в своих локальных областях памяти. Поэтому функция может получить доступ к переменным только в том случае, если данные передаются в качестве аргументов. Это позволяет защитить целостность данных.  Сценарий, представленный на рис. 1.5.3-6 содержит три независимые функции со своими локальными переменными. Загрузив сценарий, и вызвав функцию **vstfun**из ***Командного окна,*** можно видеть, что переменные функций являются локальными и недоступны как из ***Командного окна***, так и из сценария.

|  |  |
| --- | --- |
|  | |
| --> // ***Загрузка сценария* РИС15306**  --> // ***и обращение к функции* vstfun**  -->clear  --> exec('РИС15306.sce', 0);  -->  --> x = 2;  --> r = vstfun(x)  r =  8. |  |

Рис. 1.5.3-6 Отображение данных сессии в окне ***Обозреватель переменных***

В окне ***Обозреватель переменных*** отобразились только две переменные (**r** и **x**), используемые в ***Командном окне***. Таким образом, локальные переменные не доступны ни командной строке, ни другой **sce**-функции.

Один из способов получить доступ к переменным локальных функций объявить переменные ***глобальными*** (**global**). Глобальные переменные хранятся в своей ***области глобальных переменных***. Особенность глобальных переменных состоит в том, что они видны тем объектам текущего сеанса, где они описаны как глобальные.

Однако использовать глобальные переменные опасно, поскольку:

1. любая функция может получить доступ и обновить глобальную переменную, в этом случае другие функции, использующие эту переменную, могут получить неожиданные результаты.
2. «новая» глобальная переменная может случайно получить то же самое имя, что и «старая» (уже существующая) глобальная переменная. Это приводит к ошибке, которая трудно диагностируется.

Прежде чем обращаться к глобальным переменным функции из командной строки их необходимо объявить. Для этого предназначена функция **global***,* имеющая следующий формат:

**global**(*СписокГлобальныхПеременных)*

где в *СписокеГлобальныхПеременных* через запятую перечисляются имена глобальных переменных, заключенные в кавычки, например, **global('a','b','x')**. Если *СписокГлобальныхПеременных* содержит одну переменную, то допускается создание списка без скобок и кавычек.

Создадим, например, в командной строке глобальную переменную **x**, присвоив ей значение **2**. Теперь рассмотрим пример, приведенный на рис.1.5.3-7.

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15307 *и обращение к функциям* f1 *и* f2**  -->  -->clear  --> exec('РИС15307.sce');  -->a = 1;  -->f1() // ***Обращение к функции* f**1  -->a  a =  4.  -->f2() // ***Обращение к функции* f2**  -->a  a =  4. |

Рис.1.5.3-7 Пример, иллюстрирующий область видимости переменных

Обратите внимание, что при обращении к функции **f1**, переменная **а** изменила свое значение, а при обращении к функции **f2** значение **а** осталось прежним. Дело в том, что и в командной строке, и в функции **f1,** переменная, а объявлена глобальной, то есть она является видимой для **f1**. В функции **f2** переменная **а** является локальной, то есть невидимой для функции **f2**, таким образом, это две разные переменные, расположенные в разных областях оперативной памяти.

В Scilab имеется еще ряд функций для работы с глобальными переменными, имеющими тот же формат, что и функция **global**:

**clearglobal** – уничтожает глобальные переменные списка;

**isglobal** – проверяет, является ли переменная глобальной;

**gstacksize** – определяет размер области глобальных переменных.

В любом языке программирования использование глобальных переменных усложняют понимание программ, кроме того они менее защищены от изменений. Поэтому если есть возможность, данные в функции лучше передавать через параметры.

**1.5.4. Алгоритмические операторы Scilab  
и базовые программные структуры**

В программировании различают ***линейную*** и ***нелинейную*** структуру программного кода [19]. Программную структуру называют ***линейной****,* если *операторы* (*инструкции, строки программного кода функции*) выполняются строго в том порядке, в котором они написаны. Однако существует множество алгоритмов, структура которых ***нелинейная***, то есть элементы алгоритма выполняются в зависимости от определенных условий, иногда с конечным числом повторений – регулярных циклов, иногда в виде циклов, завершаемых при выполнении заданного условия. Практически любая серьезная программа имеет нелинейную структуру. Для создания таких программ необходимы специальные управляющие структуры. Они имеются в любом языке программирования высокого уровня, в том числе и в Scilab [20].

Рассмотрим эти элементы подробнее.

***Линейные программные структуры***

К линейным программным структурам можно отнести операторы присваивания, вычисления выражений и операторы ввода/вывода.

***Оператор присваивания*** подробно был рассмотрен в **п. 2.1.2**. Кратко напомним, что оператор ***присваивания*** является фундаментальным оператором системы программирования Scilab:

*ИмяПеременной* = *Выражение.*

Оператор предназначен для инициализации переменных (присвоения им конкретных значений соответствующего типа) и обозначается символом **=**, слева от которого находится имя переменной, а справа любое допустимое выражение (правила записи арифметических, логических и строковых выражений были рассмотрены в **п. 1.2**).

Все переменные, используемые в правой части оператора присваивания, должны быть предварительно проинициализированы. Если командная строка заканчивается символом точка с запятой (**;**), то результат выполнения оператора не выводится, иначе он выводится в следующей строке командного окна. Это замечание распространяется и на выполнение операторов присваивания, расположенных в **sce**-файлах.

Приведем несколько примеров использования операторов присваивания (рис.1.5.4-1).

|  |
| --- |
| -->// ***Примеры использования операторов присваивания***  -->  --> a = 2; // ***Присваивание переменной* a *числового значения (скаляр)***  --> x = a^2 - 20; // ***Присваивание переменной xзначения выражения*a2-20**  --> x  x =  -16.  -->  --> A = [2 -4 7]; // ***Присваивание переменной* A*значений вектора* [2 -4 7]**  --> A  A =  2. -4. 7.  -->  --> s1 ="Система"; // ***Присваивание переменным* s1 *и* s2**  -->s2 = "Scilab"; // ***значений строковых констант***  -->  -->s = s1 + " " + s2 // ***Присваивание переменной* s*значения выражения***  s =  Система Scilab |

Рис. 1.5.4-1 Примеры использования операторов присваивания

***Простейшие операторы ввода/вывода данных***

Часто приходится создавать программы, которые способны не просто выполнять какие-либо команды и выдавать результаты, но и обмениваться информацией с пользователем в процессе работы через интерфейс. На определенном этапе такая программа может прервать свою работу и выдать запрос пользователю, а ее дальнейшее поведение, например, продолжить или прекратить вычисления, будет зависеть от введенного пользователем ответа на этот запрос. Интерактивное взаимодействие пользователя легко реализуется путем создания интерфейса с помощью командной строки или специальных функций.

Для организации простейшего ввода в Scilab можно воспользоваться, в первую очередь, функцией:

**x** = **input**('*Подсказка*');

Функция **input** выводит в командной строке Scilab *Подсказку,* и ожидает от пользователя ввода значения, которое затем присваивается переменной **х**. На рис.1.5.4-2 представлен способ ввода исходных данных с использованием функции **input**.

|  |
| --- |
| --> // ***Пример использование функции ввода*input**  -->  -->y = input('Введите y = ')  Введите y = 5  y =  5. |

Рис. 1.5.4-2 Использованием функции **input**для ввода данных с клавиатуры

Кроме функции **input**для ввода данных можно воспользоваться специальными диалоговыми окнами для интерактивного ввода. Эту возможность можно реализовать с помощью функций **x\_dialog, x\_mdialog, x\_choose**, **x\_matrix**, **x\_message** и **x\_message\_modeless**.

Рассмотрим подробно работу функции **x\_dialog**, которая имеет следующий формат:

*Переменная***=x\_dialog('***Подсказка***','***Значение***');**

Функция **x\_dialog** выводит на экран диалоговое окно с именем   
**Ввод значения**, в котором над полем ввода отображается *Подсказка.* Если при использовании функции **x\_dialog** строка **'***Значение***'** задана пустой строкой, то поле ввода диалогового окна пусто, и пользователь может ввести в него нужное значение. Если же строка **'***Значение***'** непустая, то эта строка отображается в поле ввода диалогового окна. После щелчка по кнопке ОК значение, отображенное в поле ввода, присваивается переменной.

На рис.1.5.4-3 представлен способы ввода исходных данных с использованием диалоговое окно, при использовании функции **x\_dialog**.

|  |  |
| --- | --- |
| --> // ***Пример использования функций*x\_dialog**  -->y = x\_dialog('Введите y = ', '5 ') |  |

Рис. 1.5.4-3 Использованием функций **x\_dialog**для ввода данных с клавиатуры

При этом, если функция **input** возвращает значение числового типа, то**x\_dialog**–строкового. Поэтому при использовании для ввода числовых значений функции**x\_dialog**, возвращаемую строку следует преобразовать в число с помощью функции **evstr**. Можно также предложить следующую форму использования функции **x\_dialog**для ввода числовых значений:

*Переменная* **= evstr(x\_dialog ('***Подсказка***', '***Значение***'));**

В качестве *Значения* может быть использован вектор или матрица. Функция **evstr** распознает символы, используемые при вводе массивов (пробел, запятая или точка с запятой). Если ввод строки закончить нажатием клавиши <Enter>, то поле ввода очищается, и команда **x\_dialog** снова ожидает ввода *Значений*, и так диалог с пользователем происходит до тех пор, пока не будет нажата кнопка **ОК** (рис.1.5.4-4).

|  |  |
| --- | --- |
| -->// ***Использование функции*x\_dialog**  --> // ***для ввода значений вектора***  -->  --> V = evstr(x\_dialog('ВекторV=', ' '))  V =   1. 2. 3. 4. 5. 6.   -->  -->M = evstr(x\_dialog('Матрица М=', ' '))  М =  1. 2. 3.  4. 5. 6. |  |

Рис. 1.5.4-4 Использование команды **x\_dialog**для ввода вектора и матрицы

Для вывода значения переменной или текста можно использовать функцию **disp**, имеющую следующий формат:

**disp(b)**,

где **b** – имя переменной или заключенный в кавычки или апострофы текст.

Сделаем несколько замечаний по использованию функции **disp**. Функция **disp** осуществляет вывод значений заданных переменных или заданного текста в ***Командное окно***. Чтобы вывести значения нескольких переменных в одну строку, например, при создании таблиц данных, нужно создать единый объект, который содержал бы все эти значения. Это можно сделать, объединив соответствующие переменные в вектор, пользуясь операциями для создания вектора-строки, например, **x=[x1 x2 ...xn]**. Тогда вывод значений нескольких переменных будет осуществляться в одну строку (**Пример1**, рис. 1.5.4-5).

Другой способ форматированного вывода данных – это использование функции **mprintf** (**п. 1.2**). **Пример2**, рис. 1.5.4-5 показывает применение функции **mprintf**.

|  |
| --- |
| --> // ***Вывод значений нескольких переменных в одну стр***оку  -->  --> // ***Пример1***  -->x1 = -3.14; x2 = -2.5; x3 = 5.6; x4 = -9.33;  -->disp([x1 x2 x3 x4])  -3.14 -2.5 5.6 -9.33  -->  --> // ***Пример***2  -->disp(mprintf('Параметр1 = %g Параметр2 = %g',x1,x2))  Параметр1 = -3.14 Параметр2 = -2.5 |

Рис. 1.5.4-5 Вывод значений нескольких переменных в одну строку

Если необходимо использовать элементы форматирования, то первый шаг, который нужно сделать – решить, что именно надо отображать. Если это целое число, число с плавающей точкой или строка, то необходимо выбрать соответствующий спецификатор для базового типа формата.

Затем определяется, какой ширины должно быть печатаемое поле в символах. Обычно это определяет самое большое по величине печатаемое число, которое ожидаетcя в результате выполнения программы. Иногда этот размер определяется шириной предоставленного поля в форме или таблице. Элемент ширины необходимо задать так, чтобы он хорошо подходил как к самым большим, так и к самым малым числам. Можно разработать формат для корректного вывода самого большого числа, но если нужно обеспечить заданную точность, то лучше использовать набор условных операторов с разными вариантами формата вывода.

Чаще всего применяются элементы форматирования, приведенные на рис. 1.5.4-6.



--> // ***Примеры часто используемых элементов форматирования***

-->

--> // **%d** – ***управляющие символы для вывода целого десятичного числа***

-->f1 **=** msprintf('масса: %dграмм ', 2358)

масса: 2358 грамм

-->

--> // **%f** – ***управляющие символы для вывода дробного десятичного числа***

-->f1 **=** msprintf('Время: %f секунд', 1.432)

Время: 1.432 секунд

Рис. 1.5.4-6 Примеры часто используемых элементов форматирования

Вывод некоторых символов, таких как, например, –**'**, нужно сопровождать слешем **\'**, а, кроме того, можно добавлять различные управляющие последовательности, например, **\n** – перевод строки (***Приложение 1.2,   
табл.1.2.4-5***)*.*

***Разветвляющиеся программные структуры***

К ***разветвляющимся программным структурам*** относятся оператор **if** и операторы множественного выбора **select**.

***Условный оператор* if…end** в общем виде имеет следующую структуру:

**if** *ЛогическоеВыражение****1* then**

*Операторы****1***

**elseif** *ЛогическоеВыражение****2* then**

*Операторы****2***

*…*

**else**

*Операторы****n***

**end**

Правила записи логических выражений описаны в **п. 1.2***.***3.**

***Следует отметить, что во всех операторах разветвлений ключевое слово* then*может быть опущено.***

Эта структура допускает несколько частных вариантов. Простейший – ***усеченное разветвление***, имеет следующий вид:

**If** *ЛогическоеВыражение*

*Операторы*

**end**

Напомним, что если *ЛогическоеВыражение* принимает значение *%***T**  
(«Истина»), то выполняются *Операторы*, составляющие тело структуры **if...end**. Оператор **end** указывает на конец перечня *Операторов*. *Операторы* в списке разделяются запятой или точкой с запятой. Если *ЛогическоеВыражение – %***F** («Ложь»), то *Операторы* не выполняются. Эта конструкция может быть записана как в одну строку (однострочный **if),** так и в несколько, что соответствует приведенному выше формату оператора.

Ниже приведен пример использования простейшего усеченного разветвления (рис. 1.5.4-7.).

|  |
| --- |
| --> // ***Пример усеченного разветвления***  -->  -->a = 4;  -->ifa> 0 r = sqrt(a); end // ***Одно строковый*if*, усеченное разветвление***  --> r  r =  2. |

Рис. 1.5.4-7 Пример усеченного разветвления

Эта конструкция оператора **if** довольно часто используется при написании программ, однако не всегда удобна, поскольку имеет только одну выполняемую ветвь разветвления. Так, в примере, показанном на рис.1.5.4-7, выполняется следующее: если значение **a**положительно, то корень вычисляется, а если значение **a**не положительно, то вычисление корня не происходит, но программа не выдает сообщения об этой ситуации. Для того, чтобы отслеживать ситуации по обеим ветвям разветвления, нежно использовать другую конструкцию оператора **if**.

Вторая частная конструкция оператора **if** называется ***стандартным разветвлением***:

**If** *ЛогическоеВыражение*

*Операторы****1***

**else**

*Операторы***2**

**end**

Здесь, если *ЛогическоеВыражение истинно* выполняются *Операторы****1***, в противном случае выполняются *Операторы****2***.

Пример, в котором выполняется стандартное разветвление, приведен на рис.1.5.4-8. Здесь использованы две функции **raz1** и **raz2**, первая из которых реализует стандартное разветвление, выполненное в одну строку, а вторая – в несколько строк и при этом реализуют одинаковые действия. При **а=4** выводится значение **х=2**, а при **x=-4** – сообщение: «**Подкоренное выражение < 0**».

|  |
| --- |
| -->// ***Загрузка сценария* РИС15408 *выполнение функций*raz1 *и* raz2**  -->  -->clear  --> exec('РИС15408.sce');  --> a = 4; x = raz1(a)  x =  2.  -->  -->a = -4; x = raz2(a)  Подкоренное выражение < 0 |

Рис. 1.5.4-8 Примеры стандартных разветвлений

Рассмотрим ***вложенное разветвление*** на примере программной реализации следующего разветвления:



Для того, чтобы полностью отразить структуру сложного разветвления, была разработана функция с именем **raz**, а для того чтобы проверить правильность работы функции, организующей разветвления. В примере произведено трехкратное обращение к функции с исходными данными, соответствующими доступу к каждой ветви разветвления.

На рис.1.5.4-9 приведен текст функции **raz**, реализующей вложенное разветвление, трехкратное обращение к функции **raz** и результаты.

|  |
| --- |
| --> // ***Загрузка сценария* РИС15409 *выполнение функции* raz**  -->  --> exec('РИС15409.sce');  -->p =raz(-2, 1) // ***Проверка 1-й ветви разветвления***  p =  1.  -->  -->p = raz(3, 1) // ***Проверка 2-й ветви разветвления***  -->p=9  p =  9.  -->  -->p = raz(1, 1)// ***Проверка 3-й ветви разветвления***  p =  1. |

Рис.1.5.4-9 Примеры, реализующие вложенное разветвление

***Оператор множественного выбора –*select** используется для осуществления множественного выбора**:**

**select** *Переменная*

**case***Значение1 then Операторы1*

**case***Значение2 then операторы2*

**…**

**else***Операторы*

**end**

Если *Переменная* заголовка **select** имеет значение какого-либо *Значение...,* то выполняется соответствующий ему блок операторов **case**, в противном случае – список операторов, записанных после **else**. То есть при выполнении блока **case** исполняются те *Операторы…*, для которых *Значение* совпадает со значением *Переменной*.

Поясним использование оператора **select** следующим примером вычисления значения t:



На рис. 1.5.4-10приведена функция **multifunc**, реализующая множественное разветвление с использованием оператора **select**, и обращение к ней при исходных данных, позволяющих проверить каждую ветвь разветвления.

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15410 *и выполнение функции* multifunc**,  --> // ***реализующей множественное разветвление***  -->  -->clear  --> exec('РИС15410.sce');  --> r = multifunc(5, 1) // ***Проверка 1-ой ветви разветвления***  r =  5.  --> r = multifunc(5, 2) // ***Проверка 2-ой ветви разветвления***  r =  25.  --> r = multifunc(5, -1) // ***Проверка 3-ей ветви разветвления***  r =  0.0384615  --> r = multifunc(5, 6) //***Проверка 4-ой ветви разветвления***  r =  0.027027 |

Рис. 1.5.4-10 Обращения к функции **multifunc**

***Базовые разветвляющиеся программные структуры*** Scilabпозволяют описывать широкий класс задач: нахождение максимального и минимального значения двух переменных; нахождение максимального и минимального значений нескольких переменных; вычисление значений сложных функций, зависящих от сложных условий; выбора различных путей вычислительного процесса и другие.

Рассмотрим несколько простых примеров, реализующие базовые разветвляющие программные структуры.

Реализовать функции, которые присваивают переменной **f** наибольшее из значений двух переменных **x**, **y** (1.5.4-11)***.***

При реализации функции **fmax\_a** используется стандартное разветвление, а при реализации функции **fmax\_b–**усеченное.

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15411*и выполнение функций*fmax\_a *и* fmax\_b**  --> // ***реализующих нахождение* max *из 2-х перем****енных*  -->  -->clear  --> exec('РИС15411.sce', 0);  --> r = fmax\_a(10, 3)  r =  10.  --> r=fmax\_b(10,3)  r =  10. |

Рис. 1.5.4-11 Нахождение максимума двух переменных, реализованное с   
помощью стандартной и усеченной разветвляющихся структур

***Обратите внимание, что:***

**Ifx>ythenf = xelsef = y*соответствует функции* f = max(x,y),**

**Ifx<ythenf = xelsef = y*соответствует функции* f = min(x,y),**

**где x и y любые арифметические выражения.**

Реализовать функцию, которая вычисляет минимальное значение трех выражений **r=min(a+b,c2,)**(рис. 1.5.4-12).

В функции **minХ** выбор наибольшего (наименьшего) из значений переменных или выражений числом более двух сводится к последовательному применению усеченных разветвлений. Алгоритм, реализующий эту задачу, относится к числу ***базовых*** алгоритмов выбора наименьшего из нескольких значений. Основой алгоритма является усеченное разветвление. Здесь первоначально переменной **R** присваивается значение первого из выражений. Истинное значение наименьшего значения определяется путем последовательного сравнения со всеми остальными значениями выражений заданной последовательности.

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15412*и выполнение функций* minx**  -->  -->clear  --> a = 1; b = 5; c = 10; d = 1;  --> exec('РИС15412.sce');  -->R = minx(a, b, c, d) // ***Использование функции пользователя* minX**  R =  1. |

Рис. 1.5.4-12 Реализация вычисления минимум из элементов массива

***Регулярные циклические структуры и оператор* for**

***Оператор цикла* for...end** обычно используется для организации вычислений с заданным числом повторений циклов. Конструкция такого цикла имеет следующий вид:

**for***vаr= s : d : e*

*Оператор****1***

***….***

*Оператор****n***

**end**

где *s* - начальное значение переменной цикла *var, d* - приращение этой переменной и *е* – конечное значение управляющей переменной, при превышении которого цикл завершается. Возможна и запись в виде *s:е* (в этом случае *d=1*). Список выполняемых в цикле инструкций завершается оператором **end**.

Рассмотрим пример на рис.1.5.4-13, где при заданных действительных числах **a, b (a<b),**и целом числе **n**, **необходимо вычислить выражение s=,** где **,** если ****

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15413 *и выполнение функции* summa**,  --> // ***вычисляющее выражения* p**  -->  -->clear  -->exec('РИС15413.sce', 0);  --> a = 4; b=2; n=5;  --> summa(a, b, n)  Ошибка в исходных данных b<a  -->  --> a = 2; b = 4; n = 5;  --> p = summa(a, b, n)  p =  -0.6114314 |

Рис. 1.5.4-13. Вычисление выражения ****

Решение задачи требует разработки 2-х функций: функции **slog**, предназначенной для вычисления **i**-го слагаемого и функции **summa**, которая вычисляет заданное выражение****. Функции **slog** и **summa** помещены в сценарий, сохраненный в файле с именем главной функции **summa**. Запуск на выполнение осуществляется из командного окна обращением к функции **summa**. Предварительно переменным **a**, **b**и **n** присваиваются числовые значения. Проверка правильности ввода исходных данных, предусмотрена в функции **summa.** В случае если **b>a,** происходит вычисление значения **i-**го слагаемого, накапливание суммы и, по окончании цикла, умножение суммы на величину **h**, с последующим выводом результат в командной строке, иначе в командной строке выводится сообщение «Ошибка в исходных данных **b<a**».

Необходимо отметить, что наиболее часто **оператор цикла for...end** используется при обработке векторов и матриц.

В качестве примера использования оператора **for...end** вычислим сумму элементов вектора**х** с использованием функции **summa**и входными параметрами, которые служат вектор **x**и число элементов **n**.Значение элементов вектора определены в командном окне (рис. 1.5.4-14).

|  |
| --- |
|  |
| -->// ***Загрузка сценария* РИС15414*и выполнение функции*summa**  -->  -->clear  --> exec('РИС15414.scе');  --> x = [1 2 3 4 5]; n = length(x);  -->s = summa(x, n) // ***Обращение к функции пользователя* summa**  s =  15.  -->  --> s = sum(x) // ***Проверка с использованием встроенной функции* sum**  s =  15. |

Рис. 1.5.4-14 Функция **summa**, вычисляющей сумму элементов массива,  
и обращения к функции **summa** и встроенной функции **sum**

Количество элементов массива **х** определяется функцией **length**. Кроме обращения к функции **summa**в командном окне предусмотрена проверка результата вычислений с использованием встроенной функции **sum(x)**.

В цикле может быть использован оператор **continue**, который передает управление в следующую итерацию цикла, пропуская операторы, которые записаны за ним, причем во вложенном цикле он передает управление на следующую итерацию основного цикла. Оператор **break** может использоваться для досрочного прерывания выполнения цикла (например, при отладке участка программы). Как только он встречается в программе, цикл прерывается. Примеры использование **break** и **continue** будут приведены ниже.

Кроме простых регулярных циклов в Scilab имеется возможность организации ***вложенных циклов***. Пример вычисления суммы элементов матрицы **а**, требующий использования вложенных циклов, приведен на   
рис. 1.5.4-15.

|  |
| --- |
|  |
| -->// ***Загрузка сценария* РИС15415 *и выполнение функции* vzikl**  -->  -->clear  --> a = [1,2,3;4,5,6;7,8,9]; n = 3;  --> exec('РИС15415.sce');  -->  --> s=vzikl(a)  s =  45. |

Рис. 1.5.4-15 Использование вложенных циклов для вычисления   
суммы элементов двумерного массива

В регулярных циклах число повторений должно быть определено заранее. При этом в одних случаях это число задано явно (константой или вводимым значением переменной), а в других случаях его надо предварительно вычислить.

Возможны и такие случаи, когда число повторений цикла не фиксируется в алгоритме в явном виде, а определяется неявно граничными значениями и шагом изменения не­которых переменных.

Рассмотрим пример реализации функции, которая вычисляет   
**y(x)=sin(x)** при значе­ниях **x**, изменяющихся на отрезке **[a;b]** с шагом **h**(рис.1.5.4-16).

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15416 *и выполнение функции* ziklh**  -->  -->clear  -->a = 1; b = 2; h = 0.2; // ***Исходные данн*ые**  --> x = [a:h:b] // ***Формирование массива* х**  --> exec('РИС15416.sce', 0);  --> [x, y] = ziklh(a, b, h);  --> [x,y]' // ***Вывод таблицы значений функции* y(x)=sin(x)**  ans =  1. 1.2 1.4 1.6 1.8 2.  0.841471 0.9320391 0.9854497 0.9995736 0.9738476 0.9092974 |

Рис.1.5.4-16 Вычисление таблицы значений функции **y(x)=sin(x)**

Как уже отмечалось, в циклических структурах часто используются операторы, влияющие на их выполнение. Так, в примере рис. 1.5.4-17 показано применение оператора **break**для досрочного прерывания выполнения цикла. В качестве условия досрочного прерывания цикла используется условие **i==5**.

|  |
| --- |
| --> // Пример досрочного прерывания цикла  -->  -->clear  -->fori = 1:10  > if i==5 break, end  >end  i =  1.  i =  2.  i =  3.  i =  4.  i =  5. |

Рис. 1.5.4-17 Прерывание программы с применением оператора **break**

Оператор **continue**, прерывая цикл, передает управление в следующую итерацию цикла, про­пуская операторы, которые записаны за ним, причем во вложенном цикле он пере­дает управление на следующую итерацию основного цикла. На рис.1.5.4-18 приведен пример вычисления суммы и произведения положительных элементов матрицы **b(3,3)**.

|  |
| --- |
|  |
| --> // ***Пример использования оператора* continue**  --> // ***Загрузка сценария* РИС1541*8 и выполнение функции* prer**  -->clear  --> b = [3 6 -1; -3 5 6 ; 2 -2 4];  --> n = 3;  --> exec('РИС15418.sce');  -->  --> [s, p] = prer(a, n)  p =  362880.  s =  4. |

Рис. 1.5.4-18 Прерывание выполнения цикла с применением   
оператора **continue**

Оператор **return** обеспечивает нормальный возврат в вызывающую функ­цию и в режим работы с клавиатурой. Пример применения оператора **return** представлен на рис. 1.5.4-19.

|  |
| --- |
| -->// ***Пример использования оператора* return**  --> // ***Загрузка сценария* РИС15419 *выполнение функции* det**  -->clear  -->  --> A = [];  --> det(A)  ans =  1.  -->  -->A = [1];  -->det(A)  *Неопределённая переменная 'd' в функции 'det'.* |

Рис. 1.5.4-19 Применение оператора **return**

В данном примере показано, что если матрица **A** пустая, то будет выведено значение **1**, иначе, система выводит сообщение:

«Неопределённая переменная **'d'** в функции **'det**'».

Базовые алгоритмы регулярной циклической структуры позволяют описать широкий класс задач: *формирования значений массивов*, *вычисление значений функций от одной и более переменных с заданным диапазоном и шагом их изменения; вычисление конечных сумм и произведений значений величин; ввод, формирование, обработка и вывод элементов массива и многие другие.*

Рассмотрим несколько простых примеров, реализующие базовые алгоритмы разветвляющих структур.

***При заданных числа х* b1,b2,…,bn, *требуется вычислить их сумму*: f(b1)+f(b2)+…+f(bn),**где

**

Программная реализация задачи приведена на рис.1.5.4-20. Для решения поставленной задачи создадим функцию **fb(x)**, которая реализует алгоритм вычисления значения **y=f(x).** Функция имеет один входной параметр – **х**, который при обращении к функции принимает текущее значение элемента массива **b(i),** и один выходной параметр **y**. Обращение к функции происходит в цикле, организованном для вычисления суммы значений **fb(x)**.

|  |
| --- |
| -->//***Загрузка сценария* РИС15420 *и выполнение функции* fb*, реализующ***ей  --> // ***вложенное разветвление и вычисление суммы элементов массива***  -->clear  -->b = [3546961012];  --> exec ('РИС15420.sce');  s =  409.66667 |

Рис. 1.5.4-20**.** Вычисление сумы элементов массива, используя функцию **fb**

Для вычисления суммы значений функции f**b(х)** создан сценарий **РИС15420**, в котором сначала описывается функцияf**b(х),** а затем задаются количество чисел (**n=10**) и вектор их значений (**b**), и организован регулярный цикл для обращения в функции (**fb)** для вычисления суммы.

Перед решением задачи созданный сценарий **РИС15420**, следует загрузить с использованием команды **exec**, а затем запустить его на выполнение из  
***Командного окна***.

***Сформировать из произвольных чисел матрицу* а(3,4). *Вычислить и вывести вектор b, каждый элемент которого есть среднее арифметическое элементов соответствующей строки матрицы* а (**рис. 1.5.4-21)**.**

На рис. 1.5.4-21 приведен сценарий с именем **РИС15421**, где определена матрица **а**, состоящая из **3-**х строк и **4**-х столбцов и организован цикл по количеству формируемых элементов матрицы **b** путем обращения к вспомогательной функции **sred\_ar**. В самой функции **sred\_ar**реализован *базовый* алгоритм вычисления среднего арифметического значений элементов матрицы. В данном случае под вектором понимается **i**-я строка матрицы **а**. Алгоритм заключается в вычислении суммы нужных элементов, реализованный в регулярном цикле, с последующим делением на количество просуммированных значений, то есть на количество элементов в строке (**m**).

|  |
| --- |
| --> // ***Загрузка и выполнение сценария* РИС15421*c встроенной функцией* sred\_ar**  -->  -->b = [-1256; 0.60.10.50.55; 12.36.7248];  --> exec('РИС15421.sce', 0);  --> a'  ans =  3. 0.4375 7.755 |

Рис. 1.5.4-21 Сценарий формирования матрицы **a**,вычисления среднего арифметического значения элементов матрицы и результаты его работы

***Создать функцию, которая вычисляет сумму положительных значений элементов матрицы* b,** при **b(i,j)>0.**

На рис. 1.5.4-22 приведено решение задачи, а затем пример вычисления суммы элементов вектора и матрицы с использованием стандартной функции **sum**. В данном примере элементы матрицы заданы положительными значениями специально, чтобы проверить на совпадение результаты работы созданной функции **sumf**и встроенной функции **sum**.

|  |
| --- |
|  |
| -->// ***Загрузка сценария* РИС15422*и выполнение функции* Sumf*,***  --> // ***вычисляющей сумму положительных элементов массива*B**  -->  -->clear  -->exec('РИС15422.sce', 0);  --> B = [1 2 3; 4 5 6];  -->  --> s1 = Sumf(B)  s1 =  21.  -->  --> s1 = sum(B)  s1 =  21. |

Рис. 1.5.4-22 Вычисление суммы положительных элементов матрицы с использованием функции пользователя **Sumf**и встроенной функции **sum**

**Создать функцию**, **которая формирует из чисел матрицы A*размером* 3x4 *одномерный вектор, каждый элемент которого есть сумма элементов соответствующих столбцов матрицы* A.**

Решение задачи с использованием функции **Summ**, вычисляющей суммы элементов в столбцах матрицы **А(3,4)**, и формирующий из них элементы вектора **Х(4)**. Вызов функции и результаты выполнения приведено на   
рис. 1.5.4-23.

|  |
| --- |
|  |
| -->//***Загрузка сценария* РИС15423*и выполнение функции* Summ,** --> //***которая формирует массив*b*из сумм элементов в столбцах массива* А**  -->  -->clear  --> A = [1 3 4 5; 4 7 8 9; 9 8 7 6];  --> exec('РИС15423.sce', 0);  --> X = Summ(A);  -->  --> X'  X =  14. 18. 19. 20. |

Рис. 1.5.4-23 Программа формирования элементов массива **Х** равным   
суммам элементов соответствующих столбцов массива **А**

Кроме алгоритмов, представленных в приведенных выше примерах, к базовым алгоритмам обработки массивов можно отнести алгоритмы: формирования элементов массива, нахождения количества элементов матрицы при заданном условии, нахождения суммы и произведения значений элементов матрицы при заданном условии, обмен элементов столбцов или строк матрицы, доступ к элементам массива и многие другие.

***Итеративные циклические структуры и while***

Общий вид *итеративной циклической* структуры с предусловием **while…end***можно* представить следующим образом:

**While** *ЛогическоеВыражение*

…

*Операторы*

*…*

**end**

Отличительной особенностью этой структуры является то, что *Операторы*, расположенные в теле структуры, выполняются только в том случае, если   
*ЛогическоеВыражение* равно **– %T**(**True**). Как только *ЛогическоеВыражение* становится **%F**(**False**), происходит выход из структуры повторения, и управление передается на инструкцию, расположенную после ключевого слова **end**.

Рассмотрим простой пример вычисления длины окружности.  
Сценарий, сохраненный в файле с именем **РИС15424**, служит для многократного вычисления длины окружности по вводимому пользователем значению радиуса **r**, где диалог реализован с помощью команды**input** и **disp**. Cтроки, связанные с вводом переменной r и вычислением длины окружности, включены в управляющую структуру **while...end**. Это необходимо для циклического повторения вычислений при вводе различных значений **r**. Пока **r>0**, цикл повторяется. Но стоит задать **r<0**, вычисление длины окружности перестает выполняться, а цикл завершается. Поскольку во второй строке программы величина **r** определена равной **0**, цикл выполняется хотя бы один раз.

|  |
| --- |
| --> //***Загрузка сценария* РИС15424*и его выполнение***  -->  -->clear  -->exec('РИС15424.sce');  Введите окружность радиусом r=4  Длина окружности l =25.132741  Введите окружность радиусом r=-1  r<0 |

Рис. 1.5.4-24 Сценарий, использующий операторы  
итеративной циклической структуры

При организации цикла с предусловием блоки тела цикла, следующие за блоком, в котором проверяется условие выхода из цикла, выполняются всякий раз, когда логическое выражение принимает значение **True**. При первом невыполнении этого условия происходит выход из цикла. Таким образом, возможен случай, когда тело цикла не будет выполнено ни разу.

При вычислении значений функций или числовых последовательностей (например, арифметической прогрессии), их часто записывают в виде специальных сумм или произведений, называемых рядами. Вычисление суммы членов ряда проводится с использованием итерационных циклических алгоритмов, которые позволяют вычислять приближенное значение результата с заданной точностью. При этом для вычисления отдельных слагаемых (или сомножителей), как правило, используются рекуррентные формулы.

***Рекуррентная формула*** – это такая формула, которая позволяет вычислить значение **n**-го члена последовательности, используя значение одного или нескольких предыдущих членов этой последовательности.

Рассмотрим некоторую последовательность, содержащую бесконечное число членов: **a0,a1,a2,a3,…,ai,…,an,…** В таких задачах требуется вычислять члены последовательности до тех пор, пока очередной вычисленный член не будет удовлетворять некоторому условию.

Если задано некоторое число ε, условия окончания итерационного процесса могут, например, быть следующими:

**an<ε**– для убывающей последовательности;

**an>ε**– для возрастающей последовательности;

**|an|<ε**– для убывающей знакопеременной последовательности;

**|an+1-an|<ε** – для некоторых других типов последовательностей.

***При заданной возрастающей последовательности***



***требуется реализовать функцию, которая вычисляет все члены последовательности, до тех пор, пока значение очередного члена не превысит некоторое заданное число* d, *например*, (3<d<100).**

В нашей задаче для вычисления любого члена последовательности можно воспользоваться формулой **an=xn/3n**, где **n=0,1,2,…** – номер члена.

Во избежание зацикливания, в данном примере цикл будет выполняться не более **100** раз, даже если очередной член последовательности будет оставаться меньше **d** (рис. 1.5.4-25).

|  |
| --- |
| --> // ***Загрузка сценария* РИС15425 *и выполнение функции* Pos**,  -->// ***реализующей вычисление членов последовательности***  -->  -->clear  -->exec('РИС15425.sce', 0);  --> Pos(5,9)  n =  1.  a =  3.  n =  2.  a =  9. |

Рис. 1.5.4-25 Вычисление членов последовательности с использованием   
итеративного цикла

К сожалению, во многих задачах непосредственный подсчет очередного члена связан с вычислительными трудностями. В этом случае целесообразно использовать рекуррентную формулу, которая позволяет вычислить значение переменной на следующем шаге, используя ее значение на текущем шаге – **an+1=anq**или **an=an-1q**.

***Вычислить значение функции точност*ью ε=0.01 *при* х=0.1*, каждый член которой представляет собойзнакопеременную убывающую последовательность*:**

****

Вычисление с заданной точностью **ε** означает, что суммирование членов ряда надо продолжать до тех пор, пока очередной вычисленный член ряда не станет меньше по абсолютной величине числа **ε**(рис.1.5.4-26).

Приведем вывод рекуррентной формулы для заданного в примере ряда. Формула для **n**-го члена приведена в задании:

тогда формула (n+1)-го члена

.

Разделив **an+1**член на **an**, получим выражение для q:



Таким образом, рекуррентная формула для данного ряда:

Выбор начального значения номера члена ряда (**n**) для нашего случая будет **n = 0**, так как при подстановке этого значения в формулу **n-**го члена рядаполучим значение первого члена, равного**x-1**или  
**a0=x-1***.*

Таким образом, выражение для **q** можно получить, разделив **(n+1)-**й член последовательности на **n-**й.

|  |
| --- |
| --> // ***Загрузка сценария* РИС15426*и выполнение функции* Pos2***,*  -->// ***реализующей вычисление суммы ряда***  -->  -->clear  -->exec('РИС15426.sce', 0);  -->  --> s = Pos2(0.1,0.01)  s =  -0.9 |

Рис. 1.5.4-26 Вычисление суммы ряда c использованием  
рекуррентной формулы

**1.5.5. Средства отладки sce-файлов**

***Понятие отладки программ***

***Отладка программ*** – это процесс выявления места расположения ошибок и нахождения их причины. В Scilab для этого предусмотрено множество вспомогательных средств [13]. Рассмотрим некоторые средства отладки ***Редактора* sce-**файлов.

Одним из таких средств является возможность отображать разными цветами элементы **sce-**файлов (***переменные, операции, константы, операторы, ключевые слова языка программирования, комментарии*** и т.д.), а также,в соответствии с правилами технологии структурного и процедурного программирования, имеется возможность выравнивания строк программного кода **sce-**файлов в зависимости от используемых ключевых слов и управляющих конструкций. Одновременно с этими действиями ***Редактор*** может автоматически выполнить ***синтаксический контроль*** программного кода и сигнализировать об этом различными сообщениями и действиями.

***Обратите внимание, что семантические (смысловые) ошибки решаемой задачи система отладки Scilab идентифицировать не может!***

Простейшие программы обычно не нуждаются в использовании специальных инструментов отладки. Корректного выполнения программного кода **sce-**файла, как правило, можно достичь после нескольких запусков   
**sce-**файла с использованием контрольных (тестовых) входных данных. Можно также задать просмотр результатов промежуточных вычислений, удалив в программном коде символы точки с запятой (;), которыми завершаются инструкции, либо ввести дополнительные переменные для отображения хода вычислений, а после проверки вернуть программный код **sce-**файлов его исходному виду.

Если ошибка возникает в процессе выполнения программного кода, Scilab пытается ее идентифицировать самостоятельно, а затем отображает в ***Командном окне*** сообщение, описывающее приблизительный характер ошибки и номер строки кода, в которой она обнаружена. Если щелкнуть мышью на подчеркнутом фрагменте данного сообщения, то соответствующий **sce**-файл раскроется в окне ***Редактора***, и курсор будет указывать на строку с ошибкой.

Однако ***Отладчик*** не всегда правильно отражает реально происходящие процессы. Если вы имеете дело со сложным программным кодом, то простыми приемами проверки не обойтись. В этом случае полезно воспользоваться специальными инструментами отладки.

Одним из эффективных средств отладки является размещение в тексте программного кода так называемых ***Точек останова* (*Breakpoint*)**, что позволяет проводить пошаговое выполнение программного кода, при котором имеется возможность просмотреть содержимое различных **Рабочих областей**(то есть получить значения всех переменных, используемых в функциях). Такая отладка может быть выполнена в Scilabc помощью функции **debug** и ее команд отладки в ***Командном окне***.

Обычная последовательность действий при отладке программы заключается в следующем:

1. Подготовка к отладке.
2. Установка (отключение) точек останова.
3. Запуск **sce**-файла и анализ результатов его выполнения.
4. Изменение программного кода **sce**-файла.
5. Устранение проблем и завершение отладки.

Рассмотрим вышеперечисленные действия более подробно.

***Отладка сценариев в среде в Scilab debug***

Для отладки сценариев используется функция **debug.** Этот режим позволяет управлять точками останова, запуском выполнения с остановкой на ошибке, выполнять сценарий по шагам и некоторыми другими действиями.

После ввода команды или функции **debug**, Scilab переходит в режим отладки с приглашением **debug>**,и ждет ввода одной из следующих команд:

* **(h)help***–*получение справки отладчика;
* **(q)quit** *–* окончание режима отладки и возврат к обычному режиму Scilab;
* **(w) where** или **bt** – показ содержимого стека вызовов;
* **(e)exec** или **(r)run** – запускает на выполнение;
* **(d) disp** *Переменная****,*** или**(p) print** *Переменная* – вывод на экран значение *Переменной*;
* **(c) continue**–продолжение выполнения;
* **(a) abort**–прерывание выполнения;
* **step(n) next** или **next** – продолжение выполнения до следующего выражения;
* **step (i) in** или **in***–* продолжение выполнения следующего выражения во вложенной функции;
* **step (o) out** или **out**–продолжение выполнения следующего выражения в вызывающей функции;
* **(b) breakpointnl**или **breaknl**– остановка функции на строке **n** и по условию **l**, добавление точки останова в функцию на строке **n**, причем если указано условиеl, точка останова добавляется только в том случае, если условием является %**t**;
* **(del) delete n** *–* удаление всех точек останова, или если указано значение **n**, то удаляется точка останова с номером **n**;
* **enable n** – включение всех точек останова, или если указано **n**, включает точку останова с номером **n**;
* **disable n** – отключение всех точек останова, или если указано **n**, отключение точки останова с номером **n**;
* **(s) show n** – показ всей информации о точках останова, или если задано значение **n**, показ точки останова с номером **n.**

Рассмотрим несколько примеров (рис. 1.5.5-1).

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС15501 *и выполнение функции* debug**  -->  -->clear  -->exec('РИС15501.sce', 0);  --> // ***Начало отладки***  debug  break test\_debug  [exec](https://help.scilab.org/docs/6.0.1/ja_JP/exec.html)test\_debug  [where](https://help.scilab.org/docs/6.0.1/ja_JP/where.html)  stepin  [disp](https://help.scilab.org/docs/6.0.1/ja_JP/disp.html) val  continue  quit  function v=compute(val)  v = 1;  for i =1:val  v = v \* i;  end  endfunction  debug  break compute 4 "i == 5"  [exec](https://help.scilab.org/docs/6.0.1/ja_JP/exec.html) compute(10)  [disp](https://help.scilab.org/docs/6.0.1/ja_JP/disp.html) val  [disp](https://help.scilab.org/docs/6.0.1/ja_JP/disp.html) i  [disp](https://help.scilab.org/docs/6.0.1/ja_JP/disp.html) v  continue  quit |

Рис. 1.5.5-1 Пример протокола отладки

* + 1. **Хранение данных и функций в библиотеках**

***Файловая система***

В Scilab реализованы средства для работы с различными типами файлов, но чаще всего текстовые файлы сохраняются с расширениями **\*.sce**, в них можно хранить как сценарии, так и данные в текстовом виде. Если же пользователь решил создать личную библиотеку, например, для хранения созданных им внешних функций, то тексты отлаженных функций необходимо сначала сохранить в файлах с расширением **\*.sci**, а затем средствами Scilab создать библиотеку двоичных файлов.

Файловая система Scilab может cодержать не только текстовые файлы, двоичные библиотечные файлы и системные папки Scilab, но и папки с файлами данных пользователя и папки с программами пользователя. Причем, следует помнить, что с точки зрения значимости системных папок особенное значение для пользователя имеет системная папка ***Home***, в которой содержатся данные для настройки среды пользователя при загрузке системы Scilab.

Для создания **sci**-файлов также используется встроенный в Scilab текстовый редактор **SciNotes**.

***Сохранение и восстановление переменных***

При решении серьезных задач или задач с большим объемом данных необходимо сохранять данные на диске.

Самый простой способ сохранить значения всех переменных – это сохранить данные области ***Обозревателя переменных (текущие переменные сессии в*** двоичном файле с расширением **\*.sod)** [13]. Для этого можно использовать функцию **save**. При обращении к ней появляется диалоговое окно**,** вкотором следует указать ***Путь к файлу*** и ***Имя файла***. По умолчанию предлагается сохранить файл в **Текущей папке**, при этом всегда целесообразно файлам давать содержательные имена.

Для восстановления сохраненных ранее данных можно использовать функцию **load**. При этом следует указать ***Путь к файлу***. После этого восстановленные данные можно использовать во вновь вводимых командах.

Команды сохранения и восстановления переменных рабочей среды, можно вызвать из командной строки:

**save(***ИмяФайла***,** '*х1*'*,* '*х2*'*,...,* '*хп*'**)**

**load(***ИмяФайла***,** '*х1*'*,* '*х2*'*,...,* '*хп*'**),**

где: *ИмяФайла*–символьная строка, содержащая путь к файлу;

*х1,х2,...,хп* – список сохраняемых или восстанавливаемых переменных, каждая из которых должна быть заключена в одинарные или двойные кавычки.

Рассмотрим примеры сохранения и загрузки переменных ***Рабочей среды*** (рис. 1.5.6-1).



-->// ***Пример сохранения и загрузки переменных Рабочей среды***

-->a = [eye](https://help.scilab.org/docs/6.0.1/en_US/eye.html)(2, 2); b= [ones](https://help.scilab.org/docs/6.0.1/en_US/ones.html)(a);

-->[save](https://help.scilab.org/docs/6.0.1/en_US/save.html)('vals.dat', a, b);

-->[clear](https://help.scilab.org/docs/6.0.1/en_US/clear.html)a

-->[clear](https://help.scilab.org/docs/6.0.1/en_US/clear.html)b

-->load('vals.dat', 'a', 'b');

-->

-->// ***Запись и чтение с использованием двоичного формата*sod**  
-->

--> a = [eye](https://help.scilab.org/docs/6.0.1/en_US/eye.html)(2, 2); b = [ones](https://help.scilab.org/docs/6.0.1/en_US/ones.html)(a);

-->[save](https://help.scilab.org/docs/6.0.1/en_US/save.html)("val.sod", "a", "b");

-->[clear](https://help.scilab.org/docs/6.0.1/en_US/clear.html)b

-->load("val.sod", "a", "b");

Рис.1.5.6-1 Пример сохранения и загрузки переменных ***Рабочей среды***

Проверить список сохраненных переменных в файле можно с помощью функции **listvarinfile**, которая имеет следующий формат:

**listvarinfile**(*filename*)

**[***names,typs,dims,vols***]=listvarinfile**(*filename*),

где: *filename–*символьная строка, путь к файлу для проверки;

*names–* матрица строк, показывающая имена переменных, сохраненных в файле;

*dims –* матрица, из которого видны размеры переменных, сохраненных в файле;

*typ*s*–* числовая матрица, показывающая типы переменных, сохраненных в файле;

*vols–* числовая матрица, которая представляет размер в байтах переменных, сохраненных в файле.

Рассмотрим пример использования функции **listvarinfile  
(**рис.1.5.6-2).



-->// ***Примериспользованияфункции*listvarinfile**

-->

--> a = eye(2, 2); b = int16(ones(a)); c = rand(2, 3, 3);

-->

--> save(fullfile(TMPDIR, "vals.sod"), 'a', 'b' , 'c')

--> [names, typs, dims, vols] = listvarinfile(fullfile(TMPDIR, "vals.sod"))

vols =

32.8.144.

dims(1)

2. 2.

dims(2)

2. 2.

dims(3)

2. 3. 3.

typs =

1 8. 1.

names =

!a !

! !

!b !

! !

!c !

Рис.1.5.6-2 Пример использования функции **listvarinfile**

***Создание библиотек функций пользователя***

**Библиотека** (**library)** представляет собой набор функций, объединенных некоторой идеей или по функциональному признаку. В системе Scilab простейшая библиотека содержит только функции, написанные средствами языка Scilab, где каждая функция хранится в отдельном файле. В терминах Scilab такие функции являются ***макросами*** [13]. Они не содержит файлов справки и функций на компилируемых языках (*примитивов*).

Объеди­нение функций в библиотеку во многих практических ситуациях является весьма удобным средством использования внешних независимых функций при создании локальных приложений. Более сложной организацией внешних откомпилированных функций, написанных, в том числе, на универсальных языках программирования, является ***модуль***.Разработка собственного модуля не представляет трудностей, однако требует более детального знакомства с внут­ренним устройством пакета Scilab. Кроме того, модули также имеют в своей основе библиотеки, поэтому для создания модулей требуется понимание работы библиотек.

К преимуществам библиотеки можно отнести следующее:

* библиотека загружается и выгружается целиком, что защищает все функции, входящие в нее.
* библиотеки легко обслуживать, так как каждая законченная функция хранится в своем файле.
* разрабатывать библиотеку может группа разработчиков, так как функции соединяются воедино только на этапе сборки.
* функции библиотеки хранятся в откомпилированном виде и, следовательно, выполняются быстрее.

Рассмотрим создание простой библиотеки функций Scilab, а также способы ее автоматической загрузки при запуске пакета.

Предположим, что имеется несколько **sci**.файлов, написанных средствами Scilab. Причем в каждом файле хранится одна функция, имя которой совпадает с именем **sci**-файла. Тогда последовательность шагов по созданию библиотеки заключается в следующем:

1. Создать откомпилированные (бинарные файлы с расширением **\*.bin**) экземпляры функций, используя функцию **genlib.** Эта функция создает библиотеку из функций, описания которых расположены в некотором каталоге. Кроме того, функция **genlib**создает индексные файлы.
2. Загрузить библиотеку в Scilab, с помощью встроенная функция **lib**.

Для создания библиотеки функций следует придерживаться следующих правил:

* Файлы, содержащие определения функций, должны иметь расширение **\*.sci**. В одном **sci-**файле могут быть определены несколько функций Scilab, од­нако только первая из них считается главной и доступна извне. Иными словами, только пер­вая функция, определенная в файле, считается общедоступной, в то время как остальные неявно полагаются внутренними функциями;
* Имя **sci**-файлов должно совпадать с именем главной функции в этом файле. Например, если имя функции **myfun**, то файл, содержащий ее, должен иметь название **myfun.sci**. Это требование является обязательным, в противном случае функция **genlib** не будет работать корректно.

После создания библиотеки в каталоге появятся новые файлы:

* **bin**-файлы, которые являются скомпилированными версиями функций в **sci**-файлах. Из этих файлов и загружаются функции;
* файл **names** – служебный файл, который содержит имена функций библиотеки;
* файл с расширением **\*.lib**, который используется сценарием погрузки и фактически является головой библиотеки.

Далее рассмотрим пример создания библиотеки (рис.1.5.6-3).

Пусть в каталоге **samplelib** размещаются два файла.



--> // ***Первый файл библиотеки* samplelib*содержит две функции***

--> // **C:\samplelib/fuction1.sci**:

--> functionу = functionl(х) // ***Перваяфункция***

>у= 1 \* functionl\_support(х)

> end

-->

--> function у = functionl\_support(x) // ***Втораяфункция***

> у = 3 \* x

>end

--> // ***Второй файл содержит одну функцию***

--> // C:/samplelib/fuction2.sci:

--> function у = function2 (x)

> у = 2 \* x

>end

Рис. 1.5.6-3 Файлы с функциями, размещенные в каталоге **samplelib**

Создадим бинарную версию функций, воспользовавшись функцией **genlib (**рис. 1.5.6-4**)**. Первый аргумент функции **genlib** пред­ставляет название будущей библиотеки, а второй указывает каталог, где разме­щены файлы функций. Заметим, что в данном случае только функции **functionl** и **function**2 являются общедоступными, а функция **functionl\_support** может использоваться только внутри библиотеки, но не вне ее.



--> // ***Запись файлов в каталог* samplelib**

-->genlib("mylibrary",   "С:/samplelib")

--> mylibrary

mylibrary =

Functions   files   location:   С:\samplelib\.functionl  function2

Рис.1.5.6-4 Запись файлов в каталог **samplelib**

В результате функция **genlib** генерирует и помещает в каталог "**C:/samplelib**" следующие файлы:

**functionl.bin** –  бинарная версия файла function1.sci;

**function2.bin** –   бинарнаяверсияфайла function2.sci;

**lib**             – бинарная версия библиотеки;

**names**          – текстовый файл, содержащий имена всех

                   функций в библиотеке.

Сразу же после вызова **genlib**, две новых функции становятся доступны окружению Scilab и могут быть вызваны, как показано на рис. 1.5.6-5.



--> // ***Пример обращения к библиотечным функциям***

-->functionl(3)

 ans =

  9.

-->

-->function2(3)

 ans =

6.

Рис. 1.5.6-5 Обращение к библиотечным функциям из командного окна

Вместе с тем, на практике нет необходимости каждый раз генерировать биб­лиотеку заново. Созданную ранее библиотеку можно загрузить посредством команды **lib**, единственный аргумент которой указывает местоположение загружаемой библиотеки в файловой системе. Фрагмент на рис. 1.5.6-6 иллюстрирует загрузку ранее созданной библиотеки.

---> // ***Загрузка файлов в библиотеку***

--> mylibrary = lib("С:\samplelib\")

ans =

--> functions   files   location:   С:\samplelib\.

functionl   function2

Рис.1.5.6-6 Загрузка файлов в библиотеку с использованием функции **lib**

***Использование стартового сценария***

При большом числе загружаемых библиотек, удобно поместить инструкции **lib** в стартовом сценарии Scilab, который автоматически исполняется при за­грузке пакета [13]. В этом случае все указанные в инструкции **lib** библиотеки будут доступны сразу же после старта Scilab. Файл стартового сценария размещается в основном ка­талоге Scilab, путь к которому можно узнать, проверив значение переменной **SCIHOME (**рис. 1.5.6-7).



--> // ***Доступ к стартовому сценарию***

-->SCIHOME

SCIHOME =

C:\Users\51FB~1\AppData\Roaming\Scilab\scilab-6.0.1

--> // Загрузкасозданнойбиблиотеки

--> mylibrary = lib("С:/samplelib/")

Рис. 1.5.6-7 Доступ к стартовому сценарию

Стартовый сценарий имеет имя **scilab** и является обычным сценарием Scilab (в том числе может содержать комментарии). Для загрузки созданной ранее библиотеки добавим в файл **scilab** следующие строки:

В результате библиотека **mylibrary** будет загружаться всякий раз при стар­те пакета Scilab.

* + 1. **Контрольные вопросы**

1. Какие функциональные структуры используются в Scilab?
2. Что такое **Примитив**?
3. Что такое **scе**-сценарий и каковы его особенности?
4. Каким образом **scе**-сценарий запускается на выполнение?
5. Что такое **scе-**функция?
6. В чем отличие файла-сценария от **sce-**функции?
7. Может ли **scе**-функция иметь несколько выходных параметров?
8. Как происходит обращение к **sce**-функции?
9. Для чего используется функция **exec**?
10. Какова структура кода сценария со встроенными функциями?
11. Что такое указатель на функцию?
12. Что такое видимость переменных?
13. Для чего используется переопределение функций?
14. Какие операторы реализуют линейные программные структуры?
15. Какие простейшие операторы ввода/вывода данных известны?
16. Какие операторы реализуют разветвляющиеся программные структуры?
17. Какие операторы реализуют регулярные циклические?
18. Какие операторы реализуют итеративные циклические структуры?
19. Какие имеются средства отладки **sce**-файлов?
20. Что представляет собой файловая системаScilab?

# 2. Средства пакета Scilab для решения задач численными методами

При решении вычислительных задач с использованием средств математических пакетов мы получаем результат, который, как правило, не является точным, поскольку при постановке задачи, и при вводе исходных данных, и в ходе вычислений возникают ***погрешности***.

*Источниками погрешностей* вычислений могут быть:

1. Неточное математическое описание реальных объектов и процессов.
2. При построении или выборе математических моделей не учитываются многие параметры, которые, по мнению специалистов, оказывают несущественное влияние на результат решения задачи, что естественно, делает формализованное описание объекта исследования приближенным.
3. Неточное задание исходных данных, так как их значения, как правило, могут быть получены в результате эксперимента.
4. Применение приближенных методов решения формализованной математической задачи. Численные методы делятся на точные и приближенные. Вместе с тем, для большинства реальных задач точных методов решения вообще не существует. Получение точного решения задачи часто связано с большими, иногда и непреодолимыми трудностями, поэтому приходится прибегать к их приближенному решению.
5. Округление исходных данных, промежуточных или окончательных результатов. Возникающая при округлении погрешность называется вычислительной погрешностью или погрешностью округления.

Простейшей количественной мерой погрешности является ***абсолютная погрешность*** [22, 23], которая определяется как **Δ=|a-a\*|,** где **a\*** – приближенное значение числа, незначительно отличающееся от его точного значения (**a**), имеющего те же единицы измерения, что и число **а**.

***Относительная погрешность*** [20, 23], это погрешность, в которой значение абсолютной погрешности отнесено к модулю приближенного значения (**a\*≠0**):



Использование относительных погрешностей удобно, в частности, тем, что они не зависят от масштабов величин и единиц измерений данных, а измеряются в долях или процентах.

Если некоторые погрешности (например, погрешности, связанные с неточностью исходных данных или неточностью описания реальных процессов) являются ***неустранимыми***, то на погрешность, связанную с использованием численных методов, реализованных в математических пакетах, можно влиять путем задания допустимой погрешности в качестве исходных данных. При этом математические модели, по которым производится оценка той или иной погрешности, могут быть включены в решатели, соответствующих задач.

Если решатели (совокупность средств, реализующих численные математические модели) в качестве параметра не содержат определение допустимой погрешности, то итерационные вычисления проводятся, ориентируясь на минимальное вещественное значение с двойной точностьюи формы выражения погрешностей вычислений.

В общем случае для полного анализа погрешностей желательно учитывать все их виды, но, следует помнить, что при решении нецелесообразно задавать допустимую погрешность существенно меньшей величины, чем погрешности, заложенные при математическом описании задачи и в исходных данных.

При решении инженерных задач оперируют с приближенными значениями величин, но выполнение действий над приближенными числами приводит также к приближенному числу, и имеет практическую ценность лишь тогда, когда можно указать степень точности, то есть оценить погрешность.

Использование выражений для абсолютных и относительных погрешностей не всегда возможно, так как точное число (**a**) чаще всего бывает неизвестно. Однако можно указать число, оценивающее абсолютное значение погрешности сверху. Максимальное положительное число, являющееся границей абсолютных значений погрешности, называется ***предельной абсолютной погрешностью*** [22]. Предельная абсолютная погрешность **Δnа**превышает абсолютное значение погрешности или, в крайнем случае, равна ей: **Δnа≥Δ**.

В качестве примера определим предельную абсолютную и предельную относительную погрешности числа **а=0.33**, взятого в качестве приближенного значения числа **a\*=1/3**.

Абсолютное значение погрешности числа **а**: **Δ=|a-a\*|=|1/3-0.33|**.

Из всех чисел, удовлетворяющих неравенству, обычно выбирают наименьшее и простое по записи число, которое и будет предельной абсолютной погрешностью. В нашем случае предельная абсолютная погрешность будет составлять **1/300** или любое большее число.

Имея значение предельной абсолютной погрешности можно рассчитать и ***предельную относительную погрешность***:



Таким образом, предельная относительная погрешность составляет порядка 10% или любое большее число.

**2.1. Полиномы**

**2.1.1. Представление, создание и использование   
полиномиальных объектов**

***Представление полиномов в Scilab***

В простейшем случае ***полиномы*** – это выражения от одной независимой переменной, например, **x** с неотрицательными целыми показателями степени:

**P(x)=a0+a1x+a2x2+…+anxn,**

где **ai**– коэффициенты при неизвестных **(i=0,1,…,n).**

  В Scilab полином такого вида является одним из типов данных (**п. 1.2**). Для создания полинома необходимо сначала сформировать вектор-строку, содержащий значения коэффициентов при неизвестных, упорядоченных по восходящим степеням. В общем виде полином **n-й** степени можно описать вектор-строкой **vPn=[a0,a1,a2,…,an]**, а его решение, то есть корни полинома, представить в виде вектора **vXn=[x0,x1,x2,…,xn].**

Так, например, полином 2-й степени Р**2(x)=6-5x+x2** в Scilabможно представить вектором **vP2=[6 -5 1]**, а его корни – **[2 3].**

При этом члены полинома, которые имеют нулевые коэффициенты, обязательно должны быть представлены в векторе **vP2**нулем. Ноль действует в векторе как заполнитель для той особой степени x, которая отсутствует в математической записи полинома. Так, например, для представления многочлена **P5(x)=21+2x-3x2+4x5**, вектор коэффициентов, представляющий этот полином, должен иметь вид: **vP5=[21 2 -3 0 0 4]**.

Другим простейшим способом создания полинома является использование системных констант **%z** и **%s**, описанной в **п.1.2.1**, имеющих полиномиальный тип (рис. 2.1.1-1).

--> // ***Создание переменной* t *типа полином***

-->t = 1 + 5\*%z

t =

1 +5z

Рис. 2.1.1-1. Использование системной константы **%z** для создания полинома

В Scilab имеется специальная функция **poly** [13], предназначенная для представления полинома (переменной полиномиального типа). Эта функция имеет два принципиально разных формата.

Первый формат **poly** создает переменную типа полинома по заданному вектору коэффициентов:

*Р***=poly(***ВекторКоэфициентовПолинома***, '***ПеременнаяПолинома***',** *'***с***'***)**,

где: *ВекторКоэфициентовПолинома* – вектор коэффициентов полинома (vP);

**с** – признак формата, заключенный в одинарные кавычки, означающий, что должен быть сформирован полином с коэффициентами, хранящимися в параметре *ВекторКоэфициентовПолинома;*

*Р* – переменная, которая после присваивания ей результата выполнения функции **poly**, представляет собой полином и приобретает полиномиальный тип (**polynomial**) (п. 1.2.1)*.*

Второй формат функции **poly** предназначен для создания переменной типа полином по заданному вектору корней полинома:

*P***=poly(***ВекторКорнейПолиномов***, ‘***ПеременнаяПолинома'***,** *'* **r***'***),**

где: *ВекторКорнейПолиномов* – вектор корней полиномов (*vХ*);

**r** – признак формата, означающий, что значения параметра *ВекторКорнейПолиномов* воспринимаются функцией как корни, для которых необходимо рассчитать коэффициенты соответствующего полинома, если признак формата не задан, по умолчанию предполагается **r**.

Использование различных значений параметра формата (**c** и **r**) показано в примере, приведенном на рис. 2.1.1-2.



--> // ***Примеры использования функции* poly *для создания полиномов***

-->

--> P1 = poly([2 3], 'x', 'r')

P1 =

6 -5x +x2

-->

--> P2 = poly([1 2 3], 'x', 'c')

P2 =

1 +2x +3x2

-->

--> vP5 = [21 2 -3 0 0 4]

vP5 =

21. 2. -3. 0. 0. 4.

--> P3 = poly(vP5, 't', 'c')

P3 =

21 +2t -3t2 +4t5

-->

--> vK5 = [21 2 -3 0 0 4]

vK5 =

21. 2. -3. 0. 0. 4.

--> P4 = poly(vK5, 'y', 'r')

P4 =

-504y2 +234y3 +53y4 -24y5 +y6

Рис. 2.1.1-2. Примеры применение функции **poly**

|  |
| --- |
| --> // ***Примеры создания матриц полиномов***  -->  -->vD = [1 %z, 2 +2\*%z, 3 -5\*%z]  vD =  1 z 2 +2z 3 -5z  -->  -->mD = [1 +%z, 2 -2\*%z; 3 +3\*%z, 4 -4\*%z]  mD =  1 +z 2 -2z  3 +3z 4 -4z  -->  --> y = %z  y =  z  -->  -->s = poly(0, 'x') // **или(s = %z) *Создание полиномиальной переменной*s**  -->p = 1 –2\*s +5.2\*s^2; // ***Выражение для формирования элемента матрицы***  -->M =[p, p-1, p-1.2] // ***Создание матрицы-строки***  M =  1 -2x +5.2x2, -2x +5.2x2, -0.2 -2x +5.2x2  -->  -->v = [p; p+1; p-p^2] // ***Создание матрицы-столбца***  V =  1 -2x +5.2x2  2 -2x +5.2x2  2x -9.2x +20.8x3 -27.04x4  -->  -->C = [p, p/2; p+1, p\*2; p, p+2] // ***Создание матрицы С(3x****2)*  C =  1 -2x +5.2x2 0.5 -x +2.6x2  2 -2x +5.2x2 2 -4x +10.4x2  1 -2x +5.2x2 3 -2x +5.2x2  *-->s = poly(0, 'x'); //***Создание переменной типа полинома**  --> // ***Полиномы для формирования элементов матрицы***  -->c11 = poly([3, 2, 1], 'x', 'c');  --> c12 =poly([4, 0, 1], 'x', 'c');  --> c13 = poly([5.3, -1.01, sin(2)], 'x', 'c');  --> c21 = poly([-3.1, 1.5, log(0.15)], 'x', 'c');  --> c22 = poly([4, 2.5, 0], 'x', 'c');  --> c23 = poly([%e, 2.4, 0.56],'x', 'c');  --> C = [c11, c12, c13; c21, c22, c23]  C =  3 +2x +x2 4 +x25.3 -1.01x +0.9092974x2  -3.1 +1.5x -1.89712x2 4 +2.5x 2.7182818 +2.4x +0.56x2 |

Рис. 2.1.1-3. Создание матриц полиномов

Чтобы создать полиномиальную матрицу заданного размера и заполнить ее полиноминальными элементами можно, воспользоваться системной переменной **%z** или создать новую полиноминальную переменную, а затем сформировать матрицу одним из стандартных способов, рассмотренных ранее в **п. 1.2.3** (рис. 2.1.1-3).

Чтобы создать полиномиальную матрицу заданного размера и заполнить ее полиноминальными элементами можно, воспользоваться системной переменной **%z** или создать новую полиноминальную переменную, а затем сформировать матрицу одним из стандартных способов, рассмотренных в   
**п. 1.2.3** (рис. 2.1.1-3).

***Оценка значений полиномов***

Для оценки значений полиномов в Scilabиспользуется функция

*A*=**horner(***p,х***)**,

где *p*– полиномиальная матрица, **x**– матрица значений переменной полиномов *p***,** а *A*– значения полиномов *p*при заданных значениях *x*(рис.2.1.1-4).

|  |
| --- |
| --> // ***Оценка значений полиномов с использованием функции*horner**  -->  -->// ***Оценка полинома для заданного значения***  -->a1 = poly(1:3, 'y')  a1 =  -6 +11y -6y2 +y3  -->  --> horner(a1, 1)  ans =  0.  -->  --> // ***Оценка полинома для матрицы значений***  -->х = [12; 34]  х =  12.  34.  --> a2 = poly([1 2 3], 'x', 'c')  a2 =  1 +2x +3x2  -->  --> h = horner(a2, х)  h =  457.  3537. |

Рис. 2.1.1-4. Оценка значений полиномов с помощью функции **horner**

***Вычисление корней полиномов***

Для нахождения корней полинома можно использовать функциюScilab

*X***=roots(***p***),**

где: *p–* полином с вещественными или комплексными коэффициентами, или вектор вещественных значений **m×1** (**1×m**) полиномиальных коэффициентов в порядке *убывания степени*; *X–* вектор корней уравнения.

В данном случае используется алгоритм, основанный на *собственных значениях сопутствующих матриц* [20], и позволяющий за одно обращение к функции **roots** вычислить все корни заданного полинома (рис. 2.1.1-5).

|  |
| --- |
| --> // ***Нахождение корней полинома функцией* roots**  -->  -->a = [-1 0 8 -8 2]; // *Коэффициенты полинома*  -->p = poly(a, 'x', 'c') // *Определение полинома*  p =  -1 +8x2 -8x3 +2x4  -->  -->X = roots(p) // ***Нахождение корней полинома по заданному полиному* p**  X =  2.306563  1.5411961  -0.306563  0.4588039  -->  --> da = size(a, "\*")  da =  5.  -->y = roots(a(da:-1:1)) // ***Нахождение корней полинома***  y =  2.306563  1.5411961  -0.306563  0.4588039  -->  -->isreal(y) // ***Проверка: являются ли корни действительными? Нет***  ans =  F  -->y = real(y) // ***Преобразование из мнимых значений в действительные***  y =  2.306563  1.5411961  -0.306563  0.4588039  -->  -->isreal(y) // ***Проверка: являются ли корни действительными? Да***  ans =  T |

Рис. 2.1.1-5. Нахождение корней полинома функцией **roots**

***Обратите внимание, что при вычислении корней полинома (функция* roots) *вектор коэффициенты полинома должен быть представлен в порядке убывания степени, в то время как при определении полинома (функция* poly) *вектор коэффициенты полинома должен быть представлен в порядке возрастания степени.***

Предполагается, что функция **roots** всегда возвращает корни полинома как комплексные числа, даже если мнимая часть равна нулю. Конвертировать их в действительные значения можно с помощью функции **real**,а для проверки типа полученных корней можно использовать функцию **isreal**.

Рассмотрим пример из ТЭЦ, который позволяет проверить, является ли устойчивой электрическая цепь с передаточной функцией вида:  
[17] (рис. 2.1.1-6).

|  |
| --- |
|  |
| --> // ***Загрузка и выполнение сценария*РИС21106**  -->  -->Vk = [-1 1 2]; // ***Задание коэффициентов полинома***  -->exec('РИС21106.sce', 0)  Pk1 = 0.5 Pk2 = -1  --> //*Корень*Pk1 *оказался в правой полуплоскости*,  --> //***следовательно, цепь неустойчива*** |

Рис. 2.1.1-6. Проверка устойчивости электрической цепи

**2.1.2. Операции и функции c полиномиальными**

**данными и рациональные дроби**

***Операции над полиноминальными данными***

С полиномами можно осуществлять операции, такие как: сложение, вычитание, умножение и деление [13]. Примеры перечисленных операций приведены на рис. 2.1.2-1.

|  |
| --- |
| --> // ***Операции над полиномами***  -->  **--> a = poly([1 2 3], 'x, 'c') //***Создание полинома* a  a =  1 +2x +3x2  --> b = poly([4 1 2] ,'x', 'c') // *Создание полинома* b  b =  4 +x +2x2  -->  --> z1= a + b //***Сложения полином*ов**  z1 =  5 +3x +5x2  -->  --> z2 = a - b //***Вычитания полиномов***  z2 =  -3 +x +x2  -->  --> z3 = a \* b //***Умножения полиномов***  z3 =  4 +9x +16x2 +7x3 +6x4  -->  --> z4 = a / b//***Делениеполиномов***  **z4 =**  1 +2x +3x2  ---------  4 +x +2x2 |

Рис.2.1.2-1. Операции над полиномами

Как видно из примеров, приведенных на рис.2.1.2-1, все операции с полиномами выполняются точно так же, как c арифметическими переменными. Например, для получения полиномиальной дроби достаточно воспользоваться обычным знаком деления (**/**). При этом полиномиальную дробь, являющуюся частным от деления полиномов, представляет рациональная дробь.

***Рациональная матричная производная***

С помощью функции **derivate** можно найти рациональную матричную производную от полиномиальной или рациональной матрицы. Функция **derivat** имеет следующий формат:

*dp* = **derivat**(*p*),

где: *p* – полиномиальная или рациональная матрица;

*dp* – рациональная матричная производная.

Функция **derivat** работает с такими выражениями, как , состоит из линейных комбинаций функций с целочисленными показателями одной независимой переменной и реализует аналитическое вычисление.

Рассмотрим несколько примеров (рис.2.1.2-2):



--> // ***Вычисление рациональной матричной производной***

-->

-->s = poly(0, 's'); // ***Создание переменной типа полино***м

-->derivat(1/s) // ***Вычисляетсяпроизводная*(1/S)'=-1/s^2**

ans =

-1

---

s2

-->

--> p1 = poly([1 -2 1], 'x', 'c')

p1 =

1 -2x +x2

-->

--> derivat(p1)

ans =

-2 +2x

-->

--> p3 = poly(ones(1, 10), 'z', 'c')

p3 =

1 +z +z2 +z3 +z4 +z5 +z6 +z7 + z8 +z9

--> derivat(p3)

ans =

1 +2z +3z2 +4z3 +5z4 +6z5 +7z6 +8z7 +9z8

Рис.2.1.2-2. Вычисление рациональной матричной производной

Scilab может выполнять операции над полиномами только с одной переменной, иначе в результате выполнения появляется сообщение об ошибке:

*«Переменные не имеют одной и той же формальной переменной».*

В Scilab имеется множество вспомогательных функций по работе с полиномами [14], в том числе, для создания специальных полиномов. В качестве примера рассмотрим создание полинома Чебышева с использованием функции **chepol** (рис.2.1.2-3), имеющей следующий формат:

*Tn***=chepol(***Порядок полинома, 'Независимая переменная'***)**,

где: *Tn*– полиноминальная переменная, которая в результате присваивания ей результата выполнения функции **chepol** представляет собой полином   
Чебышева.

|  |
| --- |
| -->// ***Использование функции*chepol*для создания полинома Чебышев*а**  -->  --> T4 = chepol(4, 'x') // ***Создание полинома Чебышева 4-го порядка***  Th =   1. -8x2 +8x4   -->  --> // ***Использование функции в выражениях***  -->P = 2 \* poly(0, 'x') \* chepol(4,'x') - chepol(5, 'x')  P =  -3x +4x3 |

Рис. 2.1.2-3. Использование **chepol**для создания полинома Чебышева

Далее приведен пример использования вспомогательной функции **pol2str**, которая преобразует полином в символьную строку (рис. 2.1.2-4).



-->// ***Преобразование полинома в символьную строку***

-->

-->p = poly([1, 2, 3], "t", "c")

p =

1 +2t +3t2 // **Значение данных типа полин*ом***

-->pol2str(p)

ans =

1+2\*t+3\*t^2 // ***Значение данных типа строка***

Рис. 2.1.2-4. Пример преобразования полинома в символьную строку

***Рациональные дроби***

**Рациональная дробь –** объект Scilab **r**, который является частным двух полиномов ***r* = num/den**[13]. Внутреннее представление рациональной дроби это список **r=rlist(num,den,[])** или то же самое, что **r=num/den**. Матрица рациональных чисел может быть определена обычным синтаксисом, например, **[r11,r12;r21,r22]** является матрицей размером **2×2**, где **rij –** рациональные числа размерностью **1x1**. Матрица рациональных чисел может быть, также определена в виде списка,

**tlist(['r','num','den','dt'],num,den,[])**,

где: **num** и **den** – матрицы полиномов.

Рассмотрим несколько примеров использования рациональных дробей (рис. 2.1.2-5).



--> // ***Представление и работа с рациональными дробями***

-->

--> s = poly(0, 's')

s =

s

--> W = [1/s, 1/(s+1)]

W =

1 1

-- ------

s 1 + s

-->

--> W' \* W

ans =

1 1

-- ------

s2 s + s2

1 1

------ -----------

s + s2 1 + 2s + s2

-->

--> Num = [s, s+2; 1, s]

Num =

s 2 +s

1 s

--> Den = [s\*s, s; s, s\*s]

Den =

s2 s

s s2

-->

--> H = Num ./ Den

H =

1 2 + s

-- ------

s s

1 1

-- --

s s

-->

--> syslin('c', Num, Den) //***Определениелинейнойсистемы***

ans =

s 2 + s

-- ------

s2 s

1 s

-- --

s s2

--> [Num1, Den1] = simp(Num, Den) // ***Рациональное упрощение***

Den1 =

s s

s s

Num1 =

1 2 + s

1 1

Рис. 2.1.2-4. Работа с рациональными дробями

Полиномы, как с действительными, так и с комплексными коэффициентами, широко используются в теории электрических цепей и теории связи. Например, при анализе нелинейных элементов электрических цепей, для аппроксимации характеристик нелинейных элементов, при расчете переходных процессов в электрических цепях, при синтезе электрических фильтров, при анализе электрических цепей операторным методом, при исследовании цепей на устойчивость. В соответствующих разделах теории электрических цепей также используются полиномы Гурвица, полиномы Батерворта, полиномы Чебышева и другие.

Рассмотрим пример построения временной зависимости напряжения на конденсаторе **uC(t)** в неразветвленной **RLC**-цепи, если цепь включается на постоянное напряжение при нулевых начальных условиях [17] (рис. 2.1.2-5).

|  |  |
| --- | --- |
|  | |
| -->//***Загрузка и выполнение сценария*РИС21205**  **Колебательный режим** | R=0.8;  L=0.3e-3;  C=0.1e-6; |

Рис. 2.1.2-5 Построение временной зависимости напряжения на конденсаторе **uC(t)** в неразветвленной **RLC**-цепи

**2.1.3 Контрольные вопросы**

1. Что такое полином?
2. Является ли полином типом данных Scilab?
3. Как создать переменную типа полинома по заданному вектору коэффициентов?
4. Как создать переменную типа полином по заданному вектору корней полинома?
5. Как создать полином с использованием системных констант?
6. Какой формат имеет функция **poly**?
7. Как задать матрицу полиномов?
8. Как оценить значение полинома?
9. Как вычислить значения корней полинома?
10. Можно ли с полиномами осуществлять операции, такие как: сложение, вычитание, умножение и деление?
11. Как вычислить производную от полинома?
12. Как преобразовать полином в символьную строку?
13. Что представляет собой рациональная матричная производная?
14. Что представляет собой рациональная дробь?

**2.2. Решения задач аппроксимации и   
интерполяции функций**

**2.2.1. Постановка задач аппроксимации   
и интерполяции**

В инженерной и научной работе существенную роль играет решение задачи ***аппроксимациифункций*** (***подгонка кривых***), поскольку зачастую реальные функциональные зависимости слишком сложны для использования или, например, их значения получены в ходе некоторого эксперимента. Задача аппроксимации и состоит в замене исходной функции другой, более простой аналитической функцией, обеспечивающей на некотором множестве точек необходимое приближение, по заданному критерию, ее значений со значениями исходной функции.

В частном случае, когда аппроксимирующая функция во всех узловых точках обеспечивает полное совпадение с аппроксимируемой функцией, задачу аппроксимации функции называют ***интерполяцией***. Интерполяционные формулы имеют не только практическое, но и теоретическое значение. Они лежат в основе квадратурных формул численного интегрирования, методов решения дифференциальных и интегральных уравнений и многих других численных методов. Все перечисленные выше аспекты рассмотрены в классических учебниках по численным методам [22, 23].

Постановка задачи ***аппроксимации*** состоит в следующем. Пусть функция задана таблично, то есть имеется набор узловых точек **xk***,* где   
**k=1,2,…,n** и значения функции **y(xk)**в этих точках. Требуется заменить ее некоторой приближенной функцией **f(x,a1,a2,…,am)***,* которая кроме аргумента **х**, зависит еще и от параметров **as**, где **s=1,2,…,m**. Задача ***аппроксимации*** состоит в том, чтобы подобрать такие значения параметров **as**, при которых аппроксимирующая функция **f(x,a1,a2,…,am)** наилучшим образом описывала бы исходную функцию. Как правило, **m<<n**, поэтому добиться того, чтобы функция **f(x,a1,a2,…,am)** давала точные результаты даже в узловых точках, как правило, не удается. Следовательно, должен быть выбран такой критерий близости, который смог бы обеспечить требуемую точность. В зависимости от решаемой задачи на практике используют различные методы и, соответственно, различные критерии близости. Чаще всего для аппроксимации таблично заданной функции используется *метод наименьших квадратов*, в котором в качестве критерия близости, аппроксимирующей и аппроксимируемой функций, используется неравенство:

,

где:  – среднеквадратическое отклонение;

 – допустимая погрешность;

**уi**– значения аппроксимируемой функции в узловых точках;

**f(xi)** – значения аппроксимирующей функции в узловых точках;

**n+1** – количество узловых точек.

Задача ***интерполяции*** функции, как уже отмечалось выше, является частным случаем задачи аппроксимации. Решение задачи интерполяции требует замены таблично заданной функции (**y(xk),** где **k=1,2,…,n)** такой функцией **f(x),**которая в узловых точках **x1,x2,…,xn**принимала бы значения заданной функции, то есть другими словами выполнялось бы ***основное условие интерполяции* f(xk)=yk,** где **k=1,2,…,n**. Чаще всего в качестве интерполирующей функции **f(x)** используется полином **p(x)**, степень которого **n-1** (на единицу меньше количества узлов интерполяции), а ***интерполяционные формулы*** [24] в зависимости от метода различаются лишь формами представления полиномов.

При большом количестве базовых точек интерполяция полиномом может оказаться малоэффективной, поэтому нередко используют интерполяцию ***сплайнами***. Идея ***сплайн-интерполяции*** [24] состоит в разбиении диапазона интерполирования на отрезки, в пределах которых используются разные функции одного вида (чаще всего алгебраические многочлены). При этом сама функция и ее производные 1-го, 2-го и 3-го порядка во всем диапазоне интерполяции непрерывны. Существуют линейные и квадратичные сплайны, однако, наиболее распространенный и точный вариант интерполяции – интерполяция ***кубическими сплайнами***, в результате которой мы имеем кусочно-непрерывную интерполяционную зависимость.

Очевидно, что решение задач и аппроксимации, и интерполяции функций являются крайне трудоемкими. Это связано с тем, что их точность во многом зависит от количества используемых узловых точек на отрезке и шага интерполяции. Поэтому на практике эти задачи чаще всего решаются с использованием средств математических пакетов [25, 26].

Цель данного раздела – продемонстрировать возможности системы Scilab для изучения базовых вопросов, возникающих при решении задач аппроксимации и интерполяции функций при помощи полиномов. Именно поэтому изложение материала второй части данного учебника мы начали с представления в системах Scilab*полиномов* и возможных операций над ними.

**2.2.2. Решение задач аппроксимации и интерполяции функций средствами пакета Scilab**

Для выполнения полиномиальной аппроксимации в системе Scilab используется функция

**[***a,err***]=datafit(***R,z,a0***),**

где: *R*– функция, с помощью которой можно вычислить разность между заданными и расчетными значениями (**y-f(x,a0,a1,…,ak))** аппроксимируемой и аппроксимирующей функциями в узловых точках;

*z* – матрица исходных данных (первая строка – массив **х**, вторая строка – массив**y**);

*a0* – вектор начальных коэффициентов (нулевой вектор из **k** элементов);

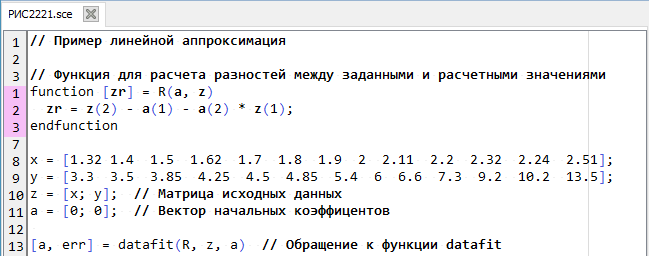
*a* – вектор коэффициентов полинома степени **k** аппроксимирующей функции (**a0,a1,…,ak**);

*err*– сумма квадратов отклонений аппроксимируемой и аппроксимирующей функции.

Функция **datafit**возвращает вектор коэффициентов полинома степени **k**, который с наименьшей среднеквадратической погрешностью аппроксимирует таблично заданную функцию. Результатом является вектор ­строка длиной **k+1**, содержащий коэффициенты полинома в порядке увеличения степеней. Как правило, степень полинома много меньше количества узлов (**k<<n**).

Перед проведением интерполяции с использованием функции **datafit***,* элементы вектора начальных коэффициентов полинома следует обнулить.

На рис. 2.2.2-1 приведен пример линейной аппроксимации таблично заданной функции. Функция **R(a,z)** определяет разности между исходной и аппроксимирующей функцией**,** а сумма квадратов отклонений возвращается через переменную **err**.





--> // ***Загрузка и выполнение файла-сценария*РИС2221**

--> clear

-->exec('РИС2221.sce', 0);

err =

17.371602

a =

-7.6955977

7.412379

--> y1 = poly(a', 'x', 'c')

y1 =

-7.6955977 +7.412379x// ***Линейный полином y1***

-->y1x = horner(y1, x) // ***Значение полинома* y1 *в точках* х**

y1x =

2.0887426 2.6817329 3.4229708 4.3124562 4.9054466 5.6466845 6.3879223

7.1291602 7.9445219 8.611636 9.5011215 8.9081312 10.909474

--> plot(x, y1x, 'mx');

--> mtlb\_hold('on');

--> plot(x, y, 'o');



Рис. 2.2.2-1. Линейная аппроксимация функцией **datafit**

**Заголовок, оси и легенду!!! Лучше линией с кружочком или крестиком!**

Рассмотрим пример кубической аппроксимации (рис.2.2.2-2).

|  |
| --- |
|  |
| --> // ***Загрузка и выполнение сценария* РИС2222 *– кубической аппроксимации***  -->  --> clear  -->exec('РИС2222.sce', 0);  err =  1.745866  a = [-47.407916 87.939048 -51.081064 10.199285]  -->P3 = poly(a', 'x', 'c')  P3 =  -47.407916 +87.939048x -51.081064x2 +10.199285x3 // ***Кубический полином y3***  -->y3x = horner(P3, x) // ***Значение полинома* P3 *в точках* х**  y3x =  3.1260106 3.5747038 3.9908489 4.3587432 4.5732778 4.8619532 5.23053  5.740204 6.5368498 7.4276265 9.0321418 7.9052934 12.787137  --> plot(x, y3x, 'mx'); mtlb\_hold('on'); plot(x, y, 'o'); |
|  |

Рис. 2.2.2-2. Кубическая аппроксимация функцией **datafit**

Если коэффициенты аппроксимирующего полинома известны, то его можно описать как встроенную функцию **def** (рис.2.2.2-3). Тогда для проведения дальнейших расчетов при вычислении значений функции в нужных точках достаточно обратиться к этой функции.

|  |
| --- |
| --> // ***Получение двух аппроксимирующих функций в явном виде***  --> a=[-7.6955977 7.412379];  --> c=[-47.407916 87.939048 -51.081064 10.199285];  --> deff('y=f1(x)','y=a(1)+a(2).\*x');  --> deff('y=f3(x)','y=c(1)+c(2).\*x+c(3).\*x.^2+c(4).\*x.^3');  --> [f1(1.85),f3(1.85)]  ans =  6.01728688.7324291  6.0173035 5.024308 |

Рис. 2.2.2-3. Вычисление значений функций в точке **x=1.85** с использованием линейной и кубической аппроксимации

Как уже было отмечено выше, интерполяция является частным случаем аппроксимации, поэтому очевидно, что для случая, когда степень аппроксимирующего многочлена на единицу меньше количества узловых точек   
(**k=n-1**), в силу единственности полинома степени **n-1**, функция **datafit** решает задачу *интерполяции*.

В следующем примерена рис. 2.2.2-4., используя в качестве узлов интерполяции**x=1,2,4,5**, требуется построить полином, задав в качестве интерполирующий функции, функцию вида **y(x)=sin(x)**, и проверить точность полученного значения полинома в точке **a=2.5**.

|  |
| --- |
|  |
| --> // **Загрузка сценария РИС2224– интерполяция в точке и его выполнение**  --> clear  -->exec('Рис2224.sce', 0)  err =  6.230D-10  c =  -0.8515488  2.7612598  -1.196154  0.1279213  P3 =  -0.8515488 +2.7612598x -1.196154x2 +0.1279213x3  --> P3x = horner(P3, 2.75)  P3x =  0.3563652  --> y = sin(2.75)  y =  0.381661 |

Рис. 2.2.2-4. Интерполяция в точке с использованием функции **datafit**

Для этого, поскольку интерполируемая функция задана аналитически, сначала вычислим значения функции **y(x)** в заданных узлах, а затем сформируем вектор **z**, заполнив его элементы вычисленными значениями интерполируемой функции. Далее выполним интерполяцию с использованием функции **datafit***,* предварительно обнулив элементы вектора начальных приближений коэффициентов **c**. В примере получено также и аналитическое выражение интерполирующей функции (полином 3-го порядка), с использованием которого вычислено значение в точке **2.5**. Следует отметить, что **err** - выходной параметр функции **datafit**, имеет небольшое значение, но не точно равен нулю, что связано с неустранимой ошибкой вычислений. Графики узловых точек и интерполяционного полинома приведены на рис. 2.2.2-4.

В Scilab для линейной сплайн-интерполяции и вычисления значения в заданной точке служит специальная функция **interpln**, которая имеет формат:

*y***=interpln(***z, x***),**

где*z*– матрица исходных данных, состоящая из вектора значений узлов и вектора значений в узлах;

*х* – точка, в которой проводится интерполяция;

*y*– результат выполнения функции **interpln***,* равный значению линейного сплайна в точке *x*.

Выполним линейную сплайн-интерполяцию и получим значение функции в точке **х=0.58** (рис. 2.2.2-5),используя таблично заданную функцию:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x** | -1 | 0 | 1 | 2 |
| **y(x)** | 4 | 2 | 6 | 1 |

|  |
| --- |
| ---> // ***Линейная сплайн-интерполяция***  -->x = [-1 0 1 2]; // ***Вектор аргументов***  -->y = [4 2 6 10]; // ***Вектор значений функции***  -->z = [x; y];  -->f = interpln(z, 0.58) // **Линейная сплайн-интерполяция в точке**  f =  4.32 |

Рис. 2.2.2-5. Линейная сплайн-интерполяция с использованием функции   
**interpln** в точке **х=0.58**

Построение кубического сплайна в Scilab состоит из двух этапов. На первом этапе вычисляются коэффициенты сплайна с помощью функции:

*koeff***=splin(***x,y***),**

где: *х* – строго возрастающий вектор значений узлов;

*y*– вектор значений интерполируемой функции в узлах, имеющий тот же размер, что и *х*;

*koeff*– выходной параметр, содержащий коэффициенты кубического сплайна.

На втором этапе рассчитывается значения интерполяционного полинома в точках, заданных вектором *t*:

*p***=interp(***t,x,y,koeff***),**

где: *х* и *y* – имеют тоже назначение, что и в функции **splin;**

*t* – вектор абсцисс, в которых требуется вычислить значения функции;

*koeff*– параметр, содержащий коэффициенты кубического сплайна;

*p* – выходной параметр, содержащий вектор ординат, являющихся значениями кубического сплайна в точках *t*.

Рассмотрим пример использования кубической сплайн-интерполяции на примере приближенного вычисления функции в точках **0.702**, **0.512** и **0.698**, не совпадающими с узлами таблично заданной функции:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x | 0.43 | 0.48 | 0.55 | 0.62 | 0.7 | 0.75 |
| **y(x)** | 1.635 | 1.732 | 1.876 | 2.033 | 2.228 | 2.359 |

На рис.2.2.2-6 приведены команды, необходимые для проведения интерполяции таблично заданной функции *кубическими сплайнами*, и вычисление значений функции в заданных точках, а также графики интерполируемой функции и интерполирующей функций.

|  |
| --- |
| --> // **Загрузка сценария РИС22206 – кубической сплайн-интерполяции** --> // **в заданных точках и его выполнение**  --> clear  --> exec('РИС22206.sce', 0)  ans =  0.702 0.512 0.608  2.2309267 1.7972605 2.0028862 |

Рис. 2.2.2-6. Выполнение кубической сплайн-интерполяции

с использованием функции **interp**в заданных точках

**2.2.3 Контрольные вопросы**

1. Как формулируется постановка задачи аппроксимация функции?
2. Как формулируется постановка задачи интерполяция функции?
3. Что такое аппроксимация функции, и в каких случаях она используется?
4. В чем отличие аппроксимации от интерполяции?
5. Какой метод аппроксимации реализован в функции **datafit**?
6. В чем суть *метода наименьших квадратов*?
7. Как оценить погрешность в *методе наименьших квадратов*?
8. Какие способы задания функций известны?
9. Что является результатом выполнения функции **datafit**?
10. В каком случае функция **datafit** решает задачу интерполяции и почему?
11. Назначение и формат функции **interpln**?
12. Что является выходным параметром функции **datafit**?
13. Каким параметром определяется тип интерполяции в функции   
    **interpln?**

**2.3. Приближенное вычисление   
производных и интегралов**

**2.3.1. Постановка задачи вычисления производных  
и конечных разностей**

Производной функции **y=f(x)** называется одна из функций:

,

которая, при каждом значении независимой переменной **х**, равна пределу отношения приращения функции к приращению независимой переменной **х** при стремлении **Δх** к нулю.

Процесс вычисления производной называется ***дифференцированием функции***. Дифференцирование происходит в соответствии с правилами дифференцирования. Производная характеризуется *порядком*, то есть производная от первой производной называется производной второго порядка, от второй – третьего порядка и так далее. Однако такое вычисление производных возможно только тогда, когда функция задана аналитически и при этом достаточно проста. Именно поэтому в случае, если функция задана таблицей значений, для нахождения производной функции в каком-либо узле заданной таблицы обычно используют интерполяционные многочлены невысоких степеней, которые хорошо приближают функцию к окрестности значений этого узла. Для этого обычно используют выражения, основанные на таблице конечных разностей. Наиболее известные из них – формулы Ньютона [22]. Но их применение возможно только, если узлы таблицы являются равноотстоящими.

Чтобы вычислить производные с использованием таблицы конечных разностей, предположим, что решается следующая задача: при заданной таблице значений функции **yi=f(xi)**, где **i=0,1,…,n**, где шаг изменения **xi** является постоянным (**h=xi-xi-1=const**), требуется определить таблицу значений производных в точках **xi**.

С использованием значений конечных разностей производные функции в точках **xi** можно определить, как:



Например, когда значения производных вычисляются с использованием правых конечных разностей:

Очевидно, что приведенная выше формула позволяет определить значения производных во всех точках, кроме **xn**. Вычислить производную в точке **xn** можно по аналогичной формуле, в которой используются левые конечные разности:



Поскольку использование этих выражений требует вычисления конечных разностей, то точность вычисления производной напрямую зависит от величины шага между узловыми точками таблицы.

Рассмотрим имеющиеся в системе Scilab средства дифференцирования функций, заданных аналитически и таблично, а также средства, позволяющие проводить вычисления конечных разностей.

**2.3.2. Вычисление производных средствами Scilab**

***Вычисление производной отаналитической функции***

Версия системы Scilab 6.01, к сожалению, не позволяет получить аналитических выражений производных от функции **f(x)**. Однако в ней имеется функция Scilab **numderivative**, предназначенная для вычисления значений производных 1-го или 2-го порядка в заданных точках от аналитических функций. Эта функция имеет следующие форматы:

**numderivative***(f,x)*

*J=***numderivative***(f,x)*

*J=***numderivative***(f,x,h)*

*h=***numderivative***(f,x,h,order)*

**[***J,H***]=numderivative(***f,x,h,order***).**

*Входные параметры*:

*f* ***–*** функция или список функций, от которых берутся производные;

*х* ***–*** вектор значений независимых переменных, в которых вычисляются производные;

*h****–*** шаг, используемый в конечно-разностных приближениях. Если *h* не указан, то шаг по умолчанию вычисляется в зависимости от *х* и порядка. Если *h*-матрица **1х1**, то она расширяется до того же размера, что и *х*;

*order****–*** порядок конечных разностей (порядок по умолчанию равен **2**), доступные значения *order*: **1, 2** или **4**.

*Выходные параметры*:

*J* ***–*** матрица (*необязательный параметр),* для функции от одной переменной размером **1х1**, для функции от двух переменных – **2х2** (значения частных производных выводятся на ее диагонали);

*H****–*** матрица (*необязательный параметр),* в которую выводятся частные производные второго порядка.

На рис.2.3.2-1 приведены два варианта вычисления значения производной в точке **х=3** от функций, описанных с использованием оператора **deff**. Причем, в первом примере значение производной вычислено с помощью функции **numderivative**, а во втором ***–*** с использованием выражения производной, полученного заранее в явном виде. Как показали приведенные вычисления, числовые значения производных в точке **х=3**, полученные двумя способами, полностью совпали.

|  |
| --- |
| --> // ***Вычисление производной от функции* f(x) *в точке* х=3**  -->  --> deff('[y] = f(x)','y = 2^x – 4 \* x'); // ***Описаниеисходной функци****и*  -->  --> numderivative(f, 3) // ***Вычисление производной функции* f *в точке* х=3**  ns =  1.5451774  -->  --> // ***Вычисление производной, заданной аналитическим выражением в точке***  -->deff('[y1] = f1(x)','y1 = 2^x\*log(2) - 4'); // ***Производная функции***  --> f1(3) // ***Значение в точке*х=3**  ans =  1.5451774 |

Рис. 2.3.2-1. Вычисление производной в точке функцией **numderivative**

На рис. 2.3.2-2 приведены примеры получения выражений производной от полинома, созданного в аналитическом виде (использование функции **derivat**)и значения производных в точках **x = [1 3 6]** (использованиефункции **numderivative**).



--> // ***Вычисление производных полинома***

-->

-->p = poly([1, 2, 3], "x", "c") // ***Определение полинома***

p =

1 +2x +3x2

-->p1 = derivat(p) // ***Определение производной от полинома***

p1 =

2 +6x

-->ps = pol2str(p) // ***Преобразование полиноминальное представление***-->// ***полинома в символьную строку функцией* pol2str**

-->

ps =

1+2\*x+3\*x^2

-->

-->deff('[sd] = fd(x)', 's = 1 + 2 \* x + 3 \* x^2') // ***Описаниефункции*fd**

--> x = [1 3 6];

--> numderivative(fd, x)

ans =

8. 0. 0.

0. 20. 0.

0. 0. 38.

-->

-->deff('sp = fp(x)', 'sp='+ ps) // ***Описание функции*fp с использованием sp**

--> numderivative(fp, x)

ans =

8. 0. 0.

0. 20. 0.

0. 0. 38.

Рис. 2.3.2-2. Вычисление производных от полинома

Функция **numderivative**позволяет вычислить и значения частных производных. На рис. 2.3.2-3приведен пример вычисления частных производных от функции трех переменных **f(x1,x2,x3)=x12x2+x22x3+x32x1** при **х1=1, x2=3** и **х3=6**.

|  |
| --- |
| --> // ***Вычисление частных производных функции трех переменных***  -->  --> deff('s = f(x)', 's = x(1)^2\*x(2) + x(2)^2\*x(3) + x(3)^2\*x(1)')  -->  -->x = [1 3 6]; // ***Вектор значений переменных***  -->numderivative(f, x, 2) // ***Вычисление второй производной***  ans =  42. 37. 21. |

Рис.2.3.2-3. Вычисление значений частных производных от функции трех переменных с использованием функции **numderivative**

Вычислить дифференциальную ёмкость нелинейного элемента с ВАХ:   
**q=2u3+u2+5** в точке **U0=1 В**. Решение приведено на рис. 2.3.2-4.

|  |
| --- |
|  |
| -->// ***Вычисление дифференциальной ёмкости* Cd ВАХ *нелинейного элемента***  -->Uo = 1;  --> // ***Загрузка и выполнение сценария* РИС2324**  --> exec('РИС2324.sce', 0)  --> mprintf('Cd = %g ' ,Cd);  Cd = 8 |

Рис.2.3.2-4. Вычисление дифференциальной ёмкости нелинейного элемента

Построить касательную к функции**f(x)=(3\*x2-7)/(2\*x+1)** (рис. 2.3.2-4)**.**

|  |
| --- |
|  |
|  |

Рис.2.3.2-5. Построение уравнения и графика производной функции

***Вычисление производной от табличной функции***

Если функция задана таблично, то для вычисления значения производной в точке, необходимо предварительно получить значения конечных разностей, поскольку в основе численного дифференцирования, как уже отмечалось выше, лежит аппроксимация функции, от которой берется производная, заменяемая интерполяционным многочленом Ньютона

.

Именно поэтому численное дифференцирование еще называют ***аппроксимированным дифференцированием***.

Для получения конечных разностей, входящих в состав интерполяционного многочлена, в Scilab используется функция **diff,** которая может иметь один из следующих форматов:

*dy***=diff(***y***);**

*dy***=diff(***y,n***);**

dy**=diff(***y,n,dim***),**

где *y*– вектор или матрица (вещественная, комплексная или полиномиальная) табличных значений функции;

*n* – порядок конечных разностей (целое число), по умолчанию значение *n* равно **1**;

*dim* – размерность матрицы, по которой происходит вычисление конечных разностей. Размерность может принимать значения как '**r**', '**c**' или **1, 2** соответственно (по строкам или столбцам), так и'**\***' **(**по всем элементам матицы). По умолчанию значение **dim** равно **"\*"**, то есть запись **diff(y,n)** равносильна записи **diff(y,n,'**\***')**.Если параметр **dim** имеет значение **'r'**, то это эквивалентно записи **dim=1**, а **dim='c'** – эквивалентно **dim=2.**

*dy* – скаляр или матрица значений конечных разностей.

Формат *dy***=diff(***y***)** предназначен для вычисления разностей между соседними значениями y (y(2:$)-y(1:$-1)), поскольку по умолчанию **n=1**. При использовании формата **dy=diff(y,n)** функция **diff** вычисляет **n**-конечные разности, а использование формата **dy=diff(y,n,dim)** позволяет вычислить **n**-ые разности при заданной размерности **dim**.

На рис.2.3.2-4 приведен пример вычисления значений производной от функции **y(x)=log10(x)**, заданной в виде таблицы. Зная аналитическое выражение для производной от функции **y(x) (y**'**=1/ln(10)/x)**, в примере проведена проверка результата, полученного при использовании функции **diff**.

|  |
| --- |
| --> // ***Вычисление значений производных для функции, заданной таблично***  -->  --> x = 10 : 5 : 25; h = 5;  --> y = log10(x); // ***Дифференцируемая функция***  -->  --> // ***Конечные разности 1-го, 2-го и 3-го порядк***а  --> dy = diff(y); dy2 = diff(y, 2); dy3 = diff(y, 3);  -->  --> // ***Приближенные значения*y'(x) *в точках*x = 10, 15, 20**  --> y10 = (dy(1) - dy2(1)/2 + dy(1)/3) / h  y10 =  0.0520729  --> Y15 = (dy(2) - dy2(2)/2) / h  Y15 =  0.0277906  -->Y20 = dy(3) / h  Y20 =  0.019382  -->  --> // ***Значения производных функции log10(x)***  --> (1/log(10)) ./ x(1:$-1)  ans =  0.0434294 0.028953 0.0217147 |

Рис. 2.3.2-4. Вычисление производных по таблице значений функции

На рис. 2.3.2-5 показана еще одна возможность приближенного вычисления производных функции **y=sin(x)**, заданной таблично. Принимая во внимание, что для исходной функции известно аналитическое выражение производной (**y**'**=cos(x)**), с помощью функции **norm** (норма матрицы), можно получить значение максимального отклонения приближенных значений производных от их соответствующих точных значений, полученных по аналитической формуле.

|  |
| --- |
| --> // ***Аппроксимированное дифференцирование***  --> x = 1 : 0.01 : 1.05; h = 0.01;  --> // ***Функция, от которой берется производная***  --> y = sin(x);  --> // ***Аппроксимированное вычисление производной функции***  --> dy = diff(sin(x) ./ h)  dy =  0.536086 0.5276177 0.5190967 0.5105238  --> y1 = cos(x) // ***Производная от функции* sin(x)**  y1 =  0.5403023 0.5318607 0.523366 0.5148188 0.5062203  --> // ***Максимальный элемент вектора модулей разностей***  --> norm(dy - cos(x(1:$ - 1)), %inf)  ans =  0.0043204 |

Рис. 2.3.2- 5. Аппроксимированное дифференцирование

Поскольку результат выполнения функции **norm(V,%inf)** равен значению максимального элемента вектора **V**по модулю, то в данном он равен значению **0.0043204**.

**2.3.3. Постановка задачи численного   
интегрирования**

***Интегрирование*** – это распространенная математическая операция для решения многих задач науки и техники. С его помощью проводят вычисление площадей, скорости, ускорения, перемещения и многое другое. Интегрирование простых функций можно легко провести аналитически, но чем сложнее подынтегральная функция, тем труднее, а иногда невозможно, получить точное значение интеграла. Следует также отметить, что, если в курсах математического анализа подынтегральная функция обычно задана аналитически, то в прикладных задачах она может быть задана в виде таблицы, поскольку здесь часто используются данные экспериментов.

Определенный интеграл в общем виде имеет вид:



где: **f(x)**– подынтегральная функция;

**а** и **b** – пределы интегрирования.

Исходя из геометрической интерпретации, значение **интеграла sчисленно равна площади области, ограниченной графиком** функции, *осью абсциссс***ОX** и *двумя прямыми***x=а** и **x=b**. При ***численном интегрировании*** эта область разбивается на несколько частей, вычисляется площадь каждой маленькой части, а затем они суммируются. Существует множество численных методов, которые отличаются способами разбиения области интегрирования на части и способами вычисления площади каждой из этих частей [22].

Суть всех формул численного интегрирования состоит в том, что на элементарных отрезках интегрирования подынтегральную функцию заменяют простейшим интерполяционным полиномом, который легко может быть проинтегрирован в аналитическом виде. Так, например, для получения формул ***прямоугольников***, ***трапеций*** и ***Симпсона*** используют полиномы соответственно нулевой, первой и второй степени.

**2.3.4. Численное вычисление определенных   
интегралов средствами Scilab**

***Вычисление определенных интегралов – inttrap***

В системе Scilab для вычисления определенного интеграла  по формуле *трапеций* используется функция **inttrap**, имеющая формат:

*s***=inttrap(***y***)**

*s***=inttrap(***x,y***),**

где *y*– подынтегральная функция *y***(***x***)**, заданная таблично**;**

*x*– вектор значений независимой переменной – *необязательный параметр*, если он отсутствует, то элементы вектора *х* принимают значения номеров вектора *y*.

Функция **inttrap** вычисляет площадь области, ограниченной функцией **y(x),** которая описана набором точек (**x,y**). Простейший пример использования функции **inttrap**приведен на рис. 2.3.4-1.

Здесь подынтегральная функция задается **11**-ю узловыми точками, где **х** меняется от **1** до **10** с шагом**0.5**, а функция **y** принимает в этих точках те же значения.

|  |
| --- |
| --> // ***Пример использования функции* inttrap**  -->  --> x = 1 : 0.5 : 10; // ***Вектор аргументов***  --> y = 1 : 0.5 : 10; // ***Вектор значений подынтегральной функции***  -->s = inttrap(x, y)  s =  49.5 |

Рис.2.3.4-1. Пример использования функции **inttrap**

На рис. 2.3.4-2 приведены несколько примеров вычисления значений определенных интегралов методом трапеций. В последнем примере для функции интегрирования использован формат **inttrap(y)**. Поэтому здесь в соответствии с форматом функции **inttrap** произошла замена элементов вектора **х** номерами элементов вектора **y,** в результате чего получено другое значение интеграла. Система Scilab не выдала ошибки, поскольку в данном случае была вычислена площадь совершенно другой фигуры.

|  |
| --- |
| --> // ***Вычисления определенных интегралов функцией*inttrap**  --> // ***Пример1***  --> a = 5; b = 13; // ***Пределы интегрирования***  --> x = a : b; y = sqrt(2\*x - 1); // ***Подынтегральная функци*я**  --> inttrap(x, y) // ***Вычисление интеграла с шагом 1***  ans =  32.655571  -->  --> // ***Пример2***  --> h = 0.1; x = a : h : b; y = sqrt(2\*x -1);  --> inttrap(x,y) // ***Вычисление интеграла с шагом 0.1***  ans =  32.666556  -->  --> // ***Пример3***  --> inttrap(y) // ***Элементы вектора* x *равны номерам элементов вектора* y**  ans =  326.66556 |

Рис. 2.3.4-2. Вычисление определенных интегралов функцией **inttrap**

***Вычисление определенных интегралов – integrate***

Для вычисления интеграла по формуле ***Симпсона*** в Scilab применяется функция **integrate**, имеющая следующий формат:

**integrate(**'*f*'**,**'*x*'**,***a***,***b***)**

**integrate(***'f'***,***'x'***,***a***,***b***,,***er1***)**

**integrate(**'*f*'**,**'*x*'**,***a***,***b***,,***er1,er2***),**

где: *f*– функция, задающая подынтегральное выражение;

*x*– независимая переменная;  
*a***,** *b*– пределы интегрирования – действительные числа;

*er1* и *er2* – *необязательные параметры* – абсолютная и относительная

погрешности вычисления интеграла – действительные числа.

Примеры использования функции **integrate**показаны нарис.2.3.4-3. Особенность использование **integrate** состоит в том, что при обращении к этой функции шаг интегрирования не задается, а используются параметр **er1** и (или) **er2** (**Пример1**). Если погрешность вычисления интеграла отсутствует, то вычисление проводится с погрешностью, установленной по умолчанию (**Пример2**). По умолчанию значение **er1** равно **1.D-8**, а значение **er2** равно **1.D-14**.

|  |
| --- |
| --> // ***Вычисление интеграла с заданнойпогрешностью***  -->  --> // ***Пример1 Вычисление интеграла с абсолютной погрешность =* .0001**  --> q1 = integrate('(2\*x - 1)^0.5', 'x', 5, 13, 0.0001)  q1 =  32.666667  -->  --> // ***Пример2 Вычисление интеграла с погрешность по умолчанию***  --> q2 = integrate('(2\*x - 1)^0.5', 'x', 5, 13)  q2 =  32.666667  -->  --> // ***Пример3 Вычисление интеграла с погрешностью по умолчанию***,  --> // ***а подынтегральное выражение задано внутренней встроенной функцией***  --> deff('y = fint((2 \* x - 1)^0.5)', 'x', 5, 13)  --> q3 = integrate('fint', 'x', 5, 13)  q3 =  32.666667 |

Рис.2.3.4-3. Вычисление определенных интегралов функцией **integrate**

***Вычисление определенных интегралов функцией intg***

Для интегрирования в Scilab имеется универсальная функция **intg**:

**[***s,er***]=intg(***a,b,f,er1,er2***),**

где: *a, b*– пределы интегрирования;

*f* – имя подынтегральной функции, которая может быть задана с помощью внешней функции или в виде набора дискретных точек и должна быть непрерывной;

*er1* и *er2*– *необязательные параметры* – абсолютная и относительная погрешности интегрирования. Если погрешность вычисления интеграла отсутствует, то вычисление проводится с погрешностью, установленной по умолчанию (значение **er1** равно **1.D-8**, а значение **er2** равно **1.D-14**).

Функция **intg**возвращает значение интеграла (*s*) и оценку абсолютной ошибки вычислений (*er*).

Внешнюю функцию можно задать функцией **deff**или **function:**

* если **f** – функция, то её определение должно иметь вид**y = f(t)**;
* если **f** – список, то этот список должен быть описан как: **list(f,x1,x2,…).**

Рассмотрим несколько примеров (рис.2.3.4-4).

|  |
| --- |
|  |
| -->// ***Загрузка сценария* РИС23404 *и выполнение функции* intg**  --> exec(**'**РИС23404.sce **'**);  --> disp([" I ", " er "], ...  > [I1, er1], [I2, er2], [I3, er3], [I4, er4], [I5, er5]);  ! **I er** !  -3.58957861311877350 0.00000000001621547  -3.58957861320974690 0.00001678794315296  -3.58957862082392730 0.0015909785599586  -3.58957862082391800 0.00159097855998436  -3.58957861311831690 0.00000000781318699 |

Рис.2.3.4-4. Вычисление определенных интегралов функцией**intg**

Точность полученного результата зависит от заданной абсолютной погрешности. Поэтому очевидно, что в **Примере1** и **Примере5** обеспечена максимальная точность.

Опишем подынтегральную функцию **y=cos(x)2/(1-x)**в функции **fi(x)** и вычислим значение определенного интеграла (с пределами **a=2** и  
**b=6)** сначала с помощью функции **intg**, а затем – функции **inttrap**, разбив при этом отрезок интегрирования на **5**, **10** и **20** частей (рис.2.3.4-5).

|  |
| --- |
| --> // ***Сравнение вычисления интеграла функциями* intg *и* inttrap*по точности***  -->  -->// ***Описание подынтегральной функции***  -->deff('[y] = fi(x)', 'y = cos(x).^2 ./ (1-x)')  -->  -->a = 2; b = 6;  --> // ***Вычисление интеграла с помощью функции* intg**  --> [s,ir] = intg(a, b, fi)  ir =  5.844D-10  s =  -0.8781216  -->  --> // ***Вычисление интеграла с помощью функции*inttrap**  --> // **s1–5 *точек* (h1=0.8); s2–10 (h2=0.4); s3–20 (h3=0.2);**  --> h1 = 0.8; x = a:h1:b; y = fi(x); s1 = inttrap(x, y)  --> h2 = 0.4; x = a:h2:b; y = fi(x); s1 = inttrap(x, y)  --> h3 = 0.2; x = a:h3:b; y = fi(x); s1 = inttrap(x, y)  --> z=[h1,s1; h2,s2; h3,s3]  z =  0.8 -0.8490697  0.4 -0.8711676  0.2 -0.8764039 |

Рис.2.3.4-5. Влияние величины шага на точность интегрирования

Из полученных результатов (рис.2.3.4-5) следует, что, если значение интеграла, полученное с использованием функции **intg** можно принять практически за точное значение (относительная погрешность **5.844D-10)**, то с помощью функции **inttrap**, использующей таблицу значений подынтегральной функции, мы только приблизились к этому значению. При этом из полученных результатов явно следует, что чем меньше значение шага интегрирования, тем точнее результат. А поскольку величина шага влияет на количество используемых узловых точек, то становится очевидным, что увеличение числа, используемых в расчете узловых точек приводит к получению более точного значения интеграла.

Рассмотрим несколько задач.

Построить график мгновенного напряжения **uL(t)** на катушке индуктивности **L=0,5 Гн** в интервале **[0 1000] мc,** если **i(t)=exp(-t)-exp(2t)**(рис. 2.3.4-6).

|  |  |
| --- | --- |
|  | L=0.5 Гн  t=[0:50:1000]\* 10D-3; |
| exec('РИС23406.sce', 0) | |

Рис.2.3.4-6. Связь между мгновенным током и напряжением в катушке индуктивности

Решим несколько задач.

Определить мощность, выделяемую в резисторе сопротивлением **R=1Ом**, если через него протекает ток гармонической формы с мгновенными значением **i = sin(t)**(Рис.2.3.4-7. ).

В ТЭЦ среднюю мощность Pможно выразить формулой:,

где **T** – период, **u**– мгновенное напряжение, **i**– мгновенный ток.

|  |  |
| --- | --- |
| exec(РИС23407.sce', 0)  er =  1.339D-11  P =  1.5571783  Pt =  0.0397965 | R = 1 ом  t=2 c |

Рис.2.3.4-7. Определение мощности, выделяемая в резисторе

**2.3.5. Контрольные вопросы**

1. Какой встроенной функцией системы Scilabвычисляется производная в заданной точке?
2. Какое назначение и какой формат функции **numderivative**используется**.**
3. Как осуществляется вычисление производной от функции, заданной аналитически?
4. Как формулируется постановка задачи численного интегрирования в Scilab?
5. Какая функция позволяет получить производную от полинома в явном виде?
6. Что называется, аппроксимированным дифференцированием?
7. Как осуществляется вычисление определенных интегралов в Scilab?
8. Назначение функции **inttrap(x,y)**?
9. Что возвращает функция **inttrap(x,y)**, если **y(x)** – матрица?
10. Можно ли использовать функцию **inttrap**, если узлы по оси х - не равноотстоящие?
11. Какая функция Scilab позволяет вычислить определенный интеграл с заданной точностью?
12. Какой численный метод заложен в функции **integrate**?
13. Какие существуют способы задания подынтегральной функции при вычислении определенного интеграла с использованием функции **integrate**?
14. Как задать допустимую погрешность вычисления определенного интеграла?
15. С какой точностью производится вычисление определенного интеграла, если погрешность не задана?

2.4. Решение нелинейных уравнений

**2.4.1. Постановка задачи решения   
нелинейных уравнений**

В общем виде ***нелинейное уравнение*** (НУ) можно представить как   
**f(x)=0**. В зависимости от вида элементарных функций, входящих в состав функции нелинейного уравнения**,** различают алгебраические и трансцендентные уравнения.

***Алгебраическими уравнениями*** называются уравнения, в которых значение функции **f(x)** представляет собой полином **n***-*й степени (**n≥2**). Всякое неалгебраическое уравнение называется ***трансцендентным***. Функция **f(x)** в таких уравнениях содержит хотя бы одну из следующих элементарных функций: *показательную, логарифмическую, тригон*о*метрическую* или *обратную тригонометрическую.*

Решением уравнения **f(x)=0** является *совокупность* корней **Х**, при которых уравнение обращается в тождество **f(Х)0**. Однако, точные значения корней могут быть найдены аналитически только для некоторого ограниченного типа уравнений. Еще меньше возможностей для получения точного решения у трансцендентных уравнений. Таким образом, задача нахождения точных значений корней не всегда корректна. Например, если коэффициенты уравнения являются приближенными числами, точность вычисленных значений корней заведомо не может превышать точности исходных данных. Эти обстоятельства заставляют рассматривать возможность отыскания корней уравнения с ограниченной точностью (приближенных значений корней).

Процесс численного решения нелинейного уравнения состоит из двух этапов:

1. ***отделение корней уравнения*** (локализация каждого корня нелинейного уравнения на отдельном отрезке);
2. ***уточнение каждого корня*** на выбранном отрезке с заданной точностью.

Этап отделения корней нужен для того, чтобы, во-первых, убедиться, что данное уравнение вообще имеет корни, а во-вторых, необходим для выбора начального значения при использовании численных методов. Важно, что численные методы при этом применимы только тогда, когда на выделенном отрезке существует единственный корень.

Для отделения корней нелинейных уравнений применяются ***графический*** и ***аналитический*** методы. При применении графического метода в области допустимых значений функции (левой части уравнения) строится ее график и визуально определяются отрезки, имеющие единственное пересечение с осью абсцисс **ОX**. Использование аналитического метода основывается на теореме о существовании и единственности корня на отрезке [24]. Согласно этой теореме надо найти такие отрезки, на концах которых функция **f(x)**имеет разные знаки, а первая производная непрерывна и не меняет свой знак (то есть функция монотонна).

Для уточнения корней с заданной степенью точности существуют множество классических численных методов (***метод Ньютона****,* ***метод хорд****,* ***метод итераций****,* ***метод половинного деления*** и многие другие, а также их модификации) [22].

Задача нахождения корня уравнения с заданной точностью **ε (ε>0**) считается решенной за **n** итераций, если найдено приближенное значение **xn**, которое отличается от точного значения корня x\* не более чем на значение **ε**: **|x\*-xn|≤ε.**

Рассмотрим средства Scilab, используемые для решения нелинейных уравнений.

**2.4.2. Решение нелинейных уравнений   
средствами Scilab**

Итак, первым этапом решения нелинейных уравнений является исследование функции решаемого НУ, в результате чего в области допустимых значений функции выявляются такие отрезки (или отрезок), которые содержат только один корень.

Рассмотрим этап отделения корней средствами Scilab на примере решения нелинейного уравнения **2x–4x=0**.

Отделение корня нелинейного уравнения целесообразно вначале произвести ***графически***. Из графика функции **f(x)=2x–4x**(рис. 2.4.2-1) следует, что уравнение имеет два действительных корня, принадлежащих отрезкам **[0;1]** и **[3.5;4.5]**.

|  |
| --- |
| -->// ***Графическое отделение корня нелинейного уравнения***  --> // ***Загрузка сценария РИС2421 и его выполнение***  -->  -->clear  --> exec('РИС2421.sce', 0); |

Рис.2.4.2-1. Графическое отделение корней нелинейного уравнения **2x–4x=0**

Проверим на выбранных отрезках выполнение ***аналитических*** условий существования и единственности корня (рис.2.4.2-2).

|  |
| --- |
| -->//*Аналитическое отделение корня нелинейного уравнения*  -->//***Загрузка сценария* РИС2422 *и выполнение функций f(x) и f1(x)***  -->  --> clear  --> a = 0; b = 5; h = 0.5;  --> exec('РИС2422.sce', 0);  -->x = a : h : b; // ***Вектор аргумент* x**  -->fx = f(x); f1x = f1(x); // ***Вектор функций* f *и 1-й производной* f1**  --> disp([x; fx; f1x]', [" x", " fx", " f1x "]);  ! **x fx f1x** !  0. 1. -3.3068528194400546  0.5 -0.5857864376269049 -3.01974185653145270  1. -2. -2.61370563888010920  1.5 -3.17157287525380970 -2.03948371306290530  2. -4. -1.22741127776021890  2.5 -4.34314575050761940 -0.0789674261258111  3. -4. 1.5451774444795623  3.5 -2.68629150101523880 3.84206514774837780  4. 0. 7.09035488895912460  4.5 4.62741699796952230 11.6841302954967560  5. 12. 18.1807097779182490 |

Рис. 2.4.2-2. Отделение корня нелинейного уравнения

Из полученной таблицы, содержащей значения аргумента, функции и производной, следует, что условия существования и единственности корня выполняются на отрезках **[0;0.5] и [3.5;4]** (на концах каждого из этих отрезков функция имеет противоположные знаки, а ее производная на каждом из этих отрезков знакопостоянна).

Для решения нелинейных уравнений в среде Scilab используется функция **fsolve**, предназначенная для нахождения ***вещественных*** корней уравнений вида **f(х)=0**, использующая модификацию гибридного метода Пауэлла [13]. Хотя здесь и в дальнейшем **fsolve**называем функцией, следует понимать, что **fsolve**по своей сути является ***решателем***, объединяющим под этим именем несколько алгоритмов.

Метод Пауэлла является методом локального поиска, поэтому очень важно правильно произвести выбор начального приближения, а по окончании поиска корня проверить индикатор завершения и остаточное значение для того, чтобы убедиться, что в результате выполнения функции получено правильное решение.

Функция **fsolve**имеет следующие основные форматы:

**fsolve(***x0,f***)**

**[***х,fx,id***]=fsolve(***x0,f,fp,er2***)**,

где: *х0 –* начальное приближение к корню;

*f–* указатель на внешнюю функцию *f(x)*уравнения **f(х)=0**;

*fp*– указатель на внешнею функцию **fp(x)** (*необязательный параметр*), представляющую производную от функции **f(x)** (при решении систем нелинейных уравнений *fp*– это Якобиан [13]);

*er2* – относительная погрешность (*необязательный параметр)* по умолчанию принимает значение **1.d-10**, и определяет условие проверки сходимости;

*х* и *fx–* выходные параметры функции **fsolve***–* значение корня уравнения, вычисленное с заданной относительной погрешностью *er2* и значение функции в точке корня;

*id–* индикатор (*необязательный параметр*), показывающий при каких условиях произошло прекращение итерационного процесса поиска корня.

Индикатор *id* может иметь следующие значения:

*0* – неверные входные параметры;

*1* – найдены значения корней с заданной точностью **tol**(или с

погрешностью, принятой по умолчанию);

*2* – допустимое количество вызовов **f**достигнуто;

*3* – *er2* слишком мало, поэтому дальнейшие приближения   
невозможны;

*4* – итерации не дают результата (корней нет).

При использовании первого формата **fsolve**происходит вычисление и вывод корня уравнения, как значение переменной **ans**, а при использовании второго*–* значение корня**x** и значение функции в точке корня **fx**.

На рис.2.4.2-3 приведен пример уточнения корня нелинейного уравнения на отрезке **[0;1]** с использованием функции **fsolve**, где в качестве начального приближения к корню выбрано значение **0**.

|  |
| --- |
| --> // ***Загрузка сценария* РИС2423*, содержащего описание*** функции f(x)  --> // ***Уточнение корня нелинейного уравнения на отрезке*[0;1]**  -->exec('РИС2423.sce', 0);  -->  --> // ***Первый способ***  -->fx = f(x);  --> [x, fx] = fsolve(0, f) // ***Начальное приближение х0 = 0***  -->disp([x, fx], [" x ", " fx "]);  ! **xfx** !  0.30990693238069056 0.  -->  --> // ***Второй способ***  -->x = fsolve(0, f); fx = f(x);  -->disp([x, fx], [" x ", " fx "]);  ! **x fx**!  0.30990693238069056 0. |

Рис. 2.4.2-3. Уточнение корня уравнения с использованием функции **fsolve**

При обращении к функции **fsolve**может использоваться и один выходной параметр *– х*, тогда происходит только вычисление корня, а значение функции может быть по необходимости вычислено позже.

В следующем примере показано решение нелинейного уравнения   
**ex/5–2(x-1)2=0**, иллюстрирующее возможность использования функции **fsolve** для решения НУ, имеющего несколько корней, при условии, что для каждого из коней задается свое начальное приближение.

Предположим, что предварительно проведено графическое отделение всех корней уравнения **ex/5–2(x-1)2=0** (рис. 2.4.2-4), из которого следует, что уравнение имеет три корня, принадлежащие соответственно следующим отрезкам: **x1€[0.5;1], x2€[1.5;2], х3€[5;5.5].**

|  |
| --- |
|  |
| -->// ***Загрузка сценария* РИС2424.sce*, содержащего описание функции* f(x)*,***  -->// ***отображение графика и***  --> // ***выполнение функции* fsolve *с различными начальными приближениями***  -->  -->clear  -->exec('РИС2425.sce', 0);  --> // ***1-ый этап: Отделение корней графическим способом:***  -->x = 0 : 0.2 : 5.5;  -->y = yf(x); // ***Левая часть НУ***  -->y1 =x \* 0; // **y1 *используется для выделения оси* Ох**  --> plot(x, y, x, y1)  --> xtitle('', 'X', 'Y(X)');  -->xgrid(); |
| --> // ***Из графика видно, что уравнение имеет три корн***я,  --> // ***принадлежащие отрезкам:* x1€[0.5;1], x2€[1.5;2], х3€[5;5*.5]***  *-->*//***Выбор начальных приближений:* x1 = 0.5, x2 = 2, x3 = 5**  -->  --> // ***2-ый этап: Уточнение корней:***  --> // ***1-й способ: вычисление каждого корня уравнения отдельным***  --> // ***обращением к функции* fsolve**  -->xx(1) = fsolve(0.5, yf); xx(2) = fsolve(2, yf); xx(3) = fsolve(5, yf);  -->yy = yf(xx); xy = [xx'; yy']'  xy =  0.5778406 -3.220D-15  1.7638701 1.554D-15  5.1476865 0.  -->  --> // ***2-й способ: вычисление корней уравнения с использованием***  --> // ***вектора начальных приближений***  --> [xx,yy] = fsolve([0.5; 2;5], yf);  -->xy = [xx'; yy']'  xy =  0.5778406 -3.220D-15  1.7638701 1.554D-15  5.1476865 0. |

Рис. 2.4.2-4. Отделение корней нелинейного уравнения графическим методом и уточнение его корней с использованием функции **fsolve**

На рис. 2.4.2-4 реализовано решение заданного нелинейного уравнения с использованием функции **fsolve** двумя способами. В первом случае функцию **fsolve** используют трижды, то есть по числу отделенных корней уравнения. При этом для каждого корня задается свое начальное приближение. Второй способ иллюстрирует возможность задания начальных приближений к корням в виде вектора, что и позволяет выполнить функцию **fsolve** всего один раз.

Приведем еще один пример использования функции **fsolve**, содержащей третий (необязательный) выходной параметр *–* индикатор (рис.2.4.2-5). Для этого используем нелинейное уравнение **sin(x)+2=0**, для которого заранее известно, что оно не имеет корней. В результате выполнения значение индикатора приняло значение 4, что говорит о том, что корень не найден, а в качестве результата выведено значение переменной, обеспечивающее максимальное приближение к корню.

|  |
| --- |
|  |
| --> // ***Загрузка сценария*** РИС2425***, содержащего описание функции* f(x**)  --> // ***Уточнение корня нелинейного уравнения и***  --> // ***использование индикатора для проверки наличия корня***  -->  -->exec('РИС2425.sce', 0);  -->  --> [xx, yy, id] = fsolve(0, fd)  id =  4.  yy =  1.0000001  xx =  -1.5703125 |

Рис. 2.4.2-5. Решение уравнения не имеющего корней

Функция **fsolve** может быть использована для решения как ***трансцендентных***, так и ***алгебраических*** уравнений. Правая часть алгебраических уравнений представляет собой полином вида . Однако, для вычисления корней полинома, как уже говорилось в п.2.1.1, чаще используется функция **roots**, так как эта функция вычисляет все корни, действительные и комплексные. Перед использованием **roots** создается список коэффициентов (даже нулевых), затем выполняется функция **poly(a,**'**x**'**,**'**с**'**)**. Решим в качестве примера одно и тоже алгебраическое уравнение **x5-x3+1=0** с использованием функций **fsolve** и **roots**, построив предварительно график функции левой части уравнения(рис. 2 4.2-6).

|  |
| --- |
| --> deff('[f] = y(x)', 'f = x^5 - x^3 + 1')  --> x = -2:0.1:2; f = y(x);  --> plot(x, f)  -->xtitle('X','Y');  -->xgrid(); |
| --> // ***Решение алгебраического уравнения с помощью функции* fsol*ve***  --> X = fsolve(-1.5, y)  X =  -1.2365057 |
| -->  --> // ***Решение алгебраического уравнения с помощью функции* roots**  -->Х = roots(poly([1 0 0 -1 0 1], 'x', 'c'))  Х =  -1.2365057  0.9590477 + 0.428366i  0.9590477 - 0.428366i  -0.3407949 + 0.7854231i  -0.3407949 - 0.7854231i |

Рис. 2.4.2-6. Уточнение корней уравнения с использованием

функций **fsolve**и **roots**

Из графика (рис. 2.4.2-6) видно, что уравнение имеет один действительный корень. Этот корень и был найден функцией **fsolve**. Решение этого же уравнения функцией **roots** показало, что кроме вещественного корня уравнение имеет еще несколько мнимых корней. Именно поэтому для вычисления всех корней полинома целесообразнее использовать функцию **roots**.

Функция **fsolve** может решать также и системы нелинейных уравнений от двух переменных. Рассмотрим следующий пример:решить систему нелинейных уравнений



Решениепроведем при начальных значениях неизвестных **x0=[-5,5]**.

Создадим sce-файл для вычисления значений **F(x)**, в котором левые части уравнений системы представлены в виде элементов вектора, а начальные приближения к неизвестным **x** и обращение к функции **fsolve** содержатся в том же сценарии, записанном в файле **РИС2427.sce** (рис.2.4.2-7).

|  |
| --- |
|  |
| -->// ***Решение системы уравненийфункцией fsolve***  -->// ***Загрузка и выполнение сценария* РИС2429*.sce***  -->  --> clear  -->exec('РИС2427.sce', 0);  x(1)=0.567143 x(2)=0.567143 fun(1)=0.000000 fun(2)=0.000000 id=1 |

Рис. 2.4.2-7. Решение системы нелинейных уравнений функцией **fsolve**

Решение системы уравнений позволило за конечное число итераций (**id=1**) получить ее корни, которые при подстановке в заданную систему уравнений обращают уравнения в тождество. Поскольку точность вычислений не была задана, то они проводились с погрешносью, принятой по умолчанию.

**2.4.3 Контрольные вопросы**

1. Что является нелинейным уравнением?
2. Этапы численного решения нелинейных уравнений?
3. Как осуществляется графическое отделение корней нелинейного уравнения средствами Scilab?
4. Как осуществляется аналитическое отделение корней нелинейного уравнения средствами Scilab?
5. Какие основные идеи заложены в методе Пауэлла для нахождения корней нелинейных уравнений?
6. Какие имеются способы задания нелинейных уравнений при их решении с использованием функции **fsolve**?
7. Каково назначение и формат функции **fsolve**?
8. Каково назначение и формат функции **poly**?
9. Каково назначение и формат функции **roots**?
10. Что является решением функций **fsolve** и **roots**?
11. Какие алгоритмы используются в функции **fsolve**?
12. Какие алгоритмы используются в функции **roots**?
13. Каким образом необходимо задать исходные данные для решения систем нелинейных уравнений?

2.5. Решение обыкновенных   
дифференциальных уравнений

**2.5.1. Постановка задачи решения  
обыкновенных дифференциальных уравнений**

Дифференциальными уравнениями описываются практически все физические процессы, которые используются в технических приложениях. Аналитически может быть решено только ограниченное число дифференциальных уравнений. Тогда как численные методы могут давать приближенное решение практически любого дифференциального уравнения.

Обыкновенным дифференциальным уравнением (ОДУ) является уравнение, которое содержит независимую переменную, зависимую переменную и производные зависимой переменной. Таким образом, ОДУ   
1-го порядка имеет следующий вид:



где **x**и **y** – независимая и зависимая переменные, соответственно.

Такое ОДУ первого порядка представлено в виде, разрешенном относительно производной **y'=f(x,y)**.

Для решения ОДУ первого порядка, в соответствии с ***задачей Коши***, должны быть заданы начальные условия его решения – **y(x0)=y0**. Тогда решением ОДУ является функция **y=f(x)**, которая будучи подставленной в исходное уравнение, обратит его в тождество, при этом будут выполняться и начальные условия.

***Задача Коши*** для решения ОДУ **n**-го порядка



формулируется аналогично, при этом в качестве начальных условий должны быть заданы 

При его решении выполняется ряд подстановок , в результате которых ОДУ **n**-го порядка заменяется системой из n обыкновенных дифференциальных уравнений 1-го порядка:



Результатом решения ОДУ ***численными методами*** является ***таблица значений* y = y(x)** на некотором заданном множестве значений аргумента**х**. Поэтому при постановке задачи численного решения ОДУ наряду с начальными условиями **x0, y0**необходимо задать область решения – отрезок интегрирования **[a;b]**, где **a=x0**, и шаг изменения аргумента – **h** (шаг интегрирования).

Для получения численного решения ОДУ используются различные ***методы Рунге-Кутты*** [24],которые различаются порядком. Чем выше порядок метода, тем точнее решение, полученное при равном шаге интегрирования.

В этом разделе описываются некоторые средства системы Scilab, предназначенные для решения обыкновенных дифференциальных уравнений.

**2.5.2. Решение обыкновенных дифференциальных уравнений средствами Scilab**

Для решения ОДУ и системы ОДУ 1-го порядка в Scilab предусмотрена функция **ode,** имеющая форматы:

**[y]=ode(y0,t0,t,f),**

**[y,w,iw]=ode(type,y0,t0,f,adams,stiff,rk,rkf,fix,rtol,adol,jac,w,iw),**

которые содержат ***обязательные*** и ***необязательные*** параметры.

Первый формат функции**ode** содержит только ***обязательные*** параметры, к которым относятся:

**y0** – вектор начальных условий;

**t0** – начальная точка интервала интегрирования;

**t** – координаты узлов сетки, в которых происходит поиск решения;

**f**– имя внешней функция, определяющей правую часть уравнения

или системы уравнений;

**y** – вектор решений (выходной параметр).

Второй формат функции **ode**содержит ряд ***необязательных*** параметров. Эти параметры позволяют выбрать метод решения или тип решаемой задачи:

**type**– строка, указывающая тип используемой программы решения, может принимать значения: **"adams"**, **"stiff"**, **"rk"**, **"rkf"**, **"fix"**, **"discrete"**или **"root"**;

**adams**– применяют при решении дифференциальных уравнений или систем методом прогноза и коррекции Адамса;

**stiff**– указывают при решении жестких задач;

**rk**– используют при решении дифференциальных уравнений или систем методом Рунге-Кутты четвертого порядка;

**rkf**– указывают при выборе пятиэтапного метода Рунге-Кутты четвертого порядка;

**fix**– тот же метод Рунге-Кутты, но с фиксированным шагом;

**rtol**, **atoll**– относительная и абсолютная погрешности вычислений, соответственно, по умолчанию **rtol=0.00001**, **atol=0.0000001** (при использовании параметров **rkf**и **fix**– **rtol=0.001**, **atol=0.0001**);

**jac**– матрица, представляющая собой якобиан правой части жесткой системы дифференциальных уравнений и заданная в виде внешней функции вида **j=jak(t,y);**

**w**, **iw**– векторы, предназначенные для сохранения информации о параметрах интегрирования, которые применяют для того, чтобы последующие вычисления выполнялись с теми же параметрами.

Описание этого формата достаточно подробно рассмотрено в справочной системе Scilab, где и приведены примеры его использования для различных типов ОДУ [13].

Остановимся на использовании первого формата функции **ode**, для чего рассмотрим решение следующего примера: найти решение ОДУ **y'=-sin(x\*y)** на отрезке **[0;5]**cшагом **0.5** при начальных условиях **y(0)=1.5**.

Решение ОДУ, полученное с применением функции **ode** и график полученного решения, представлены на рис.2.5.2-1.

|  |
| --- |
|  |
| --> // Решение ОДУy'=-sin(x\*y)  --> y0 = 1.5; t0 = 0; t = 0:0.5:5; // Начальные условия  --> // Загрузка и выполнение сценарияРИС2521  --> exec('РИС2521.sce', 0);  ans =  0. 1.5  0.5 1.3302706  1. 0.9566472  1.5 0.5574285  2. 0.2477507  2.5 0.0822207  3. 0.0208664  3.5 0.0041103  4. 0.0006303  4.5 0.0000753  5. 0.000007 |
|  |

Рис. 2.5.2-1. Решение ОДУ с использованием функции ode

Для решения ***системы обыкновенных дифференциальных уравнений*** в Scilab предназначена функция:

**y=ode(x0,t0,t,sys),**

где **х0** – вектор начальных условий ОДУ;

**t0** – начальная точка интегрирования;

**t**– вектор значений независимой переменной;

**sys**– имя функции, в которой исходя из вектора значений**t**вычисляется матрица решений**y**;

**y**– матрица решений (*выходной параметр*), первый столбец которой

**y(1)**– значение функции **y(x)**, а второй - **y(2)***–*значение производной **y'(x).**

Решение системы ОДУ формируется в матрице **y***,* и выводится на экран в виде таблицы

В качестве примера рассмотрим решение системы ОДУ:



c начальными условиями **x(0)=0**, **y(0)=0** на отрезке **[0;10]** и шагом **1.**

Решение системы ОДУ в Scilab начинается с создания функции **sys***,* описывающей систему (рис.2.5.2-2). После того, как заданы начальные условия ОДУ, производится обращение в функции **ode**, в результате которого формируется матрица решения **y**. Решение системы выводится в виде таблицы и графика. (Шапка таблицы и легенда для графика!)

|  |
| --- |
| --> // ***Решение системы ОДУ***  --> // ***Загрузка сценария РИС2522 и его выполнение***  -->clear  --> // Начальные условия  -->x0 = [0; 0]; t0 = 0; t = 0:1:10;  -->  --> exec('РИС2522.sce', 0);  ans =  0. 0. 0.  1. 0.9802401 0.533358  2. 1.4096497 0.9693978  3. 1.7429464 0.6024417  4. 2.4027415 0.2586293  5. 3.3312751 0.005761  6. 4.2504071 -0.1650347  7. 4.826261 -0.2358589  8. 5.1581587 -0.2515654  9. 5.3963437 -0.2509281  10. 5.5981318 -0.2461951 |
|  |

Рис.2.5.2-2. Решение системы ОДУ

Рассмотрим задачу построения временной зависимости тока **i(t)** в **RC**-цепи.

Построить временную зависимость тока **i(t)** в неразветвленной **RC**-цепи (рис. 3.3-4), если цепь включается на постоянное напряжение при нулевых начальных условиях.

|  |  |
| --- | --- |
|  | |
|  | **Неразветвленная RC-цепь**  Дано:  E=1 В – ЭДС источника;  R=1 Ом – сопротивление;  С=1 Ф – емкость;  t=[0, 6τ] c – временной интервал. |

Рис.2.5.2-3. Построить временную зависимость тока **i(t)** в **RC**-цепи

***Обратите внимание, что для построения зависимости искомого тока требуется решение обыкновенного дифференциального уравнения***

**2.5.3. Контрольные вопросы**

1. Как формулируется постановка задачи численного решения ОДУ?
2. Какие начальные условия должны быть заданы в соответствии с задачей Коши при решении ОДУ 1-го порядка средствами Scilab?
3. Какие начальные условия должны быть заданы в соответствии с задачей Коши при решении ОДУ n-го порядка средствами Scilab?
4. Что является численным решением ОДУ?
5. Какие численные методы реализованы в функции **ode**?
6. Какие входные параметры функции **ode** являются обязательными?
7. В какой форме должны быть записаны функции ОДУ при использовании функции **ode**?
8. Какими средствами решается система ОДУ?
9. Что представляет собой решение ОДУ 1-го и 2-го порядка при использовании функций **ode**?

2.6. Решение систем линейных уравнений

**2.6.1. Постановка задачи решения систем   
линейных уравнений**

В общем виде система линейных уравнений (СЛУ) записывается следующим образом [20]:



Другая форма записи СЛУ – ***матричная****:*



где: **А** – матрица коэффициентов, **Х** – вектор неизвестных, **В** – вектор свободных членов.

Решением СЛУ является такой набор значений неизвестных **x1,x2,…,xn**, при котором **n** уравнений системы обращаются в тождества.

Методы решения СЛУ обычно разделяют на две большие группы:

1. ***точные методы***, представляющие собой алгоритмы, позволяющие находить точные решения за конечное число арифметических операций, (например, ***правило Крамера*** или ***метод Гаусса***);
2. ***приближенные итерационные методы***, при применении которых решение получают в результате бесконечного процесса приближений (например, ***метод итераций***, в которомбесконечный сходящийся процесс при заданной точности ограничивается конечным числом итераций).

СЛУ может иметь единственное решение, не иметь решения, иметь бесконечное множество решений. Поэтому перед тем, как решать СЛУ, ее надо исследовать на предмет существования единственного решения. Исследование сводится к анализу значения определителя матрицы коэффициентов СЛУ:

Пусть .

Известно, что если **Δ≠0**,то СЛУ имеет единственное решение, а если **Δ=0**, то СЛУ не имеет решения или имеет бесконечное множество решений.

Однако часто **Δ≠0** могут иметь СЛУ, при решении которых за счет накапливания погрешностей получают неверные решения. Такие системы линейных уравнений называют плохо ***обусловленными*** системами. В случае плохо обусловленных систем решение найти трудно, а точность решения сомнительна.

**2.6.2. Решение системы линейных уравнений   
средствами Scilab**

В пакете Scilab существует специальная функция **linsolve**, предназначенная для решения СЛУ. Функция **linsolve** имеет следующий формат:

*X***=linsolve(***A,-b***),**

где *А*– матрица коэффициентов;

*b*– вектор свободных членов;

*Х* – выходной параметр – вектор неизвестных.

На рис.2.6.2-1 приведено решение СЛУ:****

Очевидно, что определитель матрицы **А** отличен от нуля, следовательно, СЛУ имеет решение. Функция решает СЛУ вида **Ax+b=0**, поэтому на   
рис. 2.6.2-1, который иллюстрирует пример решения СЛУ с использованием встроенной функции **linsolve**, знак вектора **b** изменен на обратный. В результате решения системы получены два корня **х1=-3** и **х2=2.5**.

|  |
| --- |
| -->A = [1,2; 3,4]; // ***Матрица коэффициентов***  -->b = [2; 1]; // ***Вектор свободных членов***  -->X = linsolve(A, -b) // ***Решение СЛУ***  X =  -3.  2.5 |

Рис.2.6.2-1. Решение СЛУ с использованием функции **linsolve**

В математике существует множество методов решения СЛУ. Рассмотрим, например, решение СЛУ с использованием ***метода обратной матрицы***. В этом случае СЛУ должна быть представлена в виде **A\*x=b (**где **A** – матрица коэффициентов при неизвестных, **х** – вектор неизвестных, **b** – вектор свободных членов). В ходе решения требуется выполнить следующую последовательность действий:

* создать матрицу коэффициентов **A**, где коэффициенты размещаются по столбцам (если переменная отсутствует в уравнении, то элемент матрицы определяется равным нулю);
* задать вектор правой части **b** размером **n**;
* найти решение по формуле **x=b\A**.

Полученное решение можно проверить по формуле **b1=x\*A**. В результате должно получиться равенство **b1=b**.

Запись **A\b** соответствует математической операции **A-1b**, а запись **x=A\b** эквивалентна записи **x=inv(A)\*b**.

В результате решения системы (рис. 2.6.2-2) получены два корня **х1=-3** и **х2=2.5**. Проверка, выполненная путем умножения вектора полученных корней на матрицу коэффициентов, показала, что полученное решение верно, поскольку ее результат численно равен вектору свободных членов.

|  |
| --- |
| -->A = [1 2; 3 4]; // ***Матрица коэффициентов***  -->b = [2; 1]; // ***Вектор свободных членов***  -->det(A) // ***Вычисление определителя матрицы* A**  ans =  -2.  -->x = A \ b // ***Решение СЛУ методом обратной матрицы***  x =  -3.  2.5  -->b1 = A \* x // ***Проверка***  b1 =  2.  1. |

Рис.2.6.2-2. Решение СЛУ методом обратной матрицы

В следующем примере СЛУ решается при помощи ***правила Крамера****,* которое заключается в следующем. Если определитель **Δ = detA** матрицы системы из **n** линейных уравнений с **n** неизвестными вида **A\*x = b** отличен от нуля, то система имеет единственное решение **x1, x2, … xn**, определяемое по формулам ***Крамера***: **xi=∆i/∆,** где **i=1,2,…,n**, а **∆i** – определитель матрицы, полученной из матрицы **A** путем замены **i**-го столбца матрицы **А** столбцом свободных членов **b**.

Пусть задана следующая система линейных уравнений с 4-мя неизвестными:



На рис. 2.6.2-3 приведено решение заданной СЛУ средствами Scilab по ***методу Крамера***. Решение СЛУ оформлено в виде функции **Kramer(A,b,n)**. Перед обращением к функции предусмотрен ввод матрицы коэффициентов (**A**) и вектора свободных членов (**b**). Обратите внимание, что в функции предусмотрено, что если определитель матрицы коэффициентов равен нулю, то решение не выполняется, а на экран выводится сообщение «Определитель матрицы **А** равен нулю». В нашем случае в результате выполнения функции получены четыре корня СЛУ: **Х(1)=3; Х(2)=-4; Х(3)=-1; Х(4)=1**. После вывода их значений произведена проверка правильности решения СЛУ.

|  |
| --- |
|  |
| -->// ***Решение СЛУ по методу Крамера***  -->clear  --> // ***Загрузка сценария* РИС2623 *и выполнение функции* Kramer**  -->  --> A = [2 1-5 1; 1-3 0 -6; 0 2 -1 2; 1 4 -7 6]; // ***Матрица коэффициентов***  --> b = [8; 9; -5; 0]; // ***Матрица свободных членов***  -->exec('Kramer.sce');  -->X = Kramer(A, b, 4) // ***Обращение к функции решения СЛУ по методу* Крамера**  X =  3.  -4.  -1.  1.  --> P = A \* X - b // ***Проверка***  P =  0.  0.  -8.882D-16  2.665D-15 |

Рис. 2.6.2-3. Решение СЛУ по методу Крамера

Рассмотрим следующий не менее известный метод решения СЛУ– ***метод Гаусса***, который основан на том, что от исходной системы уравнений происходит переход к эквивалентной системе, которая решается проще, чем исходная.

Метод состоит из двух этапов. Первый этап (***прямой ход***) заключается в том, что расширенная матрица (матрица коэффициентов, в которую добавлен столбец свободных членов) путем элементарных преобразований (перестановка уравнений системы, умножение уравнений на число, отличное от нуля, и сложение уравнений) приводится к ступенчатому виду. Для этой цели в Scilab используется функция **rref([A,b])**.На втором этапе (***обратный ход***) ступенчатую матрицу преобразовывают так, чтобы в первых **n** столбцах получилась единичная матрица. Последний, **n+1** столбец этой матрицы содержит решение системы линейных уравнений.

На рис.2.6.2-4 приведено решение следующей СЛУ средствами Scilab по ***методу Гаусса***:



Метод Гаусса реализован в виде функции **Gauss(A,b)**. В качестве входных параметров здесь используется матрица коэффициентов (**А**) и вектор свободных членов (**b**), а выходным параметром является вектор, содержащий решение СЛУ (**Х**).

|  |
| --- |
|  |
| -->// ***Решение СЛУ по методу Гаусса***  -->clear  --> // ***Загрузка сценария* РИС2624**  --> // ***Матрица коэффициентов и вектор свободных членов***  --> A = [2 -1 1; 3 2 -5; 1 3 -2]; b = [0; 1; 4];  --> exec('РИС2624.sce');  -->X = Gauss(A, b) // ***Обращение к функции решения* СЛУ *по методу* Гаусса**  X =  0.4642857  1.6785714  0.75 |

Рис.2.6.2-4. Решение СЛУ методом Гаусса

**2.6.3 Контрольные вопросы**

1. Как формулируется постановка задачи численного решения системы линейных уравнений?
2. Какие методы решения СЛУ известны?
3. Что называют плохо ***обусловленными*** линейными системами?
4. Какой формат имеет функция **linsolve**?
5. Что представляет собой алгоритм решения СЛУ средствами Scilab по *методу Крамера*?
6. Что представляет собой алгоритм решения СЛУ средствами Scilab по *методу Гаусса*?

## **2.7. Решение задач оптимизации средствами Scilab**

Математические модели реальных объектов или технологических процессов при их исследовании или проектировании с известной степенью детализации могут быть представлены функциональными зависимостями, при этом, как правило, нелинейными. Переменные в таких функциях связаны между собой нелинейными законами, а зачастую ограничены предельными диапазонами изменения переменных, обеспечивая максимальную приближенность модели к изучаемому объекту. Математическое моделирование имеет своей целью не только достоверно отразить процесс функционирования объекта, но и произвести выбор таких значений его параметров, которые обеспечивают оптимальную эффективность этого объекта.

**2.7.1. Постановка задачи оптимизации**

***Постановка задачи оптимизации нелинейных функций***

Нелинейные функции, описывающие в теории оптимизации реальные объекты или процессы, принято называть *целевыми* или *критериальными*, а аргументы (независимые переменные) таких функций – *параметрами оптимизации*. *Решить задачу оптимизации – значит найти оптимальное (наилучшее) решение, в смысле заданного критерия.*

По количеству независимых переменных (**n**) различают задачи ***одномерной*** оптимизации (**n=1**) и ***многомерной*** оптимизации (**n≥2**). При этом задачу нахождения максимума целевой функции можно свести к задаче нахождения минимума путем замены исходной функции на функцию с обратным знаком, поэтому в дальнейшем будем говорить только о поиске минимума функции.

При решении ***задач одномерной оптимизации*** в области допустимых значений функция **f(x)** может иметь несколько минимумов. Говорят, что функция **f(x)** имеет в точке **x\* *локальный*** минимум, если существует такая **δ-**окрестность точки **х\***, что для всех значений **х**, принадлежащих **δ-**окрестности, выполняется неравенство **f(x)≥f(x\*)**. Функция **f(x)** имеет в точке **x\* *глобальный*** минимум, если для всех **х**, принадлежащих всей области допустимых значений, справедливо неравенство **f(x)≥f(x\*)**.

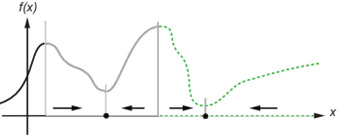


Рис. 2.7.1-1. Функция **f(x)**, имеющая локальный и глобальный минимумы

На рис. 2.7-1-1 показаны области, отмеченные различными стилями линий и стрелками направления спуска, на которых   черные точки представляют собой локальные минимумы, среди которых имеется и глобальный.

***Условия существования оптимума***

Задача ***одномерной оптимизации*** имеет единственное решение, если функция **f(x)** на отрезке **[a;b]** имеет только один экстремум, то есть она   
*унимодальна*. Для того чтобы функция на отрезке **[a;b]** была унимодальной должны выполняться следующие условия [24]:

* ***Необходимым условием*** существования экстремума дифференцируемой одномерной функции **f(x)** является выполнение равенства   
  **f′(х) = 0**;
* ***Достаточным условием*** существования единственного минимума одномерной функции **f(x)** на отрезке **[a;b]** является возможность дважды ее дифференцировать и требование положительности второй производной **f′′(х)** в любой точке этого отрезка.

Задача ***многомерной оптимизации*** сводится к нахождению оптимального значения функции **n** переменных **f(Х)=f(x1,x2,…,xn)**, определенной в **n**-мерном пространстве , где из множества допустимых решений выбирается такой вектор **Х\***, которому соответствует минимальное значение целевой функции **f(Х\*)**=**minf(Х)**.

Для всякой непрерывно дифференцируемой функции **f(Х) *достаточным*** условием того, что функция имеет точку минимума, является положительная определенность матрицы вторых частных производных (*матрицы Гессе или Гессиан*):



Известно, что матрица положительно определена в том случае, если все её угловые миноры положительны. Так, например, для функции двух переменных **f(x,y)** *матрица Гессе* имеет следующий вид:



и тогда достаточным условием существования минимума функции **f(x,y)** является выполнение неравенств:



Для решения задачи многомерной оптимизации требуется ввести понятие ***вектора-градиента*** функции **f(Х)**:



Этот вектор направлен в сторону наискорейшего возрастания функции, но поскольку все методы оптимизации предназначены для поиска минимума, то направление поиска минимума (спуска) противоположно, то есть совпадает с направлением вектора ***антиградиента:* .**

Таким образом, решение задачи многомерной оптимизации является задачей поиска:

* ***локального оптимума функции*** – где точка является точкой локального минимума функции **f(Х)** на множестве **X**, если существует такая **δ** – окрестность **Uδ** этой точки (**δ>0**), что для всех   
  **Х∈Хδ=ХUδ** выполняется неравенство **f(X\*)≤f(X)**;
* ***глобального оптимума функции*** – где точка является точкой глобального минимума функции **f(Х)** на множестве**X**, если функция достигает в этой точке своего наименьшего значения:  
  **f(X\*)≤f(Х), **.

Также известно, что ***аналитические методы поиска оптимума*** применимы только для ограниченного круга задач. В основном это связано с тем, что при использовании аналитических методов возникает необходимость вычисления производных и решения нелинейных уравнений (а для многомерных функций – систем нелинейных уравнений), которые тоже, как правило, решаются только численными методами.

Для поиска оптимального значения целевой функции численными методами кроме самой целевой функции необходимо задать точность искомого оптимума **Ɛ**и начальную точку поиска **X0**.

Выбор начального приближения является нетривиальной задачей. Поэтому прежде, чем приступить к численному решению задачи оптимизации нелинейных функций, необходимо провести исследование целевой функции.

***Исследование целевой функции***

Исследование целевой функции можно осуществлять как графически, так и аналитически.

Проведение графического исследования поведения целевой функции (которое возможно только для функций от одной и двух переменных) очень полезно с точки зрения того, что можно не только визуально удостовериться в наличии оптимума, но и провести правильный выбор начальной точки его поиска.

Так в случае ***одномерной оптимизации*** можно в области допустимых значений **x**построить график целевой функции, например,

**f(x)=x4 + 3x3 - 13x2 - 26x + 26**,

который позволяет определить отрезок, содержащий единственный минимум, а затем на этом отрезке, выбрать точку, которая будет использована в качестве начального приближения для дальнейшего уточнения координат точки минимума с требуемой точностью (рис.2.7.1-2).

|  |
| --- |
| --> // ***Исследование функции на интервале* (-100; 100)**  -->  -->scf(0)  -->x = -100:0.1:100;  --> // ***Загрузка сценария* РИС2712**  --> exec('РИС2712.sce', 0) |
| --> // ***Уточнение границ отрезка на интервале* (-5; 1)**  -->  -->scf(1)  --> x = -5:0.1:1;  --> exec('РИС2712.sce') |

Рис. 2.7.1-2. Исследование графика целевой функции **f(x)**

Из графика следует, что на отрезке **[-4.5;-3]** имеется один локальный минимум. После графического исследования, если это возможно, полезно провести аналитическое исследование функции: вычислить первую и вторую производные целевой функции, и убедиться, что на отрезке **[-4;-3] f′(х)** меняет знак с минуса на плюс, а **f′′(х)** –положительна.

Полученная таблица значений функции и производных (2.7.1-3) подтверждает правильность выбора отрезка **[-4;-3]**, следовательно, из этого отрезка и будет выбрана начальная точка для дальнейшего уточнения точки минимума.

|  |
| --- |
| --> // ***Аналитическое исследование функции* f(x) *на отрезке* [-4; -2]**  -->  --> // ***Загрузка сценария* РИС2713**  -->exec('РИС2713.sce', 0)  --> x = -4 : 0.5 : -3; y = f(x); y1 = f1(x); y2 = f2(x);  --> // ***Таблица значений функции и двух ее производных***  --> disp([x; y; y1; y2]',[' x ', ' y ', ' y1 ', ' y2 ']);  ! **x y y1 y2** !  -4. 22500. -14. 94.  -3.5 12656.25 23.75 58.  -3. 6400. 45. 28. |

Рис. 2.7.1-3. Аналитическое исследование функции **f(x)** на отрезке **[-4;-3]**

Проведем исследование поведения ***функции от двух переменных***. Например, предположим, что необходимо выбрать начальную точку для поиска минимума *функции Розенброка:*

**f(x1,x2)=100(x2-x12)2+(1-x1)2**.

Отметим, что *функция Розенброка* является тестовой функцией для многих методов оптимизации в связи с тем, что она относится к классу овражных функций, для которых результат поиска сильно зависит от выбора начальных условий.

Построим график поверхности и контурное изображение функции, которые помогут визуально оценить оптимальную область функции   
*Розенброка* (рис. 2.7.1-4).

|  |
| --- |
|  |
| -->// ***Исследование функции*rosenbrock**  -->  -->exec('РИС2714.sce', 0);  --> xdata = linspace(-2, 2, 100);  --> ydata = linspace(-2, 2, 100);  --> contour(xdata, ydata, rosenbrock, [1 10 100 1000])  --> [x, y] = meshgrid(-2:0.1:2, -2:0.1:2);  --> z = rosenbrock(x, y);  --> scf(1); surf(x, y, z); |
|  |
|  |

Рис. 2.7.1-4. Исследование *функции Розенброка* **f(x1,x2)**

Из контурного графика функции следует, что минимальным является контур со значением функции **f=1**. Следовательно, в качестве начальной точки поиска минимума может быть выбрана, например, точка с координатами **(0,0)**, определяемая в Scilab вектором **x=[0,0]**.

Построение контурного графика может послужить первым шагом для нахождения оптимального решения многомерной функции, и, кроме того, позволяет сократить количество итераций по поиску оптимума. Однако получить графическое изображение функции можно только тогда, когда у нас есть ограниченное количество входных переменных (**1** или **2**). Во всех остальных случаях приходится сразу приступать к использованию численных алгоритмов и тогда начальное приближение выбирается исходя из пространства допустимых значений конкретной функции.

**2.7.2. Численные методы оптимизации и их   
реализация в Scilab**

***Численные методы оптимизации нелинейных функций***

Среди численных методов поиска экстремума нелинейных функций можно выделить [24]:

* *методы прямого поиска*, то есть методы, в которых при поиске экстремума целевой функции используются только ее значения;
* *градиентные методы,* а также многочисленные модификации этих методов, первого или второго порядка, в которых, при поиске *экстремума функции* используются, соответственно, значения ее первых производных или первых и вторых производных функции (т.е. в основе градиентных методов лежит выбор направления поиска и шага спуска в этом направлении).

В Scilab реализованы методы поиска оптимума, которые применимы как для решения задачи одномерной, так и многомерной оптимизации для условной и безусловной нелинейной оптимизации [13].

***Безусловная оптимизация*** нелинейных функций использует два базовых алгоритма: *Квази-Ньютоновские алгоритмы* и *алгоритмы* [*Нелдера-Мида*](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9D%D0%B5%D0%BB%D0%B4%D0%B5%D1%80%D0%B0_%E2%80%94_%D0%9C%D0%B8%D0%B4%D0%B0).

***Квази-Ньютоновские алгоритмы***, применяют смешанную процедуру поиска с использованием квадратичного или кубического полиномов и алгоритм *BFGS* для аппроксимации и корректировки *Гессиана* (матрицы вторых частных производных). *Квази-Ньютоновские методы* [*оптимизации*](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) основаны на накоплении информации о [кривизне](https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B8%D0%B2%D0%B8%D0%B7%D0%BD%D0%B0) [целевой функции](https://ru.wikipedia.org/wiki/%D0%A6%D0%B5%D0%BB%D0%B5%D0%B2%D0%B0%D1%8F_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F) по наблюдениям за изменением [градиента](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%B4%D0%B8%D0%B5%D0%BD%D1%82), и принципиально отличаются от [*Ньютоновских*методов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9D%D1%8C%D1%8E%D1%82%D0%BE%D0%BD%D0%B0)тем,что *Квази-Ньютоновские* методы исключают явное формирование матрицы *Гессе*, заменяя её приближением, исходя из сделанных до этого шагов.

Метод *BFGS* – итерационный метод [численной оптимизации](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D0%BC%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)), предназначенный для нахождения локального минимума нелинейной [функции](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB) без ограничений. Также существуют модификация этого метода: метод с ограниченным использованием дополнительной памяти ([*L-BFGS*](https://ru.wikipedia.org/w/index.php?title=L-BFGS&action=edit&redlink=1)), который предназначен для решения нелинейных задач с большим количеством неизвестных, а также метод с ограниченным использованием памяти в многомерном кубе   
([*L-BFGS-B*](https://ru.wikipedia.org/w/index.php?title=L-BFGS-B&action=edit&redlink=1)). Этот метод находит минимум любой дважды непрерывно дифференцируемой выпуклой функции. Несмотря на эти теоретические ограничения, ***BFGS*** хорошо справляется и с невыпуклыми функциями.

***Алгоритм*** [*Нелдера-Мида*](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9D%D0%B5%D0%BB%D0%B4%D0%B5%D1%80%D0%B0_%E2%80%94_%D0%9C%D0%B8%D0%B4%D0%B0), также известен как ***метод деформируемого многогранника или симплекс-метод***.  Этот алгоритм является алгоритмом прямого поиска и использует только значения функции (не требует производные), а поэтому легко применим к негладким и/или зашумленным функциям. Здесь регулярным симплексом называется множество **(n+1)**-й равноудаленной точки в **n**-мерном пространстве. В двумерном пространстве симплексом является равносторонний треугольник, а в трехмерном пространстве – правильный тетраэдр.

Алгоритм, построенный на основе метода [*Нелдера-Мида*](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_%D0%9D%D0%B5%D0%BB%D0%B4%D0%B5%D1%80%D0%B0_%E2%80%94_%D0%9C%D0%B8%D0%B4%D0%B0), заключается в последовательном перемещении и деформировании [симплекса](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%BF%D0%BB%D0%B5%D0%BA%D1%81) вокруг точки экстремума. Идея метода состоит в сравнении значений функции в (**n+1**) вершинах *симплекса* и перемещении *симплекса* в направлении оптимальной точки с помощью итерационной процедуры. Существуют несколько модификаций этого метода, допускающих, чтобы симплексы были неправильными. В результате получился очень надежный метод прямого поиска, являющийся одним из самых эффективных, если **n<6**. В различных модификациях метода симплекс может перемещаться с помощью трех основных операций: ***отражения, растяжения*** и ***сжатия***.

***Средства Scilab для решения задач оптимизации***Программные средства Scilab для решения определенных задач оптимизации называют *решателями*. Безусловно, одной из самых сложных проблем, возникающих при решении задачи оптимизации, является проблема формализации – представление реального оптимизируемого объекта или процесса в виде математической модели, т.е. в виде функции, связывающей параметры оптимизации с воздействующими на неё факторами. При этом второй по сложности проблемой здесь является выбор из множества существующих алгоритмов подходящего и позволяющего обеспечить сходимость за приемлемое время. Эту задачу в системе Scilab помогают выполнить решатели, включающие различные методы, настраиваемые на максимальную скорость решения задачи оптимизации, используя доступные вычислительные ресурсы и соответствующие настройки их параметров. То есть в Scilab программные средства оптимизации реализованы как наборы объектно-ориентированных инструментов – решателей и вспомогательных компонент.

Основными программными средствами оптимизации в Scilab являются [13]:

* [**optim**](https://help.scilab.org/docs/6.0.1/en_US/optim.html)– решатель, предназначенный для задач нелинейной оптимизации как без ограничений, так и с ограничениями, основанный на *Квази-Ньютоновских методах* и использующий внешнюю функцию [**cost**](https://help.scilab.org/docs/6.0.1/en_US/NDcost.html), которая позволяет свести к минимуму проблемы, связанные с вычислением производных и их градиентов, в тех случаях когда их сложно получить в явном виде;
* ***Neldermead* –** набор объектно-ориентированных инструментальных средств, который поддерживает прямой поиск оптимума, основанный на методах *Нелдера-Мида*.

Они могут использоваться, если:

* нет необходимости или невозможно получить производные целевых функций;
* количество параметров невелико (не более 6);
* могут существовать границы и/или нелинейные ограничения.

В наборе инструментальных средств ***Neldermead***предоставлены

следующие *решатели*:

* **neldermead**– *решатель*, который реализует различные варианты алгоритмов *Нелдера-Мида и* может управлять ими для установки их компонент, таких, например, как способы вычисления начальных симплексов, критерии завершения процесса оптимизации и многие другие;
* **fminsearch** – *решатель*, который реализует упрощенный алгоритм *Нелдера-Мида*, в котором заранее заданы конкретные установки критериев прекращения оптимизации, начальные симплексы и вспомогательные установки.
* **nmplot**– решатель, обеспечивающий расширенные возможности для отображения и хранение выходных графиков хода выполнения итерации алгоритма *Нелдера-Мида*, то есть его можно рассматривать как инструмент для исследования процесса оптимизации и обучения.

Вспомогательными классами и методами для этих решателей являются:

* **optimbase**– абстрактный, определяющий такие параметры как количество переменных, минимальные и максимальные границы, количество нелинейных ограничений неравенства, критерии прекращения, целевые функции; **optimsimplex**– класс для управления симплексом из произвольного числа вершин, включая вычисление симплекса различными методами и управление его параметрами.
* **optimset** –метод, который может создавать или обновлять значения структур данных оптимизации для изменения поведения методов и результатов оптимизации;
* **optimget** – метод, который запрашивает текущие значения структуры данных оптимизации;
* **optimplotfunccount**, **optimplotx** и **optimplotfval** – методы, которые осуществляют управление возможностями вывода результатов для решателей.

Таким образом, система Scilab имея большую коллекцию доступных алгоритмов, обладает большим потенциалом для решения задач нелинейной оптимизации функций как одной, так и многих переменных.

Рассмотрим использование некоторых упомянутых выше решателей более подробнее.

**2.7.3. Решатели нелинейной оптимизации Scilab**

***Решатель* optim**

*Решатель* **optim**предназначен для решения задачи нелинейной оптимизации f**(X)**, где **X** является вектором, а f**(X)** функцией, возвращающей скаляр. Функция **optim** использует *Квази-Ньютоновские методы*, основанные на различных алгоритмах, в том числе на *BFGS-*алгоритме, который представляет собой алгоритм высокой точности для локальной оптимизации. Пространство допустимых решений этой функции может быть и ограниченное – задача нелинейной оптимизации с ограничениями (или условная оптимизация):

**minf(X)**, при**binf<= X<= bsup**,

где **binf** –  нижняя граница, а **bsup** – верхняя граница **X**.

Для вычисления значений целевой функции и её производных функция **optim**использует вспомогательную функцию **costf**. Имя этой функции входит в состав аргументов в качестве указателя на функцию. Заметим, что функция **costf** является внешней, и может быть написана на языках Scilab, C++ или Fortran.

В случае если **costf** является **sce**-функцией, то обращение к ней может иметь следующий формат:

**[f,g,ind]=costf(X,ind)**,

где **X** – вектор значений текущей точки, **ind** – параметр, предназначенный для связи функции **costf**с функцией **optim**(если значение **ind** равно **2 , 3** или **4** то функция **сostf**обеспечивает поиск минимума, если **ind=1**, то в функция **optim**не выполняется), **f**– вещественное значение целевой функции в точке **X**, а **g** – вектор, содержащий значения частных производных целевой функции (или, в случае одномерной оптимизации, значение производной), вычисляемых с использованием функции **numdervative(f,X)**.

Выходное значение параметра **ind**показывает причину завершения работы функции **costf**, причем, если **ind<0,** то это значит, что **f** не может являться оценкой **X,** а **ind=0**  означает прерывание процесса оптимизации.

Функция **optim**может быть вызвана одним из следующих форматов:

**fopt=optim(***costf,X0***)**

**fopt=optim(***costf*,<*contr*>**,***X0***,algo,df0,mem,work,<*stop*>,iprint=*iflag*)**

**[fopt,xopt]=optim(…)**

**[fopt,xopt,gopt]=optim(…)**

**[fopt, xopt, gopt, work] = optim(…)**

**[fopt, xopt, gopt, work, iters] = optim(…)**

**[fopt, xopt, gopt, work, iters, evals]= optim(…)**

**[fopt, xopt, gopt, work, iters, evals, err]=optim(…)**

**fopt,xopt,gopt,work,iters,evals,err,ti,td=optim(…,params="si","sd")**

*Набор обязательных входных параметров:*

*сostf* – функция, вычисляющая целевую функцию и частные производные (или производную в случае одномерной  
оптимизации);

*X0* – вещественный вектор, состоящий из начальных значений**X**.

*Необязательные входные параметры:*

**<***сontr***>**– *последовательность (список)* параметров, содержащий границы **X**: **"***b***"**, *binf* и *bsup*,  где  *binf* и *bsup*  – вещественные вектора с таким же размером, что и  *X0*;

*аlgo* – строка, указывающая используемые доступные алгоритмы (по умолчанию – **"qn"**):

**"qn"** – *Квази-Ньютоновский BFGS;*

**"gc"** – ограниченная память *BFGS;*

**"nd"** – не используются производные и не учитываются

ограничения по **X**.

*df0* – вещественный скаляр, представляющий значение **f** на первой   
итерации (по умолчанию **df0=1**).

*mem* – целое число переменных, используемых для матрицы *Гессе* (по

умолчанию **mem=10**);

**<***stop***>**– *последовательность (список)* параметров, управляющий

cходимостью алгоритма:

**"ar",** *nap*

**"ar",** *nap, iter*

**"ar",** *nap, iter, epsg*

**"ar",** *nap, iter, epsg, epsf*

**"ar",** *nap,iter, epsg, epsf, epsx*

где: *nap* – максимальное количество допустимых   
 вызовов  *costf*(по умолчанию **nap=100**);

*iter* – максимальное количество итераций (по умолчанию

**iter=100**);

*epsg* – порог градиентной нормы (по умолчанию **epsg=%eps**);

*epsf* – порог контроля снижения **f** (по умолчанию **epsf=0**);

*epsx* – порог контроля изменения **X** (по умолчанию **epsx=0**),

**"iprint=***iflag***"** – именованный аргумент, используемый для установки режима трассировки выходных данных (по умолчанию **iprint=0**, режим, при котором печатаются сообщения), причем, если **iprint** больше или равно **1**, выводится больше информации, в зависимости от выбранного алгоритма:

Если *iflag***<0**, то целевая функция вычисляется в каждой из **m** итераций, с **ind=1**.

*Выходные параметры*:

*fopt* – значение целевой функции в точке *xopt*;

*xopt* – вектор аргументов, обеспечивших оптимальное значение

функции;

*gopt* – градиент целевой функции (или производной в случае одномерной оптимизации) в точке *xopt*;

*work – рабочий массив перезапуска для Квази*-Ньютоновских методов.

*iters* – скаляр, число итераций, которое отображается при **iprint=2**.

*evals* – скаляр, число вычислений функции *cost***,** которое отображается

при **iprint=2**.

*err* – скаляр, индикатор завершения, может иметь следующие   
значения:

* *err***=1** – норма вычисленного градиента ниже допустимого;
* *err***=2** – на последней итерации **f** уменьшается;
* *err***=3** – оптимизация останавливается из-за слишком малых

изменений для **X**;

* *err***=4** – остановка: максимальное количество обращений к **f**;
* *err***=5** – остановка: максимальное количество итераций;
* *err***=6** – остановка: слишком маленькие изменения в   
  направлении градиента;
* *err***=7** – остановка: во время расчета направления спуска;
* *err***=8** – остановка: во время расчета оценки матрицы *Гессе*;
* *err***=9** – окончание оптимизации, успешное завершение;
* *err***=10** – успешное окончание оптимизации (линейный поиск  
  не выполняется).

Каждый алгоритм, реализованный в **optim**, имеет свои собственные критерии завершения, которые могут использовать параметры, полученные от пользователя, такие как: *nap, iter, epsg, epsf*и*epsx*.

Рассмотрим пример использования решателя **optim** для случая решения задачи оптимизации ***одномерной*** функции **f(x)=x4+3x3-13x2-6x+26**, имеющей на отрезке **[-4;-2]** единственный минимум (исследование которой приведено выше на рис. 2.7.1-2 и 2.7.1-3), выбрав в качестве начальной точки **х0=-3   
(**рис. 2.7.3-1). В нашем случае используется достаточно простой формат функции **optim** – передается имя вспомогательной функции (**costf)** и начальное значение аргумента (**х0**), а в качестве выходных параметров выступают координаты точки минимума функции **f(х)**.

|  |
| --- |
|  |
| -->// ***Нахождение координат точки минимума с использованием решателя*optim**  -->  -->exec('РИС2731.sce');  -->x0 = -3;  --> // ***Вычисление координат точки минимума***  --> [fmin, xmin] = optim(costf, x0); // ***Обращение к функции* optim**  --> disp([fmin, xmin])  -95.089413 -3.8407084 |

Рис. 2.7.3-1. Нахождение координат точки минимума функции **f(x)**с использованием решателя **optim**

В примере, приведенном на рис. 2.7.3-1 для вычисления производной используется функция **numderivative**, что позволяет устранить проблему получения аналитических выражений производных в случае сложного выражения целевой функции. Однако использовать эту функцию совсем необязательно. Так, например, на рис. 2.7.3-2 приведен сценарий, в котором для вычисления значений производной используется аналитическое выражение.

|  |
| --- |
|  |

Рис. 2.7.3-2. Вычисление производной с использованием   
ее математического выражения

В следующем примере (рис.2.7.3-3) определим координаты точки минимума и значение ***многомерной*** функции *Розенброка****:***

**f(x1,x2)=100(x2-x12)2+(1-x1)2,**

с использованием того же упрощенного формата функции **optim**. Ранее (рис.2.7.1-4) в качестве начального приближения уже был выбран вектор начальных значений **[0,0]**.

|  |
| --- |
|  |
| --> // ***Нахождение минимума многомерной функции* f(x1,x2*)***  -->  -->exec('РИС2733.sce', 0);  -->x0 = [0; 0]; // ***Начальныезначения***  --> [f, xopt] = optim(cst, x0); // ***Обращение к функции* optim**  disp(f, [xopt']);  1. 1.  .083D-30 |

Рис. 2.7.3-3. Использование функции **optim** для нахождения координат точки минимума многомерной функции **f(x1,x2)**

Отметим, что в описанных выше примерах были использованы самые простые форматы функции **optim**. Однако в процессе вычислений часто представляет интерес: сколько итераций потребовалось для нахождения минимума той или иной функции, причину прекращения итерационного процесса и многие другие моменты, которые могут быть выявлены при использовании других, описанных выше, форматов этой функции.

Именно поэтому пользователю, решающему серьезные задачи оптимизации крайне важно рассмотреть и проанализировать результаты выполнения следующего примера (рис. 2.7.3.4), показывающего возможности использования различных (необязательных) параметров функции **optim**.

|  |
| --- |
|  |
| --> // ***Загрузка сценария* РИС2734** --> // ***и обращение к решателю* optim с *дополнительными параметрами***  --> //  --> // exec('РИС2734.sce', 0)  Xopt = 1.2.3.; fopt=0.; gopt= 0.0.0.  Xopt = 1. 2. 3.; fopt=0.;gopt= 0.-1.776D-14-1.199D-14  xopt=1. 2. 3.; fopt=2.297D-28; gopt=-0.5-1.0.  xopt=0.5 1.3.;fopt=0.625; gopt=-0.5 -1. 0.  xopt=0.5 1. 3.;fopt=0.625; gopt=-0.5 -1.5559108 -0.7779554  xopt=0.5 1. 2.2220446 fopt=1.6380365 gopt=-0.5 -1.8308232 -0.9154116  xopt=0.5 1. 2.0845884 fopt=2.2199459  \*\*\*\*\* enters -qn code- (without bound cstr)  dimension=3, epsq=0.2220446049250313E-15, verbosity level: iprint=3  max number of iterations allowed: iter=100  max number of calls to costf allowed: nap=100  ------------------------------------------------  iter num 1, nb calls=1, f=6.500  linear search: initial derivative=-3.606  step length=0.1000E-01, df=-0.4261, derivative=-3.485  step length=0.1000, df=-3.611, derivative=-2.404  iter num 2, nb calls=3, f=2.889  linear search: initial derivative=-2.404  step length=1.000,df=-2.889,derivative=0.3695E-15  iter num 3, nb calls=4, f=0.9861E-30  linear search: initial derivative=-0.1380E-14  step length=1.000,df=0.4142E-29, derivative=0.3192E-14  step length=0.2996,df=-0.9861E-30,derivative=0.0000E+00  iter num 4, nb calls=6, f=0.0000E+00  \*\*\*\*\* leaves -qn code-, gradient norm=0.0000000000000000E+00  optim: Норма проектируемого градиента ниже, чем 0.  xopt=1.2.3.fopt=0. | |

x0 = 1. -1. 1.

xopt = 1. 2. 3.

fopt = 0.

gopt = 0. 0. 0.

xopt = 1. 2. 3.

fopt = 0.

gopt = 0. -1.776D-14 -1.199D-14

xopt = 1. 2. 3.

fopt = 2.297D-28

gopt = -0.5 -1. 0.

xopt = 0.5 1. 3.

fopt = 0.625

gopt = -0.5 -1. 0.

xopt = 0.5 1. 3.

fopt = 0.625

gopt = -0.5 -1.5559108 -0.7779554

xopt = 0.5 1. 2.2220446

fopt = 1.6380365

gopt = -0.5 -1.8308232 -0.9154116

xopt = 0.5 1. 2.0845884

fopt = 2.2199459

\*\*\*\*\* enters -qn code- (without bound cstr)

dimension= 3, epsq= 0.2220446049250313E-15, verbosity level: iprint= 3

max number of iterations allowed: iter= 100

max number of calls to costf allowed: nap= 100

------------------------------------------------

iter num 1, nb calls= 1, f= 6.500

linear search: initial derivative= -3.606

step length= 0.1000E-01, df=-0.4261 , derivative= -3.485

step length= 0.1000 , df= -3.611 , derivative= -2.404

iter num 2, nb calls= 3, f= 2.889

linear search: initial derivative= -2.404

step length= 1.000, df= -2.889, derivative= 0.3695E-15

iter num 3, nb calls= 4, f= 0.9861E-30

linear search: initial derivative=-0.1380E-14

step length= 1.000, df= 0.4142E-29, derivative= 0.3192E-14

step length= 0.2996, df=-0.9861E-30, derivative= 0.0000E+00

iter num 4, nb calls= 6, f= 0.0000E+00

\*\*\*\*\* leaves -qn code-, gradient norm= 0.0000000000000000E+00

optim: Норма проектируемого градиента ниже, чем 0.

xopt = 1. 2. 3.

fopt = 0.

Рис. 2.7.3-4. Применение дополнительных параметров при использовании функции **optim**

***Решатель* fminsearch**

*Решатель* **fminsearch –** вычисляет безусловный минимум функции по алгоритму *Нелдера-Мида* и имеет следующие форматы [13]:

*X***=fminsearch(***costf,х0***)**

*X***=fminsearch(***costf,х0***,options)**

**[***X,fval***]=fminsearch(***costf,х0***,options)**

**[***X,fval,exitflag***]=fminsearch(***costf,х0***,options)**

**[***X,fval,exitflag,output***]=fminsearch(***costf,х0***,options)**

Этот *решатель* является алгоритмом прямого поиска, не использующим производную целевой функции, а основан на обновлении симплекса, который является набором **k>=n+1** вершин, где каждая вершина связана с одной точкой и одним значением функции. Алгоритм создан на базе [**neldermead**](https://help.scilab.org/docs/6.0.1/en_US/neldermead.html) компонент [13].

Рассмотрим подробнее назначение параметров:

*Входные параметры:*

*сostf* – *внешняя функция*, которая возвращает значение целевой   
функции и имеет следующий заголовок **f=***costf***(X),**где **X**– текущая точка;

**f** – значение целевой функции;

*X0* – матрица начальных значений;

**options** – *структура данных*, содержащая настраиваемые параметры   
алгоритма оптимизации, значения которых и определяют   
конфигурацию алгоритма **fminsearch**, причем **fminsearch**чувствительна к следующим опциям:

**options.MaxIter** – максимальное количество итераций, значение по

умолчанию – **200\*n**, где **n**- количество переменных;

**options.MaxFunEvals** – максимальное количество оценок функции  
затрат, значение по умолчанию – **200\*n**, где **n** – количество  
переменных;

**options.TolFun** – абсолютный допуск по значению функции, значение которого по умолчанию – **1.e-4**;

**options.TolX** – абсолютный допуск на размер симплекса, значение по умолчанию – **1.e-4**;

**options.Display** – подробный уровень, для которого возможны значения: **"notify","iter","final"** или **"off"**, значение по умолчанию –  
**"notify"**;

**options.OutputFcn** – выходная функция или список выходных

функций;

**options.PlotFcns** – функция печати или список функций печати.

*Выходные параметры*:

*X* – значение аргумента, обеспечившее минимальное значение функции**;**

*fval*– минимальное значение функции.

*еxitflag* – флаг, связанный со статусом выполнения алгоритма:

**-1** – максимальное количество итераций достигнуто;

**0** – максимальное число оценок функций достигнуто;

**1** – допуск на размер симплекса и его изменения уже

достигнут: алгоритм не имеет сходимости;

**output** – структура данных, которая хранит информацию о   
выходных данных и содержит следующие поля:

**output.algorithm** – строка, содержащая название алгоритма, то есть алгоритма Нелдера*-*Мида;

**output.funcCount** – количество оценок функции;

**output.iterations** – количество итераций;

**output.message** – строка, содержащая сообщение о завершении.

***Критерии прекращения процесса оптимизации*** используют следующие переменные:

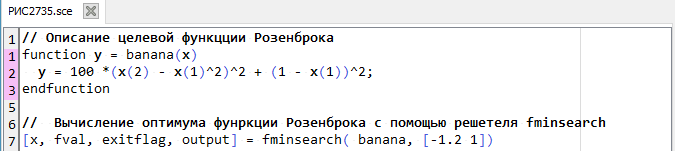
**ssize** – текущий размер симплекса;

**shiftfv** – абсолютное значение разности значения функции  
между самой высокой и самой низкой вершиной.

Процесс итерации останавливается, если выполняются следующие  
условия: **ssize<options.TolFun**&**shiftfv<options.TolFun**.

Алгоритм, заложенный в **fminsearch** использует специальный начальный симплекс, который вычисляется эвристически в зависимости от начального предположения (подробнее см. в документации о **optimsimplex [x])** .

В следующих примерах для вычисления минимума *функции Розенброка* воспользуемся решателем **fminsearch**, описав предварительно функцию под именем **banana**, и задав начальные значения поиска **[-1.2 1.0].** В данном случае для поиска минимума потребовалось выполнение **85** итераций и **159** обращений к функции оценки (рис. 2.7.3-5).





--> // ***Загрузка сценария РИС2735***

--> // **и вычисление минимума с использованием решателя fminsearch**

-->

--> exec('РИС2735.sce', 0)

[output.Output, output.funcCount, output.iterations, output.message]

exitflag =

1.

fval =

0.00000000081776611

x =

1.0000220217835567 1.0000422197517711

--> mprintf('algorithm = %c\nfuncCount = %4d\niterations = %3d',…

output.algorithm, output.funcCount, output.iterations);

algorithm = Nelder-Mead simplex direct search

funcCount = 159

iterations = 85

--> output.message

ans =

!Optimization terminated:

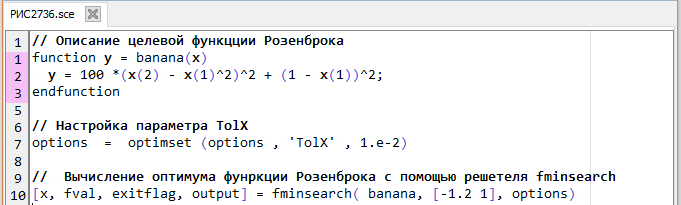
!

! the current x satisfies the termination criteria using OPTIONS.TolX of 0.0001 ! and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 0.0001 !

Рис. 2.7.3-5 Вычисление минимума *функции Розенброка*

с использованием решателя **fminsearch**

Далее рассмотрим пример вычисление минимума *функции Розенброка* решателем **fminsearch *с настраиваемыми опциями.*** Настроим абсолютный допустимый размер симплекса до большего значения, так чтобы алгоритм выполнял меньшее количество итераций. Поскольку по умолчанию значение **tolX=1.e-4**, заменим его в [**optimset**](https://help.scilab.org/docs/6.0.1/en_US/optimset.html) на большее значение, например, **1.e-2**. В данном случае потребовалось **70**итераций и **130**обращений к функции оценки (рис. 2.7.3-6).



-->// ***Загрузка сценария РИС2735, настройка некоторых опций***

-->// ***и вычисление минимума с использованием решателя*fminsearch**

--> exec('РИС2736.sce', 0)

options =

Display: [0x0 constant]

FunValCheck: [0x0 constant]

MaxFunEvals: [0x0 constant]

MaxIter: [0x0 constant]

OutputFcn: [0x0 constant]

PlotFcns: [0x0 constant]

TolFun: [0x0 constant]

TolX: [1x1 constant]

output =

algorithm: [1x1 string]

funcCount: [1x1 constant]

iterations: [1x1 constant]

message: [3x1 string]

exitflag =

1.

fval =

0.0000085

x =

1.001016 1.0017605

-->

--> mprintf('algorithm = %c\nfuncCount = %4d\niterations = %3d',…

output.algorithm, output.funcCount, output.iterations);

algorithm = Nelder-Mead simplex direct search

funcCount = 130

iterations = 70

-->

--> output.message

ans =

!Optimization terminated: !

! the current x satisfies the termination criteria using OPTIONS.TolX of 0.01

! and F(X) satisfies the convergence criteria using OPTIONS.TolFun of 0.0001

Рис. 2.7.3-6. Вычисление минимума *функции Розенброка*   
решателем **fminsearch** с настраиваемыми опциями

***Решатель* nmplot**

Кроме описанных выше решателей **optim** и **fminsearch**, в Scilab представляет интерес решатель **nmplot** [13], который использует алгоритмы *оптимизации прямого поиска*, при этом всю информацию о процессе оптимизации на каждой итерации можно отобразить на экране или сохранить в файлах для ее последующего анализа.

Решатель **nmplot** включает несколько методов оптимизации прямого поиска, основанных на симплексном методе. Он является модификацией компонента **neldermead**, с выводом результатов, и позволяет хранить историю значений данных в процессе итераций.

Этими данные могут быть:

* значения координат симплекса;
* значения функции, усредненные по вершинам;
* минимальные значения функции в симплексе;
* размеры симплекса.

Во время процесса оптимизации эти данные хранятся в нескольких файлах.

На рис. 2.7.3-7 приведен пример только графической траектории поиска минимума функции с использованием решателя **nmplot.** Подробноэтотпримериспользования решателя **nmplot** рассмотрен в [13]. Из которого следует, что выбор направления спуска определяется симплексом.

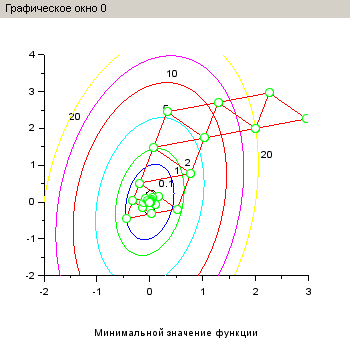


Рис. 2.7.3-7 Траектория спуска при поиске минимума с использованием   
решателя **nmplot**

**2.7.4 Контрольные вопросы**

1. Как формулируется постановка численного решения задачи оптимизации нелинейных функций?
2. Что является безусловной и условной оптимизацией нелинейных функций?
3. Что является достаточными условиями существования минимума для одномерной и многомерной функции?
4. Что такое локальный и глобальный минимумы функций?
5. Можно ли средствами Scilab вычислить глобальный минимум заданной функции?
6. Какие исследования целевой функции целесообразно произвести перед поиском локального минимума средствами Scilab?
7. Как выбрать начальные приближения в случае многомерной оптимизации?
8. Что такое решатель и в чем его обличие от функции?
9. На каких идеях основаны *Квази-Ньютоновские* алгоритмы поиска оптимума?
10. На каких идеях основаны методы деформируемого многогранника поиска оптимумам?
11. Какие средства Scilab известны для решения задач оптимизации функций?
12. Какие основные программные средствами оптимизации в Scilab известны?
13. Для чего используется решатель [**optim**](https://help.scilab.org/docs/6.0.1/en_US/optim.html)?
14. Решатель **optim** и назначение ее входных и выходных параметров?
15. Для чего при использовании функции **optim** необходимо формирование вспомогательной функции?
16. Можно ли использовать решатель **optim** без выходных параметров?
17. Можно ли с использованием решатель **optim** вычислить локальный максимум?
18. Назначение функции **costf*,*** ее входные и выходные параметры.
19. Что служит результатом выполнения решателя **optim?**
20. Какие средства Scilab используются для описания целевой функции, и как это влияет на параметры решателя **optim?**
21. Для чего используется решатель **fminsearch**
22. Каковы особенности решателя **nmplot**?

# Приложение

Приложение 1.2

*Системные встроенные функции*. Таблица 1.2.1-1

|  |  |
| --- | --- |
| **Имя функции** | **Назначение** |
| **whos**  **whos()**  **whos** -**type***Тип*  **whos -name***Имя* | ***Отображает переменные в длинной форме***  Отображает все текущие имена переменных (без учета регистра), типам и используемой памяти.  Отображает все текущие переменные с указанным типом *Тип,* где  *Тип –* текстовая строка, кодирующая тип данных.  Отображает все текущие переменные, имена которых начинаются *Имя*, где *Имя –*  имена искомых переменных, либо их начальные фрагменты. |
| **who**  **who()**  **who('local')**  **who('get')**  **who('global')**  **who('sorted')** | ***Отображает переменны****е*  Отображается текущие имена переменных и констант.  Отображается имена локальных переменных и параметры память, используемые в данных двойной точности.  Отображается имена глобальных переменных и параметры память, используемые в данных двойной точности.  Отображается имена всех переменных(если переменная является глобальной, \* появляется после имени типа). |
| **сlear**  **clear('a','b',..)** | ***Удаление из памяти объектов и  освобождение имен переменных***  Удаляются все незащищенные объекты и освобождаются имена текущей сессии  Удаляются указанные незащищенные объекты и освобождаются имена текущей сессии |
| **predef()**  **predef('***a***')**  **predef('с')**  **predef('***Список***')** | ***Защищает переменные***  Возвращается количество защищенных переменных.  Защищаются все переменные из списка **who('get').**  Снимается защита со всех переменных списка **who('get').**  Отображается *Список* защищенных переменных. |
| **xists('***Имя***')**  **exists('***Имя***',** 'l'**)**  **exists('***Имя***',** 'n'**)**  **exists('***Имя***',** 'a'**)** | ***Проверяет существование объекта с указанным именем***  Если объект заданного типа существует, то возвращается**T**, в противном случае **F**, где**'l'** – локальный,**'n'** – нелокальный,**'a**'  – все в () (по умолчанию)**.** |

*Функции, позволяющие получить номера и названия типов объектов*. Таблица 1.2.1-2

|  |  |  |
| --- | --- | --- |
| *НомерТипаОбъекта* **= type(***Oбъект***) –** Возвращает номер типа объекта. | | |
| *НазваниеТипаОбъект*а **= typeof(***Oбъект***) –** Возвращает названия типа объекта. | | |
| **№**  **Типа** | **Название Типа** | **Примеры** |
| 1 | Вещественные или комплексные значения двойной точности (**double**) | type(42)  type(%nan)  type(%inf)  type(1 + %i) |
| 2 | П[олиномиальный](https://help.scilab.org/docs/6.0.1/ru_RU/poly.html) (**polynomial**) | type(1 - %z + %z^2) |
| 4 | Л[огический](https://help.scilab.org/docs/6.0.1/ru_RU/matrices.html)  (**boolean**) | type(%t) |
| 8 | Ц[елочисленный](https://help.scilab.org/docs/6.0.1/ru_RU/int8.html) (**integer**):  хранятся в 1 (**int**8), 2 (**int16**), 4 (**int32**) или 8 (**int64**) байтах | g = [int8](https://help.scilab.org/docs/6.0.1/ru_RU/int8.html)([1 - 120127312])  type(g)  type(1.23 \* [int8](https://help.scilab.org/docs/6.0.1/ru_RU/int8.html)(4)) |
| 9 | [Графические дескрипторы](https://help.scilab.org/docs/6.0.1/ru_RU/graphics_entities.html)  (указатели) | type([gdf](https://help.scilab.org/docs/6.0.1/ru_RU/gdf.html)()) |
| 10 | [Символьный](https://help.scilab.org/docs/6.0.1/ru_RU/strings.html)(**string**) | type("Текст")  type('Текст') |
| 13 | Компилированные  функции (**function**) | [deff](https://help.scilab.org/docs/6.0.1/ru_RU/deff.html)('[y] = f(x)',['a = 3 \* x + 1']);  type(f) |
| 14 | [Библиотеки](https://help.scilab.org/docs/6.0.1/ru_RU/library.html) функций (**library**) |  |
| 15 | Простые списки ([**list**](https://help.scilab.org/docs/6.0.1/ru_RU/list.html)) | l = [list](https://help.scilab.org/docs/6.0.1/ru_RU/list.html)(1,["a" "b"]);  type(l) |
| 16 | Типизированные списки ([**tlist**](https://help.scilab.org/docs/6.0.1/ru_RU/tlist.html)) | e = [tlist](https://help.scilab.org/docs/6.0.1/ru_RU/tlist.html)(["lt","f1","f2"], [], []);  type(e) |
| 17 | Матрично-ориентированные  типизированные списки ([**mlist**](https://help.scilab.org/docs/6.0.1/ru_RU/mlist.html))  (**Структуры,Ячейки, Полиномы,**Рациональные дроби) | h=[mlist](https://help.scilab.org/docs/6.0.1/ru_RU/mlist.html)(['V','n','v'],['a','b';'c''d'],[12;34]);  type(h)  [clear](https://help.scilab.org/docs/6.0.1/ru_RU/clear.html)s,s.r=%pi  type(s) // структуры  c={%t% pi% i%z"abc "s  type(c)} //cell-массивы  r=[%z/(1 - %z)(1 - %z) / %z^2]  type(r) // рациональныедроби |
| 130 | Встроенные функции (**fptr**) | type([disp](https://help.scilab.org/docs/6.0.1/ru_RU/disp.html)) |

*Функции для работы с комплексными данными*. Таблица 1.2.2-2

|  |  |  |
| --- | --- | --- |
| **Функции** | **Назначение** | **Примеры** |
| **complex(a, b)** | ***Создает комплексное***  ***число*** | -->complex(5, 9)  ans =  5. + 9.i |
| **сonj(a, b)** | ***Создает комплексно-сопряжённое число*** | -->b = complex(5, 9)  b =  5. + 9.i  -->conj(b)  ans =  5. - 9.i |
| **imag(Z)** | ***Выделяет мнимую часть числа*** | -->imag(complex(5,9))  ans =  9. |
| **real(Z)** | ***Выделяет вещественную часть числа*** | -->real(complex(5,9))  ans =  5. |
| **gcd(V)**  **lcm(V)** | ***Вычисляет наибольший общий делитель и наименьшее общее кратное*** | --> V = uint16([2^2 \* 3^5,…  > 2^3 \* 3^2, 2^2 \* 3^4 \* 5])  V =  972 72 1620  -->y = gcd(V)  y =  36  --> lcm(V)  ans =  9720 |
| **atan(imag(Z),real(Z))**  **atan(imag(Z),real(Z))…**  **\*180/%pi** | ***Возвращает фазу угла в радианах***  ***Возвращает фазу угла в градусах*** | -->argZ = atan(imag(1 + %i),…  > real(1 + %i)) \* 180 / %pi  argZ =  45. |
| **abs(Z)** | ***Вычисляет модуль  комплексного числа.*** | -->Z = 1 + %i;  -->modZ = abs(Z)  modZ =  1.4142136 |
| **isreal(Z)**  **isreal(a)**  **isreal(b)** | ***Возвращает логическое значение* T*, если число действительное и* F *– если комплексное*** | -->Z = complex(5, 9);  -->a = 67.76;  -->isreal(b)  ans =  F  -->isreal(Z)  ans =  F |

*Системные константы*. Таблица 1.2.2-3

|  |  |  |
| --- | --- | --- |
| **Системная**  **константа** | **Назначение** | **Значения системных  констант** |
| **%i** | Мнимая единица | sqrt(-1) |
| **%pi** | Число π | 3.1415926… |
| **%eps** | Погрешность числа с плавающей точкой | 2-52 |
| **%e** | Основание натурального логарифма | 2.71828182 |
| **%inf** | Значение машинной бесконечности |  |
| **%nan** | Указание на нечисловой характер данных **(Not-a-Number)** |  |
| **%s**  **%z** | Переменные, используемые для определения полиномов | --> z = poly(0, **"z");**  --> s = poly(0, **"s");** |
| **ans** | Переменная, хранящая результат последней операции |  |

*Функции, определяющие структуру матрицы.* Таблица 1.2.2-4

|  |  |  |
| --- | --- | --- |
| **Функция** | **Назначение** | **Пример использования** |
| **length(М)** | ***Возвращает число элементов в матрице*** | -->М = [1 2 3;2 3 4;4 5 6];  --> length(M)  ans =  9.  --> V = [3 4 5 6 7];  --> length(V)  ans =  5. |
| **length(Y(:, 1))** | ***Возвращает число строк  матрицы*** | --> X = [1 2;2 3;4 5];  -->length(X(:, 1))  ans =  3. |
| **length(X(1, :))** | ***Возвращает число столбцов матрицы*** | --> X = [1 2;2 3;4 5];  -->length(М(1, :))  ans =  2. |
| **size(М)**  **[n, m] = size(М)**  **size(M, 1)**  **size(M, 2)** | ***Возвращает вектор, содержащий количество строк и столбцов матрицы М, или только число строк, или только число столбцов*** | --> M = [1 3; 2 4; 4 6];  --> size(М)  ans =   1. 3.   --> [n, m] = size(M)  m =  3.  n =  2.  --> size(M, 1)  ans =  2.  --> size(M, 2)  ans =  3. |
| **ndims(T)**  **n = ndims(T)** | ***Возвращает число измерений матрицы*** | --> T = [2 3 4; 4 3 2; 5 7 8];  --> ndims(T)  ans =  2. |

*Алгебраические матричные операции и функции.* Таблица 1.2.2-5

|  |  |  |
| --- | --- | --- |
| **Операции**  **и функции** | **Назначение** | **Описания** |
| **+** | ***Сложение*** | **A + B** складывает матрицы **A** и **B**. |
| **+** | ***Унарный плюс*** | **+A** возвращает **A**. |
| **-** | ***Вычитание*** | **A - B** вычитает **B** из **A.** |
| **-** | ***Унарный минус*** | **-A**  меняет знак**A.** |
| **\*** | ***Матричное  умножение*** | **C = A \* B** – алгебраическое произведение матриц **A** и **B**, при условии, что количество столбцов  **A** равно числу строк **B**. |
| **^** | ***Матричное  возведение в степень*** | **A^B** – возведение матрицы **A** в степень **B**, если **B** является скаляром. Для других значений **B** вычисления включают собственные значения и собственные вектора. |
| **/** | ***Деление матриц слева направо*** | **X = B / A** – решение уравнения **X \* A = B**, при условии, что матрицы **A** и **B** имеют одинаковое количество столбцов. С точки зрения операций деления слева и транспонирования  **B / A = (A' \ B')'.** |
| **\** | ***Обратное (справа налево) деление матриц*** | **Х = A \ B** – решение уравнения **A \* X = B**, при условии, что матрицы **A** и **B** имеют одинаковое количество строк. |
| **'** | ***Транспонирование***  ***матрицы*** | **B = A'** – комплексно-сопряженное транспонирования матрицы **A**. Для комплексных матриц эта операция не предполагает сопряжения. |
| **d = det(mA)** | ***Вычисление  определителя  матрицы*** | --> A = [3 2; 4 3];  --> det(A) //***Определитель матрицы***  ans = 1 |
| **t = trace(A)** | ***Вычисление следа матрицы, то есть суммы элементов главной диагонали*** | --> A = [1 2 3; 4 -2 1; 0 3 -1]  --> trace(A) //***СледматрицыА***  ans =  -2  --> // ***то же что и***  --> sum(diag(A))  ans =  -2 |

*Арифметические поэлементные операции над матрицами.* Таблица 1.2.2-6

|  |  |  |
| --- | --- | --- |
| **Операция** | **Назначение** | **Описание** |
| **+** | ***Сложение*** | **A + B** поэлементное сложение **A** и **B** |
| **+** | ***Унарный плюс*** | **+A** возвращает **A** |
| **-** | ***Вычитание*** | **A - B** поэлементное вычитание **B** из **A** |
| **-** | ***Унарный минус*** | **-A** поэлементное присвоение в **A** |
| **.\*** | ***Поэлементное***  ***умножение*** | **C = A .\* B**  поэлементное умножение  **A** и **B** |
| **.^** | ***Поэлементное***  ***возведение в степень*** | **A .^ B**  поэлементное возведение **A** в степень **B** |
| **.\** | ***Поэлементное обратное деление массивов*** | **X = A .\ B** – поэлементное обратное деление **A** и **B** |
| **./** | ***Поэлементное деление*** | **X = B ./ A** поэлементное деление **A** и **B**. |
| **.'** | ***Транспонирование массива*** | **A .'**– поэлементная операция транспонирования **A** |

*Наиболее часто используемые математические функции*. Таблица 1.2.2-7

|  |  |
| --- | --- |
| ***Тригонометрические функции (аргумент задается в радианах)*** | |
| **sin, cos, tan, cot** | Синус, косинус, тангенс и котангенс |
| **sec, csc** | Секанс, косеканс |
|  | |
| **asin,acos,atan, atan2, acot** | Арксинус, арккосинус, арктангенс и арккотангенс |
| **asec, acsc** | Арксеканс, арккосеканс |
| ***Алгебраические и арифметические функции*** | |
| **abs** | Модуль |
| **exp** | Экспоненциальная функция |
| **log, log2, log10** | Логарифм натуральный, по основанию **2** и **10** |
| **sqrt** | Квадратный корень |
| **fix** | Целая часть числа |
| **floor** | Округление до ближайшего целого значения, которое не превышает аргумент |
| **mod(x, y), rem(x, y)** | Остаток от деления x на y. Целая часть определяется соответственно функциями **floor** и **fix** |
| **sign** | Знак числа |
| **factorial** | Вычисление факториала числа |
| ***Гиперболические функции*** | |
| **sinh, cosh, tanh, coth** | Гиперболические синус, косинус, тангенс и котангенс |
| **sech, csch** | Гиперболические секанс и косеканс |
| **asinh, acosh, atanh, acoth** | Гиперболические арксинус,  арккосинус, арктангенс и арккотангенс; |

*Преобразование данных числового тип*а. Таблица 1.2.2-8

|  |  |  |
| --- | --- | --- |
| Ф**Функции** | **Назначение** | **Примеры** |
| [**ceil**](https://help.scilab.org/docs/6.0.1/en_US/ceil.html)**(A)** | ***Возвращает матрицу целых чисел, состоящую из элементов, округлённых в сторону +∞*** | -->ceil([1.3 1.5 1.7 2.5 3.7])  ans =  2. 2. 2. 3. 4.  -->ceil([-1.3 -1.5 -1.7 -2.5 -3.7])  ans =  -1. -1. -1. -2. -3. |
| [**fix**](https://help.scilab.org/docs/6.0.1/en_US/fix.html)**(A)** | ***Возвращает матрицу целых чисел, состоящую из элементов, округлённых в сторону нуля***  y= sign(x) .\* floor(abs(x)) (то же самое, что и int). | -->fix([1.3 1.5 1.7 2.5 3.7])  ans =  1. 1. 1. 2. 3.  -->fix([-1.3 -1.5 -1.7 -2.5 -3.7])  ans =  -1. -1. -1. -2. -3. |
| [**floor**](https://help.scilab.org/docs/6.0.1/en_US/floor.html) | ***Возвращает матрицу целых чисел, состоящую из элементов, округлённых в сторону -∞*** | -->floor([1.3 1.5 1.7 2.5 3.7])  ans =  1. 1. 1. 2. 3.  -->floor([-1.3 -1.5 -1.7 -2.5 -3.7])  ans =  -2. -2. -2. -3. -4. |
| [**int**](https://help.scilab.org/docs/6.0.1/en_US/int.html)**(A)** | ***Возвращает матрицу целых чисел, состоящую из элементов, округлённых в сторону нуля***  То же самое, что и fix. | --> int([1.3 1.5 1.7 2.5 3.7])  ans =   1. 1. 1. 2. 3.   --> int([-1.3 -1.5 -1.7 -2.5 -3.7])  ans =  -1. -1. -1. -2. -3. |
| [**round**](https://help.scilab.org/docs/6.0.1/en_US/round.html) | ***Округляет до ближайшего целого по правилам математики*** | -->round([1.3 1.5 1.7 2.5 3.7])  ans =  1. 2. 2. 3. 4.  -->round([-1.3 -1.5 -1.7 -2.5 -3.7])  ans =  -1. -2. -2. -3. -4. |
| **ieee()** | ***Устанавливает режим предупреждающий о исключительных ситуациях при операциях с плавающей точкой*** | --> ieee(1); 1/0  Предупреждение : деление на нуль...  ans =  Inf  -->ieee(2); 1/0, log(0)  ans =  Inf  ans =  -Inf |
| [**isinf**](https://help.scilab.org/docs/6.0.1/en_US/isinf.html)**()** | ***Проверяется значения на бесконечность* inf** | --> realmax + .0001E+308  ans =  Inf  --> -realmax - .0001E+308  ans =  -Inf |
| [**isnan**](https://help.scilab.org/docs/6.0.1/en_US/isnan.html)**()** | ***Проверяется значения  на* nan** | x = [1 2 %nan 3 %nan 4]  k = [find](https://help.scilab.org/docs/6.0.1/ru_RU/find.html)(~isnan(x))  y = x(k) |

*Операции отношений.* Таблица 1.2.3-1

|  |  |  |
| --- | --- | --- |
| **Операция** | **Описание** | **Примеры \*** |
| **==** | Равно | --> x == y  ans =  T T F  --> a == b  ans =  F |
| **~=** | Не равно | --> x ~= y  ans =  F F T  --> a ~= b  ans =  T |
| **<** | Меньше чем | --> x < y  ans =  F F T |
| **>** | Больше чем | -->x > y  ans =  FFF |
| **<=** | Меньше или равно | --> x <= y  ans =  T TT |
| **>=** | Больше или равно | -->x >= y  ans =  TTF |

\* Примеры в этой таблице предполагают, что

**x = [2, 3, 4]**, **y = [2, 3, 5]**, **a = 3 +2 \* i**,  **b = 3 + 4 \* i**.

*Логические операции*. Таблица 1.2.3-2

|  |  |  |
| --- | --- | --- |
| **Операция** | **Описание** | **Примеры \*** |
| **b = or(A)** | ***Логическое сложение (операция ИЛИ) над элементами вектора или матр*ицы**  Где **b** - логический скаляр, если **or(A)** используется без каких-либо вариантов **"r"**, **"c"**, **n** (по умолчанию).  **b**равно **%F**, если все элементы **A** имеют значение **%F** или ноль, включая **%nan**.  **b** равно **%Т**, в противном случае (по крайней мере один элемент **A** имеют значение **%T** или ненулевой. | --> or([])  ans =  F  --> or(0)  ans =  F  --> or(0+%i \* 0)  ans =  F  --> or(%nan)  ans =  F  --> or([T, F T])  ans =  T  --> or([1 0 1; 0 4 3])  ans =  T |
| **B = or(A, 'r')**  **B = or(A, 1)**  **B = or(A, 'c')**  **B = or(A, 2)** | ***Логическое сложение (операция ИЛИ) над элементами столбцов или строк матрицы***  Где **B** - логический вектор.  Если **n = 1** или| **"r"**: **or** – возвращает вектор-строку логических значений  **B(j) = or(A(:, j)).**  Если **n = 2** или **" c"**: **or** возвращает вектор-столбец логических значений  **B(i) = or(A(i, :)**) | -->**B = or(A, 1)**  B =  T T F  -->**B = or(A, 2)**  B =  T  T  T |
| **L = A | B** | ***Поэлементное логическое сложение (операция ИЛИ) над элементами векторов или матриц***  Где **A** и **B** вектора или матрицы с элементами целого или логического типа, которые должны иметь одинаковые размеры.   Если **A** или **B** является скаляром, она заранее расширяется до размера другого операнда.  **L** – вектор или матрица логических значений | --> x = [1 0 0];  --> y = [1 1 1];  --> L = x | y  L =  T T T  --> y = [1 0 1];  --> L = x | y  L =  T F T |
| **b = and(A)** | ***Логическое умножение (операция И) над элементами вектора или мат*рицы**  Где **b** - логический скаляр, если  **and(A)** используется без каких-либо вариантов **"r"**, **"c"**, **n** (по умолчанию).  **B** равно **%F**, если хотя бы один из элементов **A** имеют значение **%F** или ноль.  **B** равно **%Т**, в противном случае (все элементы **A** имеют значение **%T**, ненулевое или **%nan**.  Где **A** вектор или матрица булевых,   целочисленных действительных или комплексных значений | -->and([])  ans =  T  -->and(0)  ans =  F  --> and(0 + 0\*%i)  ans =  F  -->and(%nan)  ans =  T  --> and([1, 0 1])  ans =  F  --> and([1 0 1; 0 4 3])  ans =  F |
| **B = and (A, 'r')**  **B = and (A, 1)**  **B = and (A, 'c')**  **B = and (A, 2)** | ***Логическое умножение (операция И) над элементами столбцов или строк  матрицы***  Где **B** – вектор логических значений.  **and**  возвращает вектор-строку логических значений  **b(j) = and(A(:, j));**  **and** возвращает вектор-столбец логических значений  **b(i) = and(A(i, :))** ; | -->**B = and (A, 1)**  B =  T F F  -->**B = and (A, 2)**  B =  T  F  F |
| **L = A & B** | ***Логическое умножение (операция И) над векторами или матрицами***  Где **A** и **B** вектора или матрицы с элементами целого или логического типа, которые должны иметь одинаковые размеры.   Если **A** или **B** является скаляром, она заранее расширяется до размера другого операнда.  **L** – вектор или матрица логических значений | --> x = [1 0 0];  --> y = [1 1 1];  --> L = x & y  L =  T F F  --> y = [1 0 1];  --> L = x & y  L =  T F F |
| **~A** | ***Логическое отрицание (операция НЕ) над элементами матрицы логических значений*** | --> x = [1 0 0];  --> ~x  ans =  T F F  --> y = [1 1 1];  --> oy = ~y  oy =  T F F |

\*В примерах используются:

вектора **x = [1 0 0] и y = [1 1 1];**

матрицы **А = [5 7 0; 3 2 4; 5 0 0]** и **B = [6 6 0; 1 3 5; -1 0 0].**

*Символы преобразования и примеры их применения*. Таблица 1.2.4-1

|  |  |
| --- | --- |
| **Символ** | **Описание** |
| **%c** | Выводит одиночный символ |
| **%d** | Выводит десятичное число |
| **%e** | Выводит десятичные числа с плавающей точкой в экспоненциальном виде |
| **%E** | Так же, как **%e**, но, используя прописные |
| **%f** | Выводит десятичные числа с плавающей запятой |
| **%g** | Более компактное **%e** или **%f**, незначительные нули не выводятся. |
| **%G** | Выводит десятичного целого числа (представление числа с основанием **10**) |
| **%i** | Выводит целую часть десятичного числа (представление числа с основанием **10**) |
| **%s** | Выводит строку символов |
| **%u** | Выводит целую часть десятичного числа без знака (представление числа с основанием **10**) |

*Ширина и точность полей*. Таблица 1.2.4-2

|  |  |  |
| --- | --- | --- |
| **Характер** | **Описание** | **Пример** |
| Ширина поля | Строка цифр, указывающая минимальное количество цифр для печати | ('%5d', 0) 0  ('%5d', 7) 7  ('%5d', -9) -9  ('%5d', 45622) 45622  ('%5d', 4562237) 45622 |
| Точность | Строка цифр, включающая точку (.) указание количества цифр, которые должны быть напечатаны справа от десятичной точки | ('%.0f', e) 3  ('%.0f.', e) 3  ('%.1f', e) 2.7  ('%.2f', e) 2.72  ('%.5f', e) 2.71828  ('%f', e) 2.718282 |

*Флаги форматирования*. Таблица 1.2.4-3

|  |  |  |
| --- | --- | --- |
| **Характер** | **Описание** | **Пример** |
| Знак минус (**-**) | Левое выравнивание | ('%-5.1f', e) 2.7\_\_ |
| Знак плюс (**+**) | Всегда печатает знак (+или -) | ('%+5.1f', e) \_+2.7 |
| Ноль (**0**) | Отображаются нул, а не пробелы | ('%05.1f', e) 002.7 |
| Знак плюс (**+**) или (**-**) |  | ('%+-5.1f', e) \_+2.7\_ |
| Знак плюс (**+**) и ноль (**0**) |  | ('% 05.1f', e) +02.7\_ |

Таблица 1.2.4-4

| **Символ** | **Влияние на текст** |
| --- | --- |
| **''** | Одиночная кавычка |
| **%%** | Одиночный знак процента |
| **\\** | Однократная обратная косая черта |
| **\a** | Аварийная сигнализация |
| **\b** | Возврат на одну позицию |
| **\n** | Новая строка |
| **\r** | Возврат каретки |
| **\t** | Горизонтальная табуляция |
| **\v** | Вертикальная табуляция |
| **\xN** | Шестнадцатеричное число, N |
| **\N** | Восьмеричное число, N |

Символы, специальные операции и примеры их использованияТаблица 1.2.2-1

| **Сим**  **вол** | **Название символа** | **Роль** | **Описание** | **Примеры** |
| --- | --- | --- | --- | --- |
| **.** | **Точка** | * Десятичная точка. * Операции. * Доступ к элементам структуры. | Точка разделяет целую и дробную части числа, используется в поэлементных операциях, позволяет получать доступ к полям в структуре. | Десятичная точка:  **102.5543**  Поэлементные операции:  **A.\* B**  Доступ к полям структуры:  **Struct.f1** |
| **...** | **Многоточие** | Продолжение строки. | Три или более точки в конце строки продолжают текущую команду на следующей строке. | Продолжает ввод выражения на следующей строке:  --> f=5\*x^7+%e\*(35\*sin(x-2)+...  > 7+x-a(5)/8); |
| **,** |  | Разделитель. | Запятые разделяют элементы строки в массиве, индексы массива, аргументы функций и команд. | Отделяет элементы строки для создания массива:  **mA = [12,13; 14,15]**  Отделяет индексы:  **mА(1,2)**  Отделяет входные и выходные параметры:  **[Y, I] = max (A, [], 2)**  Отделение нескольких команд в одной строке:  **mА(1, 2), [Y, I] = max(A, [], 2), А.^2** |
| **:** | **Двоеточие** | * Создание * вектора. * Индексирование. * Итерация для цикла. | Оператор двоеточия позволяет создавать регулярные интервалы векторов, производить индексирования в массивах и определять границы цикла**for**. | Создает вектор:  **x = 1:10, x = 1: 3: 19**  Изменяет матрицу на вектор столбец:  А (:)  Присваивает новые значения массиву:  **A = rand(3,4);A(:) = 1:12;**  Определяет диапазон индексов:  **А(:), А (:, 3)**  Определяет границы цикла**for**:  **x = 1;**  **for k = 1:25**  **x = x + x.^ 2;**  **end** |
| **;** | **Точка с запятой** | * Разделитель. * Знак конца строки. * Подавление вывода строки. | Точка с запятой разделяетописание строкипри созданиимассива, подавляет отображения строки кода и вывод результата. | Отделяет строки при создании массива:  **A = [12,13; 14,15]**  Подавляет вывод:  **Y = max(A);**  Разделяет несколько команд на одной строке:  **А = 12.5; B = 42.7, C = 1.25;**  **B =**  **42.7000** |
| **()** | **Круглые скобки** | * Последовательность операций. * Заключает списокпараметров функций и индексы. | Круглые скобки определяют последовательность операций в выражениях, внутри скобок перечисляются параметры функций и индексы в массиве. | Определяет последовательность операций:  **(A.\* (B./C)) - D**  Заключают списокпараметров функций:  **fun(X, Y ,Z)**  Заключают список индексов массивов:  **А (3, :), А (1,2), А (1: 5,1)** |
| **[]** | **Квадратные скобки** | * Конструктор массива. * Удаление пустой матрицы и ее элемента. * Определение вектора выходных параметров. | Квадратные скобки осуществляют построение и конкатенацию массивов, создание пустых матриц, удаление элементов массивов и определяют векторвыходных параметров функций. | Конструирует вектора:  **X = [10 12 -3]**  Создает пустую матрицу:  **A = []**  Удаляет столбец матрицы:  **A (:, 1) = []**  Определяет выходные параметры функций:  **[C, iA, iB] = uni5(A, B)** |
| **{}** | **Фигурные скобки** | Конструктор массива ячеек. | Фигурныескобки осуществляют построение массива ячеек или определяют доступ к ним. | Конструирует массив ячеек:  C = {[2.6 4.7 3.9], rand(8) \* 6, 'Мама '}  Реализует индексациюв массиве ячеек:  A = C {4,7,2} |
| **//** | **Слеш** | Комментарии | // Определяет комментарии в конце строки или в целой строке. | Добавляет в программный код комментарий:  **//Цель этого цикла - вычислить** |
| **/\***  **\*/** | **Слеш**  **звездочка** | Блок  комментариев | /\* Комментарии \*/ определяет блок комментариев, которые располагаются на нескольких строках | Добавляет в программный код блок комментариев:  **/\***  **Комментарий ...**  **\*/** |
| **%** | **Процент** | * Указатель системных констант | Знак процента используется для указания на системную константу. | Указатель на системную константу:  **%e, %pi, %i** |
| **$** | **Доллар** | * Команда операционной системы | Последний индекс элемента массива. | Указатель последнего индекса массива  **A(1:2:$)** |
| **%{ %}** | **Процентная фигурная скобка** | Блок  комментариев | %{Комментарии %} определяет блок комментариев , которые располагаются на несколькихстроках программного кода | Добавляет многострочные комментарий:  **% {**  **Комментарий ...**  **%}** |
| **'** | **Одинарные**  **кавычки** | Конструктор  строк. | Используется для создания символьной переменной. | Создает символьную переменную:  **chr = 'Привет'** |
| **"** | **Двойные кавычки** | Конструктор  строк | Используется для создания строковых скаляров типа **string**. | Создает символьную переменную:  S = "Привет, Ректору" |
| **~** | **Тильда** | * Логическое НЕ. * Заполнитель аргументов | Используется для представления логического отрицания или для подавления конкретных входных или выходных параметров. | Определяет неравенство:  **A = [1 -1; 0 1]; B = [1 -2; 3 2];**  **А ~ = В**  Возвращает только третье выходное значение:  **[~, ~, iB] = Fun3 (A, B)** |
| **=** | **Знак равенства** | Присваивание. | Используется для присваивания значений переменной. | Создает матрицу A и B, определяет b[ тип:  **A = [1 0; -1 0];**  **B = A;** |

Приложение 1.3

*Функции, используемые для создания специальных матриц.* Таблица 1.3.2-1

| **Функции** | **Описания** | **Примеры** |
| --- | --- | --- |
| **A = ones()**  **A = ones(n,n)**  **A = ones(n,m)**  **A = ones(B)** | ***Создание матриц со  значениями* 1**  Возвращается скаляр со значениями **1**.  Возвращается матрица**n×n** со значениями элементов матрицы равными**1**.  Возвращается матрица n**×**m со значениями ми матрицы равные 1.  Возвращается матрица со значениями элементов матрицы равными**1** и характеристиками, аналогичными характеристиками матрицы **B**. | --> A = ones()  A =  1.  -->A = ones(2,3)  A =  1. 1. 1.  1. 1. 1.  --> B = [2 3; 4 5];  -->A = ones(B)  A =  1. 1.  1. 1. |
| **A = zeros()**  **A = zeros(n,n)**  **A = zeros(n, m)**  **A = zeros(B)** | ***Создание матриц со  значениями*** *0*  Возвращается скаляр со значениями **0**.  Возвращается матрица**n×n** со значениями элементов матрицы равными **0**.  Возвращается матрица **n×m**со значениями элементов матрицы равными **0**.  Возвращается матрица со значениями элементов матрицы равными **0**и характеристиками, аналогичными характеристиками матрицы **B**. | --> A= zeros()  A =  0.  -->A = zeros(2,3)  A =  0. 0. 0.  0. 0. 0.  --> B = [2 3; 4 5];  -->A = zeros(B)  A =  0. 0.  0. 0. |
| **D = eye()**  **D = eye(n,n)**  **D = eye(n,m)**  **D = eye(V)** | ***Создание матрицы со значениями 1 на главной диагонали и 0 во всех других элементах***  Возвращается единичная матрица неопределенного размера со значениями **1**. Размеры будут определены, когда данная единичная матрица будет просуммирована с матрицей фиксированного размера.  Возвращается матрица **n×n**со значениями элементов на главной диагонали равные **1** и**0**в остальных элементах.  Возвращается матрица **n×m** со значениями элементов на главной диагонали равные **1**и**0**в остальных элементах.  Возвращается матрица со значениями элементов на главной диагонали равные **1**и**0**в остальных элементах. Размер матрицы определяется размером вектора **V**. | --> D1 = eye()  D1 =  -->eye \*  1.  --> D2 = eye(3,3)  D =  1. 0. 0.  0. 1. 0.  0. 0. 1.  --> D3 = eye(2,3)  D3 =  1. 0. 0.  0. 1. 0.  --> V = [1252 14];  --> D4 = eye(V)  D4 =  1. 0. 0.  0. 1. 0.  0. 0. 1. |
| **A = diag(V)**  **A = diag(V, k)**  **V = diag(A)**  **V= diag(A, k)** | ***Создание диагонали квадратной матрицы из заданного вектора; создание диагональной матрицы и вектора из диагонали квадратной матрицы***  Возвращается матрица**n×n**со значениями элементов вектора **V**на главной диагонали.  Помещаются значения элементов вектора **V**на **k**-ю диагональ, где k**=0**представляет собой главную диагональ, **k>0**представляет диагональ, которая находится выше главной диагонали и **k<0**представляет диагональ, которая находится ниже главной диагонали.  Возвращается вектор-столбец главных диагональных элементов матрицы A.  Возвращает вектор-столбец элементов на k-й диагонали A.  **V** – вектор или матрица, **k** – целое число (значение по умолчанию равно **0**), **A**– вектор или матрица. | --> V = [13 62 94];  --> A = zeros (3,3);  --> A = diag(V)  A =  13. 0. 0.  0. 62. 0.  0. 0. 94.  --> A = diag(V,-1)  A =  0. 0. 0. 0.  13. 0. 0. 0.  0. 62. 0. 0.  0. 0. 94. 0.  --> V= diag(A,-1)  V =  13.  62.  94. |
| **A =** [**rand**](https://www.mathworks.com/help/matlab/ref/rand.html)**(n,m)**  **A =** [**rand**](https://www.mathworks.com/help/matlab/ref/rand.html)**(B)** | ***Создание матрицы с равномерно распределенными случайными  числами***  Возвращается матрицу с равномерно распределенными случайными числами в диапазоне **[0; 1]**.  Возвращается матрицу с равномерно распределенными случайными числами в диапазоне **[0; 1]** того же размера, что и матрица **В**. | --> B=[2 4; 3 5];  --> A=rand(3, 2)  A =  0.84155 0.87841  0.4062 0.11384  0.40948 0.19983  --> A=rand(B)  A =  0.56187 0.6854  0.58962 0.89062 |
| [**rand**](https://www.mathworks.com/help/matlab/ref/rand.html)**(n,m, 'normal')**  [**rand**](https://www.mathworks.com/help/matlab/ref/rand.html)**n(A, 'normal')** | ***Создание матрицы c нормально***  ***распределенными случайными  числами в диапазоне***  Возвращается матрицу вещественных или комплексных случайных чисел с математическим ожиданием**0**, дисперсией **1** и размера **[n; m]**.  Возвращается матрицу вещественных или комплексных случайных чисел с математическим ожиданием **0**, дисперсией **1** того же размера, что **A** | -->B=rand(3,2,'normal')  B=  -1.7211 0.18423  -0.0047 0.1023  -1.71576 -1.03329  --> A=[2 3;4 6];  --> B=rand(A,'normal')  B =  -1.28586 0.61078  0.59712 -1.05679 |
| **grand**  **D = repmat(A, m, n)** | ***Создание матрицы повторяющимися значениями***  Возвращается матрица размера **m×n**, состоящая из копий матрицы **A**, заданной вектором или матрицей. | --> D = repmat(1:2, 2, 2)  D =  1. 2. 1. 2.  1. 2. 1. 2.  --> A=[2 3;3 4];  --> D = repmat(A, 2, 2)  D =  2. 3. 2. 3.  3. 4. 3. 4.  2. 3. 2. 3.  3. 4. 3. 4. |
| **C=cat(dim,A,B,…)** | ***Сцепление матриц в указанном  измерении***  Возвращается матрица, в которой происходит сцепление матриц, перечисленных в параметрах. Если **dim = 1**, то сцепление происходит по строкам входных параметров, а если **2** – по столбцам. | -->В=[1 2 3 4;5 6 7 8];  --> C=cat(1, A, B)  C =  1. 2. 3. 4.  5. 6. 7. 8.  1. 2. 3. 4.  5. 6. 7. 8.  --> C=cat(2, A, B)  C =  1. 2. 3. 4. 1. 2.3. 4.  5. 6. 7. 8.5. 6. 7. 8. |
| **V=linspace(х1,х2,n)**  **М=linspace(c1,c2,n)** | ***Генерируется заданное  количество чисел между 2-мя границами***  Возвращается вектор-строка из **n** значений, равномерно распределённых точно в заданном диапазоне. | -->linspace(1,2,3)  ans =  1. 1.5 2.  -->linspace([1:2]',[3:4]',3)  ans =  1. 2. 3.  2. 3. 4. |
| **V=**[**logspace**](https://help.scilab.org/docs/6.0.1/ru_RU/logspace.html)**(d1,d2,n)**  **М=linspace(c1,c2,n)** | ***Возвращается вектор-строку с  интервалами между элементами в логарифмическом масштабе***  Возвращается вектор-строка из **n** значений, равномерно распределённых в логарифмическом масштабе точно между **10d1** и **10d2**. Если **d2 = %pi**, то точки располагаются между **10d1** и **%pi**. | -->logspace(1,2,3)  ans =  10. 31.62277 100.  -->logspace([1:2]',[3:4]',3)  ans=  10. 100. 1000.  100. 1000. 10000. |
| **testmatrix('magi',n)**  **testmatrix('frk',n)**  **testmatrix('hilb',n)** | ***Создание специальных матриц***  Возвращается матрица магического квадрата размера **nхn**.  Возвращается матрица Франка.  Возвращается матрица Гильберта  (Hij=1/(i+j-1)). | --> M=testmatrix('magi',3)  M =  8. 1. 6.  3. 5. 7.  4. 9. 2. |

*Функции преобразования индексаций.* Таблица 1.3.3-1

|  |  |  |
| --- | --- | --- |
| Функция | Описание | **Примеры** |
| I = sub2ind(size(A), K, L) | ***Возвращается эквивалент  матричных индексов  (K и L) в линейную  индексацию***  Где **K** вектор номеров строк, а **L** вектор номеров столбцов. | --> A = [2 6; 4 8; 3 5];  -->l=sub2ind(size(A),3,2)  lin =  6. |
| [K, L] = ind2sub(size(A),I) | ***Возвращается эквивалент  линейных индексов в строки и столбцы  матричной индексации***  Где **I** линейные индексы, а **K** и **L** вектора соответствующих индексов. | --> A = [2 6; 4 8; 3 5];  -->[r c]=ind2sub(size(A),6)  c =  2.  r =  3. |

*Примеры применения операции двоеточия при адресации*. Таблица 1.3.3-2

|  |  |  |
| --- | --- | --- |
| Функция | Описания | Примеры |
| V(:)  V(n:m) | Адресация для вектора  Обращение ко всем элементам вектора.  Обращение к элементам вектора от n номера до m номера. | --> V = [3 6 7 8 1 5];  V(:) =  3 6 7 8 1 5  --> V(3:5) =  7 8 1 |
| mА(:, m)  mА(n, :)  mА(:, m1:m2)  mА(n1:n2, :)  mA(n1:n2, 1:m2) | Адресация для матрицы  Обращение ко всем элементам m столбца матрицы mA.  Обращение ко вcем элементам n столбца матрицы mA.  Обращение ко всем элементам столбцов от m1 до m2 матрицы mA.  Обращение ко всем элементам строк от n1 до n2 матрицы mA.  Обращение ко всем элементам строк от n1 до n2 и столбцов от m1 до m2 матриц mA. | --> mB = [1 2 3; 4 5 6];  --> mB(:, 2)  ans =  2.  5.  --> mB(1, 2 : 3)  ans =  2. 3.  --> mB(:, 1 : 2)  ans =  1. 2.  4. 5.  --> mB(1 : 2, 2 : 3)  ans =  2. 3.  5. 6. |
| $ | Адресация для вектора  Обращение ко всем элементам вектора.  Обращение к элементам вектора до последнего номера. | --> M = [2 6 9; 4 2 8; 3 51];  --> M(1 : $, :)  ans =  2. 6. 9.  4. 2. 8.  3. 5. 1. |

*Функции, вычисляющие параметры матриц.* Таблица 1.3.4-1.

|  |  |  |
| --- | --- | --- |
| Функции | Описания | Примеры |
| B = prod(A)  B = prod(A, d)  B = prod(А, t) | ***Нахождение произведения  элементов матрицы***  Произведение элементов матрицы по столбцам **(d=1)** или строкам столбцам **(d=2)**.  Произведение элементов вектора или матрицы с управлением способа умножения ('native'- целочисленное, или 'double' – с плавающей точкой). | --> V = [1 3 5];  --> prod(V)  ans =  15.  --> A = [2 6 9; 4 2 8];  --> prod(A)  ans =  6912.  -->prod(A, 1)  ans =  8. 12. 72.  -->prod(A, 'native')  ans =  6912. |
| B = sum(A)  B = sum(A, d)  B = sum(А, t) | ***Нахождение суммы элементов  матрицы***  Сумма элементов матрицы по столбцам (d=1) или строкам (d=2).  Сумма элементов вектора или матрицы с управлением способа суммирования('native'– целочисленное, или 'double' – с плавающей точкой) | --> V=[1 3 5];  --> sum(V)  ans =  9.  --> A = [2 6 9; 4 2 8];  -->sum(A)  ans =  31.  -->sum(A, 1)  ans =  6. 8. 17.  --> sum(sum(A))  ans =  31. |
| **a = min(A)**  **[a, k] = min(A)**  **[a, k] = min(A, 'c')**  **[a, k] = min(A, 'r')**  **[a, k] = min(A1, 2,…)** | ***Нахождение минимальных  элементов матрицы и их  индексов***  Для A, вещественного вектора или матрицы, min(A) является наименьшим элементом A. [a, k] = min(A) даёт дополнительно индекс минимума.  При использовании второго параметра: 'r' требуется для получения вектора-строки, где,а(j)содержал минимум j-того столбца A, k(j)даёт индекс строки, которая содержит минимум для столбца j; 'c' используется для аналогичной операции на строках A | --> A = [2 6 9;1 2 8];  --> a = min(A)  a =  1.  --> [a,k]=min(A)  k =  2. 1.  a =  1.  -->min([2,6,9; 1 2 8])  ans =  1. |
| **a = max(A)**  **[a, k] = max(A)**  **[a, k] = max(A, 'c')**  **[a, k] = max(A, 'r')**  **[a, k] = max(A, 'm')**  **[a, k] = max(A1,A2,…)** | ***Нахождение максимальных  элементов матрицы и их  индексов***  Для A, вещественного вектора или матрицы, max(A) является наибольшим элементом A. [a, k] = max(A) даёт дополнительно индекс максимума. Использование второго аргумента: 'r' требуется для получения вектор-строки а такого, что, а(j) содержит максимум j-того столбца A, k(j) даёт индекс строки, которая содержит максимум для столбца j; 'c' используется для аналогичной операции на строках A; 'm' используется для совместимости с Matlab. | --> A= [2 6 9; 1 2 8];  --> a = max(A)  a =  9.  --> [a, k] = max(A)  k =   1. 3.   a =  9.  -->max([2,6,9; 1,2 8])  ans =  9. |
| **Y = mean(A)**  **Y = mean(A, 'r')**  **Y = mean(A, 'c')**  **Y = mean(A, 'm')** | ***Нахождение средних значений элементов матрицы***  Возвращает среднее значение элементов вектора или матрицы (A). Использование второго параметра позволяет найти: 'r'- среднее значение по столбцам; 'c' - среднее значение по строкам;'m'используется для совместимости с Matlab. | --> V = [1 3 5];  --> y = mean(V)  y =  3.  --> A = [2 6 9; 4 2 8];  --> y = mean(A)  y =  5.1666667  --> y = mean(A, 'r')  y =   1. 4. 8.5   --> y = mean(A, 'c')  y =  5.6666667  4.6666667 |

*Примеры логического индексирования массивов.* Таблица 1.3.5-1

|  |  |
| --- | --- |
|  | Реализация |
| **Нулевые и ненулевые элементы  матрицы.**  Найти индексы ненулевых и нулевых элементов матриц **3x3**. | --> X = [1 0 2; 0 1 1; 0 0 4]  X =  1. 0. 2.  0. 1. 1.  0. 0. 4.  --> k1 = find(X)  K1 =  1. 5. 7. 8. 9.  --> k2 = find(~X)  k2 =  2. 3. 4. 6. |
| **Равенство конкретных значений  элементов.**  Чтобы найти конкретное целое значение, используй операцию**==** . К примеру, найти элемент равный **13** в **1×10** вектор.  Чтобы найти нецелые значения, используй значение погрешности, на основе данных. В противном случае из-за ошибки округления значений с плавающей запятой результатом может быть пустая строка. | -->x = 1: 2 : 20  x =  1. 3. 5. 7. 9. 11. 13. 15. 17. 19.  --> k1 = find(x==13)  k1 =  7.  --> y = 0 : 0.1 : 1  y =  0. 0.1 0.2 0.3 0.4 0.50.6 0.7 0.8 0.91.  -->k2 = find(y==0.3)  K2 =  []  -->k3 = find(abs(y - 0.3) < 0.0001)  K3 =  4. |
| **Элементы, удовлетворяющие  нескольким условиям.**  Найти первые три элемента в матрице**4×4**, большие **0** и меньшие, чем **10**. Укажите два выхода для возвращения строк и столбцов индексов к элементам.  Первый элементом является **X(2,1)=8**. | -->X = [18 3 1 11; 8 10 11 3;  > 9 14 6 1; 4 3 15 21]  X =  18. 3. 1. 11.  8. 10. 11. 3.  9. 14. 6. 1.  4. 3. 15. 21.  --> [row, col] = find(X > 0 & X < 10, 3)  col =  1. 1. 1.  row =  2. 3. 4. |
| **Значения для ненулевых элементов и их индексы.**  Найти ненулевые элементы в матрицу **3×3**и указать индексы строк, индексы столбцов. | --> X = [3 2 0; -5 0 7; 0 0 1]  X =  3. 2. 0.  -5. 0. 7.  0. 0. 1.  --> [row, col] = find(X)  col =  1. 1. 2. 3. 3.  row =  1. 2. 1. 2. 3. |
| **Значения вектораmAсоответствующие значениям логической матрицыmB**. | -->mA = [1 2 3; 4 5 6; 7 8 9]  mA =  1. 2. 3.  4. 5. 6.  7. 8. 9.  -->mB = [%F %T %F; %T %F %T; %F %F %T]  mB =  F T F  T F T  F F T  --> mC = mA(mB)  mC =  4.  2.  6.  9.  --> find(mB)  ans =  2. 4. 8. 9. |

*Описание функции* ***find*, gsort, vectorfind.** Таблица 1.3.5-2

|  |  |  |
| --- | --- | --- |
| **Функции** | **Описания** | **Примеры** |
| **[i] = find(х)**  **[i] = find(х, nmax)**  **[i1, i2] = find(х)** | ***Нахождение ненулевых  элементов в матрицах***  ***Нахождение истинных  значений логической  матрицы***  Возвращается вектор индексов **i,** для которых **x(i)** "истина".  Если нет истинного элемента,  **find** возвращает пустую матрицу.  **nmax**– целое число, задающее максимальное число возвращаемых индексов.  Значение по умолчанию **-1**, что означает "все"  Возвращается вектор индексов **i1** (строк) и **i2** (столбцов для которых **x(i1,i2,..)**  имеют значение "истина". Если нет истинного элемента, **find** возвращает пустую матрицу. |  |
| **i = vectorfind(m,v,"r")**  **i = vectorfind(m,v,"c")** | ***Поиск строки или столбцы  матрицы, совпадающие  с заданным вектором***  **m** - матрица любого типа (того же типа, что и **v**).  **v** - вектор любого типа (того же типа, что и **m**).  "**r**" для поиска совпадений строк или "**c**" для поиска совпадений столбцов. По умолчанию значение "**r**".  **i** - вектор-строка, содержащая индексы совпадений строк или столбцов |  |
| **gsort(A)**  **B=gsort(A)**  **[B,k]=gsort(A)**  **[B,k]=gsort(A,o)**  **[B, k] = gsort(A, o, n)** | ***Сортировка элементов  матрицы***  Упорядочивается элементы векторов или столбцовматрицы.Если **о='r'**сортируется каждый столбец**A**; если **о='c**': сортируется каждая строка **A**; **о ='g'**: сортируются все элементы**A**.  **n** задаёт направление сортировки:  **n ='i'**устанавливает порядок возрастания,а**n='d'**устанавливает порядок убывания (по умолчанию) | --> V = [1 3 5];  -->gsort(V)  ans =  5. 3. 1.  --> A=[2 6;4 2;3 5];  -->B=gsort(A,'c','i')  B =  2. 6.  2. 4.  3. 5. |

Приложение 1.4

*Функции для работы с графиками функций одной переменной.* Таблица 1.4.1-1

|  |  |
| --- | --- |
| **Функции** | **Назначение** |
| **scf(n)** | Созданиенового графического окна, где**n**– номер графического окна.  При первом обращении присваивается номер ноль (**0**). |
| **winsid()** | Функция, которая возвращает список открытых графических окон. |
| **plot(x, y)** | Построение графика функции **y = f(x).** |
| **plot(x1, y1, x2, y2,...)** | Построение графиков функций **y1=f1(x1),  y2=f2(x2)…** в одном окне. |
| **xtitle('title', 'xstr', 'ystr')** | Функция, добавления к графику заголовка (**title**) и подписи осей (**xstr, ystr**). |
| **xgrid(*Список параметров*)** | Функция позволяющая отобразить координатную сетку. |
| **legend(leg1,...,legn, pos)** | Функция вывода «легенды».  **leg1,leg2,...,legn –** названия графиков**,**  **pos –** необязательный параметр (по умолчанию равен **1):**  **pos = -1 -** в правом верхнем углу над областью графика; **pos = 0 -** место выбирается автоматически; **pos = 1 -** в правом верхнем углу;**pos = 2**в левом верхнем углу области графика;  **pos = 3** - в левом нижнем углу области графика; **pos = 4 -** в правом нижнем углу области графика. |
| **mtlb\_hold('on')** | Функция, позволяющая строятся графики в одном окне. |
| **plot2d(x,y)** | Функция, строящая график по форматам, ранее определенным пользователем. |
| **plot2d2(x,y)** | Функция, предназначенная для построения графика в виде ступенчатой функции. |
| **plot2d3(x,y)** | Функция, предназначенная для построения графика в виде вертикальных полосок. |
| **plot2d4(x,y)** | Функция, предназначенная для построения графика с указанием направления. |
| **сhamp(x, y, fx, fy)** | Воспроизведение изображения в двумерном пространстве в виде векторных полей: **x,y**– вектора, определяющие сетку координат;**fx**– матрица, описывающая**x**- компоненту каждого поля вектора;**fy**– матрица, описывающая**y**-компонентой в точке **(x(i),y(i)).** |
| **histplot(n, d)** | Функция для построения гистограммы на плоскости:**n**-количество отрезков;**d** – матрица значений функции. |

*Функции для работы с графиками функций двух переменных.* Таблица 1.4.1-2

|  |  |
| --- | --- |
| **Функции** | **Название** |
| **mesh(X,Y,Z)** | Построение сетчатого графика. |
| **contour(X,Y,Z)** | Построение графика контурных линий. |
| **surf(X,Y,Z)** | Построения графика сплошной поверхности. |
| **plot3d(X,Y,Z)** | Построение точек, соединенных отрезками прямых и других линий. |
| **subplot(n, m, k)** | Функция построения в одном графическом окне нескольких графиков: **n**– количество строк в окне;**m –** количество столбцов в окне;**k –**номер области построения текущего графика. |
| **polarplot(phi, ro)** | Построение графика в полярных координатах: **phi**– диапазон значений угла;**ro**– функция от полярного угла. |

*Глобальные свойства графических объектов.* Таблица 1.4.1-3

|  |  |
| --- | --- |
| **Свойство** | **Описание** |
| **CData**  **ColorData** | Это свойство определяет цвет каждой точки вещественной матрицей значений– **Z=f(x,y)**. Используется только при рисовании поверхностей. |
| **CDataMapping**  **ColorDataMapping** | Это свойство присваивает каждой вершине поверхности индекс цвета, определяемый **Z**. Способ раскраски поверхности определяется строковыми значениями, которые используются только в функции **fac3d.** |
| **Clipping**  **(Обрезка)** | Это свойство определяет способ обрезания графика. Возможные значения:   * **'on'** - объекты обрезаются за пределами области **clip\_box**; * **'off' –** объекты не обрезаются; * **'сlipgrf'** – объекты обрезаются за пределами **axes**. |
| **Color**  **(Цвет)**  **Foreground**  **(ПереднийПлан)** | Это свойство определяет цвет ломанных линий. |
| **EdgeColor**  **(Цвет кромки)**  **Foreground**  **(Передний План)** | Это свойство определяет цвет поверхности. |
| **FaceColor**  **(ЦветФронтальный)** | Это свойство определяет цвет фронтальной части поверхности. Возможные значения:   * **'none'**, нет, рисуется только каркас; * **'flat'**, плоская, одноцветные грани; * **'interp'**, интерполяция, затенение у вершин. |
| **LineStyle**  **(ВидЛинии)** | Это свойство определяется стиль линии может принимать следующие значения:  **-** сплошная линия (по умолчанию);  **--** штриховая линия;  **:** штрихпунктирная линия (две точки);  **-.** штрихпунктирная линия (одна точка);  **none** нет линии. |
| **Marker**  **MarkStyle**  **(ТипМаркера)** | Это свойство определяется тип маркера и может принимать следующие значения:  **+** знак "плюс"  **o** кружок  **\*** звёздочка  **.** точка  **x** крестик  **'square'** или **'s'** квадрат  **'diamond'** или **'d'** ромб  **^** треугольник, указывающий вверх  **v** треугольник, указывающий вниз  **>** треугольник, указывающий вправо  **<** треугольник, указывающий влево  **'pentagram'** или **'p'** пятиконечная звезда (пентаграмма)  **'none'** нет маркера (по умолчанию)  По умолчанию маркер не отображается. |
| **MarkerEdgeColor MarkerForeground** | Это свойство определяется цвет переднего плана (контур) маркера. |
| **MarkerFaceColor MarkerBackground** | Это свойство определяется цвет заднего плана (заливка) маркера. |
| **MarkerSize MarkSize** | Это свойство определяется размер маркера, которое может принимать целое число от **0** до **5**и которое соответствуют размеры **8pt, 10pt, 12pt, 14pt, 18pt, 24pt.** |
| **Visible** | Это свойство определяется видимость и принимает значения **'on'** (да), **'**off**'** (нет). По умолчанию **'on'**. |
| **X data** | Это свойство определяется вектор или матрица значений **Х**. |
| **Y data** | Это свойство определяется вектор или матрица значений Y. |
| **Z data** | Это свойство определяется матрица значений **Z**. |

Список свойств компонентов (Style). Таблица 1.4.3-1

|  |  |
| --- | --- |
| **Имя свойства** | **Описание свойства и допустимые принимаемые значения** |
| **BackgroundColor**  **(ЦветФона)** | Это свойство устанавливает цвет фона компонента, значения которого может быть вещественным вектором**1×3** или строкой. |
| **Borde**  **(Граница)** | Это свойство устанавливает границы для компонента **frame.** |
| **CallBack**  **(Обратный вызов)** | Это свойство осуществляет вызов функции при активации uicontrol (например, при нажатии на кнопку).  Это свойство неможет быть использовано для компонентов **layer, frameа** и **text**. |
| **Enable**  **(Включить)** | Это свойство включает или отключает **uicontrol**.  Если это свойство имеет значение **"on"**(по умолчанию), **uicontrol** работает, но если это свойство имеет значение **"off"**, **uicontrol** не будет реагировать на действия мыши и будет серым цветом**({on} | off).** |
| **FontAngle**  **(НаклокШрифта)** | Это свойство устанавливает наклон шрифта для компонентов, содержащих некоторый текст**({normal} | italic | oblique ).** |
| **FontSize**  **(ЕдИзмРазмераШрифта)** | Это свойство устанавливает для компонентов, содержащих некоторый текст, значения единицы измерения, в которых указан размер шрифта |
| **FontUnits**  **(РазмерШрифта)** | Это свойство устанавливает для компонентов, содержащих некоторый текст, размер шрифта **({points} | pixels | normalized).** |
| **FontWeight**  **(ВесШрифта)** | Это свойство устанавливает для компонентов, содержащих некоторый текст, вес используемого шрифта (**light / {normal} / demi / bold**). |
| **FontName**  **(НазваниеШрифта)** | Это свойство определяет имя шрифта, выбранного для отображения текстакомпонентов. |
| **ForegroundColor**  **(ЦветПереднегоПлана)** | Это свойство, значение которого является вещественный вектор**1×3**или строка, устанавливает цвет переднего плана **компонента**, элементы которого определяют Красный, Зеленый и Синий цвет и могут принимать значения в диапазоне**[0,1]**. Цвет может быть задан как вещественный вектор -**[R, G, B]** или строка –**"R|G|B"**.  Установка этого свойства в значение **[-1 -1 -1**]позволяет установить цвет переднего плана по умолчанию. |
| **Groupname**  **(ИмяГруппы)** | Это свойство устанавливает **ИмяГруппы и** используется для компонентов**radiobutton**и **checkbox**. Это свойство позволяет управлять одной группы в целом. |
| **HorizontalAlignment**  **(ГоризонтальноеВыр)** | Это свойство выравнивае текста по горизонтали  **(left | {center} | right).** |
| **Layout**  **(Макет)** | Это свойство задает макет, используемый для размещения **frame** дочерних компонентов. |
| **Layout\_options**  **(Параметр\_Макета)** | Это свойство задает параметры макета, используемого для размещения **frame** дочерних объектов. |
| **ListboxTop**  **(ПервыйЭлСписка)** | Это свойство , являющейся скалярная величина, определяет какой элемент списка отображается в первой строке видимой области списка.  Это свойство используется только для **listbox**. |
| **Max**  **(Макс)** | Это свойство указывает наибольшее значение свойства **Value.**  Оно имеет различный смысл для каждого компонента:   * **checkbox**и **radiobutton**: **Max**– это значение, которое принимает свойство **Value** при проверке элемента управления. * **slider** и **spinner**: максимальное значение компонента. * **listbox**: если **(Max-Min)>1** Список допускает множественный выбор. * **edit**: если **(Max-Min)>1**, то **edit** допускает отображение нескольких строк. |
| **Min**  **(Мин)** | Это свойство указывает наименьшее значение свойства **Value.** Он имеет тот же смысл что и **Max.** |
| **Parent**  **(Родитель)** | Это свойство является дескриптор родителя uicontrol. Изменение этого свойства позволяет  перемещаться с одной фигуры (компонента) на другую. |
| **Position**  **(Позиция)** | Это свойство используется для установки или получения геометрической конфигурация компонента и задается вещественным вектором **[x y w h]**, где  **x** расположение левого нижнего угла, **y** расположение левого нижнего угла, ширина и высота  или строкой – **"x|y|w|h"**. |
| **Scrollable (Прокручиваемый)** | Это свойство указывает, должен ли компонент иметь возможности прокрутки **(%T)** или нет (по умолчанию **%F)**и используется для компонентов **frame**и **edit**.  Для **frame** этого необходимо установить значение при создании. |
| **String**  **(Строка)** | Это свойство представляет текст, появляющийся в компонентах. Для таблиц значение этого свойства является строковой матрицей. Для списков и **PopupMenus**, значение может быть вектором строки или строкой, в которой находятся элементы разделено"|". Для текстовых компонентов эта строка может содержать HTML-код для форматирования текста.   * Для **pushbutton**или **text**, если текст заключен между двумя **$** (знак доллара), то он будет рассматривается как выражение **LaTeX**, и если оно заключено между < и >, оно будет считаться как **MathML.** * Для **Layer и Tab** значение указывает **тег** выбранного дочернего элемента. * Для **Image**значение указывает путь к файлу образа. * Для **Table** значение указывает все табличные данные**.** |
| **Title\_position**  **(ПоложениеВкладок)** | Это свойство устанавливает положение вкладок **tab({top} | left | bottom | right).** |
| **Title\_scroll** | Это свойства указывает, необходимо ли вкладки **tab** обернуть **(title\_scroll=%F)** или прокрутить **(title\_scroll=%T).** |
| **TooltipString**  **(ТексПодсказки)** | Это свойство представляет текст (в виде cтроки или вектора строки) подсказки компонентов, появляющийся при наведении мыши на соответствующий компонент. |
| **Units**  **(ЕдиницыИзм)** | Это свойство задает единицы измерения, используемые для задания свойства "Позиция"**({points} | pixels | normalized)**. |
| **Userdata**  **(ДанныеПолзователя)** | Это свойство используется для связывания некоторых объектов Scilab **(string, String matrix, matrix mxn)**. |
| **Value**  **(Значение)** | Это свойство устанавливает значениякомпонентов (скалярные или векторные). Точное значение зависит от конкретного компонента:   * **checkbox** и **radiobutton**: значение **Max** когда включено и **Min** когдавыключено; * **listbox** и **popupmenu**: вектор соответствующий индексам выбранных записей в списке (1 - первый пункт списка); * **layer и tab**: индекс отображаемого компонента; * **image**: значение используется для установки некоторых свойств изображения   **[X-Scale Y-Scale X-Shear Y-Shear RotationAngle].** |
| **Verticalalignment**  **(ВертикальноеВыр)** | **верх / {середина} / низ**  Это свойство устанавливает вертикальное выравнивание текста**(top | {middle} | bottom)**. |
| **Visible**  **(Видимый)** | Это свойство устанавливает видимость компонентов. Если это свойство имеет значение **"on"**(по умолчанию), компоненты отображается, но, если для этого свойства задано значение**"off**", компоненты не будет отображаться на родительском рисунке ({on} | off). |

###### Список литературы

1. Дьяконов, В. П. Maple 10/11/12/13/14 в математических расчетах. - М.: ДМК Пресс, 2014. - 800 c.
2. Дьяконов, В. П. Mathematica 5/6/7. Полный самоучитель. - М.: ДМК Пресс, 2012. - 624 c.
3. Шакин В.Н., Семенова Т.И. Основы работы с математическим пакетом Matlab, Учебное пособие/ МТУСИ, 2016. -133с.
4. Семенова Т.И., Шакин В.Н., Математический пакет Scilab: учебное пособие для бакалавров. -М.: ЭБС МТУСИ, 2017.-127 с. Режим доступа http://www.mtuci.ru/structure/library/catalogue/download.php?book\_id=1834
5. Васильев А.Н. MATLAB. Самоучитель. Практический подход. – СБУ: Наука и Техника, 2012. – 448 с. : ил.
6. Дьяконов, В. П. MATLAB и SIMULINK для радиоинженеров. - М.: ДМК Пресс, 2016. - 976 c.
7. Кетков Ю.Л., Кетков А.Ю., Шульц М.М. Matlab 6.x: программирование численных методов. – СПб.: БХВ-Петербург, 2004. – 672 с., ил.
8. Алексеев Е. Р. Scilab: Решение инженерных и математических задач / Е.Р.Алексеев, О.В.Чеснокова, Е.А. Рудченко. — М.: ALT Linux ; БИНОМ. Лаборатория знаний, 2008 — 269 с.
9. Тропин И.С., Михайлова О.И., Михайлов А.В. Численные и технические расчеты в среде Scilab (ПО для решения задач численных и технических вычислений): Учебное пособие. - Москва: 2008 –65 с.
10. Трохова, Т. А. Введение в Scilab : практикум по курсу «Информатика» для студентов техн. специальностей днев. и заоч. форм обучения – Гомель : ГГТУ им. П. О. Сухого, 2016. – 56 с. - Режим доступа: <https://elib.gstu.by>
11. Андриевский А.Б., Андриевский Б.Р., Капитонов А.А., Фрадков А.Л. Решение инженерных задач в среде Scilab. Учебное пособие/ СПб.: НИУ ИТМО, 2013. - 97с.
12. Боден М. Программирование в SciLab / Вики документация SciLab, 2010. –152с. URL: Режим доступа: http://forge.scilab.org/index.php/p/docprogscilab/downloads
13. Документация Scilab – Режим доступа: https://help.scilab.org/docs/6.0.1/ru\_RU/index.html
14. Семенова Т.И., Загвоздкина А.В., Загвоздкин В.А. Графическое представление результатов расчетов в Scilab. 2018 г. В сборнике «Материалы II Международной научно-практической конференции» г.Донецк, Донецкая Народная Республика, 2018г, —4-10 с.
15. Фриск В.В. Основы теории цепей. –М.: РадиоСофт, 2002. – 288 с.
16. Смирнов Н.И., Фриск В.В. Теория электрических цепей. Учебник для вузов –М.: Горячая линия - Телеком, 2019. – 286 с.
17. Смирнов Н.И., Фриск В.В. Теория электрических цепей: конспект лекций. –М.: Горячая линия - Телеком, 2016. – 270 с.
18. Шакин В.Н., Загвоздкина А.В., Сосновиков Г.К. Объектно-ориентированное программирование на Visual Basic в среде Visible Studio .NET : Учебное пособие — М. : ФОРУМ : ИНФРА-М, 2018. - 398 с.
19. Шакин В.Н. Базовые средства программирования на VisualBasic в среде VisualStudio .NET.: Учебное пособие — М. : ФОРУМ : ИНФРА-М, 2018. — 304 с.
20. Семенова Т.И., Юскова И.Б., Юсков И.О. Проведение расчетов в среде пакета Scilab: Практикум.-М.:ЭБС МТУСИ, 2018.-35с. Режим доступа: http://www.mtuci.ru/structure/library/catalogue/download.php?book\_id=1963
21. Семенова Т.И., Юсков И.О., Юскова И.Б., Алгоритмизация вычислительных задач: Электронное учебное пособие. -М.:ЭБС МТУСИ, 2017.- 64 с. Режим доступа:

http://www.mtuci.ru/structure/library/catalogue/download.php?book\_id=1833

1. Демидович Б.П., Марон И.А. Основы вычислительной математики./ –М.: Наука, 3-е издание, 1966. -664с.
2. Копченова Н.В., Марон И.А. Вычислительная математика в примерах и задачах, М., Лань, 2008. -367с.
3. Семенова Т.И., Кравченко О.М., Шакин В.Н. Вычислительные модели и алгоритмы решения задач численными методами. Учетное пособие / МТУСИ. –М., 2017. - 84с. Режим доступа: http://www.mtuci.ru/structure/library/catalogue/download.php?book\_id=1819
4. Семенова Т.И., Загвоздкина А.В., Загвоздкин В.А. Изучение численных методов с использованием средств пакета Scilab //Экономика и качество систем связи, 2017, №4 (6), С.60-69.
5. Семенова Т.И., Загвоздкина А.В., Загвоздкин В.А. Использование пакета Scilab при изучении методов вычислительной математики //Международный сборник научных трудов «Новые технологии в науке, образовании и производстве» по материалам международной научной производственной конференции 10-13 ноября 2017, РИБиУ (Региональный институт бизнеса и управления) г. Рязань, 2017, – с. 471-482.

Оглавление

[Предисловие 2](#_Toc6900127)

[Глава 1 Основы работы с математическим пакетом Scilab 5](#_Toc6900128)

[**1.1** **Рабочая среда Scilab** 5](#_Toc6900129)

[**1.1.1 Графический интерфейс пользователя** 5](#_Toc6900130)

[**1.1.2** **Основные элементы Рабочей среды Scilab** 6](#_Toc6900131)

[**1.1.3** **Основное меню и его элементы** 7](#_Toc6900132)

[**1.1.4** **Основные окна Рабочей среды** 13](#_Toc6900133)

[**1.1.5** **Контрольные вопросы** 18](#_Toc6900134)

[**1.2** **Основные объекты системы Scilab** 19](#_Toc6900135)

[**1.2.1** **Объекты Scilab и их типы** 19](#_Toc6900136)

[**1.2.2.** **Числовые выражения, их вычисления и создание переменных** 21](#_Toc6900137)

[***Числовые выражения*** 21](#_Toc6900138)

[***Арифметические операции и функции*** 24](#_Toc6900139)

[***Преобразование данных к арифметическому типу*** 28](#_Toc6900140)

[***Приоритет выполнения математических операций*** 29](#_Toc6900141)

[***Замечания по использованию данных типа double*** 29](#_Toc6900142)

[**1.2.3.** **Логические выражения в Scilab** 30](#_Toc6900143)

[**1.2.4.** **Строки символов и форматирование данных** 31](#_Toc6900144)

[***Строки символов*** 31](#_Toc6900145)

[***Форматирование строковых данных*** 34](#_Toc6900146)

[**1.2.5. Списки, структуры и ячейки** 38](#_Toc6900147)

[***Списки*** 38](#_Toc6900148)

[***Объекты типа структур и ячеек*** 42](#_Toc6900149)

[***Объекты типа полином и рациональное число*** 44](#_Toc6900150)

[**1.2.6.** **Определение и использование внутренних встроенных функций** 45](#_Toc6900151)

[***Функции, используемые в Scilab*** 45](#_Toc6900152)

[***deff – Описание и вызов однострочной функции*** 46](#_Toc6900153)

[***function – Описание и вызов многострочной функции*** 48](#_Toc6900154)

[**1.2.7.** **Контрольные вопросы** 50](#_Toc6900155)

[**1.3** **Матричные операции создания, доступа, извлечения и модификации. Векторизация и индексирование** 52](#_Toc6900156)

[**1.3.1.** **Представление данных в Scilab** 52](#_Toc6900157)

[**1.3.2.** **Создание векторов и матриц** 53](#_Toc6900158)

[***Способы создания векторов и матриц*** 53](#_Toc6900159)

[***Создание векторов и матриц с помощью операции –* []** 53](#_Toc6900160)

[***Создание матриц путем объединения существующих*** 56](#_Toc6900161)

[***Создание матриц с помощью операции двоеточие –* :** 57](#_Toc6900162)

[***Динамическое изменение размера матрицы*** 58](#_Toc6900163)

[***Создание массивов ячеек*** 60](#_Toc6900164)

[**1.3.3.** **Индексирование и векторизация** 61](#_Toc6900165)

[***Понятия индексирования и векторизации*** 61](#_Toc6900166)

[***Индексирование векторов*** 62](#_Toc6900167)

[***Стандартное индексирование матриц*** 63](#_Toc6900168)

[***Векторное (линейное) индексирование матриц*** 64](#_Toc6900169)

[**1.3.4.** **Операции и функции с матрицами** 70](#_Toc6900170)

[**1.3.5.** **Логическая индексация** 72](#_Toc6900171)

[***Логическое индексирование матрицы*** 72](#_Toc6900172)

[***Использование функции find*** 75](#_Toc6900173)

[**1.3.6.** **Создание и использование массивов структур и массивов ячеек** 76](#_Toc6900174)

[***Тип данных массивов структур*** 76](#_Toc6900175)

[***Массивы ячеек и доступ к элементам массива ячеек*** 77](#_Toc6900176)

[**1.3.7.** **Контрольные вопросы** 81](#_Toc6900177)

[**1.4.** **Средства визуализация в системе Scilab** 83](#_Toc6900178)

[**1.4.1. Высокоуровневые графические средства. Средства отображение графиков функций и простейших геометрических фигур** 84](#_Toc6900179)

[*Основные понятия высокоуровневой графики* 84](#_Toc6900180)

[*Графические окна* 85](#_Toc6900181)

[*Построение графиков функций от одной переменной* 87](#_Toc6900182)

[*Построение графиков функций от двух переменных* 97](#_Toc6900183)

[*Глобальное свойство графических объектов* 105](#_Toc6900184)

['*ИмяСвойства*1',*ЗначениеСвойст*ва1, …, '*ИмяСвойства*n',*ЗначениеСвойства*n, 105](#_Toc6900185)

[*Примеры построения содержательных графиков* 107](#_Toc6900187)

[Построить график функции корреляции случайного процесса (рис.1.4.1-32). 109](#_Toc6900193)

[*Построение простых геометрических форм* 112](#_Toc6900197)

[**1.4.2. Низкоуровневая графические средства и основные графические объектов** 115](#_Toc6900200)

[*Иерархия графических объектов* 115](#_Toc6900201)

[*Редактор объектов* 117](#_Toc6900202)

[*Графические объекты и их свойства* 118](#_Toc6900203)

[**1.4.3. Построение графических интерфейсов пользователя (GUI)** 120](#_Toc6900204)

[***Понятия графического интерфейса пользователя*** 120](#_Toc6900205)

[***Построение графического интерфейса пользователя*** 120](#_Toc6900209)

[1.4.5 Контрольные вопросы 125](#_Toc6900210)

[**1.5. Средства программирования Scilab** 127](#_Toc6900211)

[**1.5.1. Общие понятия идеологии программирования** 127](#_Toc6900212)

[**1.5.2*.* Функциональные структуры – сценарий, функция и средства их создания** 133](#_Toc6900213)

[***Уровни функциональных программных структур*** 133](#_Toc6900214)

[***Средства Редактора SciNotes*** 135](#_Toc6900215)

[***Создание, сохранение и выполнение сценариев*** 137](#_Toc6900216)

[***Сценарии и встроенные функции*** 139](#_Toc6900217)

[**1.5.3. Общая структура функций и сценариев. Области видимости переменных** 144](#_Toc6900218)

[***Общая структура кода сценария*** 144](#_Toc6900219)

[***Имя функции как тип переменной*** 145](#_Toc6900220)

[***Видимость переменных*** 148](#_Toc6900221)

[**1.5.4. Алгоритмические операторы Scilab и базовые программные структуры** 150](#_Toc6900222)

[***Линейные программные структуры*** 151](#_Toc6900223)

[***Простейшие операторы ввода/вывода данных*** 152](#_Toc6900224)

[***Разветвляющиеся программные структуры*** 155](#_Toc6900225)

[***Регулярные циклические структуры и оператор* for** 161](#_Toc6900226)

[***Итеративные циклические структуры и while*** 171](#_Toc6900227)

[**1.5.5. Средства отладки sce-файлов** 175](#_Toc6900228)

[***Понятие отладки программ*** 175](#_Toc6900229)

[***Отладка сценариев в среде в Scilab debug*** 176](#_Toc6900230)

[**1.5.6.** **Хранение данных и функций в библиотеках** 178](#_Toc6900231)

[***Файловая система*** 178](#_Toc6900232)

[***Сохранение и восстановление переменных*** 178](#_Toc6900233)

[***Создание библиотек функций пользователя*** 180](#_Toc6900234)

[***Использование стартового сценария*** 183](#_Toc6900235)

[**1.5.7.** **Контрольные вопросы** 184](#_Toc6900236)

[2. Средства пакета Scilab для решения задач численными методами 185](#_Toc6900237)

[**2.1. Полиномы** 187](#_Toc6900238)

[**2.1.1. Представление, создание и использование полиномиальных объектов** 187](#_Toc6900239)

[***Представление полиномов в Scilab*** 187](#_Toc6900240)

[***Оценка значений полиномов*** 190](#_Toc6900241)

[***Вычисление корней полиномов*** 191](#_Toc6900242)

[**2.1.2. Операции и функции c полиномиальными** 192](#_Toc6900243)

[**данными и рациональные дроби** 192](#_Toc6900244)

[***Операции над полиноминальными данными*** 192](#_Toc6900245)

[***Рациональная матричная производная*** 193](#_Toc6900246)

[***Рациональные дроби*** 195](#_Toc6900247)

[**2.1.3 Контрольные вопросы** 198](#_Toc6900248)

[2.2. Решения задач аппроксимации и интерполяции функций 199](#_Toc6900249)

[**2.2.1. Постановка задач аппроксимации и интерполяции** 199](#_Toc6900250)

[**2.2.2. Решение задач аппроксимации и интерполяции функций средствами пакета Scilab** 200](#_Toc6900251)

[**2.2.3 Контрольные вопросы** 208](#_Toc6900252)

[2.3. Приближенное вычисление производных и интегралов 209](#_Toc6900253)

[**2.3.1. Постановка задачи вычисления производных и конечных разностей** 209](#_Toc6900254)

[**2.3.2. Вычисление производных средствами Scilab** 210](#_Toc6900255)

[***Вычисление производной от аналитической функции*** 210](#_Toc6900256)

[***Вычисление производной от табличной функции*** 214](#_Toc6900259)

[**2.3.3. Постановка задачи численного интегрирования** 216](#_Toc6900260)

[**2.3.4. Численное вычисление определенных интегралов средствами Scilab** 217](#_Toc6900261)

[***Вычисление определенных интегралов – inttrap*** 217](#_Toc6900262)

[***Вычисление определенных интегралов – integrate*** 218](#_Toc6900263)

[***Вычисление определенных интегралов функцией intg*** 219](#_Toc6900264)

[**2.3.5. Контрольные вопросы** 223](#_Toc6900265)

[2.4. Решение нелинейных уравнений 224](#_Toc6900266)

[**2.4.1. Постановка задачи решения нелинейных уравнений** 224](#_Toc6900267)

[**2.4.2. Решение нелинейных уравнений средствами Scilab** 225](#_Toc6900268)

[**2.4.3 Контрольные вопросы** 233](#_Toc6900269)

[2.5. Решение обыкновенных дифференциальных уравнений 234](#_Toc6900270)

[**2.5.1. Постановка задачи решения обыкновенных дифференциальных уравнений** 234](#_Toc6900271)

[**2.5.2. Решение обыкновенных дифференциальных уравнений средствами Scilab** 235](#_Toc6900272)

[**2.5.3. Контрольные вопросы** 240](#_Toc6900273)

[2.6. Решение систем линейных уравнений 241](#_Toc6900274)

[**2.6.1. Постановка задачи решения систем линейных уравнений** 241](#_Toc6900275)

[**2.6.2. Решение системы линейных уравнений средствами Scilab** 242](#_Toc6900276)

[**2.6.3 Контрольные вопросы** 246](#_Toc6900277)

[**2.7. Решение задач оптимизации средствами Scilab** 247](#_Toc6900278)

[**2.7.1. Постановка задачи оптимизации** 247](#_Toc6900279)

[***Постановка задачи оптимизации нелинейных функций*** 247](#_Toc6900280)

[***Условия существования оптимума*** 248](#_Toc6900281)

[***Исследование целевой функции*** 250](#_Toc6900282)

[**2.7.2. Численные методы оптимизации и их реализация в Scilab** 254](#_Toc6900283)

[***Численные методы оптимизации нелинейных функций*** 254](#_Toc6900284)

[***Средства Scilab для решения задач оптимизации*** . 256](#_Toc6900285)

[**2.7.3. Решатели нелинейной оптимизации Scilab** 257](#_Toc6900286)

[***Решатель* optim** 257](#_Toc6900287)

[***Решатель* fminsearch** 265](#_Toc6900288)

[***Решатель* nmplot** 269](#_Toc6900289)

[**2.7.4 Контрольные вопросы** 270](#_Toc6900290)

[Приложение 271](#_Toc6900291)

[Приложение 1.2 271](#_Toc6900292)

[Приложение 1.3 286](#_Toc6900293)

[Примеры 294](#_Toc6900296)

[Приложение 1.4 295](#_Toc6900297)