Федеральное агентство по образованию
Нижегородский государственный университет им. Н.И. Лобачевского

Национальный проект «Образование»
Инновационная образовательная программа ННГУ
Образовательно-научный центр «Информационно-телекоммуникационные системы:
физические основы и математическое обеспечение»

Н.Ю. Золотых

Использование пакета MATLAB в научной и учебной работе

Учебно-методические материалы по программе повышения квалификации
«Информационные технологии и компьютерная математика»

Нижний Новгород
2006

Учебно-методические материалы подготовлены в рамках инновационной образовательной программы ННГУ: *Образовательно-научный центр «Информационно-телекоммуникационные системы: физические основы и математическое обеспечение»*

Золотых Н.Ю. Использование пакета MATLAB в научной и учебной работе

В пособии описывается система для научно–технических расчетов MATLAB. Освещаются простейшие команды, научная графика, типы данных, программирование функций, основные типовые численные методы.

Для преподавателей, научных работников, аспирантов и студентов, использующих или желающих освоить систему MATLAB.

Оглавление

Предисловие

История развития системы MATLAB (сокращение от «Matrix Laboratory») насчитывает почти три десятка лет. «Классический» MATLAB был написан Кливом Моулером (университет Нью-Мехико) в 1977 г. Он представлял собой интерактивную матричную лабораторию, позволяющую вызывать подпрограммы из пакетов LINPACK и EISPACK. До 1984 г. появлялись новые некоммерческие версии MATLAB'а. В 1984 г. К. Моулер, С. Бангерт и Дж. Литтл образовали фирму MathWorks. С этого момента начинают выходить коммерческие версии системы. К настоящему моменту последней является версия MATLAB 7.0 R 2006 b, вышедшая в свет в сентябре 2006 г. Сейчас MATLAB представляет собой мощный математический пакет со своим языком программирования, гибкими графическими возможностями, средствами сопряжения с другими языками и несколькими десятками пакетов приложений.

В пособии описывается простейшие команды MATLAB'а, научная графика, типы данных, программирование функций, основные типовые численные методы. Для дальнейшего изучения MATLAB'а мы рекомендуем книги, перечисленные в библиографии.

1. Простейшие команды

1.1. Краткое введение в MATLAB

После запуска системы MATLAB на экране появляется основное окно, содержащее несколько подокон. Одно из них имеет заголовок COMMAND WINDOW — это командное окно, в котором пользователь набирает команды, а MATLAB выдает результаты. Результаты выполнения команд, содержащие графический вывод, выдаются в одно или несколько графических окон. Команды пользователя нужно вводить после приглашения системы, которое выглядит следующим образом:

» Например,

$>>$ (76 + 21 − 85)*3/4

(ввод заканчивается клавишей ENTER). MATLAB выдаст ответ:

ans =

9

Теперь наберем:

$>>$ (1 + $sqrt(5)$)/2

получим:

ans =

1.6180

В этом примере мы использовали функцию *sqrt* для нахождения квадратного корня; *ans* — это специальная переменная, в которую всегда засылается результат последней команды. Эту переменную можно использовать в следующей команде. Например,

$>>$ 1/ans

ans =

0.6180

Пользователь может создавать свои переменные. Например, команда

$>>$ e=2 + 1/2 + 1/6 + 1/24 + 1/120 + 1/720

e =

2.7181

создает переменную с именем e и значением 2.7181. Теперь переменную e можно использовать. Например,

$$>> \ err \ = \ e \ - \ exp(1)$$

Получим:

$$err \ =$$
$$-2.2627\text{e}{-4}$$

Функция *exp* вычисляет экспоненту e^x. Запись $-2.2627\text{e}{-4}$ — это представление числа в форме с плавающей точкой. Его нужно понимать следующим образом:

$$-2.2627\text{e}{-4} = -2.2627 \cdot 10^{-4} = -0.00022627.$$

Перед тем как использовать переменную, ее нужно инициализировать, т. е. присвоить ей некоторое значение (так и было в двух предыдущих примерах с переменными e и err). Использовать неинициализированные переменные запрещено. Например, если p и/или q ранее не встречались в левой части присваивания, то следующая команда

$$>> \ x \ = \ -p \ + \ sqrt(p\text{^}2 \ - \ q)$$

приведет к сообщению об ошибке.

До сих пор знаком окончания команды являлся символ конца строки: ввод команды заканчивался клавишей ENTER. В результате мы всегда получали эхо (отклик). В конце команды, перед тем как ввести ENTER, можно поставить знак «;» (точка с запятой). В этом случае отклика системы мы не получим. Например, после того как мы введем

$$>> \ e \ = \ 2 \ + \ 1/2 \ + \ 1/6 \ + \ 1/24 \ + \ 1/120 \ + \ 1/720;$$

переменной e будет присвоено вычисленное значение 2.7181 и мы сразу получим новое приглашение:

$$>>$$

Далее в примерах знак приглашения мы печатать не будем.

В одной строке можно набирать несколько команд. Их нужно отделять либо символом «,» (запятая), либо символом «;» (точка с запятой). В первом случае отклик будет, во втором — нет.

Именами переменных могут быть любые последовательности латинских букв (в любом регистре), цифр и знаков подчеркивания. Первым символом в имени может быть

либо буква, либо символ подчеркивания. Ограничений на длину имени нет, но при сравнении имен роль играют только первые символы (31 символ). Кроме того, важен регистр: например, переменные *Err* и *err* — разные.

На машинах, поддерживающих IEEE-арифметику, в системе MATLAB под действительный скаляр отводится число двойной точности с плавающей точкой.

Кроме *ans* в системе MATLAB есть другие встроенные переменные. Перечислим некоторые из них.

Переменная *eps* хранит расстояние между 1 и следующим за ним действительным числом, представимым в компьютере. На машинах, поддерживающих IEEE-арифметику, оно больше машинного эпсилон ε_{M} в 2 раза: $eps = \beta^{-t} = 2^{-52}$, где β — основание машинной арифметики, t — разрядность (длина мантиссы), в то время как $\varepsilon_{\mathrm{M}} = \beta^{-t}/2 = 2^{-53}$. В документации *eps* названо *относительной точностью арифметики с плавающей точкой* (floating-point relative accuracy). Заметим, что часто в формулах, в которых встречается ε_{M}, важно не точное его значение, а порядок. В этом случае ε_{M} можно заменить на *eps*.

Максимальное представимое вещественное число хранится в переменной *realmax*, минимальное положительное нормализованное вещественное число — в переменной *realmin*.

Итак, в IEEE-арифметике

$$eps = 2.2204 \cdot 10^{-16},$$

$$realmax = 1.7977 \cdot 10^{308}, \quad realmin = 2.2251 \cdot 10^{-308}.$$

Величины Inf, $-Inf$ в IEEE-арифметике служат для представления $\pm\infty$. «Нечисло» обозначается NaN.

Переменная *pi* хранит приближение к числу π.

Комплексные числа представляются парой двух вещественных. Переменные i и j представляют мнимую единицу. Поэтому, например 1 + 2*i — это комплексное число $1 + 2i$.

Упомянутые выше переменные *eps*, *realmax*, *realmin*, Inf, NaN, *pi*, i, j могут появляться в правой части присваивания. После этого они теряют свое первоначальное значение. Например, переменные i, j часто используются как счетчики в циклах. После этого 1 + 2*i будет уже не комплексным числом $1 + 2i$, а чем-то другим. Однако запись $1 + 2i$ (мы опустили знак умножения) всегда означает комплексное число $1 + 2i$. Вернуть первоначальное значение предопределенных констант можно командой *clear*.

Например,

clear i

возвращает значение константе i как мнимой единице.

Мы уже встречались с функциями *sqrt* и *exp*. В систему MATLAB встроено большое число других стандартных математических функций. Приведем небольшой список некоторых из них (список всех элементарных математических функций, доступных в системе MATLAB, можно получить набрав команду *help elfun*):

sin, *cos*, *tan*	обычные тригонометрические функции
acos, *asin*	обратные тригонометрические функции
exp, *log*	экспонента и натуральный логарифм
sqrt	квадратный корень
round	округление до ближайшего целого
fix	округление с отбрасыванием дробной части
abs	модуль вещественного или комплексного числа
angle	аргумент комплексного числа
real, *imag*	вещественная и мнимая части комплексного числа
conj	комплексное сопряжение

1.2. Основные функции для работы с матрицами

Основной объект в системе MATLAB — это матрицы, или массивы. Даже скалярные величины, с которыми мы имели дело в предыдущем разделе, рассматриваются системой как матрицы 1×1.

Вектор (одномерный массив) представляет собой строку, т. е. матрицу размера $1 \times n$, или столбец, т. е. матрицу размера $m \times 1$. Чтобы задать вектор, достаточно перечислить его элементы, заключая их в квадратные скобки. Элементы векторов-строк разделяются символами «,» (запятая) или « » (пробел). Элементы векторов-столбцов разделяются символом «;» (точка с запятой) или символом перехода на новую строку, который нужно ввести клавишей ENTER. Например, команда

$a = [1\ 2\ 3\ 4]$

задает строку $a = (1, 2, 3, 4)$, а команда

$$b = [1;\ 2;\ 3;\ 4]$$

задает столбец

$$a = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}.$$

Векторы, состоящие из последовательных членов арифметической прогрессии, легко задавать с помощью команды «:». Команда $a{:}h{:}b$ определяет вектор

$$(a, a + h, a + 2h, \ldots, b).$$

Команда $a{:}b$ определяет отрезок арифметической прогрессии с шагом 1.

Чтобы задать матрицу (двумерный массив), достаточно перечислить ее элементы построчно, разделяя элементы в одной строке пробелами или запятыми, а сами строки — точкой с запятой или символом перехода на новую строку. Например, команда

$$A = [1\ \ 2;\ 3\ \ 4]$$

определяет матрицу

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Строки должны содержать равное число элементов, в противном случае MATLAB выдаст сообщение об ошибке.

Для генерирования матриц полезны следующие простые функции:

zeros(*m*,*n*)	нулевая матрица
ones(*m*,*n*)	матрица, состоящая из одних единиц
eye(*m*,*n*)	матрица с 1 на диагонали и 0 вне диагонали
rand(*m*,*n*)	матрица со случайными элементами, равномерно распределенными на отрезке $[0, 1]$
randn(*m*,*n*)	матрица со случайными элементами, распределенными по нормальному закону с математическим ожиданием, равным 0, и средним квадратическим отклонением, равным 1.

В приведенной таблице параметры m и n определяют число строк и столбцов матрицы соответственно. Каждую из предыдущих функций можно вызвать с одним параметром — в этом случае генерируется квадратная матрица указанного порядка с соответствующим свойством. Например, команда $eye(m)$ генерирует единичную матрицу порядка m.

К матрицам применимы все стандартные математические функции: они применяются покомпонентно к каждому элементу. В отличие от этих функций операции, обозначаемые символами, подобными $+$, $*$, выполняются как *матричные* операции. Приведем список операций:

$a+b$	сложение скаляров, векторов или матриц
$a-b$	вычитание скаляров, векторов или матриц
$a*b$	умножение скаляров; матричное умножение
$a.*b$	покомпонентное умножение элементов матриц
$a\hat{}b$	возведение скаляра или матрицы в степень
$a.\hat{}b$	возведение каждого элемента матрицы в степень
a/b	деление скаляров; правое деление матриц, т. е. $a \cdot b^{-1}$
$a./b$	покомпонентное деление элементов матриц
$a\backslash b$	левое деление матриц, т. е. $a^{-1} \cdot b$
a'	транспонирование матрицы

Если размеры операндов (матриц, участвующих в операции) не согласованы, то выдается сообщение об ошибке.

Нумерация строк и столбцов в матрицах начинается с 1. Чтобы получить доступ к элементу матрицы A, стоящему в i-й строке, j-м столбце, достаточно ввести $A(i, j)$. Чтобы получить доступ к k-й компоненте вектора a, досточно ввести $a(k)$. Команда $A(k)$ работает и для матриц: MATLAB ищет k-й элемент матрицы A, предполагая, что элементы нумеруются по столбцам.

Что произойдет, если в команде есть ссылка на несуществующий элемент матрицы? Например, матрица A имеет размеры 2×2 и происходит обращение к элементу $A(3, 5)$. Все зависит от того, в какой части от знака присваивания расположена ссылка на элемент $A(3, 5)$. Если $A(3, 5)$ находится где-то в выражении справа от знака присваивания, это приведет к сообщению об ошибке. Если $A(3, 5)$ стоит слева, то размеры матрицы автоматически переопределяются до 3×5, при этом новым элементам матрицы присваиваются нулевые значения, а элементу $A(3, 5)$ — вычисленное значение.

Для доступа к последнему столбцу или последней строке можно использовать ключевое слово **end**. Например, $A(\mathbf{end},\ k)$ — это элемент матрицы A, стоящий в последней строке и k-м столбце.

Для доступа ко всем строкам или столбцам матрицы используется команда «:» (двоеточие). Например, $A(:,\ k)$ — это k-й столбец, а $A(k,\ :)$ — k-я строка матрицы A. Подобные выражения могут встречаться и в левой части присваивания. Например, команда

$A(k,\ :)$=3

присваивает всем элементам k-й строки значение 3.

Результатом операции $A(:)$ является длинный столбец, составленный из столбцов матрицы A. Таким образом, $A(k)$ — это k-й элемент вектора $A(:)$.

Функции $max(a)$ и $min(a)$ возвращают максимальный и соответственно минимальный элементы вектора a. Если кроме самого́ максимального или минимального значения b нам нужен его индекс i, то необходимо вызвать функцию с двумя выходными параметрами:

$$[b,\ i] = max(a)$$
$$[b,\ i] = min(a)$$

Если максимальных (минимальных) значений несколько, то возвращается номер первого из них.

Если A — матрица, то $max(A)$, $min(A)$ сформируют вектор-строку b, j-й элемент которой равен максимальному и соответственно минимальному элементу в j-м столбце матрицы A. Функции

$$[b,\ i] = max(A)$$
$$[b,\ i] = min(A)$$

кроме b возвращают также строку i, в j-й позиции которой записан индекс максимального (соответственно минимального) элемента в j-м столбце. Для нахождения максимума среди всех элементов матрицы A можно использовать $max(max(A))$ или $max(A(:))$.

Функция $size(A)$ возвращает двухкомпонентный вектор, содержащий число строк и число столбцов матрицы A. Функция $size(A,\ 1)$ возвращает число строк, функция $size(A,\ 2)$ — число столбцов; $length(A)$ — максимум из этих двух чисел, поэтому для векторов — это число компонент в них.

В системе MATLAB легко реализуются блочные операции над матрицами. Так, команды $C = [A\ B]$, $C = [A;\ B]$ формируют из матриц A и B блочные матрицы

$$C = (A\ B) \quad \text{и} \quad C = \begin{bmatrix} A \\ B \end{bmatrix}$$

соответственно. Размеры матриц A и B должны быть согласованы.

Пусть u, v — векторы, в которых записаны номера некоторых строк и столбцов соответственно (быть может, с повторениями) матрицы A. Тогда $A(u,\ v)$ — это матрица, составленная из элементов исходной матрицы, стоящих на пересечении строк с номерами u и столбцов с номерами v.

Система MATLAB поддерживает работу с пустыми матрицы, т. е. матрицами, в которых число строк или/и число столбцов равно нулю. Один из способов задать такую матрицу — воспользоваться функцией типа *zeros*. Например,

$$A = zeros(0,\ 5)$$

Определить матрицу размера 0×0 можно с помощью операции [].

Операция [] помогает также при удалении строк или столбцов матриц. Команда

$$A(i,\ :) = []$$

удаляет i-ю строку, а команда

$$A(:,\ j) = []$$

удаляет j-й столбец матрицы A.

В системе MATLAB есть удобные средства для работы с диагоналями матриц. Пусть d — вектор из n компонент. Команда

$$A = diag(d,\ k)$$

возвращает квадратную матрицу A порядка $n + |k|$ с элементами d на k-й диагонали; $k = 0$ соответствует главной диагонали матрицы, $k > 0$ — k-й наддиагонали, $k < 0$ — $|k|$-й поддиагонали. Если аргумент k не указан, то d размещается на главной диагонали матрицы A.

Команда

$$d = diag(A,\ k)$$

возвращает вектор-столбец d, содержащий элементы k-й диагонали матрицы A. Если аргумент k не указан, то возвращается главная диагональ матрицы A.

В системе MATLAB есть следующие (бинарные) операции отношения: «меньше» $<$, «больше» $>$, «не больше» $<=$, «не меньше» $>=$, «равно» $==$, «не равно» $\tilde{}=$. Они выполняют поэлементное сравнение двух массивов одинаковых размеров и возвращают так называемый *логический* массив того же размера. Его элементы равны 1 или 0 в зависимости от того, истинно или нет рассматриваемое отношение для соответствующей пары элементов из двух массивов. Один из аргументов может быть скаляром. В этом случае, как и для арифметических операций, вместо него будет рассмотрена матрица, заполненная этим скаляром.

Логические операции — это бинарные операции «и» &, «или» | и унарная операция «не» $\tilde{}$. Аргументами этих операций выступают, как правило, логические массивы, но могут выступать и обычные числовые массивы. В этом случае нулевое значение интерпретируется как ложь, а ненулевое — как истина. A & B — это логический массив, элементы которого равны 1 на тех позициях, на которых оба соответствующих элемента в A и B имеют ненулевые значения, и равны 0 на остальных позициях. $A \mid B$ — это логический массив, элементы которого равны 1 на тех позициях, на которых по крайней мере один из соответствующих элементов в A и B имеет ненулевое значение, $\tilde{}A$ — это логический массив, элементы которого равны 1 на тех позициях, где соответствующие элементы массива A нулевые, и равны 0 в противном случае. Матрицы A и B должны иметь одинаковые размеры или один из аргументов — скаляр.

Операции отношения и логические операции часто используются вместе с функциями *find*, *any*, *all*.

Функция *find* возвращает индексы и значения ненулевых элементов. Функция имеет следующий синтаксис

$$[i,\ j,\ v] = \mathit{find}(A)$$

где i, j, v — векторы, содержащие соответственно номера строк, столбцов и значения ненулевых элементов матрицы A.

Приведем небольшой пример. Пусть f — вектор, содержащий протабулированные значения некоторой функции в точках, хранящихся в x. Требуется найти корни функции. Это можно сделать следующим образом:

$$n = \mathit{length}(x);$$
$$w = 1 : n - 1;$$
$$x(\mathit{find}(f(w)\ .*\ f(w{+}1) < 0 \mid f(w) == 0))$$

Если a — вектор, то функция $all(a)$ возвращает 1, когда все элементы массива a не равны нулю, и 0 в противном случае. Если A — матрица, то $all(A)$ выполняет функцию all для каждого столбца матрицы A и возвращает полученные значения в векторе-строке.

Если a — вектор, то функция $any(a)$ возвращает 1, когда среди элеметов массива a есть ненулевой, и 0 в противном случае. Если A — матрица, то $any(A)$ выполняет функцию any для каждого столбца матрицы A и возвращает полученные значения в векторе-строке.

Пусть например,

$$A = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}.$$

Тогда $any(A)$ возвращает вектор $(0, 1, 1, 1)$, а $all(A)$ — вектор $(0, 0, 0, 1)$.

1.3. Массивы символов

MATLAB позволяет работать со строками текста. Чтобы определить такую строку, достаточно заключить строку символов в кавычки. Например,

s = '*Isaac Newton*'

Строковые значения рассматриваются системой как массивы. В приведенном примере s — это вектор-строка из 12 элементов-символов. (никакого завершающего нулевого символа нет).

Символы можно объединять в двумерные массивы. Это позволяет хранить набор строковых значений одинаковой длины. Например, после ввода команды

S = ['*Isaac Newton* '
'*Blaise Pascal*']

MATLAB создает следующий двумерный массив 2×13:

$$\begin{bmatrix} \text{'I'} & \text{'s'} & \text{'a'} & \text{'a'} & \text{'c'} & \text{' '} & \text{'N'} & \text{'e'} & \text{'w'} & \text{'t'} & \text{'o'} & \text{'n'} & \text{' '} \\ \text{'B'} & \text{'l'} & \text{'a'} & \text{'i'} & \text{'s'} & \text{'e'} & \text{' '} & \text{'P'} & \text{'a'} & \text{'s'} & \text{'c'} & \text{'a'} & \text{'l'} \end{bmatrix}$$

В данном примере мы специально добавили к имени Ньютона один пробел, чтобы уравнять число элементов в каждой строке массива. Если этого не сделать, MATLAB выдаст сообщение об ошибке.

Команда $S(1,:)$ в данном примере возвращает строку '*Isaac Newton* ', команда $S(2,:)$ — строку '*Blaise Pascal*'. Вообще, к массивам символов применимо большое число команд из предыдущего раздела. Так, например, доступны блочные операции, команды выделения подмассивов и др. В примере

a = '*Matrix*';
b = '*Laboratory*';
c = [a(1:3) b(1:3)]

переменной c будет присвоено символьное значение '*MatLab*'.

Внутри системы MATLAB символы представлены в формате UNICODE. Для хранения одного символа используется 2 байта.

1.4. Форматированный вывод

MATLAB предоставляет широкие возможности для управления форматом вывода числовых значений в командном окне. По умолчанию используется формат *short*. В нем используются следующие правила:

- Если *все* элементы массива — целые числа не более чем из 9 цифр каждое, то то они выводятся как есть.
- Если *все* элементы массива по абсолютной величине меньше 1000 и не меньше 0.0001, то они выводятся как есть.
- В остальных случаях MATLAB выводит элементы массива с использованием общего множителя (исключение составляют скалярные величины). Элементы выводятся в формате $\pm d.ddddesppp$, где d — цифры мантиссы, p — цифры показателя. Если число не ноль, то старшая цифра перед десятичной точкой не равна нулю. Если число отрицательное, то впереди, разумеется, ставится знак $-$.

Например, после ввода команды

1/7

MATLAB напечатает

0.1429

Управлять форматом вывода можно либо с помощью пункта меню FILE | PREFERENCES | COMMAND WINDOW | NUMERIC FORMAT или с помощью команды *format* с параметрами. Действие этой команды распространяется для всех последующих выдач, пока не

будет введена новая команда *format* с другим параметром. Приведем список некоторых из возможных параметров:

Формат	*Описание*	*Пример*
format	то же, что и *short*; установлен по умолчанию	3.1416
format short	формат представления чисел с фиксированной точкой: 5 значащих цифр	3.1416
format long	формат представления чисел с фиксированной точкой: 15 значащих цифр	3.14159265358979
format short e	формат представления чисел с плавающей точкой: 5 значащих цифр	3.1416e+000
format long e	формат представления чисел с плавающей точкой: 15 значащих цифр	3.141592653589793e+000
format rat	аппроксимация чисел рациональной дробью	355/113

Подчеркнем, что команда *format* влияет только на вывод числовых данных и ни коим образом не влияет на то, как хранятся эти данные. Это относится, конечно, и к формату *rat*: всякий раз, когда нужно вывести матрицу X, MATLAB находит к каждоиу ее элементу x_{ij} наилучшее рациональное приближение p_{ij}/q_{ij}, такое, что

$$|p_{ij}/q_{ij} - x_{ij}| \le tol \cdot |x_{ij}|, \qquad \text{где} \qquad tol = 10^{-6} \sum_{i,j} |x_{ij}|.$$

1.5. Справка и документация

Система MATLAB предоставляет пользователю удобный справочный навигатор с развитой системой поиска необходимой информации. Навигатор доступен через пункт меню HELP|MATLAB HELP. Справка охватывает все разделы ядра MATLAB'а и пакеты расширений, включает перекрестные ссылки, программы-примеры, ссылки на демонстрационные приложения.

Быстрый способ открыть нужный раздел справки, касающийся некоторой функции, — набрать в командном окне

doc имя_функции

Ту же информацию, но непосредственно в командном окне, можно получить по команде

help имя_функции

Например, набрав

help inv

вы получите справку по функции *inv*:

INV Matrix inverse.
INV(X) is the inverse of
the square matrix X.
A warning message ...

See also SLASH, PINV ...

Overloaded methods ...

Заметим, что в справке командного окна все функции набраны в верхнем регистре. Это сделано лишь для того, чтобы выделить их в тексте. В действительности, имена всех встроенных функций системы MATLAB в нижнем регистре.

1.6. Среда MATLAB

1.6.1. Рабочее пространство командного окна

Значения всех создаваемых в командном окне переменных хранятся в отведенной области памяти, которую мы будем называть *рабочим пространством командного окна* или *основным рабочим пространством*. Помимо основного рабочего пространства MATLAB создает и в нужное время удаляет рабочие пространства вызываемых функций. В этих рабочих пространствах размещаются внутренние переменные функций.

Имена всех переменных, находящихся в настоящий момент в основном рабочем пространстве отображаются в подокне WORKSPACE. Список имен переменных можно также получить командой *who*, а список переменных с их значениями — командой *whos*.

Команда

clear список_переменных

исключает (стирает) указанные переменные из рабочего пространства. При этом их значения теряются. Команда

clear all

стирает все переменные рабочего пространства.

Рассмотрим пример:

$A = rand(100);$
$B = rand(100);$
$C = A*B;$
who

MATLAB напечатает:

Your variables are:

A B C

Далее наберем:

clear A B
whos

Получим:

Name	*Size*	*Bytes*	*Class*
C	100x100	80000	*double array*

Grand total is 10000 *elements using* 80000 *bytes*

1.6.2. Сохранение и загрузка переменных

После выхода из системы MATLAB все переменные уничтожаются и поэтому не доступны в следующий сеанс работы. Запомнить на диске все переменные рабочего пространства можно с помощью команды

save имя_файла

При этом все переменные сохраняются в бинарном формате в указанном файле. По умолчанию расширение этого файла — *mat*, поэтому файлы такого формата называются *mat*-файлами. Не рекомендуется для *mat*-файлов использовать другие расширения.

Чтобы запомнить не все, а только часть переменных из рабочего пространства, воспользуйтесь командой

save имя_файла список_переменных

(имена переменных отделяются друг от друга пробелами).

Прочитать *mat*-файл можно с помощью команды

load имя_файла

которая загружает все переменные из файла, или с помощью команды

load имя_файла список_переменных

которая загружает из файла лишь указанные переменные.

Рассмотрим пример:

```
x = [0; 1; 2; 3; 4];
W = [x.^0, x.^1, x.^2, x.^3];
save Wndrmd x W
clear all
load Wndrmd
who
```

MATLAB напечатает:

```
Your variables are:
W x
```

С помощью команд

save имя_переменной $-ascii$

можно сохранить значение переменной в одноименном *текстовом* файле. В данным файле в текстовом формате построчно будут записаны элементы матрицы. Такие файлы можно готовить и править в любом текстовом редакторе. Не зависимо от того, как был создан такой файл, его можно прочитать командой

load имя_файла $-ascii$

Значения из файла будет загружены в переменную, имя которой совпадает с именем файла (без расширения).

1.6.3. Команды dir, type, delete, cd

На панели управления в основном окне MATLAB расположено выпадающее меню CURRENT DIRECTORY, в котором можно выбрать текущую папку.

Команда

dir

выводит на экран список файлов из текущей папки.

Команда

type имя_файла

выводит на экран распечатку указанного файла из текущей папки.

Команда

delete имя_файла

удаляет указанный файл.

Команда

cd новый_каталог

изменяет текущую папку. То же действие можно проделать, воспользовавшись выпадающим меню в подокне Current Directory или меню на панели управления.

1.6.4. Дневник работы

Чтобы записать в файл все, что отображается в командном окне: и ваши команды и ответы системы, — воспользуйтесь командой *diary*. Команда

diary имя_файла

открывает дневник, т.е. указывает системе, что все, что появится после этой команды на экране до следующей команды *diary* будет записано в упомянутый текстовый файл. Прерывает запись в дневник команда открытия нового дневника или команда

diary off

1.6.5. Запуск внешних программ

Запустить на выполнение любую команду ДОС или любую внешнюю программу, находящуюся в текущей папке, можно набрав

!имя_программы

1.7. Сценарии

Сценарием, или, просто, *скриптом*, называется последовательность команд системы MATLAB, записанных в текстовом файле. Файл должен иметь расширение m. Команды отделяются друг от друга, как и в командном окне, символами «,», «;» или символом перехода на новую строку. Коментарии начинаются с символа % и продолжаются вплоть до конца строки. Группа из трех подряд идущих точек «...», поставленных в конце строки, означает, что текущая команда продолжается на следующей строке.

Чтобы выполнить команды, записанные в программе–сценарии, достаточно имя файла (без расширения m) набрать в командном окне. Сценарии можно также вызывать из других программ–сценариев.

Дальнейшая информация о создании пользовательских программ в MATLAB'е приводится в главе 3.

2. Научная графика

В этой главе мы рассмотрим несколько простейших функций с графическим выводом. Весь графический вывод в системе MATLAB поступает в одно или несколько графических окон. Обычно пользователь не должен беспокоиться об их открытии и закрытии: графические функции высокого уровня, о которых пойдет речь в настоящей главе, обычно ведут себя достаточно разумным образом.

2.1. Функция plot

Если x и y — два вектора одинаковой длины, то функция

plot(x, y)

в графическом окне строит ломаную по точкам с абсциссами, записанными в x, и ординатами, записанными в y. Масштаб по обеим осям выбирается автоматически, так, чтобы ломаная целиком убиралась на графике. Если до выполнения этой команды ни одно графическое окно открыто не было, то такое окно открывается автоматически. В противном случае вывод будет происходить в последнее (текущее) графическое окно и по умолчанию будет стирать старое изображение

С помощью функции *plot* легко построить график функции. Например:

```
x = 0 : pi/100 : 2*pi;
y = sin(x);
plot(x, y)
```

Функции *xlabel*, *ylabel* добавляют подписи к оси абсцисс и ординат соответственно, а *title* определяет заголовок.

Применим эти функции для нашего примера:

```
xlabel('x = 0:2\pi')
ylabel('Sine of x')
title('Plot of the Sine Function')
```

Результат см. на рис. 2.1.

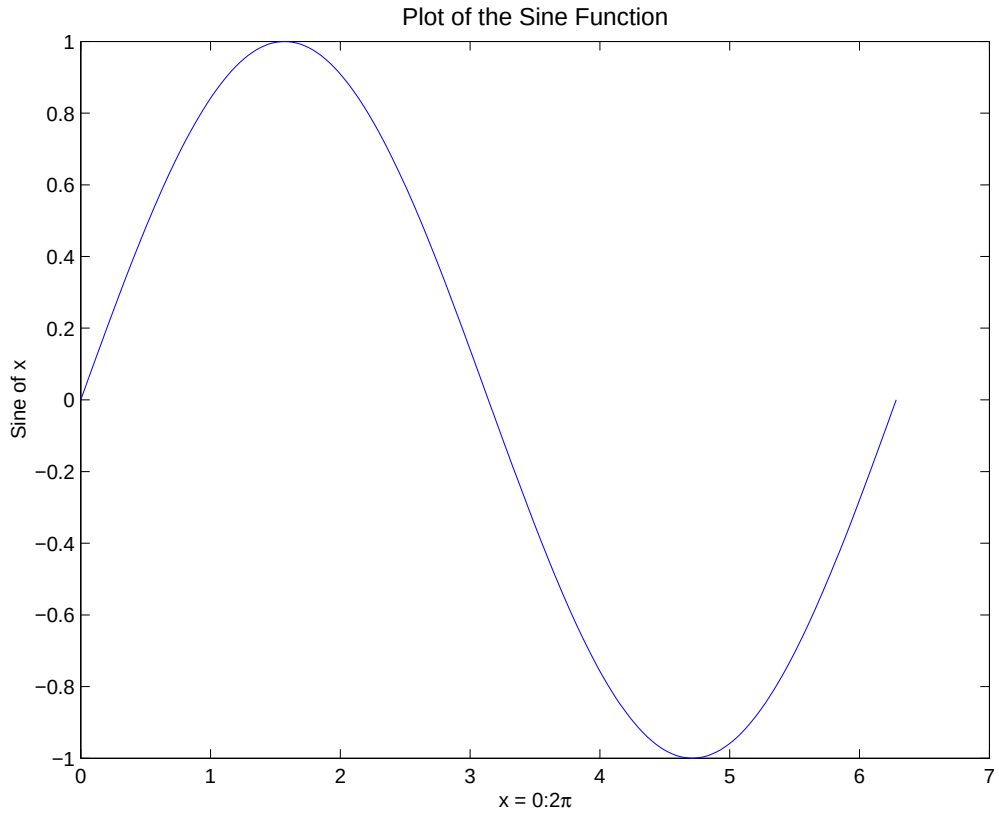


Рис. 2.1. График функции $\sin x$

2.1.1. Команда figure

Команда *figure* создает новое графическое окно и делает его текущим

Команда *figure*(n) делает текущим окно с номером n

2.1.2. Несколько кривых на графике

Каждая новая функция *plot* стирает старое изображение. Для того, чтобы нарисовать несколько графиков, мы можем использовать команду *hold on* («заморозить»), включающей режим сохранения предыдущего графического результата.

Например, построим графики трех функций (см. рис. 2.1.2):

Выйти из режима *hold on* можно с помощью команды *hold off*.

Другой способ вывести несколько графиков в одном окне — применить функцию

$$plot(x1,\ y1,\ \ldots\ ,\ xn,\ yn)$$

с несколькими парами параметров x, y. Попробуем построить те же графики:

$$plot(x,\ y1,\ x,\ y2,\ x,\ y3)$$

Отличие от предыдущего примера в том, что графики нарисованы разными цветами. Ниже мы подробно рассмотрим правила чередования цветов.

Команда *legend* добавляет легенду — пояснение к графикам. У нас три графика, поэтому число параметров функции *legend* должно быть тоже три:

$$legend('sin(x)',\ 'exp(-x\hat{}2)',\ '0.5\ atan(x)')$$

Легенда появится в правом верхнем углу осей координат. В данном случае это не совсем удачно, так как она перекроет часть графиков. Чтобы изменить положение легенды, можно перетащить ее мышкой или воспользоваться функцией *legend* еще с одним параметром:

$$legend('sin(x)',\ 'exp(-x\hat{}2)',\ '0.5\ atan(x)',\ 2)$$

Получим изображение на рис. 2.1.2. Этим параметром может быть число от 0 до 5. Числа от 1 до 4 соответсвуют номеру квадранта, в котором будет размещена легенда: 1 — правый верхний, 2 — левый верхний, 3 — левый нижний, 4 — правый нижний. Если указан 0, то MATLAB сам ищет наиболее удачное место на графике. Если указана 5, то MATLAB размещает легенду снаружи от осей координат.

Если x — вектор длины m, а Y — матрица с размерами $m \times n$, то команда

$$plot(x,\ Y)$$

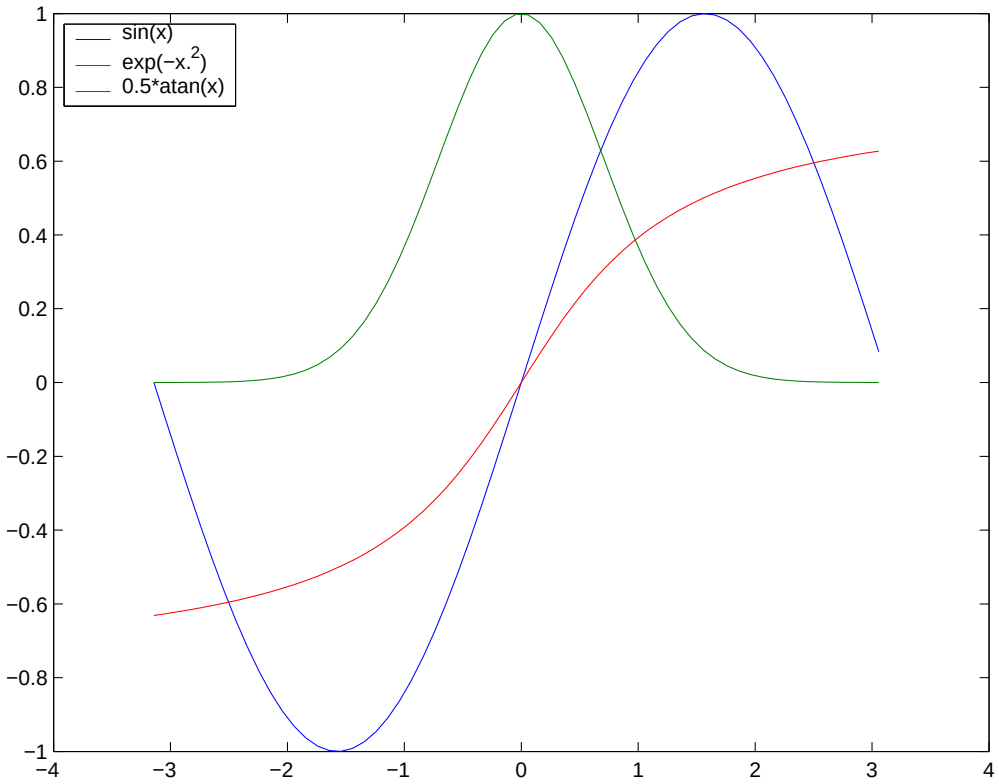


Рис. 2.2. Три графика в одних осях

эквивалентна команде

$plot(x,\ Y(:,1),\ x,\ Y(:,2),\ \ldots,\ x,\ Y(:,n))$

Например, графики из предыдущего примера можно получить с помощью команды

$plot(x,\ [y1,\ y2,\ y3])$

Нарисуем 8 синусоид с разными амплитудами:

$a\ =\ 1{:}8;$
$x\ =\ linspace(0,\ pi,\ 100);$
$y\ =\ sin(x)';$
$plot(x,\ y{*}a);$

и понаблюдаем за порядком чередования цветов. Он следующий:

синий—зеленый—красный—бирюзовый—лиловый—желтый—черный
blue—green—red—cyan—magenta—yellow—black

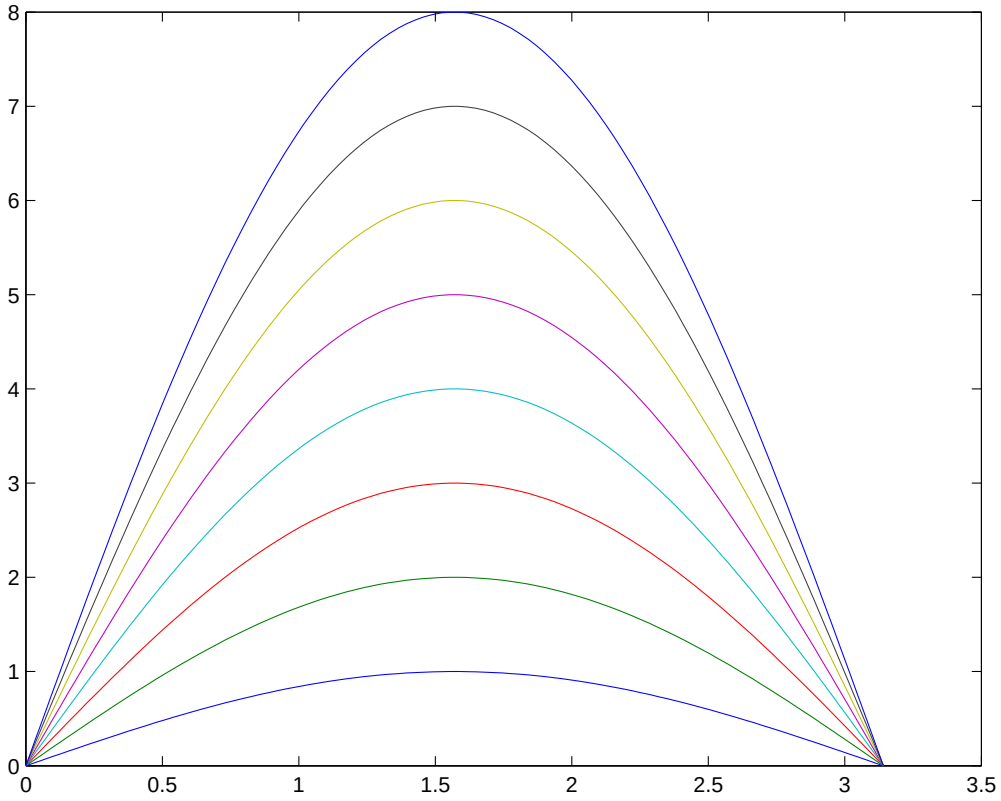


Рис. 2.3. Синусоиды с разными амплитудами

2.1.3. Стиль и цвет линий

Рассмотрим еще один вариант вызова функции *plot*:

plot(x, y, стиль)

Здесь стиль — это строка, состоящая от 1 до 4 символов, обозначающих цвет и стиль линии и тип маркера:

- Цвет: c, m, y, r, g, b, w, and k
- Стиль линии: $-$, $--$, :, $-.$
- Тип маркера: $+$, o, *, x, s, d, ^, v, $>$, $<$, p, h

Цвет линии и маркера определяется одной буквой:

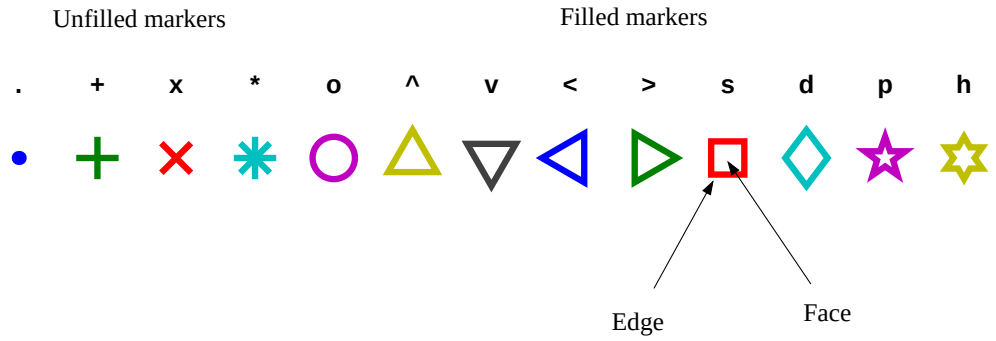


Рис. 2.4. Типы маркеров

Символ	Цвет
’b’	синий
’g’	зеленый
’r’	красный
’c’	бирюзовый
’m’	лиловый
’y’	желтый
’k’	черный

Рассмотрим возможные стили линии:

Символ	Стиль линии
’-’	сплошная линия (по умолчанию)
’_’	штрих-линия
’:’	пунктирная линия
’-.’	штрих-пунктирная линия
’none’	нет линии

Можно строить и задавать стиль сразу нескольким графикам:

plot(*x1*, *y1*, стиль1, *x2*, *y2*, стиль2 ...)

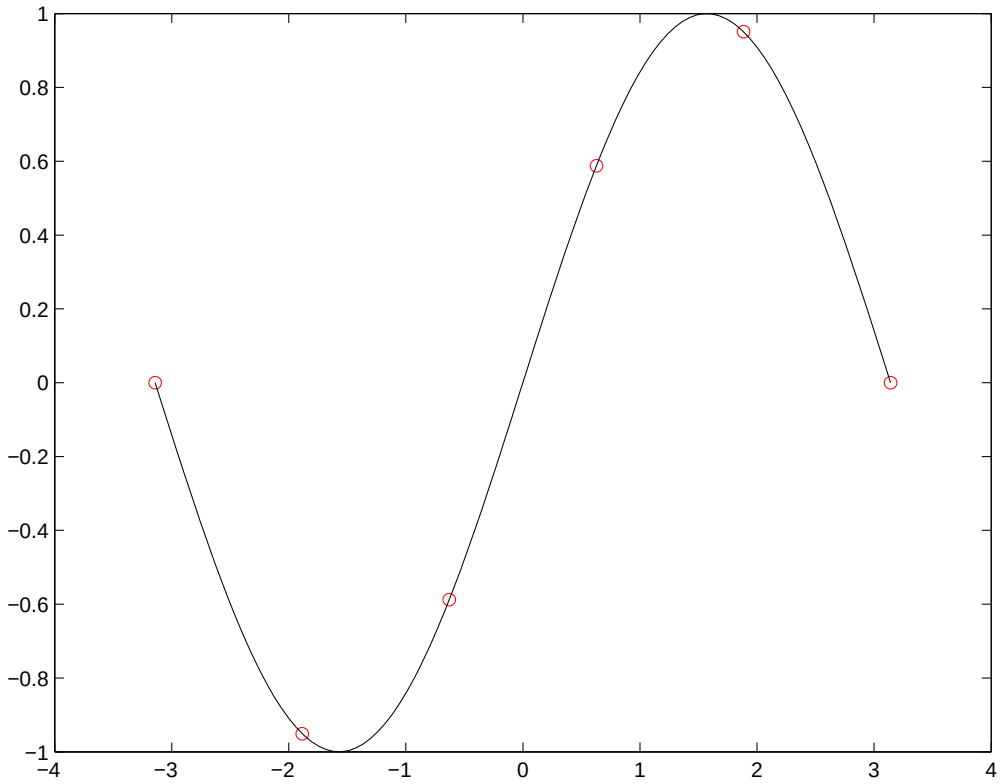


Рис. 2.5. Задание стиля для нескольких графиков

Рассмотрим пример

$x = linspace(-pi,\ pi,\ 6);$
$y = sin(x);$
$xx = linspace(-pi,\ pi,\ 100);$
$yy = sin(xx);$
$plot(x,\ y,\ 'or',\ xx,\ yy,\ '-k')$

Результат видим на рисунке 2.5

2.1.4. Команды axis и grid

Функция

$xlim([xmin,\ xmax]$

задает диапазон изменения на графике координаты x; а

$ylim([ymin,\ ymax]$

устанавливает диапазон изменения на графике координаты y. Можно сразу задать пределы изменения для обеих координат:

$axis([xmin,\ xmax,\ ymin,\ ymax])$

Командой *axis auto* можно восстановить режим, в котором пределы вычисляются автоматически.

Команда *axis square* устанавливает одинаковые пределы по всем осям. Команда *axis equal* устанавливает одинаковый масштаб по всем осям. Команда *axis off* убирает изображение осей координат (метки осей и белый прямоугольник графического вывода). При этом все графики остаются. Команда *axis on* восстанавливает изображение осей координат.

Команда *grid on* включает, а команда *grid off* выключает режим отображения решетки.

В качестве примера построим графики функций, описывающих затухания колебаний разных линейных осцилляторов:

$t = [0{:}.1{:}10]';$
$y1 = 1.03 * exp(-0.25{*}t) .* sin(0.97{*}t);$
$y2 = 1.07 * exp(-0.35{*}t) .* sin(0.94{*}t);$
$y3 = 1.15 * exp(-0.5{*}t) .* sin(0.87{*}t);$
$y4 = 0.45 * (exp(-0.38{*}t) - exp(-2.62{*}t));$
$y5 = 0.22 * (exp(-0.21{*}t) - exp(-4.80{*}t));$
$Y = [y1,\ y2,\ y3,\ y4,\ y5];$
$plot(t,\ Y)$
$ylim([-0.4,\ 0.8])$
$grid\ on$

2.1.5. Графики многозначных функций

В системе MATLAB нетрудно получить график обратной функции: достаточно в функции *plot* поменять местами аргументы. В следующем примере мы строим графики функций $y = \sin x$ и $x = \cos x$

$x = -3{*}pi{:}.01{:}3{*}pi;$
$y = cos(x);$
$plot(x,\ y,\ y,\ x);$
$legend('y = cos(x)',\ 'x = cos(y)');$

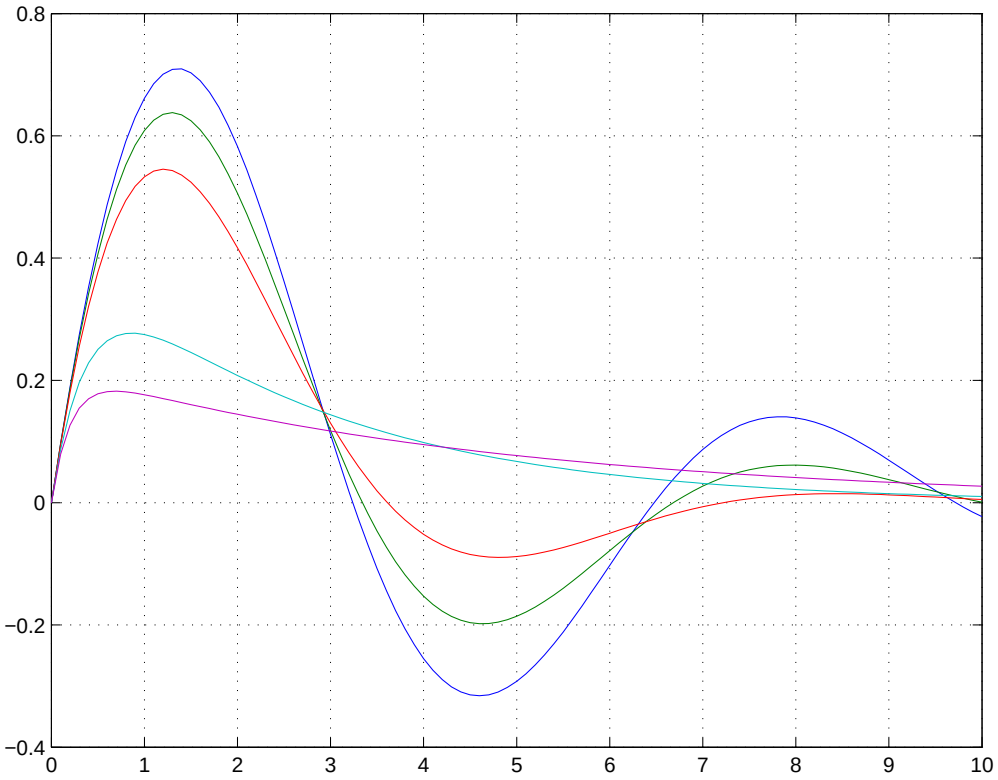


Рис. 2.6. Линейный осциллятор

2.1.6. Кривые, заданные параметрически

Функцию *plot* нетрудно использовать для отображения кривых, заданных параметрически. Например, нарисуем архимедову спираль

$$\begin{cases} x = t\cos t, \\ y = t\sin t, \end{cases} \qquad 0 \le t \le 50.$$

2.1.7. Графики в полярных координатах

Функция

$polar(phi,\ r)$

строит график функции, заданной в полярных координатах. Векторы *phi* и r должны иметь одинаковую длину и содержать значения полярного угла и радиуса соответственно.

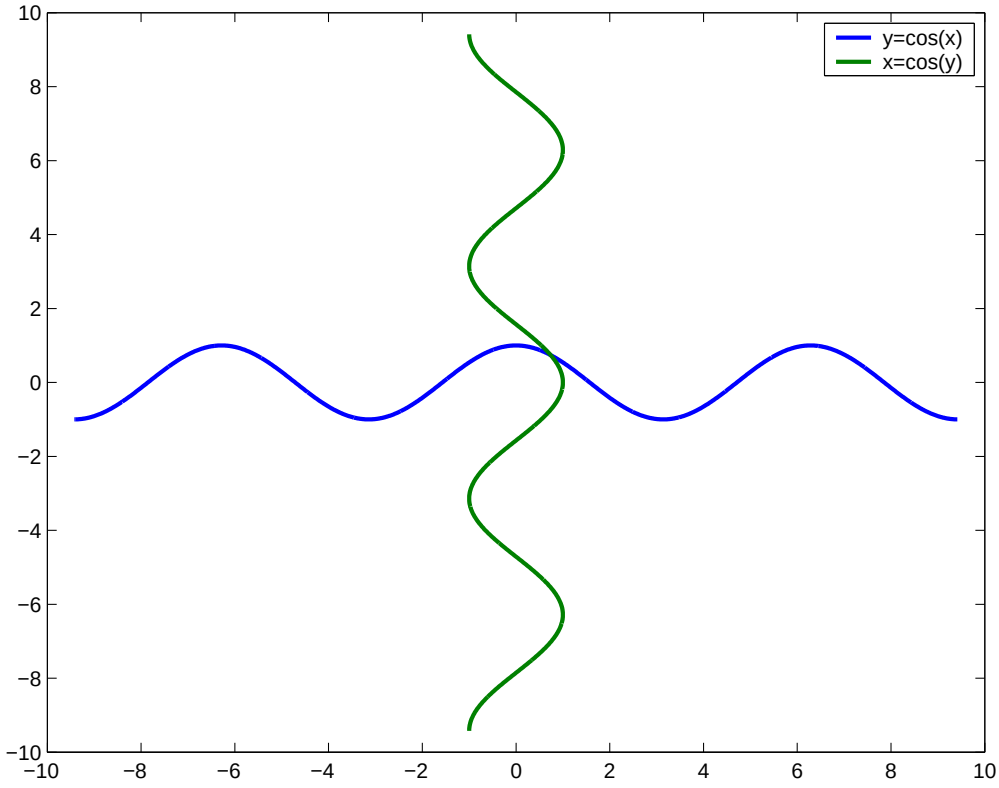


Рис. 2.7. Графики многозначных функций

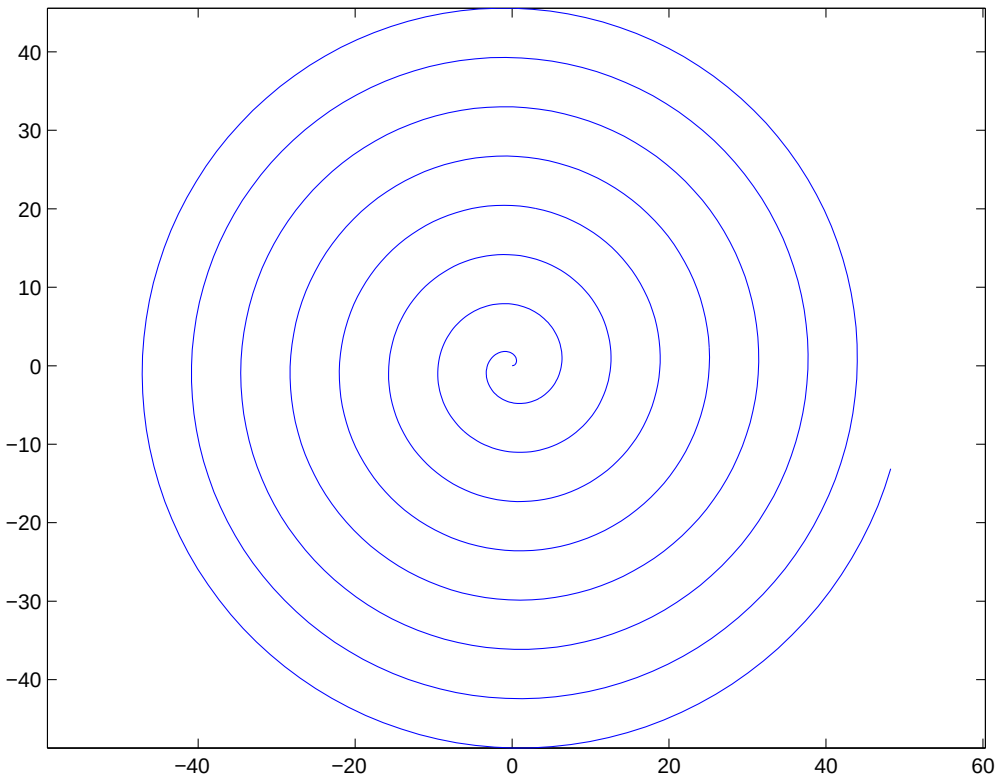


Рис. 2.8. Архимедова спираль

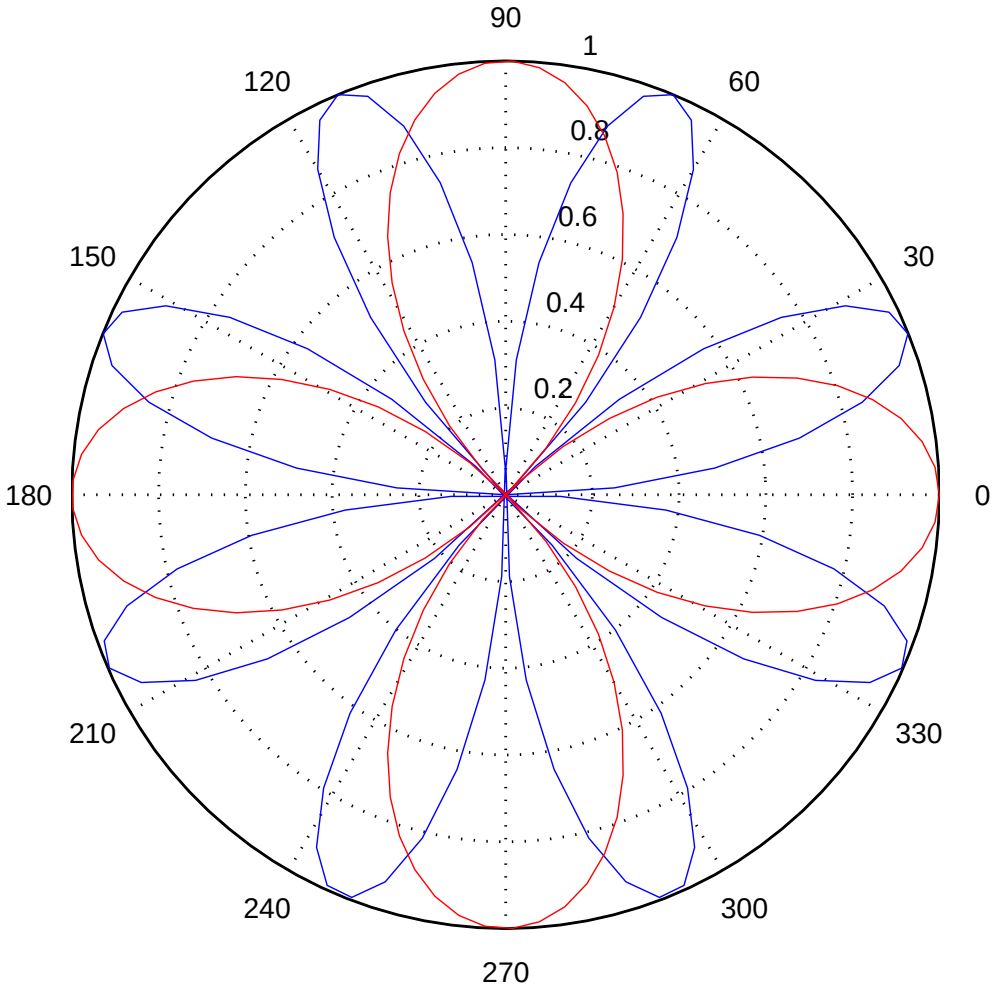


Рис. 2.9. График в полярных координатах

Например:

```
phi = linspace(0, 2*pi, 200);
polar(phi, sin(4*phi));
hold on;
polar(phi, cos(2*phi), 'r');
hold off;
```

2.2. Трехмерная графика

2.2.1. Пространственные кривые

Функция

plot3(x, y, z, стиль)

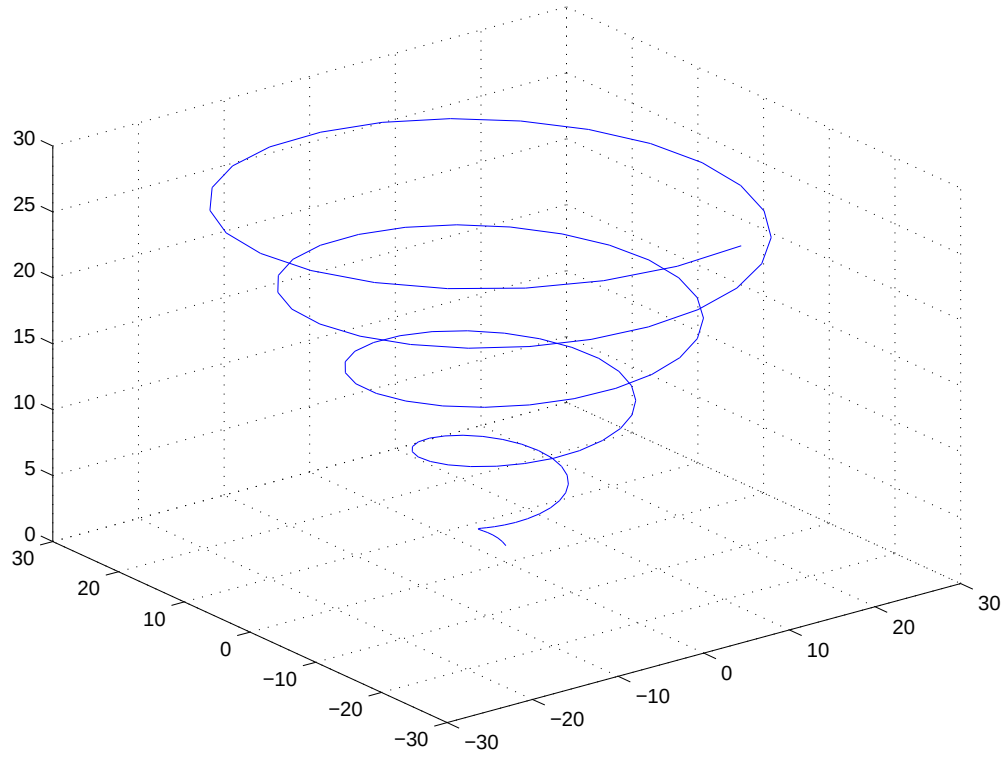


Рис. 2.10. Раскручивающаяся спираль

предназначена для построения кривых в пространстве и ее синтаксис аналогичен синтаксису функции *plot*. Нарисуем, например, раскручивающуюся спираль, заданную параметрически:

$$\begin{cases} x = t\cos(t), \\ y = t\sin(t), \qquad 1 \le t \le 30. \\ z = t, \end{cases}$$

$t = 1{:}.25{:}30;$
$x = t.{*}cos(t);$
$y = t.{*}sin(t);$
$z = t;$
$plot3(x,\ y,\ z)$
$grid$

Результат видим на рисунке 2.10.

2.2.2. Команда meshgrid

Познакомимся с функцией, незаменимой при изображении поверхностей. Пусть x и y — векторы длины n и m соответственно, тогда результатом команды

$[X,\ Y] = meshgrid(x,\ y)$

являются две матрицы размеров $m \times n$ следующего вида:

$$X = \begin{bmatrix} x(1) & x(2) & \dots & x(n) \\ x(1) & x(2) & \dots & x(n) \\ \dots\dots\dots\dots\dots\dots \\ x(1) & x(2) & \dots & x(n) \end{bmatrix}.$$

$$Y = \begin{bmatrix} y(1) & y(1) & \dots & y(1) \\ y(2) & y(2) & \dots & y(2) \\ \dots\dots\dots\dots\dots\dots \\ y(m) & y(m) & \dots & y(m) \end{bmatrix}.$$

2.2.3. Команды mesh, surf, surfl

Для рисования графиков функций двух переменных в системе MATLAB имеется несколько функций. Вот некоторые из них:

$mesh(X,\ Y,\ Z)$
$surf(X,\ Y,\ Z)$
$surfl(X,\ Y,\ Z)$

Параметры всех трех приведенных функций имеют одинаковый смысл: X, Y, Z — это двумерные массивы, определяющие координаты x, y, z поверхности.

Функция *mesh* рисует сетчатую (проволочную) поверхность, соединяя прямыми отрезками «соседние» точки, т. е. точки, координаты которых расположены либо в соседних строках, либо в соседних столбцах матриц X, Y, Z (см. рис. 2.2.3).

$[X,\ Y] = meshgrid(-3{:}.25{:}3,\ -3{:}.25{:}3);$
$Z = sin(X).*sin(Y);$
$mesh(X,\ Y,\ Z);$

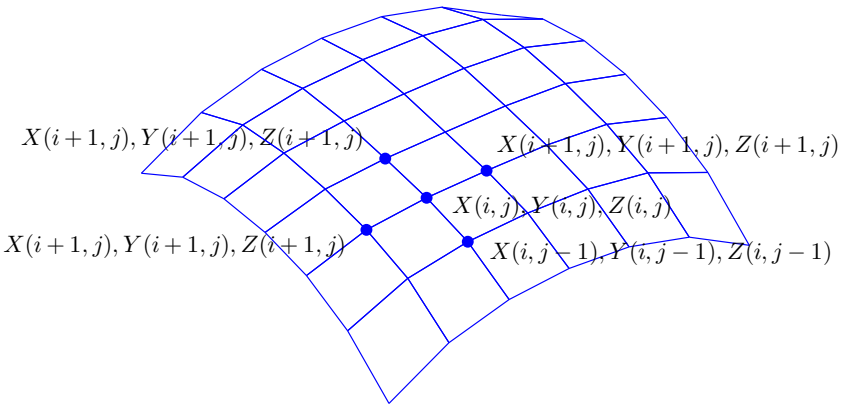


Рис. 2.11. Параметры X, Y, Z функции mesh

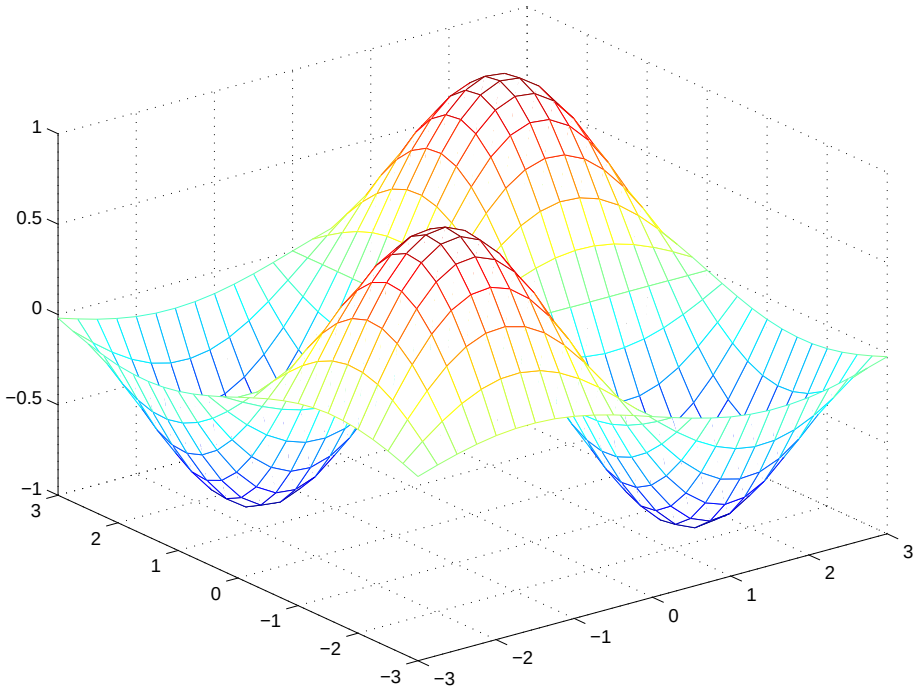


Рис. 2.12. Сеточная модель. Невидимые линии удалены

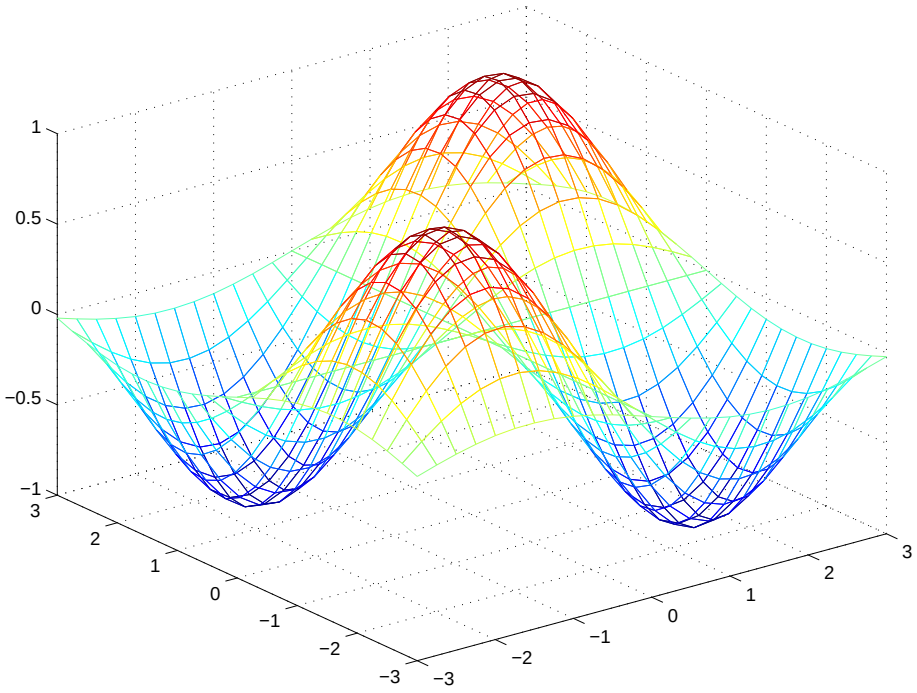


Рис. 2.13. Проволочная модель поверхности

По умолчанию невидимые линии не отображаются. Команда

hidden off

отключает режим удаления невидимых линий

Сплошную (закрашенную) поверхность рисует функция

surf(*X*, *Y*, *Z*)

Команда

colormap палитра

позволяет выбрать для закраски поверхности определенный набор цветов, называемый *палитрой*. Доступны следующие палитры: *winter*, *spring*, *summer*, *autumn*, *bone*, *copper*, *hot*, *cool*, *gray*, *pink* и др.

Задать способ окраски поверхности можно с помощью следующих команд:

shading faceted
shading flat
shading interp

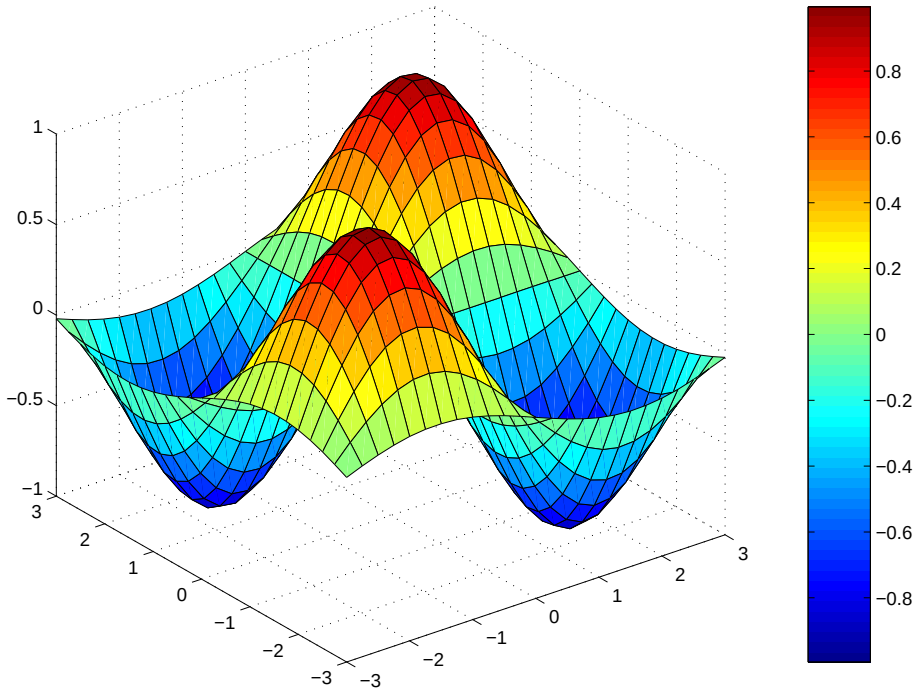


Рис. 2.14. surf

shading faceted окрашивает каждый четырехугольник (грань), из которых составлена поверхность, одним цветом. Ребра окрашиваются черным цветом. Этот режим действует по умолчанию. Действие *shading faceted* аналогично действию предыдущей команды, но при этом ребра не видимы. Команда *shading interp* каждую грань закрашивает неравномерно, используя линейную интерполяцию цвета.

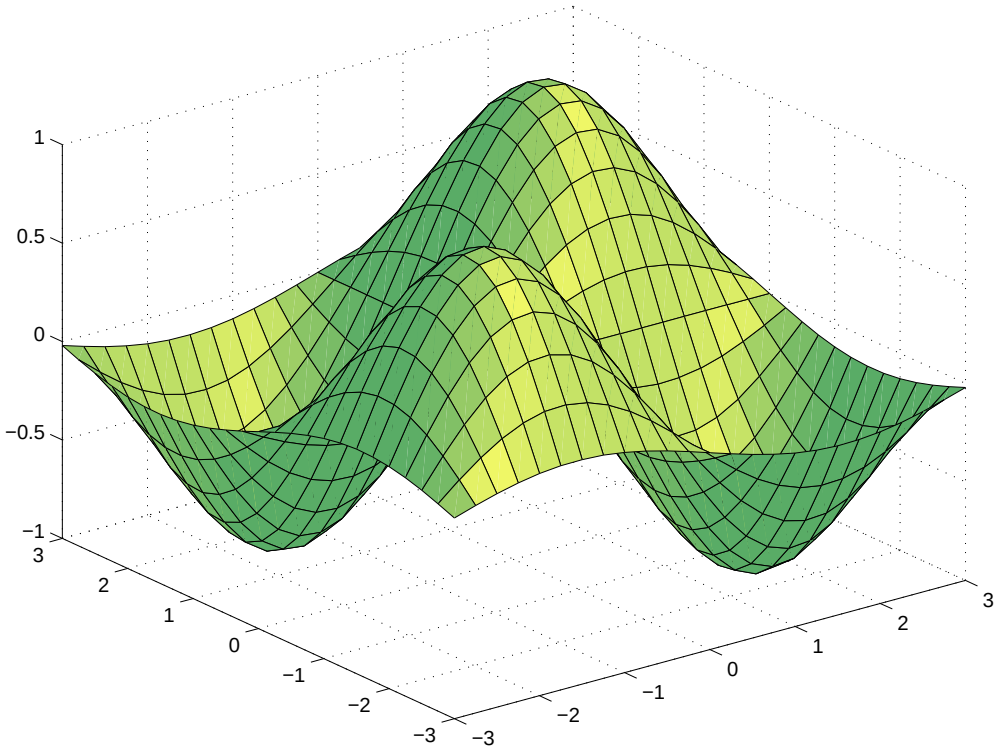
Команда

alpha a

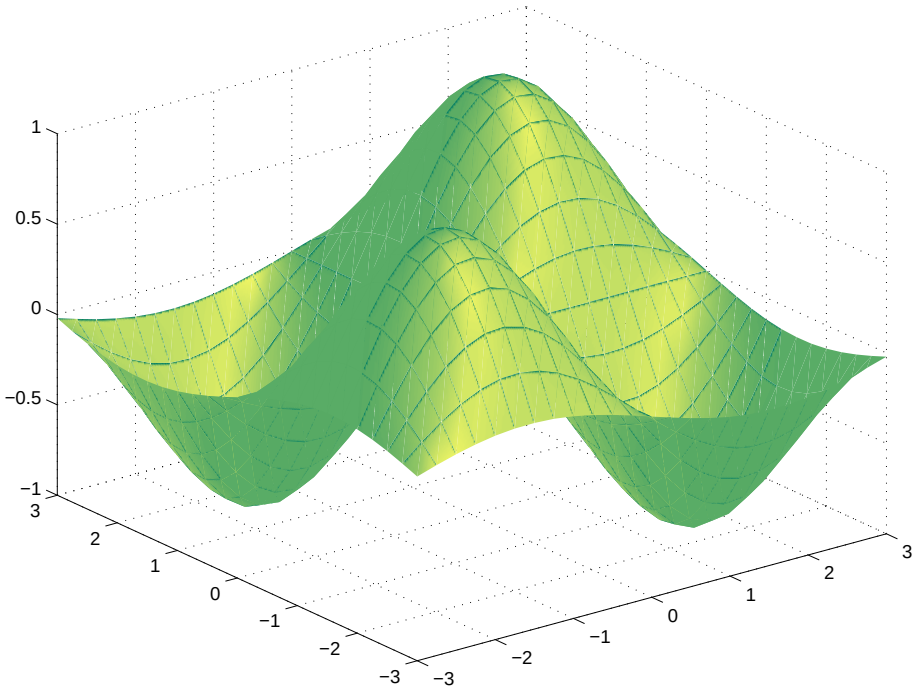
позволяет задать степень прозрачности поверхности. Значение $a = 1$ соответсвует непрозрасной поверхности. Значение $a = 0$ — полностью прозрачной (невидимой).

Рассмотрим пример:

$[X,\ Y] = meshgrid(-2{:}.02{:}2);$
$Z = sin(X.\hat{}2 + Y.\hat{}2) + exp(-X.\hat{}2);$
$surf(X,\ Y,\ Z)$
axis off
shading interp
alpha .8
$view(-17,\ 19)$



Puc. 2.15. shading faceted



Puc. 2.16. shading flat

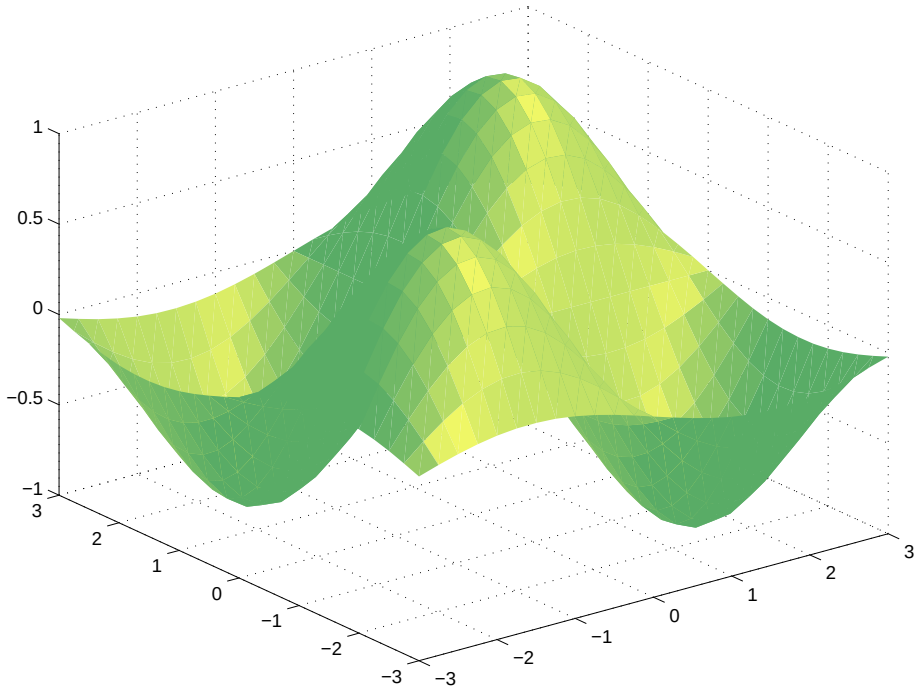


Рис. 2.17. shading interp

Результат приведен на рис. 2.18.

Функция *surfl*$(X,\ Y,\ Z)$ аналогична функции *surf*$(X,\ Y,\ Z)$, но в ней для отрисовки используются цвета, имитирующие освещение. Функцию *surf* рекомендуется использовать вместе с палитрами *gray*, *bone*, *copper*, *pink*.

Функция *view*$(az,\ el)$ определяет точку нахождения камеры (наблюдателя) Азимут az — угол поворота камеры (в градусах) вокруг оси Oz, измеряемый от отрицательного направления оси Oy. Положительные значения азимута соответствуют повороту против часовой стрелки. Возвышение el — это угол (в градусах), который составляет вектор идущий из начала коорлинат к камере, с плоскостью Oxy. Положительные значения возвышения соответствуют точкам над плоскостью Oxy, отрицательные значения — точкам под плоскостью Oxy.

2.3. Примеры

Рассмотрим еще несколько примеров.

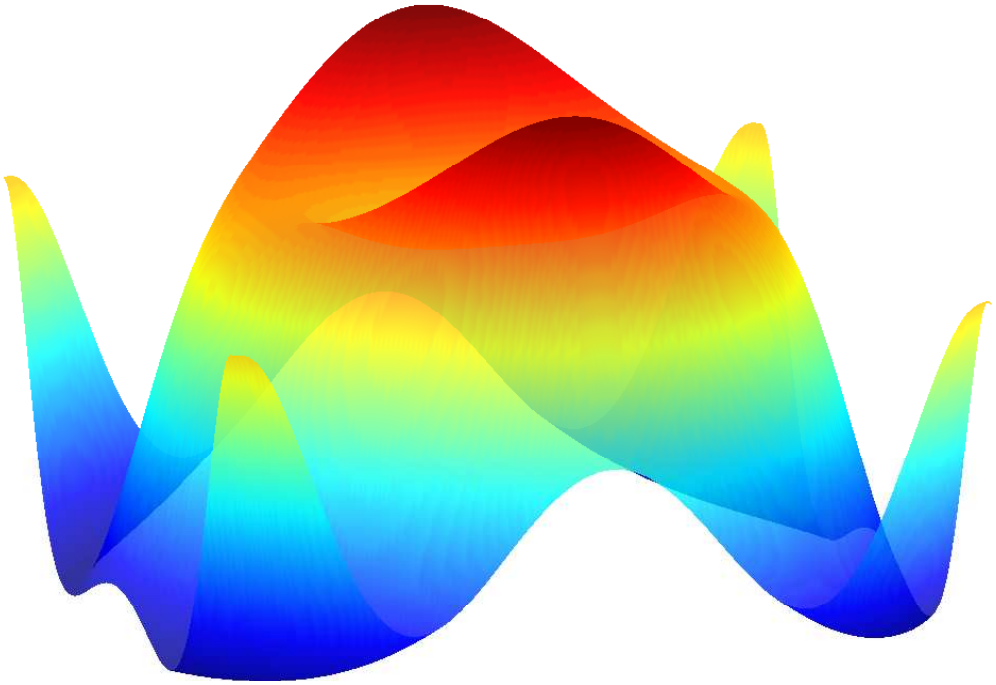


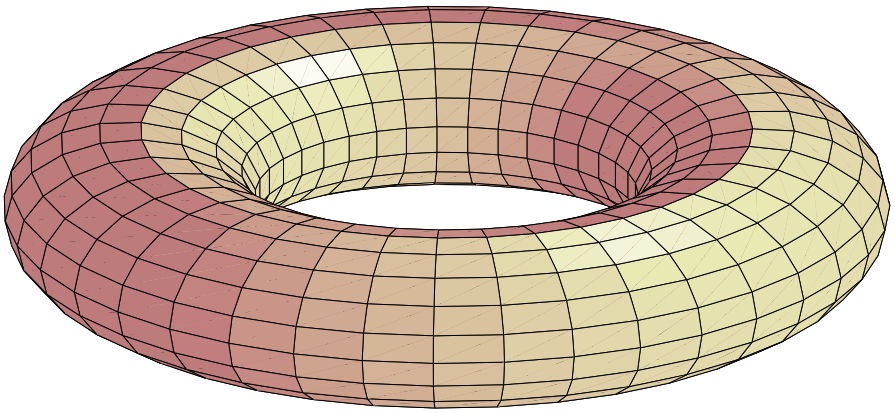
Рис. 2.18. Полупрозрачная поверхность

2.3.1. Тор

Тор можно задать параметрически:

$$\begin{cases} x = (R + r\cos v)\cos u, \\ y = (R + r\cos v)\sin u, \\ z = r\sin v, \end{cases} \qquad (0 \le u \le 2\pi,\ 0 \le v \le 2\pi),$$

где r — «малый», а R — «большой» радиусы тора.



Puc. 2.19. Top

$R = 5;$
$r = 2;$
$n = 40;$
$m = 20;$

$[U,\ V] = meshgrid(linspace(0,\ 2*pi,\ n),\ linspace(0,\ 2*pi,\ m));$

$X = (R + r.*cos(V)).*cos(U);$
$Y = (R + r.*cos(V)).*sin(U);$
$Z = r.*sin(V);$

$surfl(X,\ Y,\ Z)$
$axis([-5\ 5\ -5\ 5\ -5\ 5])$
$axis\ \ off$
$colormap\ \ pink$

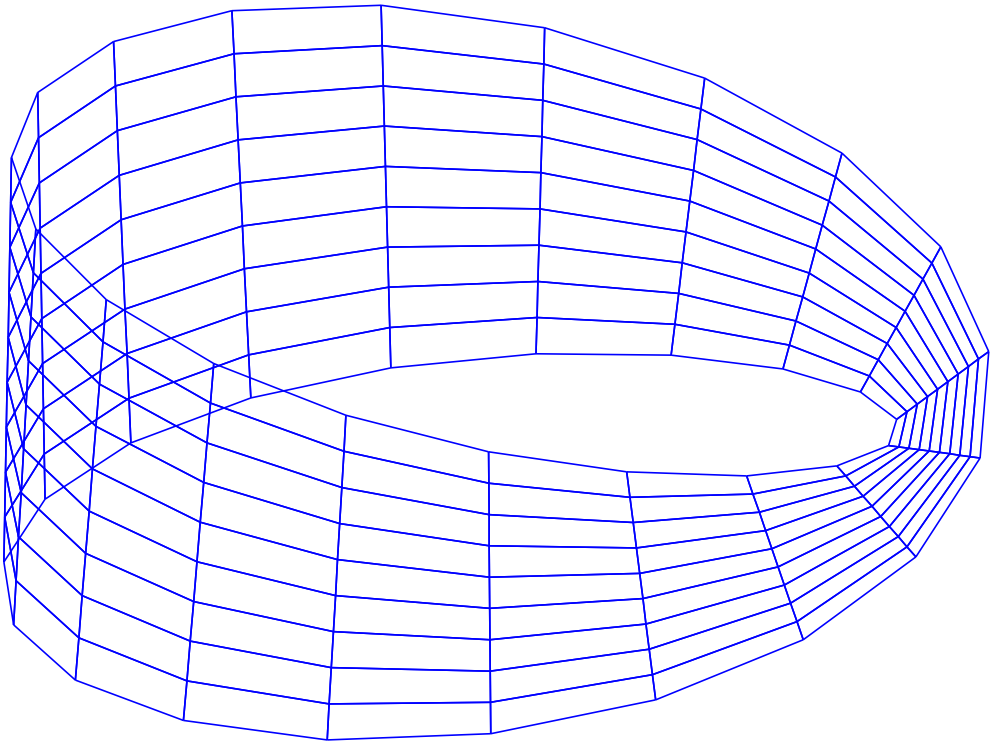


Рис. 2.20. Лист Мебиуса

2.3.2. Лист Мебиуса

Лист Мебиуса можно задать параметрически, например:

$$\begin{cases} x = \cos u + v\cos u/2. * \cos u, \\ y = \sin u + v\cos u/2. * \sin u, \qquad (0 \le u \le 2\pi,\ -h \le v \le h), \\ z = v\sin u/2 \end{cases}$$

```
[u, v] = meshgrid(linspace(0, 2*pi, 20), linspace(-.1, .1, 10));
mesh( ...
    cos(u) + v.*cos(u/2).*cos(u), ...
    sin(u) + v.*cos(u/2).*sin(u), ...
    v.*sin(u/2), ...
    'Edgecolor', 'blue');
view(15, 56);
axis off;
hidden off;
```

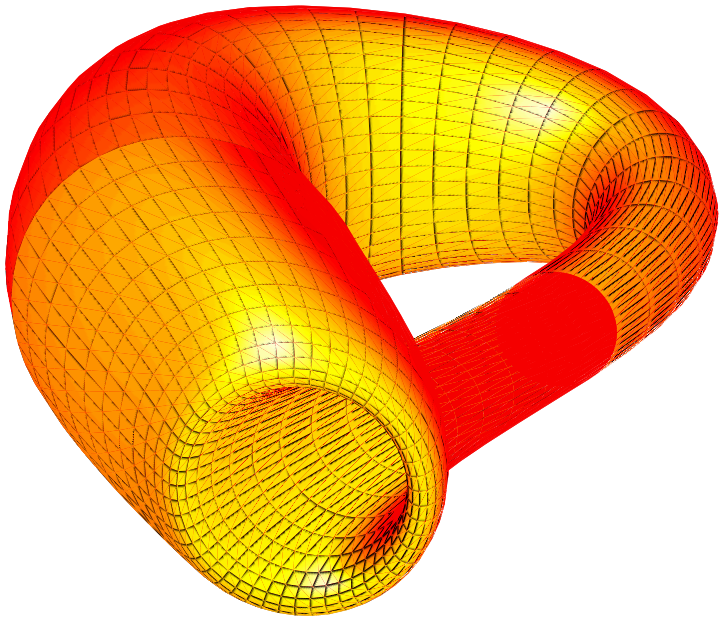


Рис. 2.21. Бутылка Клейна

2.3.3. Бутылка Клейна

Бутылку Клейна можно склеить из двух поверхностей, каждую из которых удается задать параметрически:

$$\begin{cases} x = a\cos u(1+\sin u) + r\cos u\cos v, \\ y = b\sin u + r\sin u\cos v, \\ z = r\sin v \end{cases} \qquad (0 \le u \le \pi,\ 0 \le v \le 2\pi),$$

$$\begin{cases} x = a\cos u(1+\sin u) + r\cos(v+\pi), \\ y = b\sin u, \\ z = r\sin v \end{cases} \qquad (\pi \le u \le 2\pi,\ 0 \le v \le 2\pi),$$

где

$$r = 1 - \cos\frac{u}{2}.$$

```
[u, v] = meshgrid(linspace(0, pi, 25), linspace(0, 2*pi, 50));

r1 = 4*(1 - cos(u)/2);
x1 = 6*cos(u).*(1 + sin(u)) + r1.*cos(u).*cos(v);
y1 = 16*sin(u) + r1.*sin(u).*cos(v);
z1 = r1.*sin(v);

[u, v] = meshgrid(linspace(pi, 2*pi, 25), linspace(0, 2*pi, 50));

r2 = 4*(1 - cos(u)/2);
x2 = 6*cos(u).*(1 + sin(u)) + r2.*cos(v + pi);
y2 = 16*sin(u);
z2 = r2.*sin(v);

surfl(x1, y1, z1)
hold on
surfl(x2, y2, z2)
view(9, 21)
colormap hot
shading interp
hold off
axis equal
axis off
```

2.4. Линии уровня

Функция

```
contour(X, Y, Z);
```

рисует линии уровня для функции $z = f(x, y)$, значения которой хранятся в матрицах X, Y, Z. Например:

```
[X, Y] = meshgrid(−3:.25:3, −3:.25:3);
Z = sin(X).*sin(Y);
contour(X, Y, Z);
```

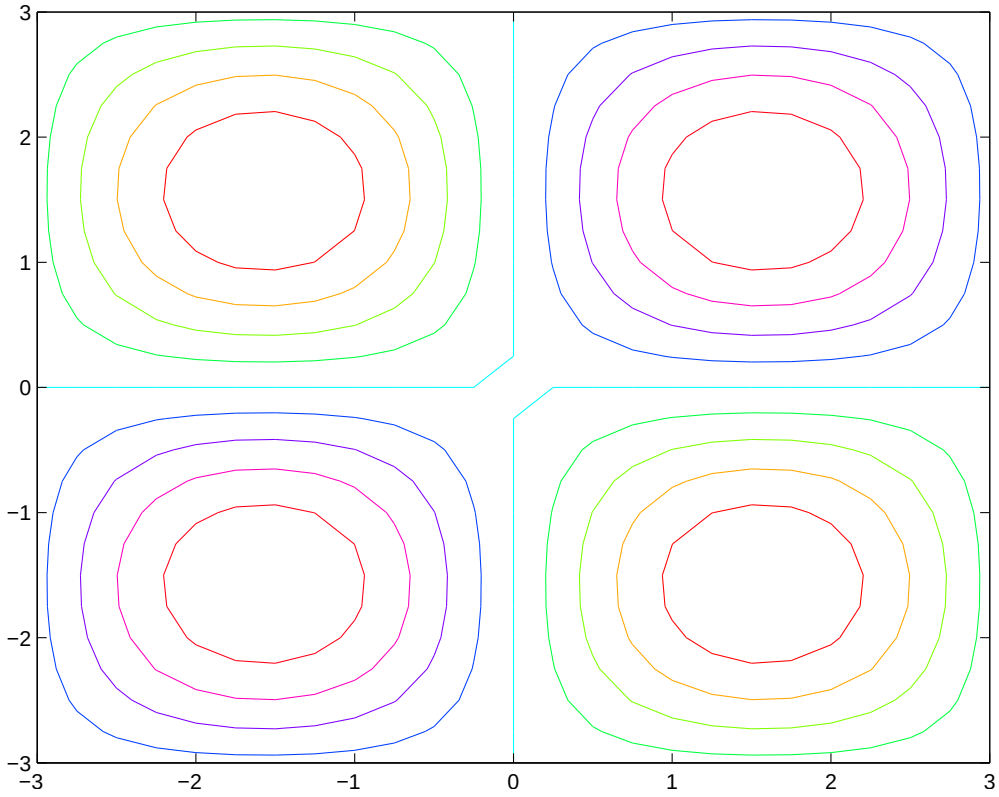


Рис. 2.22. Линии уровня

Количество и величины значений функции, для которых будут нарисованы линии уровня, определяются автоматически. Задать нужное количество k линий уровня можно, определив значение четвертого аргумента:

$$contour(X,\ Y,\ Z,\ k);$$

Если четвертый аргумент — вектор, то он интерпретируется как набор значений функции, для которых должны быть построены линии уровня.

Функции

$$contourf(X,\ Y,\ Z);$$
$$contour(X,\ Y,\ Z,\ k);$$

рисуют линии уровня и закрашивают промежутки между ними в цвета из текущей палитры. Рассмотрим пример с функцией

$$z = (x^2 + y^2)^3 - 4x^2y^2.$$

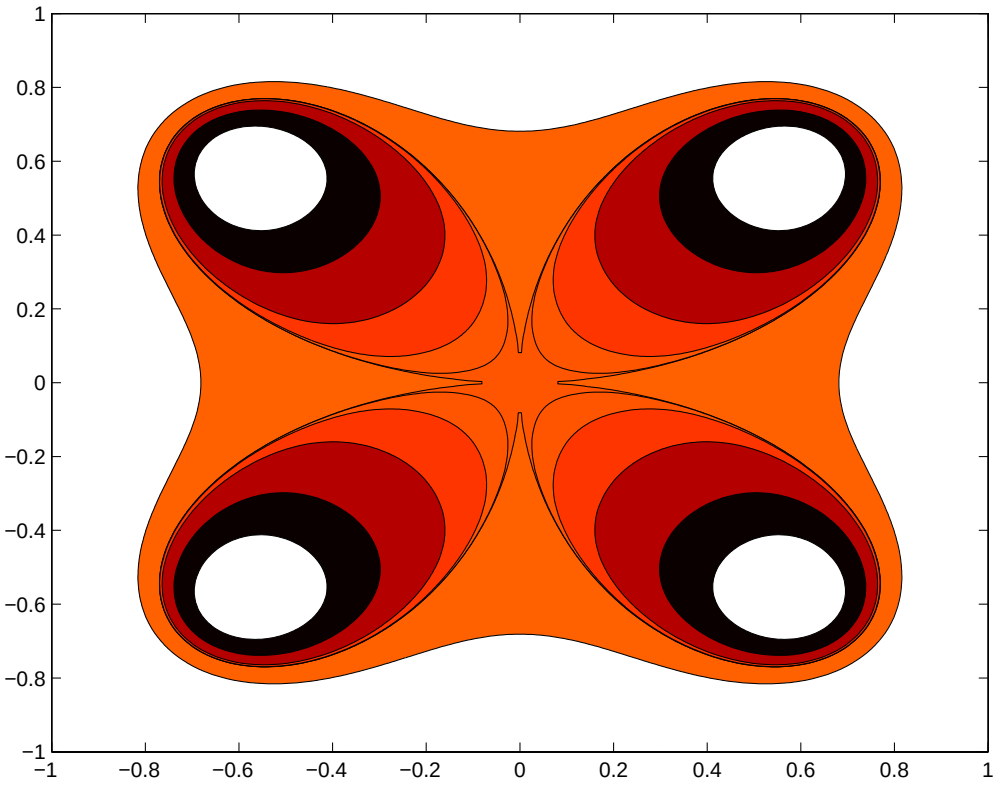


Рис. 2.23. Линии уровня с закрашенными промежутками

$[X,\ Y] = meshgrid(linspace(-1,\ 1,\ 300));$
$Z = (X.\hat{}2 + Y.\hat{}2).\hat{}3 - 4*X.\hat{}2.*Y.\hat{}2;$
$contourf(X,\ Y,\ Z,\ [-0.1\ -0.05\ -0.01\ -0.001\ -0.00005\ 0\ 0.1])$
$colormap\ hot$

С помощью функции *contour* можно строить кривые, заданные уравнениями. Для предыдущего примера

$contour(X,\ Y,\ Z,\ [0\ 0]);$

рисует четырехлепестковую розу $(x^2+y^2)^3 - 4x^2y^2 = 0$. Нам пришлось продублировать 0 в векторе, являющимся последним аргументом в функции *contour*, так как

$contour(X,\ Y,\ Z,\ 0);$

было бы проинтерпретировано, как запрос построить $k = 0$ линий уровня.

2.5. Make it easier

Если известно аналитическое задание функции в виде обыкновенного или параметрического уравнения, то иногда удобно воспользоваться ez-вариантом соответствующей

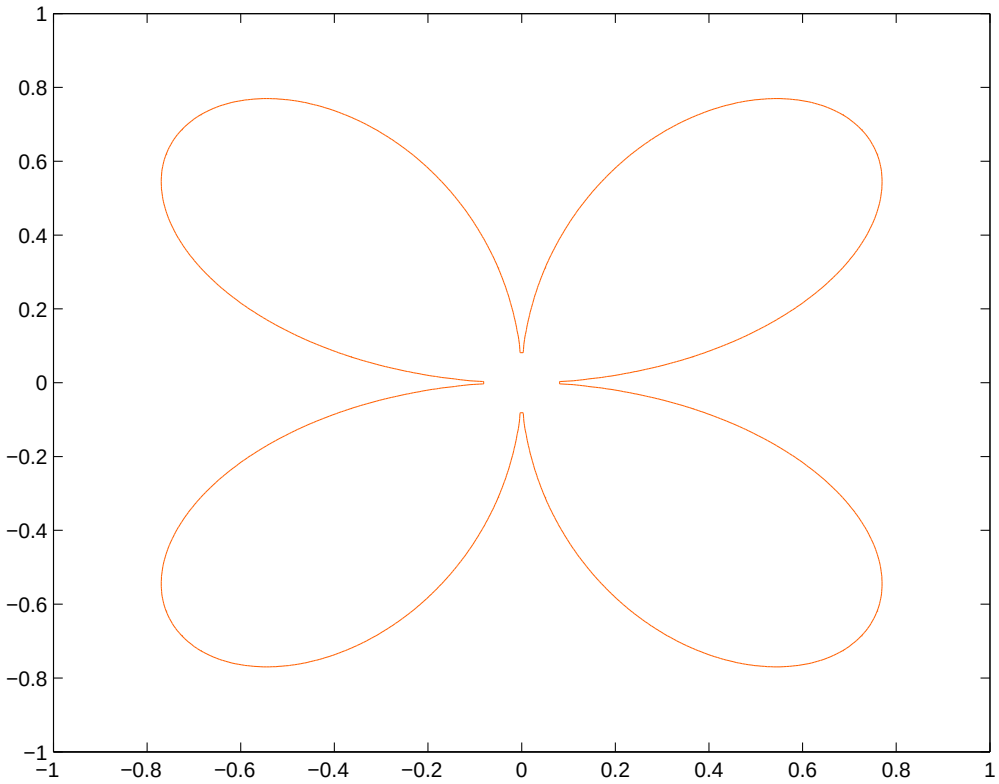


Рис. 2.24. Четырехлепестковая роза. Кривая задана уравнением $(x^2+y^2)^3 - 4x^2y^2 = 0$.

графической команды. К таким ez-функциям относятся *ezplot*, *ezpolar*, *ezplot3*, *ezmesh*, *ezsurf*, *ezcontour*. Используя их, пользователь не должен заботится о предварительной табуляции функции. Выбор узлов и сами вычисления значений функции в этих узлах проводятся автоматически.

Проиллюстрируем использование ez-функций на примерах.

Построим график функции $y = \cos(\operatorname{tg} x)$, $-3 \le x \le 3$:

ezplot('*cos*(*tan*(*x*))', −3, 3)

Нарисуем окружность $x^2 + y^2 = 1$:

ezplot('*x*^2 + *y*^2 − 1')

Изобразим кривую $x = \sin y$:

ezplot('*x* − *sin*(*y*)')

Нарисуем кривую, заданную параметрически,

$$\begin{cases} x = \frac{\sin t}{t}, \\ y = \frac{\cos t}{t}, \end{cases} \qquad 1 \le t \le 10\pi :$$

ezplot('*sin*(*t*)/*t*', '*cos*(*t*)/*t*', [1, 10**pi*])

Нарисуем пространственную кривую, заданную параметрически,

$$\begin{cases} x = t\sin t, \\ y = t\cos t, \qquad 0 \le t \le 6\pi : \\ z = t. \end{cases}$$

ezplot3('*t***sin*(*t*)', '*t***cos*(*t*)', '*t*', [0, 6**pi*])

Построим графики и линии уровня для функции

$$z = \sin(x^2 + y^2) + e^{-x^2}, \qquad -2 \le x \le 2, \quad -2 \le y \le 2 :$$

```
f = 'sin(x^2 + y^2) + exp(-x^2)';
d = [-2, 2, -2, 2];
ezmesh(f, d);
ezsurf(f, d);
ezcontour(f, d);
```

Нарисуем кривую

$$r = e^{\cos\varphi} - 2\cos 4\varphi + \left(\sin\frac{\varphi}{12}\right)^5, \qquad 0 \le \varphi \le 6\pi$$

в полярной системе координат:

ezpolar('*exp*(*cos*(*phi*)) − 2**cos*(4**phi*) + *sin*(*phi*/12)^5', [0, 6**pi*])

3. Программирование

3.1. Типы данных

MATLAB поддерживает работу с различными типами данных. С двумя мы уже познакомились. Это тип *double*, представляющий матрицы, элементы которых — действительные или комплексные числа с плавающей запятой двойной точности. Другой тип — *char* — представляет массивы символов.

3.1.1. Разреженные матрицы

Система MATLAB обеспечивает эффективную работу с разреженными матрицами. *Разреженная матрица* — матрица, у которой «много» нулевых элементов. В структурах данных, представляющих такие матрицы, хранится информация только о ненулевых элементах: их значения и положение. Это позволяет использовать меньше памяти и получать более производительные алгоритмы.

Над разреженными, как и над плотными, матрицами могут совершаться любые арифметические, логические и индексные операции и многие функции. Возможно выполнение смешанных операций — над плотными и разреженными матрицами. Операции над однородными операндами возвращают матрицу того же типа. Операции над смешанными операндами, как правило, приводят к плотным матрицам, хотя в некоторых случаях, когда разреженность сохраняется, возвращается разреженная матрица. Например, если S — разреженная, а A — плотная, то $S + A$, $S * A$, $S \setminus A$ — плотные, а $S .* A$, $S \,\&\, A$ — разреженные. В некоторых случаях результат может быть разреженным, даже если матрица содержит много ненулевых элементов.

Функция

$$S = sparse(A)$$

преобразует плотную матрицу A в разреженную S. Функция

$$A = full(S)$$

осуществляет обратное преобразование. Функция

$$S = sparse(i,\ j,\ s,\ m,\ n)$$

создает разреженную матрицу S размера $m \times n$, ненулевые элементы которой перечислены в массиве s, а их позиции, т. е. номера строк и столбцов — в массивах i и j соответственно. Обращение вида

$$S = sparse(i,\ j,\ s)$$

предполагает, что $m = max(i)$, $n = max(j)$. Функция

$$S = sparse(m,\ n)$$

создает нулевую разреженную матрицу заданного размера.

Чтобы создать единичную разреженную матрицу, воспользуйтесь функцией $speye(m,\ n)$ или $speye(n)$. В первом случае вы получите прямоугольную матрицу размера $m \times n$, во втором — квадратную матрицу порядка n.

Функция $spones(S)$ возвращает матрицу с тем же портретом, что и у S, но с единицами вместо ненулевых элементов.

Функция $nnz(S)$ возвращает число ненулевых элементов матрицы S. Функция $spy(S)$ отображает портрет матрицы S.

3.1.2. Многомерные массивы

MATLAB поддерживает работу с многомерными массивами. Например,

$ones(3,3,3)$

создает трехмерный массив размеров $3 \times 3 \times 3$, заполненный единицами, а команда

$zeros(1,2,3,4,5)$

создает пятимерный массив размеров $1 \times 2 \times 3 \times 4 \times 5$, заполненный нулями.

Доступны команды [], :, **end**, *meshgrid* и т. п.

В следующем примере формируется нулевой массив размера $4 \times 4 \times 24$. Затем с помощью функции *magic* строится магический квадрат порядка 4. Функция *perms* находит все перестановки элементов вектора 1:4. Далее в цикле в каждый «слой» $M(:,\ :,\ k)$ массива M записывается магический квадрата, с переставленными столбцами. Нетрудно видеть, что всякий раз будут снова получаться магические квадрата. Это можно проверить с помощью Функций $sum(M,\ 1)$ и $sum(M,\ 2)$, вычисляющие суммы элементов массива вдоль указанных размерностей.

```
M = zeros(4, 4, 24);
size(M)
A = magic(4);
p = perms(1:4);
for k = 1:24
  M(:, :, k) = A(:, p(k, :));
end
M(:, :, 3);
sum(M, 1)
sum(M, 2)
```

3.1.3. Массивы структур

Структурой называется абстрактный тип данных, представляющий собой коллекцию значений (*полей*) разных типов, доступ к которым осуществляется по имени (*имени поля*).

Массив структур — это коллекция структур, доступ к которым происходит по индексу. Возможны многомерные массивы структур.

В MATLAB'е не нужны предварительные объявления структур. Чтобы создать ее, нужно просто указать значения соответсвующих полей. Имя поля отделяется от имени переменной–структуры точкой:

```
S.name = 'Isaac Newton';
S.age = 38;
```

Мы получили структуру (а точнее массив структур 1×1) со следующими полями и значениями полей:

name	*age*
'Isaac Newton'	38

Как мы видели, набор полей структуры может изменяться динамически. Также динамически могут меняться размеры массива структур:

```
S(2).name = 'Blaise Pascal' ;
S(2).age = 23;
```

Теперь S — это массив структур размера 1×2:

№	*name*	*age*
1	‘Isaac Newton’	38
2	‘Blaise Pascal’	23

С помощью функции *struct* можно задать знаяения сразу нескольким полям структуры:

```
S(3) = struct('name', 'Carl F. Gauss', 'age', 43); %]
```

Имеем:

```
\begin{center}
\begin{tabular}{|c|c|c|}
\hline
\No & {\it name} & {\it age} \\
\hline
1 & ‘Isaac Newton’ & 38 \\
\hline
2 & ‘Blaise Pascal’ & 23 \\
\hline
3 & ‘Carl F. Gauss’ & 43 \\
\hline
\end{tabular}
\end{center}
```

Добавим еще одно поле:

```
%[
    S(3).profession = 'mathematician'
```

Получим:

№	*name*	*age*	*profession*
1	‘Isaac Newton’	38	[]
2	‘Blaise Pascal’	23	[]
3	‘Carl F. Gauss’	43	‘mathematician’

3.1.4. Массивы ячеек

Ячейкой (cell) называется контейнер, который может содержать в себе объект про-

извольного типов данных (т. е. это может быть массив чисел с плавающей запятой, массив символов, массив структур и др.)

Массив ячеек — это коллекция ячеек, доступ к которым происходит по индексу. Таким образом, массив ячеек может объединять разнотипные данные. Массив может быть одномерным, двумерным или многомерным.

Доступ к ячейкам осуществляется указанием после имени массива индекса элемента в *фигурных* скобках.

```
for n = 1:5
    M{n} = hadamard(2^n);
end
M{2}
```

Другой способ создать массив ячеек — это перечислить его элементы построчно, разделяя элементы в одной строке пробелами или запятыми, а сами строки — точкой с запятой или символом перехода на новую строку. Все элементы должны быть заключены в фигурные скобки. Например,

```
A = hadamard(4)
M = {A, sum(A), prod(A), 'matrix A'}
```

Чтобы создать массив пустых ячеек достаточно воспользоваться функцией `cell` с указанием размеров массива:

```
M = cell(5, 2, 3)
```

Одно из самых распространенных применений массива ячеек — его использования для хранения символьных строк. Например,

```
M = {'Isaac Newton', 'Blaise Pascal', 'Carl F. Gauss', 'Nikolai I. Lobachevski'}
```

Обращение к элементам массива ячеек с помощью индекса (или индексов, а также индексных выражений), заключенных в *круглые* скобки, приводит к созданию *срезов* массивов. Для последнего примера $M\{2\}$ — это строка *'Blaise Pascal'*, а $M(2)$ — массив ячеек 1×1, содержащий строку *'Blaise Pascal'*. Другой пример:

```
M(2:end)
```

— это массив ячеек {*'Blaise Pascal'*, *'Carl F. Gauss'*, *'Nikolai I. Lobachevski'*}.

Заметим, что, выражения вида $M\{2$:**end**$\}$ и т. п. (индексное выражение стоит в *фигурных* скобках) также возможны. Их результатом являются уже не срезы, а *списки*

значений. Они могут появляться в списках элементов массивов, в списках входных и выходных аргументов функций и др.

Например,

$M = \{225,\ 13,\ 49\};$
$A = [M\{:\}];$

эквивалентно

$A = [225,\ 13,\ 49];$

а

$args = \{x,\ y,\ 'b-'\};$
$plot(args\{:\});$

равносильно

$plot(x,\ y,\ 'b-');$

3.2. Управляющие конструкции

В любом языке программирования, в том числе в языке, предоставляемом системой MATLAB, есть специальные конструкции, предназначенные для управления порядком выпонения команд. Такие конструкции иногда называют управляющими операторами.

В MATLAB'е к ним относятся:

- условный оператор **if**,
- оператор цикла **while**,
- оператор цикла с параметром **for**,
- оператор выбора **switch**.

3.2.1. Оператор if

Оператор **if** — это оператор ветвления. Самая простейшая его форма:

```
if условие
    команды
end
```

Проверяется заданное условие. Если оно выполнено, то выполняются команды, следующие за этим условием. Если не выполнено, то управление передается командам после оператора **if** (после ключевого слова **end**). условие можно получить в результате логических операций «меньше» $<$, «больше» $>$, «равно» $==$, «меньше или равно» $<=$, «больше или равно» $>=$ «не равно» $\tilde{}=$, «и» &, «или» |, «не» $\tilde{}$.

Например, в следующем фрагменте, на экране печатается x — целое, если x — целое число:

```
if fix(x) == x
    disp('x — целое')
end;
```

Возможен более развернутый вариант оператора **if**:

```
if условие
    команды1
else
    команды2
end
```

Проверяется условие. Если оно выполнено, то выполняется блок команд 1. Если не выполнено, то выполняется блок команд 2.

Например, в следующем фрагменте, на экране печатается x — целое, если x — целое число; в противном случае будет напечатано x — дробное:

```
if fix(x) == x
    disp('x — целое')
else
    disp('x — дробное')
end;
```

Операторы **if** могут быть вложенными. В следующем фрагменте программы слово «ворона» печатается в нужно числе и падеже в зависимости от значения переменной x,

в котором хранится количество ворон: 1 ворона, 2 вороны, 3 вороны и т. д.

```
if rem(fix(x/10), 10) == 1
    disp([num2str(x) ' ворон']);
else
    if rem(x, 10) == 1
        disp([num2str(x) ' ворона']);
    else
        if rem(x, 10) >= 2 && rem(x, 10) <= 4
            disp([num2str(x) ' вороны']);
        else
            disp([num2str(x) ' ворон']);
        end;
    end;
end;
```

Самая общая схема использования оператора **if** следующая:

```
if условие1
   команды1
elseif условие2
   команды2
elseif условие3
   команды3
...
else
   команды
end
```

Вначале проверяется условие 1. Если оно выполнено, то выполняется блок команд 1. Если не выполнено, то проверяется условие 2. Если оно выполнено, то выполняется блок команд 2. В противном случае проверяется условие 3 и т. д. Если ни одно из условий не выполнено, выполняется блок команд, следующий за ключевым словом **else**.

Пример с воронами лучше переписать с использованием такого расширенного вари-

анта оператора **if**:

```
if rem(fix(x/10), 10) == 1
    disp([num2str(x) ' ворон']);
elseif rem(x, 10) == 1
    disp([num2str(x) ' ворона']);
elseif rem(x, 10) >= 2 && rem(x, 10) <= 4
    disp([num2str(x) ' вороны']);
else
    disp([num2str(x) ' ворон']);
end;
```

Возможен вариант оператора **if** без последнего блока **else** команды:

```
if условие1
   команды1
elseif условие2
   команды2
elseif условие3
   команды3
...
end
```

3.2.2. Оператор while

Оператор **while** используется в составе следующей конструкции:

```
while условие
  команды
end
```

Вначале проверяется условие. Если оно выполнено, то выполняются команды внутри тела цикла. Далее снова проверяется условие, и если оно выполнено, снова выполняются команды в теле цикла и т. д. до тех пор, пока не выполнится условие. Как только условие перестанет быть выполненным, произойдет выход из цикла и управление будет передано следующим за блоком **while** (за ключевым словом **end**) командам.

В качестве примера вычислим константу *eps*:

```
e = 1;
while 1 + e ~= 1,
   e = e/2;
end;
e = 2*e
```

3.2.3. Оператор for

Оператор **for** используется в составе следующей конструкции:

```
for переменная = выражение
   команды
end
```

Если результат вычисления выражения — вектор, то указанной переменной по очереди будет присвоена каждая из компонент этого и вектора и вскияй раз будут выполнены команды, расположенные в теле цикла.

Для примера вычислим матрицу Гильберта:

```
n = 10;
H = zeros(n,n);
for i=1:n
   for j=1:n
      H(i,j)=1/(i+j-1);
   end;
end;
H  % матрица Гильберта
```

Если результат вычисления выражения — матрица, то указанной переменной по очереди будет присвоен каждый из столбцов этой матрицы.

3.2.4. Оператор switch

Оператор **switch** — это оператор выбора. Схема его использования следующая:

switch выражение
case {список значений 1}
    команды1
case {список значений 2}
    команды2
...
otherwise
    команды
end

Вычисляется выражение и по очереди сравнивается с перечисленными после ключевых слов **case** значениями. Если найдено совпадение, то выполняется соответствующий блок команд. После этого управление передается на следующую после блока (после ключевого слова **end**) команду. Если совпадений не найдено, выполняются команды за ключевым словом **otherwise**. Блок **otherwise** команды может отсутствовать. Значения в фигурных скобках разделяются запятыми. В случае, если какой-либо список содержит только одно значение, то фигурные скобки можно опустить.

Рассмотрим пример:

```
switch lower(method)
  case {'linear', 'bilinear'}
    disp('Method is linear')
  case 'cubic'
    disp('Method is cubic')
  case 'nearest'
    disp('Method is nearest')
  otherwise
    disp('Unknown method')
end
```

В зависимости от значения символьного массива *method* в командном окне будет напечатано одно из перечисленных сообщений. Функция

lower(*str*)

заданную строку переводит в нижний регистр.

3.3. М-файлы

Программой на языке MATLAB мы будем называть последовательность команд, записанную в файле. Чтобы система принимала программу, у файла должно быть расширение *.m*, поэтому программы в MATLAB'е часто называют *m*-файлами. Об одном типе *m*-файлов — сценариях — мы уже говорили в разделе 3.3.1. К программам можно обращаться из командного окна и из других программ. При первом обращении к программе MATLAB ищет ее на диске по имени файла (без расширения). В первую очередь поиск производится в текущей папке. Как уже отмечалось, сменить текущую папку можно командой

cd имя_папки

или с помощью меню. Программу можно подготовить во внешнем редакторе (например, блокноте Windows), а можно воспользоваться встроенным редактором-отладчиком (Editor-Debugger). Для его вызова воспользуйтесь пунктом меню FILE|NEW|M-FILE или командой *edit*.

После того, как программа найдена на диске, производится ее синтаксический анализ. Если в результате этого анализа обнаружены ошибки, информация о них выдается в рабочем окне. В случае успешного выполнения синтаксического анализа программы создается ее псевдокод, который загружается в рабочее пространство и исполняется. Если во время исполнения происходят ошибки, то сообщения о них также отражаются в командном окне.

При повторном обращении к программе, она будет найдена уже в рабочем пространстве и поэтому не потребуется времени на ее синтаксический анализ. Удалить псевдокод из рабочего пространства можно командой

clear имя_функции

В общем случае поиск очередного встретившегося в командах имени (это может быть имя переменной или программы) начинается с рабочего пространства. Если имя не найдено, то оно ищется среди встроенных (built-in) функций. Исходный код этих функций не доступен пользователю. Далее поиск продолжается в каталогах, записанных в списке доступа. Изменить эти пути можно либо через меню, либо с помощью команды *addpath*. Команда

addpath folder1 folder2 ... −begin

добавлет указанные директории в начало списка, а команда

addpath folder1 folder2 ... **−end**

добавляет директории в конец.

Команды в программе отделяются друг от друга так же, как и в командном окне: либо символом перехода на новую строку, либо знаками ; , — отличие двух последних такое же, как и в командном окне. Специальным образом обозначать конец программы никак не требуется. Внеочередное прекращение работы программы выполняется командой **return**. Символ % означает начало комментариев: все, что записано после него и до конца строки при синтаксическом анализе игнорируется. Все, что записано в первых строках-комментариях, автоматически подключается в систему справки и может быть вызвано с помощью команды

help имя_файла

По использованию переменных и оперативной памяти программы делятся на программы-сценарии и программы-функции.

3.3.1. Программы-сценарии

С программами сценариями мы уже встречались в разделе . Программа-сценарий (script) — простейший тип программы. Программа-сценарий может использовать не только создаваемые ей самой переменные, но и использовать все переменные в рабочем пространстве командного окна. Созданные переменные так же располагаются в этом рабочем пространстве. Получить к ним доступ можно с помощью команды *keyboard*, которую нужно записать в программу. Эта команда передает управление в командное окно, в котором меняется вид приглашения:

K>>

Теперь в окне можно выполнять любые действия, в том числе просматривать значения переменных и изменять их. Чтобы продолжить выполнение программы необходимо выполнить команду **return**. Вместо команды *keyboard* для временного приостановления работы программы-сценария можно воспользоваться средствами встроенного отладчика: установить точку прерывания (breakpoint).

3.3.2. Программы-функции

После возможных пустых строк и строк, содержащих только комментарии, первая строка программы-функции должна иметь вид

function [*y1*, *y2*, ..., *ym*] = *ff*(*x1*, *x2*, ..., *xn*)

где *ff* — имя программы-функции (оно должно совпадать с именем файла), *x1*, *x2*, ..., *xn* — входные формальные параметры, *y1*, *y2*, ..., *yn* — выходные формальные параметры. Эту строку мы будем называть заголовком функции. Если функция содержит только один выходной формальный параметр, то окружающие его квадратные скобки в заголовке функции можно опустить. Функция может не содержать вовсе входных или/и выходных параметров. В этом случае скобки (круглые — для входных параметров, квадратные — для выходных) также можно опустить. Формальные параметры (входные и выходные) используются в функции как локальные переменные. Конечно же, функция может создавать и использовать другие локальные переменные, которые, как и обычно, объявлять специальным образом не нужно.

MATLAB'овские функции мы иногда будем называть процедурами или методами, чтобы не путать их с математическими функциями.

Вызов программы осуществляется командами

$$[u1,\ y2,\ \ldots,\ ul] = ff(v1,\ v2,\ \ldots,\ vk)$$

где *v1*, *v2*, ..., *vk* и *u1*, *u2*, ..., *ul* — фактические входные и выходные параметры функции. При вызове функции фактические входные параметры засылаются по порядку в соответствующие выходные формальные параметры. Выполнение функции заканчивается после выполнения ее последней команды. Досрочный выход осуществляется командой **return**. После того, как функция завершила свою работу, формальные выходные параметры засылаются в фактические выходные параметры. Количество фактических параметров должно быть не больше количества формальных параметров, но может с ним и не совпадать. Это удобно, если используются значения аргументов по умолчанию. Количество фактических входных параметров и фактических выходных параметров, с которыми была вызвана функция, всегда можно определить с помощью обращения к функциям *nargin* (количество входных параметров), *nargout* (количество выходных параметров).

Еще один способ вызова функции — это использование ее имени внутри выражения, например: $a = ff(k,\ 2).\hat{}2$. Здесь первый выходной параметр функции возводится в квадрат и результат присваивается переменной *a*. Заметим, что количество выходных параметров функции может быть больше, но остальные аргументы при таком обращении к функции теряются. Если выражение состоит только из одного имени функции:

$$ff(v1,\ v2,\ \ldots\ ,\ vk)$$

то первый выходной параметр присваивается перменной *ans*.

В качестве примера рассмотрим MATLAB'овскую функцию *rank*:

```
function r = rank(A, tol)
% RANK Matrix rank.
%   RANK(A) provides an estimate of the number of linearly
%   independent rows or columns of a matrix A.
%   RANK(A,tol) is the number of singular values of A
%   that are larger than tol.
%   RANK(A) uses the default tol = max(size(A)) * norm(A) * eps.

%   Copyright 1984-2000 The MathWorks, Inc.
%   Revision : 5.9     Date : 2000/06/0102 : 04 : 15

s = svd(A);
if nargin==1
   tol = max(size(A)') * max(s) * eps;
end
r = sum(s > tol);
```

Распечатать ее текст можно командами

```
type rank
```

Файл можно также открыть и во встроенном редакторе, например, командами

```
edit rank
```

Если вы это сделали, ничего не меняйте в исходном тексте!

Функция *rank* вычисляет ранг заданной матрицы A. Для этого MATLAB вычисляет ее сингулярное разложение и в качестве ранга берет количество сингулярных чисел, отличающихся от нуля больше, чем на величину tol. Функцию можно вызвать либо с одним входным аргументом:

```
rank(A)
```

либо с двумя:

```
rank(A, tol)
```

Количество фактических выходных аргументов внутри функции *rank* определяется с помощью обращения к функции *nargin*. Если параметр *tol* на входе не задан, то его значение вычисляется по формуле

```
tol = max(size(A)') * max(s) * eps;
```

В качестве еще одного примера рассмотрим стандартную функцию *humps*:

```
function [out1,out2] = humps(x)
%HUMPS A function used by QUADDEMO, ZERODEMO and FPLOTDEMO.
%   Y = HUMPS(X) is a function with strong maxima near x = .3
%   and x = .9.
%
%   [X,Y] = HUMPS(X) also returns X. With no input arguments,
%   HUMPS uses X = 0:.05:1.
%
%   Example:
%      plot(humps)
%
%   See QUADDEMO, ZERODEMO and FPLOTDEMO.

%   Copyright 1984-2002 The MathWorks, Inc.
%   Revision : 5.8     Date : 2002/04/1503 : 34 : 07

 if nargin==0, x = 0:.05:1; end

 y = 1 ./ ((x-.3).^2 + .01) + 1 ./ ((x-.9).^2 + .04) - 6;

 if nargout==2,
   out1 = x; out2 = y;
 else
   out1 = y;
 end
```

Функция вычисляет значения y некоторой тестовой функции в точках, заданных в

векторе x. Возможные способы вызова функции:

$y = humps;$
$y = humps(x);$
$[x,\ y] = humps;$
$[x,\ y] = humps(x);$

В двух последних случаях ($nargout == 2$) возвращается также сам вектор x. Если входных аргументов нет ($nargin == 0$), то в качестве x берется вектор $0 : 0.05 : 1$.

После того, как функция выполнила свою работу и произошло присваивание формальных выходных параметров фактическим, все локальные переменные функции удаляются. Переменные можно открыть для использования в командном окне, скриптах и других функциях (т. е. сделать *глобальными*) с помощью команды

global v1 v2 ... vk

Это команда должна появиться во всех функциях, которые хотят использовать одни и те же переменные, упомянутые в списке. Для получения доступа к переменным функции из командного окна (или программы-сценария) необходимо выполнить эту команду в программе-сценарии или в командной строке. Использовать глобальные переменные не рекомендуется.

В MATLAB'е возможно создание функций с неопределенным числом входным или/и выходных аргументов. Чтобы создать функцию с неопределенным числом входных аргументов, нужно список ее формальных входных аргументов завершить ключевым словом *varargin*. Например:

function *myfun*(*arg1*, *arg2*, *varargin*)

Здесь *arg1*, *arg2* — первый и второй аргумент функции, остальным аргументам соответствует *varargin*. Внутри функции к *varargin* можно обращаться как к массиву ячеек, содержащему знаячения «хвостовых» входных параметров. В частности, можно использовать *varargin*{:}, чтобы, например, передать список аргументов на обработку другой функции. Список аргументов может содержать только слово *varargin*.

Например, создадим функцию *plotter*, рисующую график (или графики) и делающую сверху подпись. Первым аргументов функции пусть будет строка символов, содержащая заголовок, остальные аргументы будут передаваться MATLAB'овской функции

plot:

```
function plotter(caption, varargin)

    title(caption)
    plot(varargin{:})
```

Теперь мы можем попробовать *plotter* в работе:

```
x = linspace(-2*pi, 2*pi);
plot('Trigonometric functions', x, sin(x), 'b', x, cos(x), 'r', x, tan(x), 'k')
```

Чтобы создать функцию с неопределенным числом выходных аргументов, нужно список ее формальных выходных аргументов завершить ключевым словом *varargout*. Работа с *varargout* аналогична работе с *varargin*.

3.3.3. Подфункции

M-файл с программой-функцией может содержать описание не одной, а нескольких функций. Имя первой функции должно совпадать с именем файла. Только эта функция (*основная*) доступна извне. Остальные функции будем называть *подфункциями*. Каждая из них может быть вызвана либо из основной функции, либо из другой подфункции того же самого *m*-файла. Конец описания основной функции или подфункций никаким специальным образом не обозначается: описание подфункции заканчивается либо в конце файла, либо непосредственно перед началом описания следующей подфункции (т. е.

перед строкой, содержащей ключевое слово **function**). Например:

```
function [...] = main(...)

    a = ...;
    b = ...;

function [...] = sub1(...)

    a = ...;
    b = ...;

function [...] = sub2(...)

    a = ...;
    b = ...;
```

Все переменные, используемые внутри подфункции, являются локальными: их область видимости ограничивается только этой подфункцией. Все переменные, используемые в основной функции, также являются локальными: их область видимости распространяется только на саму функцию, но не ее подфункции. В примере выше переменные с одинаковым именем a в основной функции и двух подфункциях различны. То же относится к переменной b.

В следующем примере m-файл содержит одну подфункцию *hilb*, которая вызывается из основной функции, а также вызывает сама себя рекурсивно.

Листинг m/hilbertfractal.m

```
    hilbertfractal(depth) кривая Гильберта
    depth задает глубину рекурсии. По умолчанию, 3
function hilbertfractal(depth)

if nargin < 1
    depth = 3;
end;

if depth > 7
    error('The depth of recursion is too large. This requires a lot of time')
```

```
end;

figure
axis([-0.25 1.25 -0.25 1.25])
axis equal
hold on
hilb([0 0], [1 0], [0 1], depth);

function hilb(A, B, C, depth)

if depth <= 0
    return;
end;

sz = 2^depth - 1;
dAB = (B - A) / sz;
dAC = (C - A) / sz;

D = (A + C - dAC) / 2;
E = (A + B - dAB) / 2;
F = D + dAC;
G = F + E - A;
H = G + dAB;
I = F + B - A;
J = C + H - F;
K = I - dAC;
L = H - dAC;

hilb(A, D, E, depth - 1);
hilb(F, G, C, depth - 1);
hilb(H, I, J, depth - 1);
hilb(K, B, L, depth - 1);
plot([D(1), F(1)], [D(2), F(2)]);
plot([G(1), H(1)], [G(2), H(2)]);
plot([I(1), K(1)], [I(2), K(2)]);
```

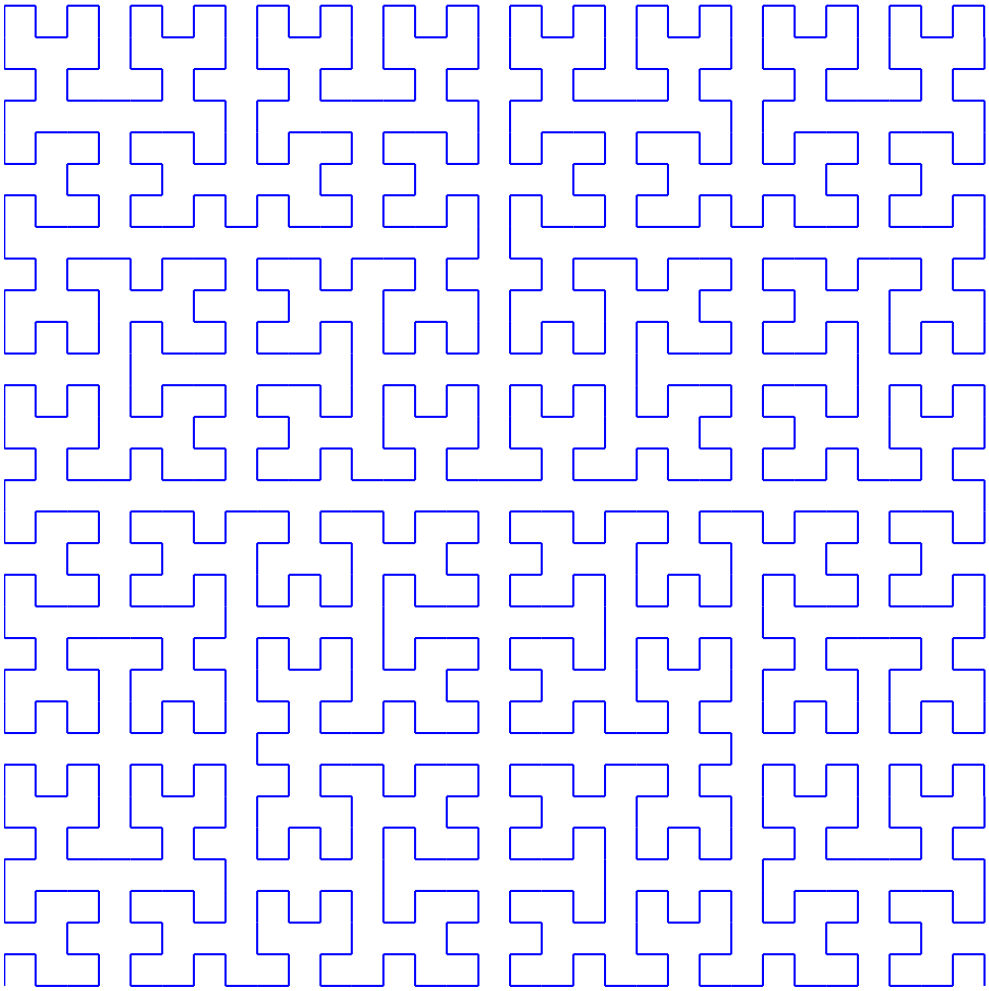


Рис. 3.1. Кривая Гильберта. Глубина рекурсии равна 5

3.3.4. Вложенные функции

Существует другой способ написания m-файлов, содержащих несколько функций. Функции можно вкладывать внутрь других функций. В этом случае каждая из функций, описанных в файле должна заканчиваться ключевым словом **end**. Все команды после заголовка функции (строка **function** ...) и до соответствующего ключевого слова **end** составляют *тело* функции. Чтобы определить *вложенную функцию* (nested function), нужно разместить ее тело внутри тела другой функции (в любом месте). Количество уровней вложенности не ограничено. Например:

```
function [...] = main(...)

    a = ...;
    b = ...;

    function [...] = sub(...)

        c = ...;

        function [...] = subsub(...)

        end;

    end;

    function [...] = sub2(...)
        c = ...;
    end;

end
```

Область видимости переменных функции распространяется на все вложенные в нее функции (а также функции, вложенные во вложенные функции, и т. д.). В примере выше во вложенных функциях *sub*, *subsub* и *sub2* есть доступ к переменным a и b из основной функции *main*. Во вложенной функции *subsub* есть доступ к переменной c

из *sub*. В то же время переменные с одинаковым именем *c* в функции *sub* и функции *sub2* различны (предполагается, что в основной функции *main* нигде не встречается переменная *c*).

Следующая программа реализует алгоритм поиска пути в лабиринте. Массив *visited* посещенных узлов лабиринта инициализирован в основной функции и при каждом обращении к вложенной функции *find_path* доступен. Инициализация этого массива внутри функции *find_path* привела бы к тому, что у нас был бы не один, а *много* массивов (по количеству вызовов функции *find_path*: эта функция вызывает себя рекурсивно) и программа не правильно бы работала. Передача массива *visited* с помощью параметров (входного и выходного) функции *find_path* привела бы к большим накладным расходам.

Листинг m/hampton.m

Функция находит путь в лабиринте из точки (*startx*, *starty*) в (*stopx*, *stopy*)

```
function path = hampton

    plan = [
        '8888888888888888888888888888888888'
        '8          8    8      8           8'
        '8 8888888 8 8888 8888 8 888888888 8'
        '8 8       8 8       8 8         8 8'
        '8 8 8888888 8 88888 8 888888888 8 8'
        '8 8 8       8   8   8         8 8 8'
        '8 8 8 888888888 8 88888 88888 8 8 8'
        '8 8 8 8   8     8     8 8 8 8 8 8 8'
        '8 8 8 8 8 8 888888888 8 8 8 8 8 8 8'
        '8 8   8 8 8 8       8 8 8 8 8   8 8'
        '8 888 8 8 8 8       8 8 8 8 8 888 8'
        '8   8 8 8 8 8       8 8 8 8 8 8   8'
        '888 8 8 8 8 8       8 8 8 8 8 8 888'
        '8   8 8 8 8 8       8 8 8 8 8 8   8'
        '8 888 8 8 8 8888 8888 8 8 8 8 8 8 8'
        '8 8   8 8 8   8 8    8 8 8   8 8 8'
        '8 8 8 8 8 8888 8 8 8888 8 88888 8 8'
        '8 8 8   8    8 8 8        8     8 8'
        '8 8 8888888888 8 8888888888888888 8'
        '8 8            8                  8'
```

```
    '8 88888888888888888888888888888888'
    '8                                  8'
    '88888888888888888 88888888888888888'];

startx = 18;
starty = 23;
stopx = 17;
stopy = 14;

[m, n] = size(plan);

visited = zeros(m, n);
path = find_path(startx, starty);

function path = find_path(x, y)

    if x > n || x < 1 || y > m || y < 1
        path = [];
    else
        if plan(y, x) == '8' || visited(y, x)
            path = [];
        elseif x == stopx && y == stopy
            path = [x, y];
        else
            visited(y, x) = 1;
            for d = [1, 0; -1, 0; 0, 1; 0, -1]'
                path = find_path(x + d(1), y + d(2));
                if ~isempty(path)
                    path = [x, y; path];
                    return;
                end;
            end;
        end;
    end;
end;
```

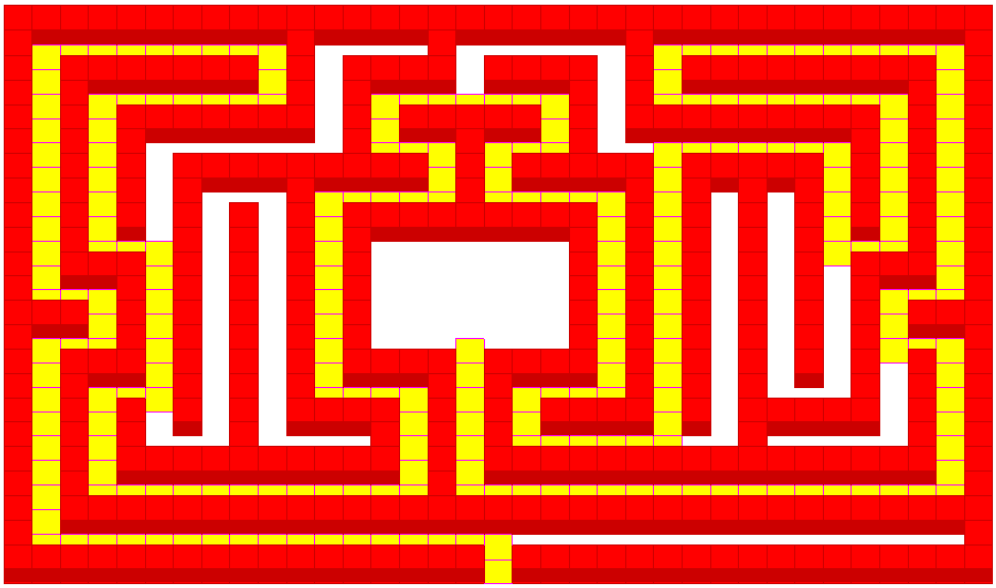


Рис. 3.2. Лабиринт в Хэмптон–Корте

end

Итак, MATLAB допускает два способа описания нескольких функций внутри одного файла:

- основная функция и ее подфункции;
- основная функция и вложенные функции.

Пользоваться обеими этими способами в одном *m*-файле нельзя.

3.3.5. Частные функции

Частные функции — это функции, размещенные в папке с именем *private*. Частные функции доступны только из функций, расположенных в самой этой папке и в родительской папке. Папку *private* не следует указывать в путях доступа.

4. Основные численные методы

4.1. Суммы и произведения

4.1.1. Суммы

Функция $sum(a)$ возвращает сумму элементов вектора a. Если a — это матрица, то функция возвращает сумму элементов в каждом столбце.

Функция $cumsum(a)$ возвращает вектор той же длины, что и a, причем i-ая компонента возвращаемого вектора — это сумма первых i элементов вектора a. Для матриц $cumsum$ работает постолбцам, а именно: если a — матрица, то функция возвращает матрицу таких же размеров, элемент i-й строки и j-го столбца которой равен сумме первых i элементов, стоящих в j-м столбце матрицы a.

Например,

$$cumsum(1:5)$$

возвратит вектор $[1,\ 3,\ 6,\ 10,\ 15]$.

Исследуем сходимость ряда:

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$$

Листинг start/dirichlet.m

Экспериментальное исследование скорости сходимости ряда

$$\sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6}$$

$$n = 100;$$
$$k = 1 : n;$$
$$a = k.\hat{}(-2);$$
$$s = cumsum(a);$$
$$l = pi\hat{}2/6$$
$$s(\textbf{end})$$

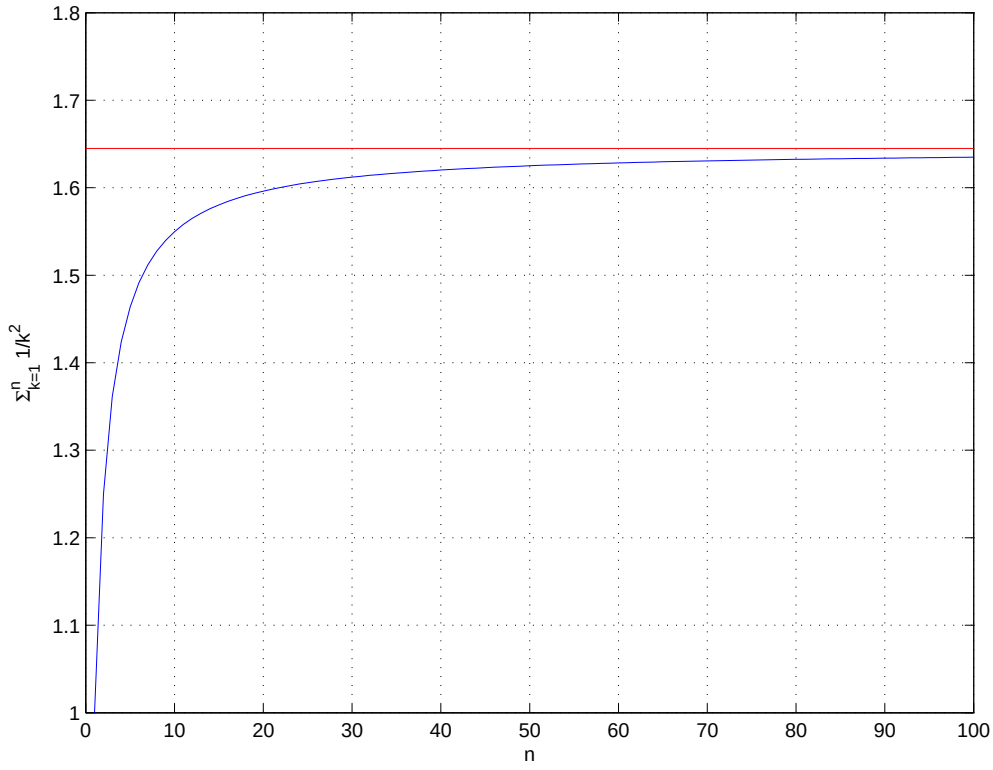


Рис. 4.1. Исследование скорости сходимости ряда $\sum\limits_{k=1}^{\infty} \dfrac{1}{k^2} = \dfrac{\pi^2}{6}$

```
plot(k, s, 'b', [0 n], [l l], 'r');
grid;
xlabel('n');
ylabel('\Sigma_{k=1}^n {1}/{k^2}');

relerr = abs(s - l)/l;

figure
semilogy(relerr)
xlabel('n');
ylabel('Relative error');
grid
```

Программа выдает значения $\pi^2/6 = 1.6449$ и частичную (при $n = 100$) сумму ряда 1.6350. Из графика относительной ошибки видно, что сходимость рассматриваемого ряда крайне медленная — ряд сходится медленнее, чем со скоростью геометрической прогрессии.

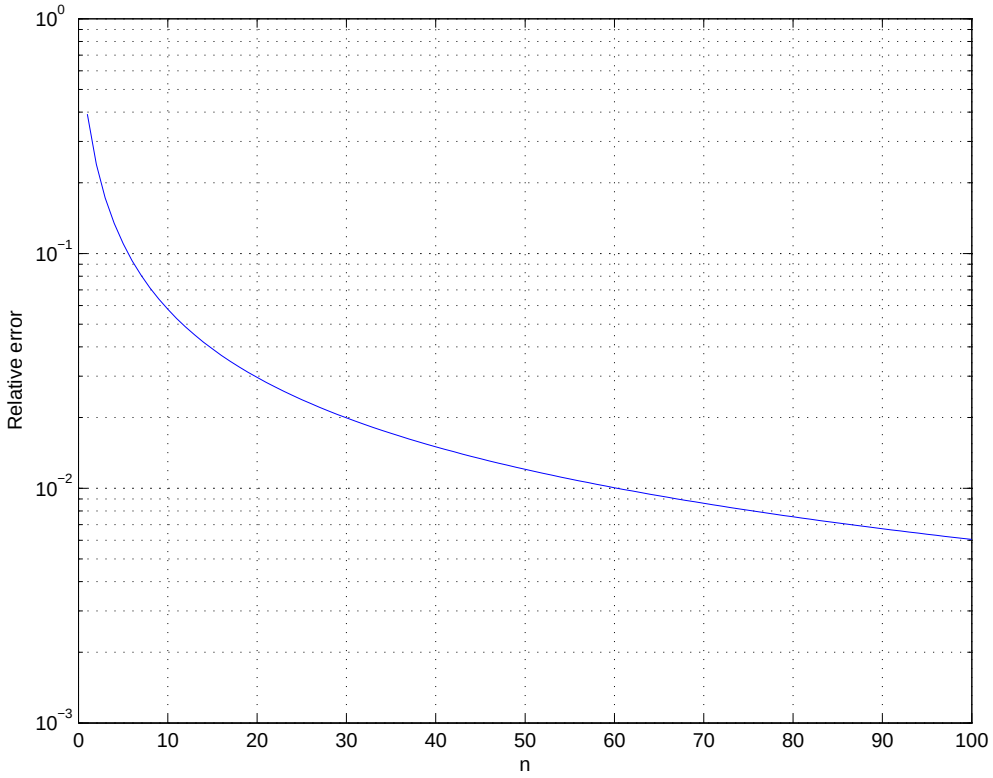


Рис. 4.2. Ряд $\sum\limits_{k=1}^{\infty} \frac{1}{k^2}$. График относительной ошибки

4.1.2. Произведения

Для нахождения произведения элементов массивов в системе MATLAB есть функции

$prod(a)$,
$cumprod(a)$,

аналогичные функциям $sum(a)$, $cumsum(a)$.

Рассмотрим бесконечное произведение

$$\prod_{k=2}^{\infty}\left(1-\frac{1}{k^2}\right)=\frac{1}{2} \tag{1}$$

и экспериментально исследуем его скорость сходимости:

Листинг start/half.m

Вычисление бесконечного произведения $\prod\limits_{k=2}^{\infty}\left(1-\frac{1}{k^2}\right)=\frac{1}{2}$

$k = 2{:}100;$
$a = 1 - 1.{*}k.\hat{}(-2);$
$p = cumprod(a);$

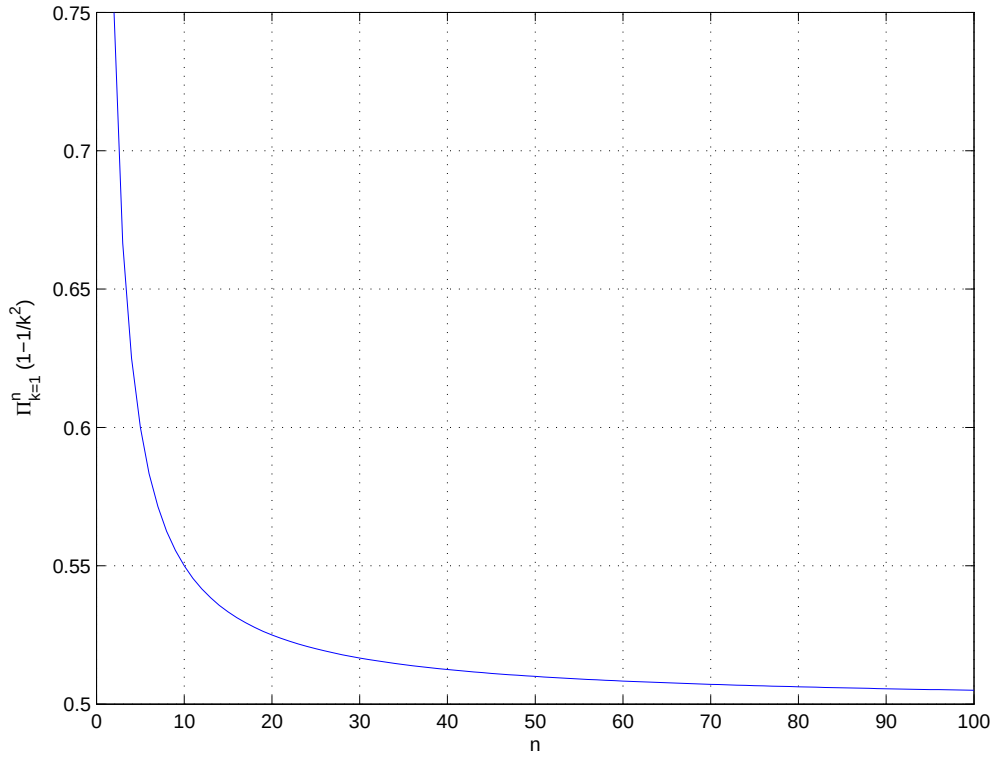


Рис. 4.3. Исследование скорости сходимости бесконечного произведения (1)

```
p(end)
plot(k, p);
xlabel('n');
ylabel('\Pi_{k=1}^n (1-1/{k^2})');
grid;

relerr = abs(.5 - p)./.5

figure
semilogy(k, relerr);
xlabel('n');
ylabel('Relative error');
grid;
```

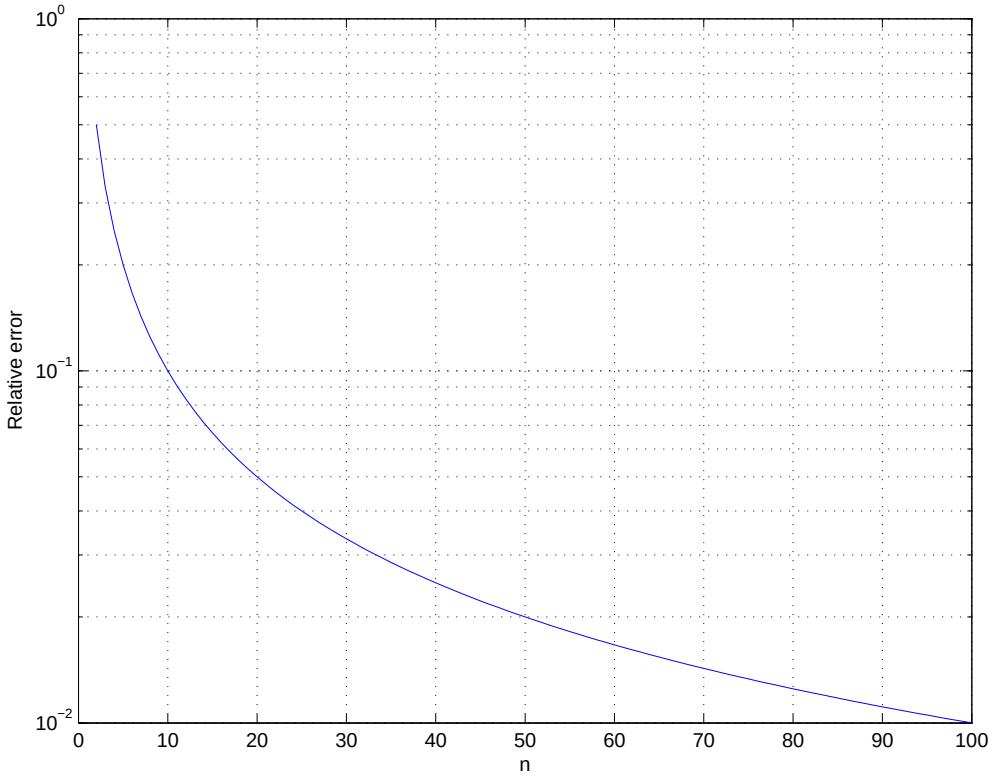


Рис. 4.4. Вычисление произведения (1). График относительной ошибки

4.1.3. Факториал

Чтобы найти факториал числа m, можно воспользоваться командой $prod(1{:}m)$. Найдем, для какого максимального значения m, вычисление $m!$ не дает переполнения:

$n = 200;$
$m = 1{:}n;$
$mf = cumprod(m);$
$semilogy(mf)$
$grid$
$sum(isinf(mf))$
$sum(isfinite(mf))$

Мы получим график значений факториала, а также два числовых значения: 30 и 170. Функция *isinf* возвращает 1, если ее аргумент — это *inf*, и возвращает 0, в противном случае. Функция *isfinite* возвращает 1, если ее аргумент — это не *inf* и не *NaN*, и возвращает 0, в противном случае. Итак, максимальное значение m, для которого вычисление $m!$ не дает переполнения — это 170.

4.2. Линейная алгебра

4.2.1. Нормы векторов и матриц

Если a — вектор, то функция $k = norm(a)$ возвращает его евклидову норму

$$\|a\|_2 = \sqrt{|a_1|^2 + |a_2|^2 + \cdots + |a_n|^2},$$

а $k = norm(a,\, p)$ — норму

$$\|a\|_p = \left(|a_1|^2 + |a_2|^2 + \cdots + |a_n|^2\right)^{1/p}, \quad p \geq 1$$

Параметр p может быть равен Inf. Норма $\|a\|_\infty$ определяется следующим образом:

$$\|a\|_\infty = \max\left\{|a_1|, |a_2|, \ldots, |a_n|\right\}.$$

Если A — матрица, то функция $k = norm(A)$ возвращает ее спектральную норму $\|A\|_2$, равную максимальному собственному числу матрицы $A^{\mathrm{T}}\overline{A}$ (т. е. максимальному сингулярному числу матрицы A), а $k = norm(A,\, p)$ — норму $\|A\|_p$. В общем случае норма $\|A\|_p$ определяется через соответствующую векторную норму следующим образом:

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}.$$

Фробениусовой нормой называется

$$\|A\|_F = \sqrt{|a_{ij}|^2}.$$

Возможные значения параметра p функции $norm(A,\, p)$: 1, 2, Inf, '*fro*'. Последнее из перечисленных отвечает фробениусовой норме.

4.2.2. Число обусловленности

Числом обусловленности матрицы A называется величина

$$\mathrm{cond}_p A = \|A\|_p \cdot \|A^{-1}\|_p.$$

Функция $k = cond(A)$ возвращает $\mathrm{cond}_2 A$ — спектральное число обусловленности, функция $k = cond(A,\, p)$ возвращает $\mathrm{cond}_p A$ — число обусловленности матрицы A относительно p-нормы. Возможные значения p: 1, 2, Inf, '*fro*'.

Функция $c = condest(A)$ вычисляет верхнюю оценку для $\mathrm{cond}_1 A$. Используется оценщик Хэйджера в модификации Хаема. Как правило, эта оценка очень хорошо приближает значение $\mathrm{cond}_1 A$.

Сравним результаты работы функций *cond*, *rcond*, *condest*:

$A = gallery('invol', 3)$
$cond(A, 1)$
$cond(A)$
$cond(A, Inf)$
$cond(A, 'fro')$
$condest(A)$

Получаем:

$$A = \begin{bmatrix} -3 & 1/2 & 1/3 \\ -36 & 8 & 6 \\ 30 & -15/2 & -6 \end{bmatrix}.$$

$cond(A, 1)$	$4.7610 \cdot 10^3$
$cond(A)$	$2.3966 \cdot 10^3$
$cond(A, Inf)$	$2.5000 \cdot 10^3$
$cond(A, 'fro')$	$2.3976 \cdot 10^3$
$condest(A)$	$4.7610 \cdot 10^3$

4.2.3. Системы линейных уравнений

Пусть A — квадратная матрица порядка n, а B — прямоугольная матрица размера $n \times k$. Команда $A\backslash B$ находит решение X системы матричных уравнений $AX = B$. В частности, если b — столбец длины n, то $A\backslash b$ возвращает решение x системы линейных уравнений $Ax = b$. Если A вырождена или близка к вырожденной, то выдается соответствующее предупреждение.

В качестве примера решим систему линейных уравнений

$$\begin{bmatrix} 3 & 2 & 2 \\ 0 & 3 & -1 \\ -3 & -5 & 3 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ 2 \\ -5 \end{bmatrix}.$$

$A = [3\ 2\ 2;\ 0\ 3\ -1;\ -3\ -5\ 3]$
$b = [7\ 2\ -5]'$
$x = A\backslash b$

Получим

$$x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}.$$

Аналогично, команда B/A находит решение X системы уравнений $YA = B$, где A — матрица размера $n \times k$, B — матрица размера $m \times k$, Y — неизвестная матрица размера $m \times n$.

Таким образом, если A — квадратная и невырожденная, то $X{=}A\backslash B$ соответствует $X = A^{-1}B$, а $Y{=}B/A$ соответствует $X = BA^{-1}$, поэтому $A\backslash B$ называется *левым матричным делением*, а B/A — *правым матричным делением*.

Следующая программа генерирует системы линейных уравнений с с матрицами, коэффициенты которых распределены по нормальному закону. Строятся графики зависимости числа обусловленности матрицы, невязки и относительной ошибки вычисленного решения от размерности задачи. Эксперимент подтверждает следующее практическое правило: если машинное эпсилон и число обусловленности системы составляют величины порядка 10^{-p} и 10^{q} соответственно, то относительная ошибка в компонентах решения системы приближенно равна 10^{-p+q} (т. е. компоненты решения содержат около $p - q$ верных десятичных цифр).

Листинг linsys/xlinsolve.m

```
nvec = 5:5:1000;
relerr = [];
discr = [];
condc = [];
condestc = [];

for n = nvec
    n
    A = randn(n);
    x = ones(n, 1);
    b = A*x;

    xc = A\b;
    relerr = [relerr; norm(x - xc)/norm(x)];
```

```
    discr = [discr; norm(A*xc - b)];

    condc = [condc; cond(A)];
    condestc = [condestc; condest(A)];
end;

subplot(2, 2, 1);
title('Relative error')
semilogy(nvec, relerr, '.')
legend('Relative error', 2)
grid

subplot(2, 2, 3);
title('Discrepancy')
semilogy(nvec, discr, '.')
legend('Discrepancy', 2)
grid

subplot(1, 2, 2);
title('Condition number');
semilogy(nvec, [condc, condestc], '.')
legend('cond', 'condest', 2)
grid
```

Проведем небольшой вычислительный эксперимент с матрицами Гильберта. Получить матрицу Гильберта заданного порядка n в MATLAB'е можно с помощью функции $hilb(n)$. Для матриц Гильберта удается аналитически найти обратную. Получить ее для заданного порядка n можно по команде $invhilb(n)$. Например,

```
format rat
H = hilb(3)
IH = invhilb(3)
```

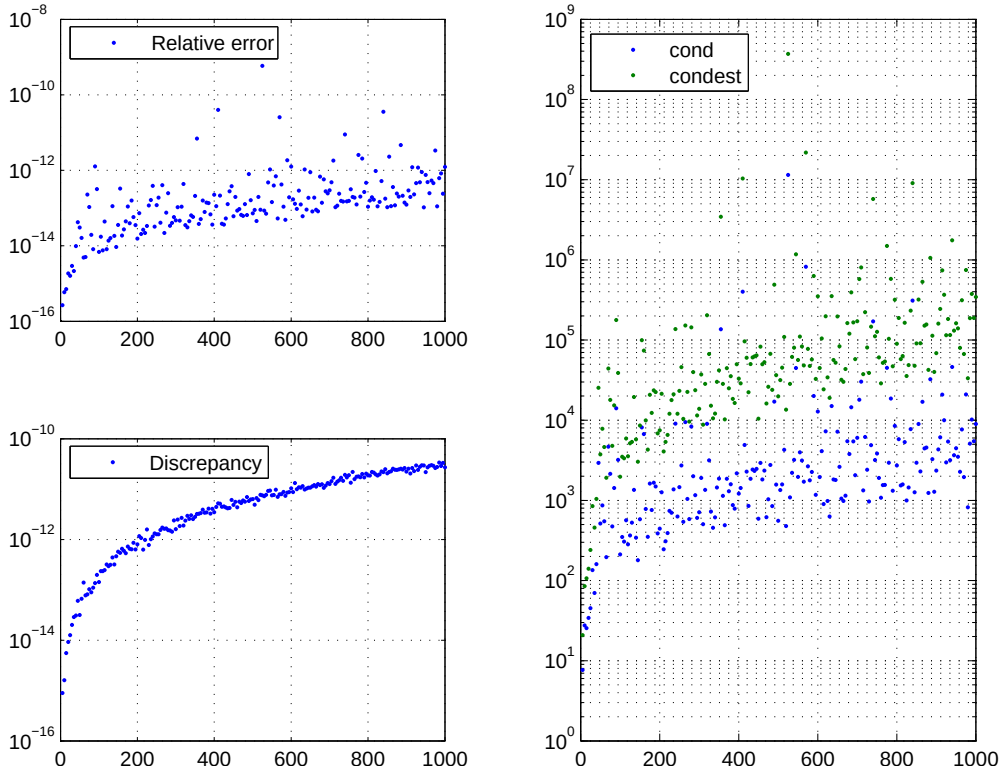


Рис. 4.5. Решение систем линейных уравнений со случайными матрицами: относительная ошибка, невязка число обусловленности

выдают

$$H = \begin{bmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{bmatrix}, \qquad IH = \begin{bmatrix} 9 & -36 & 30 \\ -36 & 192 & -180 \\ 30 & -180 & 180 \end{bmatrix}.$$

Матрицы Гильберта являются примером чрезвычайно плохо обусловленных матриц.

Для $n = 1, \ldots, 20$ следующая программа решает систему линейных уравнений с матрицей Гильберта n-го порядка. Строятся графики зависимости относительной ошибки и невязки полученного численного решения от размерности задачи. Также вычисляются число обусловленности по формулам: $cond(hilb(n)$ и $norm(hilb(n),\ 1)$*$norm(hilb(n),\ 1)$ — и оценка числа обусловленности $condest(hilb(n)$ и строятся соответсвующие графики.

Листинг linsys/xhilb.m

```
N = 20;
relerr = zeros(N, 1);
discr = zeros(N, 1);
condc = zeros(N, 1);
```

```
condestc = zeros(N, 1);
condp = zeros(N, 1);

for n = 1:N
    A = hilb(n);
    x = ones(n, 1);
    b = A*x;

    xc = A\b;
    relerr(n) = norm(x - xc)/norm(x);
    discr(n) = norm(A*xc - b);

    condc(n) = cond(A);
    condestc(n) = condest(A);
    condp(n) = norm(A)*norm(invhilb(n));
end;

shg;
clf;

subplot(2, 2, 1);
title('Relative error')
semilogy(1:N, relerr)
legend('Relative error', 2)
grid

subplot(2, 2, 3);
title('Discrepancy')
semilogy(1:N, discr)
legend('Discrepancy', 2)
grid

subplot(1, 2, 2);
title('Condition number');
semilogy(1:N, [condc, condestc, condp])
```

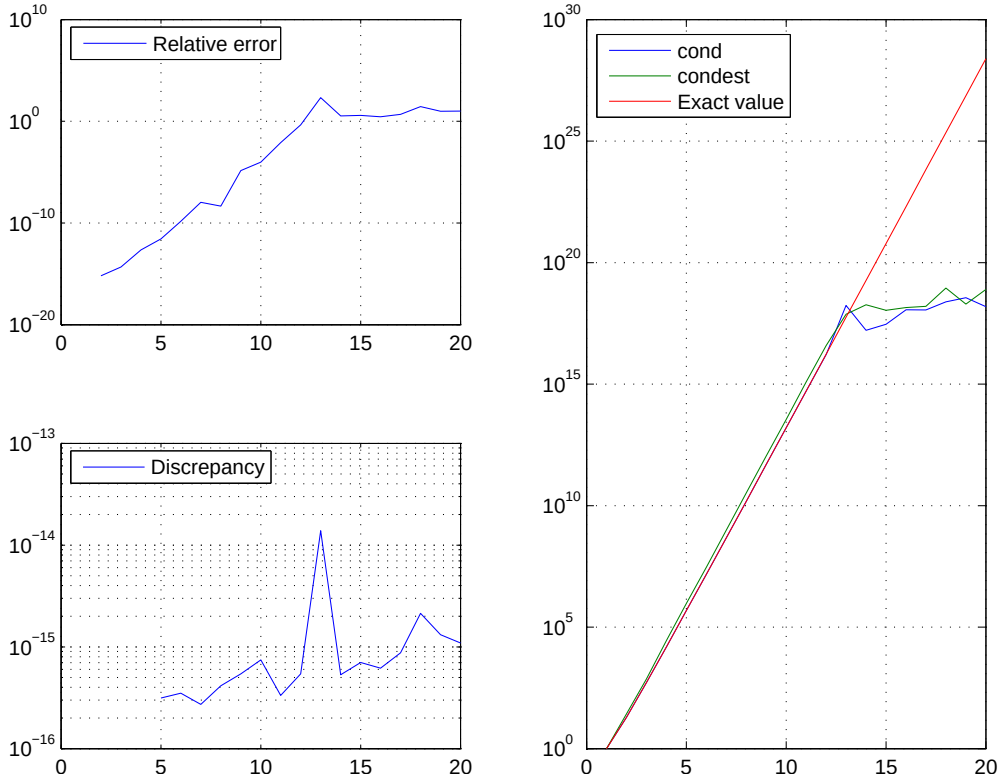


Рис. 4.6. Решение систем линейных уравнений с матрицей Гильберта: относительная ошибка, невязка число обусловленности

legend('cond', 'condest', 'Exact value', 2)
grid

Графические результаты представлены на рисунке 4.6. На графике справа мы видим, что при $n \geq 12$ число обусловленности матрицы Гильберта превышает 10^{16}. В силу ошибок округлений алгоритмы *cond* и *condest* не смогли верно определить (и оценить) $\mathrm{cond}_1 H$. Как и следовало ожидать, относительная ошибка при $n \geq 12$ больше 1 (т. е. компоненты решения не содержат ни одной верной значащей цифры), однако норма невязки даже при огромных значениях числа обусловленности очень мала.

4.2.4. Переопределенные системы

Если A — прямоугольная матрица с линейно независимыми столбцами, то $A\backslash b$ находит (единственное) псевдорешение системы $Ax = b$. Напомним, что *псевдорешением* системы линейных уравнений $Ax = b$ называется вектор $\widehat{x}$, на котором минимизируется евклидова норма невязки:

$$\|A\widehat{x} - b\|_2 = \min \|Ax - b\|_2.$$

Для примера найдем нормальное псевдорешение системы $Ax = b$, в которой

$$A = \begin{bmatrix} 1 & 5 \\ 2 & 6 \\ 3 & 7 \\ 4 & 8 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 2 \end{bmatrix}.$$

Команда

$$A\backslash b$$

возвратит вектор

$$\begin{bmatrix} 0.1750 \\ 0.1250 \end{bmatrix}$$

.

4.2.5. Обратная и псевдообратная матрицы

Если A — квадратная невырожденная матрица, то функция $B = inv(A)$ возвращает обратную к ней матрицу $B = A^{-1}$.

Если A — произвольная прямоугольная матрица, то функция $B = pinv(A)$ возвращает псевдообратную к ней матрицу $B = A^{+}$. Напомним, что матрица $B = A^{+}$ называется псевдообратной к A, если для любого столбца b формула $x = A^{+}b$ определяет нормальное псевдорешение системы $Ax = b$.

Функция $B = pinv(A,\ tol)$ при нахождении псевдообратной матрицы использует допуск tol: все сингулярные значения матрицы A, меньшие tol, зануляются. По умолчанию,

$$tol = \max\{m, n\} \cdot \|A\|_2 \cdot eps.$$

Для примера найдем нормальное псевдорешение системы $Ax = b$, в которой

$$A = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 2 \end{bmatrix}.$$

Операция \ здесь не сработает, так как матрица A не имеет полного ранга: на команду

$x = A\backslash b$

MATLAB выдаст предупреждение «Warning: Rank deficient, $rank = 2$, $tol = 1.8757\text{e}{-014}$» и ответ

$$x = \begin{bmatrix} 0.2375 \\ 0 \\ 0.0625 \end{bmatrix}.$$

Как мы увидим дальше, x является псевдорешением, но не является нормальным псевдорешением.

Для нахождения нормального псевдорешения воспользуемся функцией $pinv$:

$y = pinv(A)*b$

Получим

$$y = \begin{bmatrix} 0.1875 \\ 0.1000 \\ 0.0125 \end{bmatrix}.$$

Проверим, что нормы невязок на векторах x и y совпадают, т. е. x также является псевдорешением: команды

$norm(A*x - b)$
$norm(A*y - b)$

выдают одно и то же значение 0.5477.

Покажем, как важен правильный выбор параметра tol в функции $pinv(A, tol)$. Выполним

$z = pinv(A, norm(A)*max(size(A))*eps*1\text{e}{-2})*b$

Получим

$$z = 10^{12} \cdot \begin{bmatrix} 1.2771 \\ -2.5542 \\ 1.2771 \end{bmatrix}.$$

Для вектора z норма невязки уже больше: команда

$norm(A^*z\ -\ b)$

возвращает 0.5727.

4.2.6. Собственные числа и собственные векторы

Пусть A — квадратная матрица. Функция

$d\ =\ eig(A)$

возвращает вектор, составленный из собственных чисел матрицы A, а

$[Q,\ D]\ =\ eig(A)$

находит матрицу Q, в столбцах которой записаны собственные векторы, и диагональную матрицу D с собственными числами на диагонали.

Пусть

$$A = \begin{bmatrix} 9 & 0 & -1 \\ -5 & 8 & -10 \\ 2 & 5 & 6 \end{bmatrix}$$

Функция

$eig(A)$

возвратит собственные числа 9.4050, $6.7975 + 7.2065i$, $6.7975 - 7.2065i$. Функция

$[Q,\ D]\ =\ eig(A)$

найдет матрицу Q, в столбцах которой записаны собственные векторы:

$$Q = \begin{bmatrix} 0.7855 & 0.0755 - 0.0148i & 0.0755 + 0.0148i \\ -0.5309 & 0.8109 & 0.8109 \\ -0.3182 & 0.0597 - 0.5770i & 0.0597 + 0.5770i \end{bmatrix}$$

и диагональную матрицу D с собственными числами на диагонали:

$$D = \begin{bmatrix} 9.4050 & 0 & 0 \\ 0 & 6.7975 + 7.2065i & 0 \\ 0 & 0 & 6.7975 - 7.2065i \end{bmatrix}.$$

Проверим равенство $D = Q^{-1}AQ$, для этого найдем $norm(Q\backslash A^*Q - D, 1)$. Получим 2.2204×10^{-16}.

Если матрицы A не подобна диагональной, то собственные векторы, возвращаемые функцией $[Q, D] = eig(A)$, будут линейно зависимыми. Рассмотрим пример. Пусть

$$A = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

тогда $[Q, D] = eig(A)$ найдет

$$Q = \begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}, \qquad D = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}.$$

4.3. Интерполяция

4.3.1. Полиномиальная интерполяция

Пусть про некоторую функцию $g(x)$ известно, что она в точках $x_1, x_2, \ldots, x_n$ ($x_1 < x_2 < \cdots < x_n$) принимает значения $y_1, y_2, \ldots, y_n$ соответственно. *Задача интерполяции* заключается в построении такой функции $f(x)$ из известного класса, которая в точках x_i принимает те же значения y_i, что и функция $g(x)$. Функцию $f(x)$ часто называют *интерполянтом*, точки x_i — *узлами интерполяции*, величины y_i — *значениями интерполяции*. Выделим тот случай, когда $f(x)$ — многочлен. Он называется *интерполяционным*. Известная теорема из теории аппроксимации говорит о том, что для произвольной *таблицы интерполяции*

x_1	x_2	$\ldots$	x_n
y_1	y_2	$\ldots$	y_n

интерполяционный многочлен $f(x)$ степени меньше n существует и единственен. Если интерполируемая функция $g(x)$ в некоторой области имеет n непрерывных производных то для любой точки x из этой области погрешность интерполяции равна

$$g(x) - f(x) = \frac{g^{(n)}(\xi)}{n!}(x - x_1)(x - x_2)\ldots(x - x_n), \quad \text{где } \xi \in [x_1, x_n].$$

В MATLAB'е коэффициенты интерполирующего многочлена можно найти с помощью функции *polyfit*. Если x — массив, содержащий n узлов, а y — массив, содержащий n значений, то

polyfit$(x,\ y,\ n\ -\ 1)$

возвращает вектор коэффициентов интерполяционного многочлена.

Рассмотрим пример. Функцию $y = \ln x$ будем интерполировать кубическим полиномом по точкам 0.4, 0.5, 0.7, 0.8. Оценим погрешность интерполяции в точке 0.6. Для нахождения интерполянта выполним команды:

$x = [0.4\ 0.5\ 0.7\ 0.8];$
$y = log(x);$
$f = polyfit(x,\ y,\ 3)$

Получим:

$f =$
$1.6836 \quad -4.5239 \quad 5.2760 \quad -2.4106$

Таким образом, интерполянт имеет вид:

$$f(x) = 1.6836x^3 - 4.5239x^2 + 5.2760x - 2.4106.$$

Функция $polyval(f,\ x)$ возвращает значение многочлена, коэффициенты которого записаны в векторе f, в точке x. Параметр x может быть вектором, тогда функция возвратит вектор значений. Например,

$polyval(f,\ 0.6)$

дает значение -0.5100.

Выражение для погрешности дает:

$$\ln(0.6) - f(0.6) = -\frac{6}{\xi^4}\frac{1}{4!}(0.6 - 0.4)(0.6 - 0.5)(0.6 - 0.7)(0.6 - 0.8),$$

$0.4 < \xi < 0.8.$

Следовательно, ошибка для погрешности не больше

$$\frac{6}{0.4^4}\frac{1}{24}0.0004 \approx 0.0039.$$

Найдем фактическую абсолютную ошибку:

$polyval(f,\ 0.6)\ -\ log(0.6)$

Получим, что $|p(0.6) - \ln 0.6| = 0.00085$.

В качестве еще одного примера рассмотрим *функцию Рунге*

$$R(x) = \frac{1}{1 + 25x^2}$$

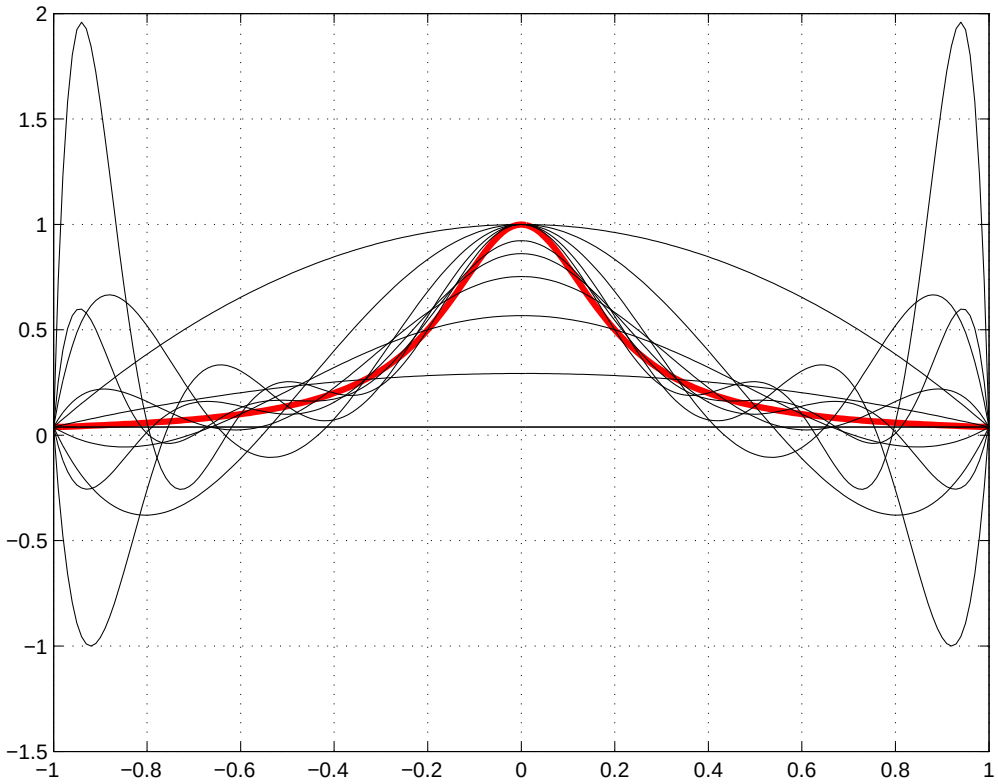


Рис. 4.7. Полиномиальная интерполяция функции Рунге с равномерным распределением узлов

и попробуем на отрезке $[-1, 1]$ найти интерполянт к ней по $n = 1, 2, \ldots, 12$ равномерно распределенным точкам:

```
xx = −1:.01:1;
yy = 1./(1 + 25*xx.^2);
plot(xx, yy, 'r', 'LineWidth', 3);
grid;

for n = 1:12,
    x = linspace(−1, 1, n);
    y = 1./(1 + 25*x.^2);
    p = polyfit(x, y, n − 1);
    yp = polyval(p, xx);
    hold on;
    plot(xx, yp, 'k');
end;
```

Попробуем теперь сгустить точки около концов отрезка $[-1, 1]$. В качестве узлов

интерполяции возьмем корни полиномов Чебышева:

$$x = \cos\frac{(2i-1)\pi}{2n} \quad (i = 1, 2, \ldots, n).$$

```
hold off;
xx = -1:.01:1;
yy = 1./(1+25*xx.^2);
plot(xx, yy, 'r', 'LineWidth', 3), grid;
hold on;

for n = 15:20;
  i = 1:n;
  x = cos((2*i-1)*pi/(2*n));
  sort(x);
  y = 1./(1 + 25*x.^2);
  p = polyfit(x, y, n-1);
  yp = polyval(p,xx);
  plot(xx, yp, 'k');
end;

plot(x, zeros(size(x)), 'o');

hold off;
```

Затруднение с функцией Рунге исчезло. Абсолютная ошибка интерполяции этой функции полиномом 19 степени составляет 0.038, а относительная — лишь 0.004:

```
max(abs(yp - yy))
max(yp/yy)
```

4.3.2. Кусочно-полиномиальная интерполяция

Функция называется кусочно-полиномиальной, если ее область определения разбивается на отрезки, на каждом из которых функция представляет собой многочлен. *Кусочно-полиномиальная интерполяция* заключается в построении такой кусочно-полиномиальной функции, которая в заданных точках $x_1, x_2, \ldots, x_n$ принимает заданные значения $y_1, y_2, \ldots, y_n$ соответственно.

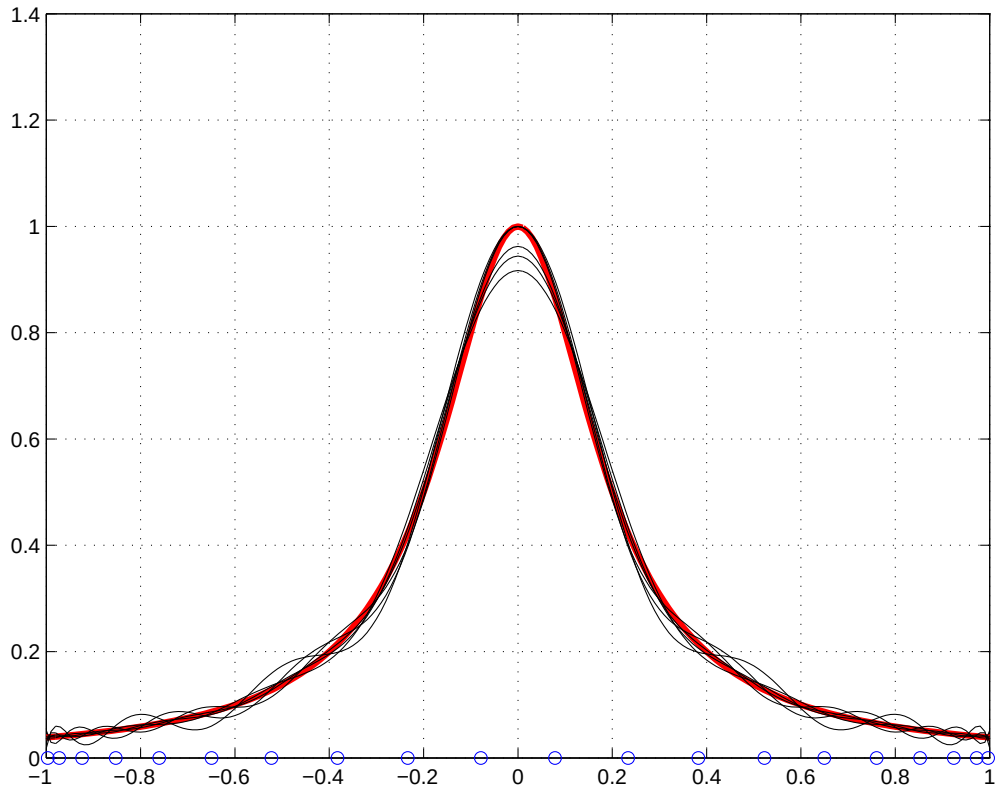


Рис. 4.8. Полиномиальная интерполяция функции Рунге с чебышевским распределением узлов

MATLAB предоставляет следующие возможности по построению кусочно-полиномиальных интерполянтов:

- функция *interp1*(x, y, xx, '*nearest*') строит ступенчатый интерполянт: для каждой промежуточной точки ищется ближайшая табличная точка x_i; в качестве значения интерполянта в промежуточной точке берется y_i;

- *interp1*(x, y, xx) или *interp1*(x, y, xx, '*linear*') строит кусочно-линейный интерполянт: на каждом отрезке $[x_i x_{i+1}]$ интерполянт представляет собой линейную функцию $L_i(x)$, при этом $L_i(x_i) = y_i$ и $L_i(x_{i+1}) = y_{i+1}$;

- *spline*(x, y, xx) или *interp1*(x, y, xx, '*spline*') находит кубический сплайн: на каждом отрезке $[x_i x_{i+1}]$ интерполянт представляет собой кубическую функцию $S_i(x)$, при этом $S_i(x_i) = y_i$ и $S_i(x_{i+1}) = y_{i+1}$; кроме того интерполянт имеет непрерывную первую и вторую производные во всех узловых точках и непрерывную третью производную в узлах, соседних с концевыми;

- *pchip*(x, y, xx) или *interp1*(x, y, xx, '*pchip*') или *interp1*(x, y, xx, '*cubic*') ищет эрмитов кубический интерполянт; как и кубический сплайн, кубический интерполянт

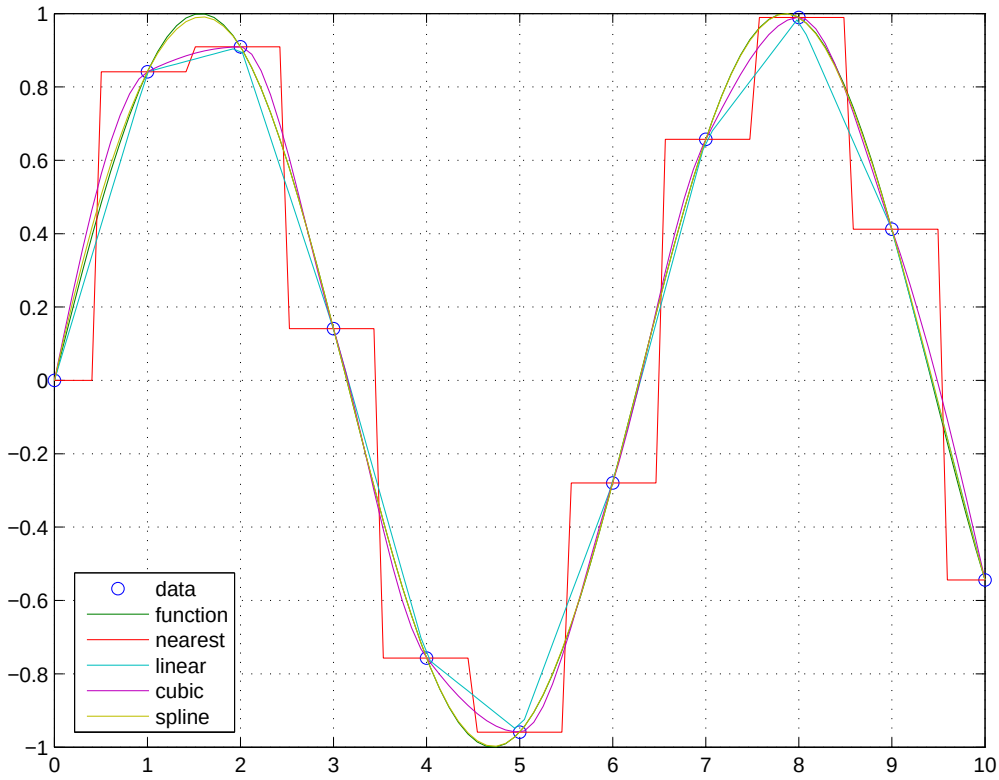


Рис. 4.9. Ступенчатая, линейная, кубическая интрполяция и интерполяция сплайном

на каждом отрезке $[x_i x_{i+1}]$ представляет собой кубическую функцию $C_i(x)$, при этом $C_i(x_i) = y_i$ и $C_i(x_{i+1}) = y_{i+1}$; эта функция имеет непрерывную первую производную, но, в отличие от кубического сплайна, может иметь разрывную вторую производную; с другой стороны, кубический эрмитов интерполянт может лучше воспроизводить поведение данных, чем кубический сплайн (особенно для негладких данных): интерполянт является монотонным на всех участках, где данные монотонны, и имеет экстремумы в тех точках, где имеются экстремумы данных.

Все перечисленные выше функции по узлам x и значениям в узлах y находят кусочно-полиномиальный интерполянт и возвращают его значения в точках xx.

При равномерной сетке x можно использовать более быстрые методы «со звездочкой»: '*nearest**', '*linear**', '*spline**, '*pchip**' — например:

interp1(*x*,*y*,*xx*,'*spline**')

Построим интерполянты функции $y = \sin x$ по 11 точкам и вычислим абсолютные ошибки интерполяции:

```
x = 0:10;
y = sin(x);
xx = linspace(0, 10);
yy = sin(xx);
yn = interp1(x, y, xx, 'nearest');
yl = interp1(x, y, xx, 'linear');
yc = interp1(x, y, xx, 'cubic');
ys = interp1(x, y, xx, 'spline');
plot(x, y, 'o', xx, yy, xx, yn, xx, yl, xx, yc, xx, ys)
grid
legend('data', ...
       'function', ...
       'nearest', ...
       'linear', ...
       'cubic', ...
       'spline', 3);
max(abs(yn - yy))
max(abs(yl - yy))
max(abs(yc - yy))
max(abs(ys - yy))
```

Мы видим, что минимальную ошибку (0.0239) дал сплайн, на втором месте — кубический эрмитов интерполянт (0.1063), далее — линейный интерполянт (0.1220) и на последнем месте — ступенчатый интерполянт (0.4822).

4.3.3. Многомерная интерполяция

В MATLAB'е есть процедуры для интерполяции функций 2-х, 3-х и n переменных:

```
ZZ = interp2(X, Y, Z, XX, YY, method)
UU = interp3(X, Y, Z, U, XX, YY, ZZ, method)
UU = interpn(X1, X2, X3, ..., Xn, U, XX1, XX2, XX3, ..., XXn, UU)
```

Здесь *method* — символьная строка, определяющая алгоритм интерполяции:

- 'linear' — линейная интерполяция (по умолчанию);
- 'cubic' — кубический эрмитов интерполянт;
- 'spline' — кубический сплайн;

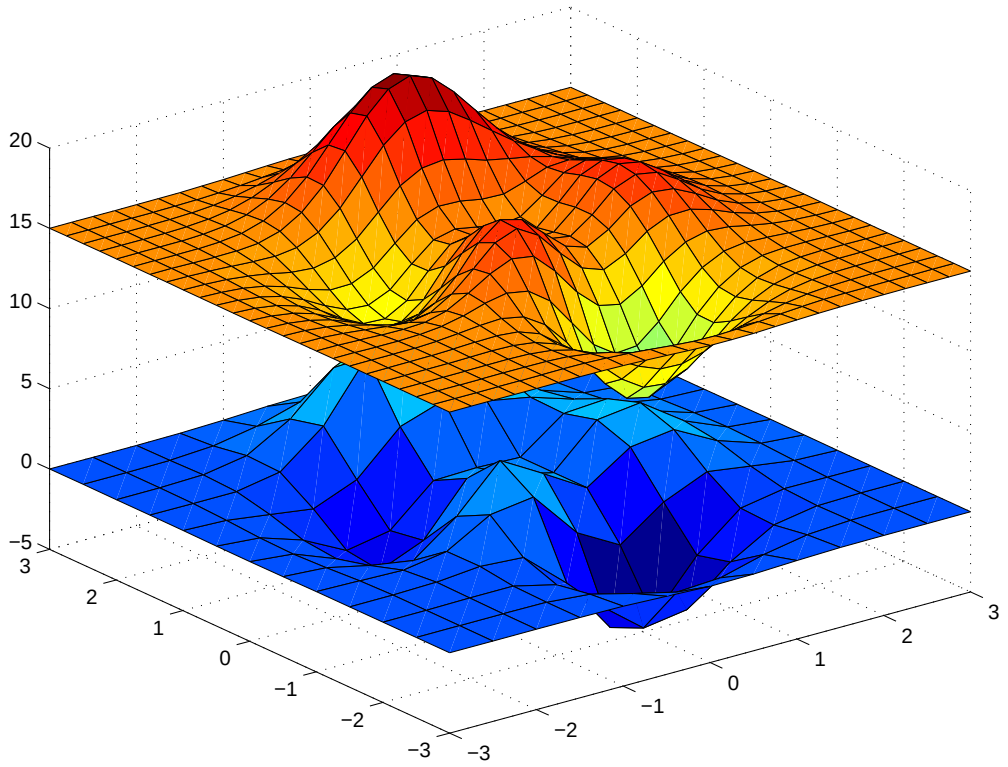


Рис. 4.10. Двумерная интерполяция

- 'nearest' — ступенчатая интерполяция.

Покажем, как работать с *interp2* на примере стандартной MATLAB'овской тестовой функции *peaks*.

Листинг interp/peaksinterp.m

```
[X, Y] = meshgrid(-3:0.5:3);
Z = peaks(X, Y);
[XX, YY] = meshgrid(-3:.25:3);
ZZ = interp2(X, Y, Z, XX, YY, 'spline');
surf(X, Y, Z)
hold on
surf(XX, YY, ZZ + 15)
hold off
axis([-3 3 -3 3 -5 20])
```

Теперь рассмотрим *interp3*. В качестве тестовой функции рассмотрим MATLAB'овскую стандартную функцию *flow*. Функции от трех аргументов визуализируем с помощью процедуры *slice*.

Листинг interp/flowinterp.m

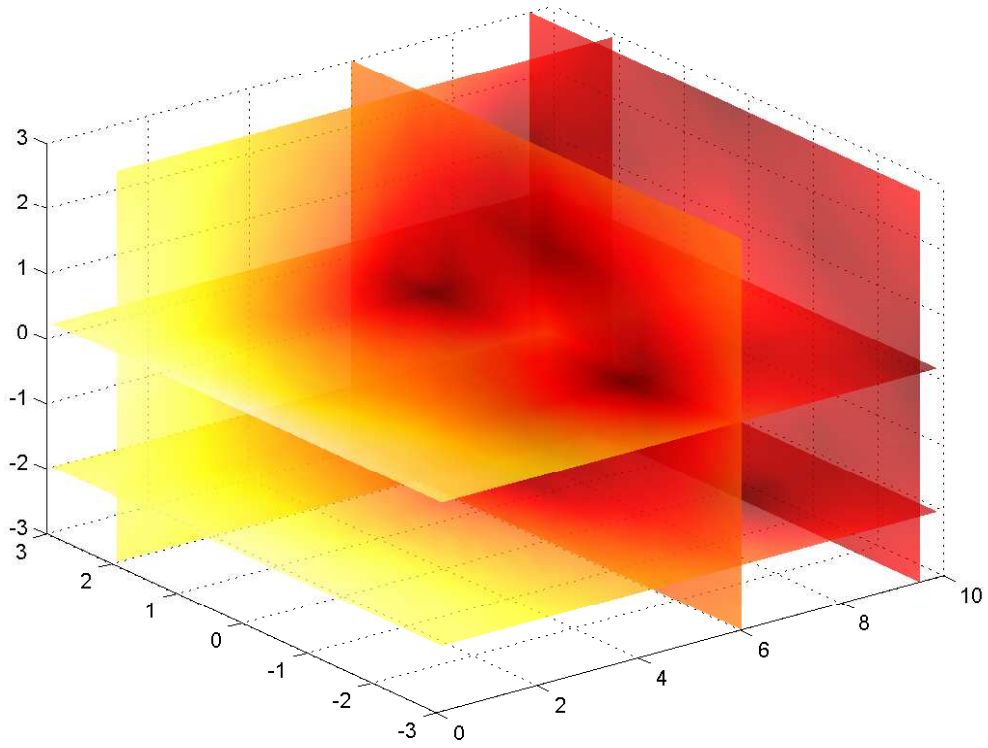


Рис. 4.11. Трехмерная интерполяция

$[X,\ Y,\ Z,\ U] = flow(5);$
$[XX,\ YY,\ ZZ] = meshgrid(.1{:}.25{:}10,\ -3{:}.25{:}3,\ -3{:}.25{:}3);$
$UU = interp3(X,\ Y,\ Z,\ U,\ XX,\ YY,\ ZZ);$
$slice(XX,\ YY,\ ZZ,\ UU,\ [6\ 9.5],\ 2,\ [-2\ .2]);$
$colormap\ hot$
$shading\ interp$
$alpha\ 0.7$

4.4. Численное интегрирование

Задача численного интегрирования заключается в нахождении некоторого приближения к значению определенного интеграла

$$I = \int_a^b f(x)dx,$$

где $[a, b]$ — отрезок интегрирования, а $f(x)$ — заданная функция. В основе методов численного интегрирования лежит суммирование (с некоторыми весами) значений функ-

ции в узлах $x_0, x_1, \ldots, x_n$, выбираемых на отрезке интегрирования:

$$\int_a^b f(x)dx = \sum_{k=0}^{n} A_k f(x_k) + R,$$

где R — остаточный член. Обозначим

$$h = \max\{x_k - x_{k-1} : \ k = 1, 2, \ldots, n\}.$$

4.4.1. Формула прямоугольников

Пусть

$$a = x_0 < x_1 < \cdots < x_{n-1} < x_n = b, \tag{2}$$

тогда

$$\int_a^b f(x)dx \approx \sum_{k=0}^{n-1} f(x_k) \cdot (x_k - x_{k-1}).$$

Эта формула называется *формулой левых прямоугольников*. В случае равноотстоящих узлов ($x_k - x_{k-1} = h$) получаем:

$$\int_a^b f(x)dx \approx h \sum_{k=0}^{n-1} f(x_k).$$

Она соответствует приближенной замене площади криволинейной трапеции площадью ступенчатой фигуры. Аналогична *формула правых прямоугольников*:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{n} f(x_k) \cdot (x_k - x_{k-1}).$$

Для равноотстоящих узлов имеем

$$\int_a^b f(x)dx \approx h \sum_{k=1}^{n} f(x_k).$$

Если $f(x)$ — непрерывно дифференцируемая на отрезке $[a, b]$ функция, то для остаточного члена формул правых и левых прямоугольников справедливо:

$$R = \frac{b-a}{4} f(\xi)\, h, \quad \text{где } \xi \in [a, b].$$

Небольшая модификация позволяет улучшить точность метода. В качестве узлов возьмем точки

$$x_{k+\frac{1}{2}} = \frac{x_k - x_{k-1}}{2} \quad (k = 0, 1, \ldots, n-1),$$

тогда

$$\int_a^b f(x)dx \approx \sum_{k=0}^{n-1} f\left(x_{k+\frac{1}{2}}\right) \cdot (x_k - x_{k-1}).$$

Эта формула называется *формула прямоугольников*. В случае равноотстоящих узлов получаем:

$$\int_a^b f(x)dx \approx h \sum_{k=0}^{n-1} f\left(x_{k+\frac{1}{2}}\right).$$

Если $f(x)$ — дважды непрерывно дифференцируемая на отрезке $[a, b]$ функция, то для остаточного члена формулы прямоугольников справедливо:

$$R = \frac{b-a}{24} f''(\xi)\, h^2, \quad \text{где } \xi \in [a, b].$$

Вычисления по формуле прямоугольников с равноотстоящими узлами в MATLAB'е легко реализовать с помощью функции *sum*. Рассмотрим, например, интеграл

$$I = \int_0^3 \frac{x}{\sin x} dx \tag{3}$$

Вычислим его численно:

```
n = 100;
h = 3/n;
x = h/2:h:3;
y = x./sin(x);
h*sum(y)
```

Получим значение $I \approx 8.4495$.

Легко получить график функции

$$I(x) = \int_0^x \frac{t}{\sin t} dt$$

```
plot(x, h*cumsum(y))
grid
```

Результат см. на рис. 4.12.

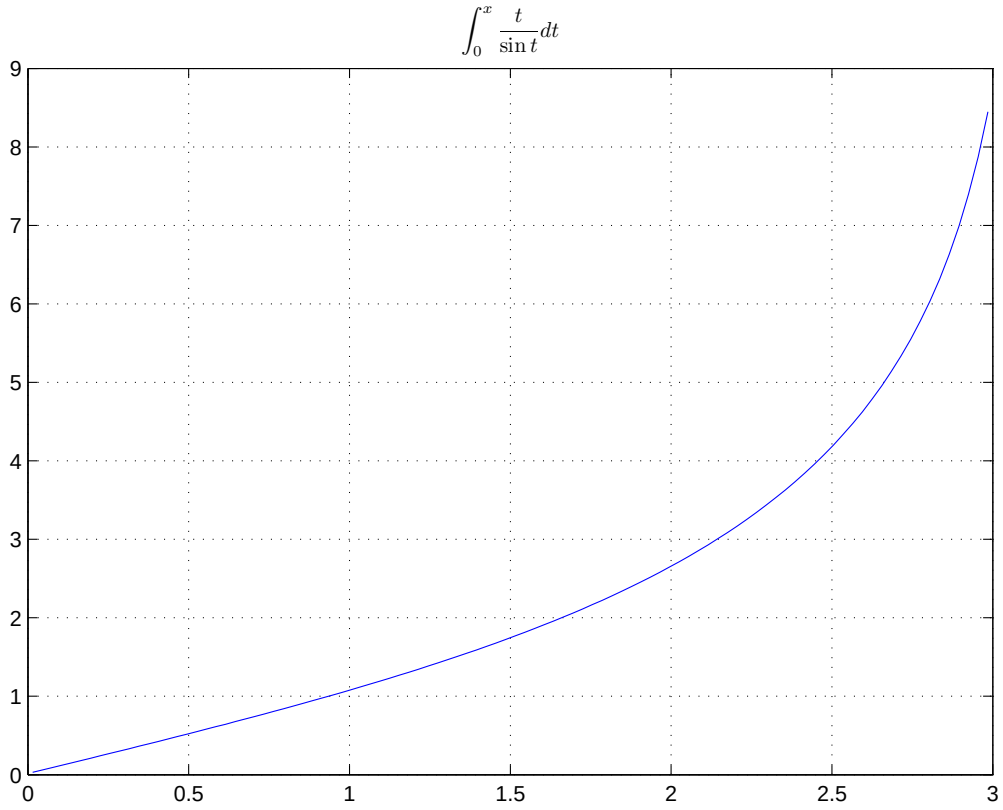


Рис. 4.12. Численное вычисление интеграла с переменным верхним пределом

4.4.2. Формула трапеций

Пусть точки x_k $(k = 0, 1, \ldots, n)$ удовлетворяют условиям (2). Тогда

$$\int_a^b f(x)dx \approx \sum_{k=1}^{n} \frac{f(x_k) + f(x_k)}{2} \cdot (x_k - x_{k-1}).$$

Эта формула называется *формулой трапеций*. Она соответствует замене площади криволинейной трапеции на сумму площадей прямоугольных трапеций. Для равноотстоящих узлов имеем

$$\int_a^b f(x)dx \approx h \left(\frac{f(x_0) + f(x_n)}{2} + f(x_1) + f(x_2) + \cdots + f(x_{n-1}) \right).$$

Если $f(x)$ — дважды непрерывно дифференцируемая на отрезке $[a, b]$ функция, то для остаточного члена формулы трапеций справедливо:

$$R = \frac{b-a}{12} f''(\xi)\, h^2, \quad \text{где } \xi \in [a, b].$$

В MATLAB'е метод трапеций реализует функция *trapz*. Есть два варианта обращения к ней: с одним входным аргументом и с двумя. Если вектор x содержит узлы, а y —

значения функции в этих узлах, то

$trapz(x,\ y)$

численно вычисляет значение определенного интеграла по формуле трапеций.

Например, для численного вычисления интеграла (3) по формуле трапеций введем:

$n\ =\ 100;$
$x\ =\ linspace(eps,\ 3,\ n\ +\ 1);$
$y\ =\ x./sin(x);$
$trapz(x,\ y)$

Получим значение 8.4669. В точке 0 функция имеет (устранимую) особенность, поэтому мы отступили от 0 на величину *eps*.

Вариант функции *trapz* вида

$trapz(y)$

предназначен для интегрирования по формуле трапеций с равноотстоящими узлами. Чтобы получить численное значение интеграла, необходимо значение $trapz(y)$ домножить на шаг интегрирования h. Для нашего примера h*$trapz(y)$ даст, конечно же, то же значение 8.4669.

4.4.3. Правило Симпсона

Пусть точки x_k $(k = 0, 1, \ldots, n)$ удовлетворяют условиям (2), n четно. Заменяя на каждом отрезке $[x_k, x_{k+2}]$ $(k = 0, 2, 4, \ldots, n)$ подынтегральную функцию $f(x)$ ее интерполяционным многочленом $f_k(x)$ 2-го порядка, построенным по узлам x_k, x_{k+1}, x_{k+2}, и интегрируя эти многочлены, мы приходим к *формуле Симпсона*. В частности, для случая равноотстоящих узлов получим:

$$\int_a^b f(x)dx \approx \frac{h}{3}\Big(y_0+y_n+2\big(f(x_2)+f(x_4)+\cdots+f(x_{n-2})\big)+4\big(f(x_1)+f(x_3)+\cdots+f(x_{n-1})\big)\Big).$$

Если функция $f(x)$ имеет непрерывную на отрезке $[a, b]$ производную 4-го порядка, то для остаточного члена формулы Симпсона справедливо:

$R = \dfrac{b-a}{180} f^{IV}(\xi)\, h^4, \quad$ где $\xi \in [a, b]$.

MATLAB'овская Функция *quad* реализует правило Симпсона *с автоматическим выбором шага*. Основную часть вычислительной работы в этой функции выполняет подфункция *quadstep*, в которой используется правило Симпсона с 3 и 5 узлами. Найденные при этом значения сравниваются. Если разница между ними больше заданной

погрешности, то отрезок делится на две равные части и функция *quadstep* рекурсивно применяется к каждой из них.

Форма обращения к функции *quad* следующая:

$$q = quad(f,\ a,\ b)$$

здесь a, b — начало и конец отрезка интегрирования, а f — указатель на подынтегральную функцию, который можно задать одним из следующих способов:

- именем m-функции, заключенным в одинарные кавычки;
- указателем @*fun*, где *fun* — имя m-функции;
- строкой, заключенной в одинарные кавычки, содержащей любую формулу, зависящую от одной переменной.

Заметим, что функция $y = f(x)$ должна быть написана так, чтобы принимать векторный аргумент x и возвращать вектор y, содержащий соответсвующие значения подынтегральной функции.

Для функции *quad* можно в качестве четвертого входного аргумента задать абсолютную погрешность *tol*:

$$q = quad(f,\ a,\ b,\ tol)$$

По умолчанию погрешность равна 10^{-6}.

Пример:

```
format long
quad('x./sin(x)', eps, 3)
quad('x./sin(x)', eps, 3, 1e-3)
```

Получим 8.45527031551890 и 8.45626111362747 соответственно.

Функция *quad*, вызванная с двумя выходными аргументами, во втором из них возвращает количество вычисленных алгоритмом значений подынтегральной функции.

Для нашего примера

```
[q, n] = quad('x./sin(x)', eps, 3)
[q, n] = quad('x./sin(x)', eps, 3, 1e-3)
```

возвратят $n = 85$ и $n = 21$ соответственно.

4.4.4. Метод Лобатто

Функции *quadl* реализует метод интегрирования 8-го порядка. Возможны следующие способы обращения к этой функции:

$q = quadl(f,\ a,\ b)$
$q = quadl(f,\ a,\ b,\ tol)$
$[q,\ n] = quadl(f,\ a,\ b)$
$[q,\ n] = quadl(f,\ a,\ b,\ tol)$

аналогичные соответствующим способам обращения к функции *quad*.

Пример:

$format\ long$
$[q,\ n] = quadl('x./sin(x)',\ eps,\ 3)$
$[q,\ n] = quadl('x./sin(x)',\ eps,\ 3,\ 1e{-}3)$

Получим $q = 8.45527024500206$, $n = 138$ и $q = 8.45527276850248$, $n = 48$ соответственно.

4.4.5. Двойные и тройные интегралы

Идея правила прямоугольников для численного вычисления двойного инеграла

$$I = \int_a^b \int_c^d f(x, y)\, dxdy$$

заключается в следующем. Прямоугольная область $a \le x \le b$, $c \le x \le d$ разбивается на меньшие прямоугольники. Для каждого прямоугольника рассматривается параллелепипед, построенный на этом прямоугольнике как на основании, и с высотой, равной значению подынтегральной функции в центре прямоугольника. Интеграл заменяется на сумму объемов этих параллелепипедов. Аналогично численно вычисляются тройные и многократные интегралы.

Рассмотрим пример

$$I = \int_{-2}^{2} \int_{-2}^{2} \cos(x^2 + y^2) e^{-x^2-y^2}\, dxdy.$$

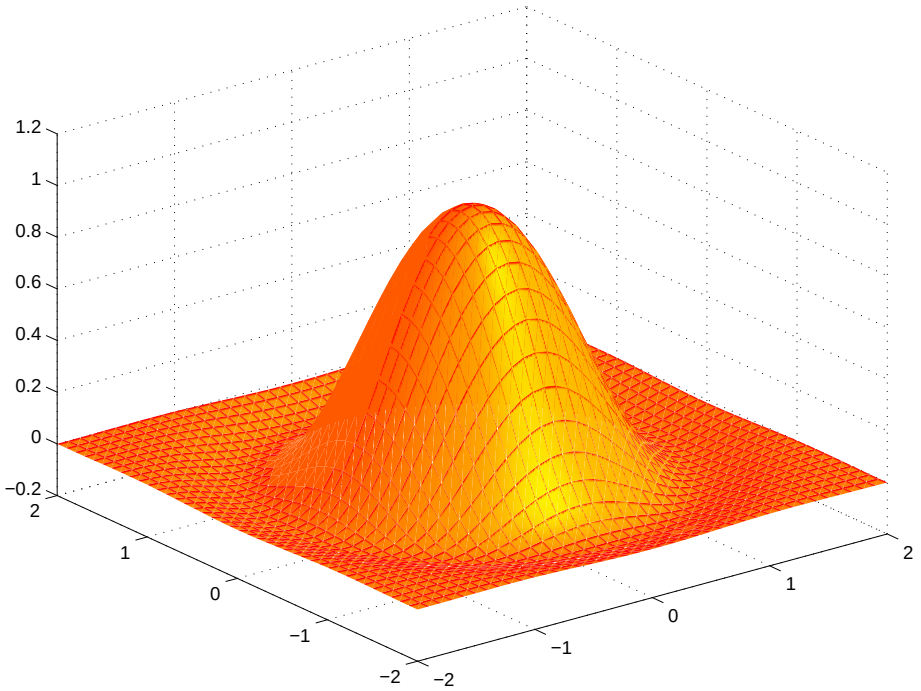


Рис. 4.13. График функции $z = \cos(x^2 + y^2)e^{-x^2-y^2}$.

График подынтегральной функции, полученный командами

$h = 0.1;$
$x = -2{:}h{:}2;$
$[X,\ Y] = meshgrid(x);$
$F = cos(X.\hat{}2 + Y.\hat{}2)\ .*\ exp(-X.\hat{}2 - Y.\hat{}2);$
$surfl(X,\ Y,\ F)$
$colormap\ \ autumn$
$shading\ \ interp$

изображен на рис. 4.13. Теперь вычислим интеграл, заметив, что достаточно найти его значение только для области $0 \le x \le 2$, $0 \le y \le 2$, а затем домножить полученную величину на 4:

$h = 0.05;$
$x = h/2{:}h{:}2;$
$[X,\ Y] = meshgrid(x);$
$F = cos(X.\hat{}2 + Y.\hat{}2)\ .*\ exp(-X.\hat{}2 - Y.\hat{}2);$
$format\ \ long$
$4*h\hat{}2*sum(F(:))$

Получим, что $I \approx 1.56315187656794$. Те же вычисления с шагом $h = 0.05$ приводят к значению интеграла $I \approx 1.56332380183120$.

MATLAB'овская функция

dblquad(*f*, *a*, *b*, *c*, *d*)

вычисляет двойной интеграл, используя функцию *quad*. Как и для функции *quad* можно задать погрешность *tol*:

dblquad(*f*, *a*, *b*, *c*, *d*, *tol*)

по умолчанию *tol* равно 10^6. Пользователь явно может указать MATLAB'у, какой метод использовать в *dblquad* для численного вычисления одномерных интегралов:

dblquad(*f*, *a*, *b*, *c*, *d*, *tol*, *method*)

Здесь *method* — указатель на соответствующую функцию, например, @*quad*.

Для нашего примера

dblquad('*cos*(*x*.^2 + *y*.^2) .* *exp*(−*x*.^2 − *y*.^2)', −2, 2, −2, 2)
dblquad('*cos*(*x*.^2 + *y*.^2) .* *exp*(−*x*.^2 − *y*.^2)', −2, 2, −2, 2, 1e−3)

возвратят соответственно 1.56338069061089 и 1.56253648593122.

Для вычислений тройных интегралов в MATLAB'е есть функция *triplequad*. Возможные способы обращения к ней:

triplequad(*f*, *a*, *b*, *c*, *d*, *e*, *f*)
triplequad(*f*, *a*, *b*, *c*, *d*, *e*, *f*, *tol*)
triplequad(*f*, *a*, *b*, *c*, *d*, *e*, *f*, *tol*, *method*)

аналогичны соответсвующим формам обращения к *dblquad*.

4.5. Численное дифференцирование

Рассмотрим задачу приближения производной заданной дифференцируемой функции $f(x)$. Используя хорошо известную формулу Лагранжа

$$f(a+h) = f(a) + f'(a)h + \frac{f''(\xi)}{2}h^2, \qquad \text{где } \xi \in [a, a+h],$$

получаем

$$f'(a) = \frac{f(a+h) - f(a)}{h} - f''(\xi)\frac{h}{2}.$$

(предполагается, что $f''(x)$ непрерывна). Представляется, что в арифметике без ошибок округления чем меньше h, тем приближение $f'(a)$ разностным отношением

$$f'(a) \approx \frac{f(a+h) - f(a)}{h} \tag{4}$$

точнее, так как для его ошибка равна

$$E_1 = \left| f'(a) - \frac{f(a+h) - f(a)}{h} \right| = \frac{|f''(\xi)|h}{2}.$$

Разностное отношение (4) иногда называют *правой разделенной разностью.*

Напишем программу, численно вычисляющую значение производной в точке по формуле (4).

Листинг start/numdiff.m

Сравнение найденных значений производной функции $f(x) = e^x$ в точке $a = 1$, вычисленных по формуле $f'(a) = (f(a+h) - f(a))/h$ при разных h

```
a = 1;
h = logspace(-1, -16, 16);
d = (exp(a + h) - exp(a))./h;
relerr = abs(d - exp(a))/exp(a);

loglog(h, relerr);
xlabel('h');
ylabel('relerr');
grid;
```

Из полученного графика видно, что относительная ошибка не убывает монотонно с убыванием шага h. Она достигает своего минимального значения, равного 2×10^{-8} при $h = h_{\min} = 10^{-8}$, а затем с уменьшением h начинает возрастать. Это можно объяснить следующим образом. Помимо ошибки «усечения» E_1, метод, реализованный на компьютере содержит ошибки округления. Предположим, что единственная ошибка этого типа происходит только при записи вычисленных значений $f(a)$ и $f(a+h)$ в ячейку памяти. Легко видеть, что абсолютная ошибка при вычислении разностного отношения составит

$$E_2 \le \frac{|f(a+h)| + |f(a)|}{h}\varepsilon_{\text{M}} \approx \frac{2|f(a)|\varepsilon_{\text{M}}}{h}.$$

Таким образом, ошибка округления в разностном отношении растет с уменьшением h. Общая ошибка (ошибка «усечения» + ошибка округления) поэтому составит

$$E_1 + E_2 \approx \frac{M_2 h}{2} + \frac{2|f(a)|\varepsilon_{\text{M}}}{h}.$$

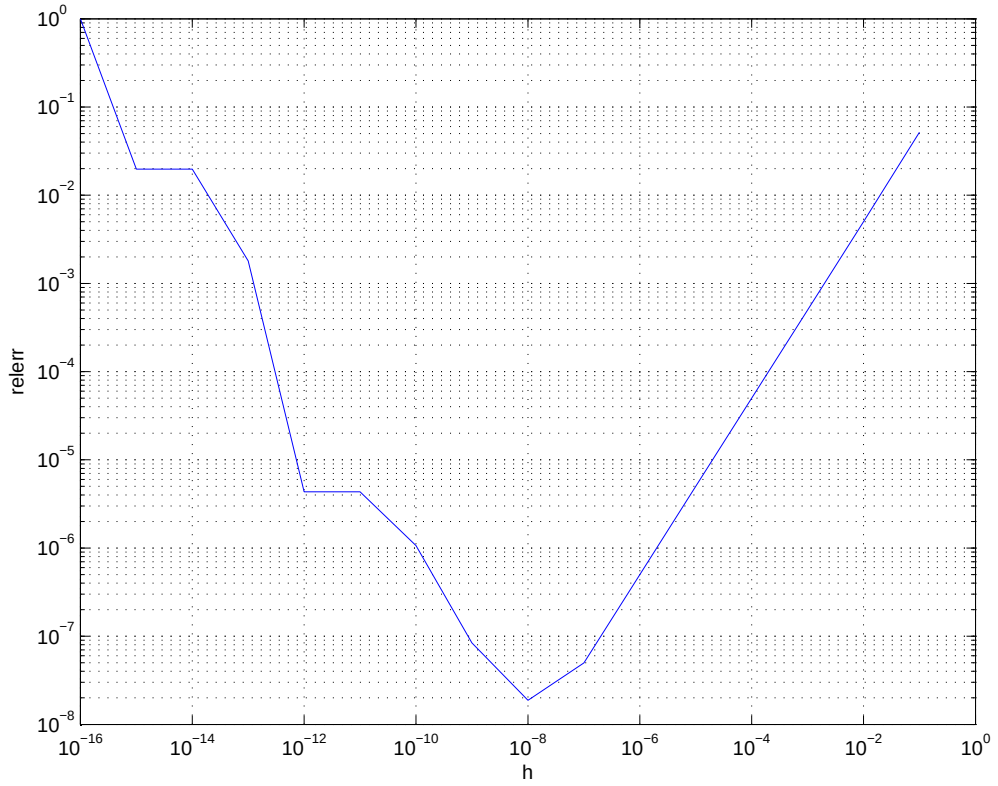


Рис. 4.14. График зависимости точности вычисленной производной от размера шага

Правая часть этого приближенного равенства достигает своего минимального значения при

$$h = h_{\min} = 2\sqrt{\frac{|f(a)|\varepsilon_{\mathrm{M}}}{M_2}}.$$

Если $|f(a)| \approx M_2$, то $h_{\min} \approx \sqrt{\varepsilon_{\mathrm{M}}}$. Конечно, приводимые здесь приближенные равенства очень грубы, но нас устраивает приближение $h_{\min}$ с точностью до порядка. Действительно, в рассматриваемом примере экспериментально мы установили, что $h_{\min} = 10^{-8} \approx \sqrt{\varepsilon_{\mathrm{M}}}$.

Наряду с правой разделенной разностью (4) для апроксимации производной можно использовать *левую разделенную разность*

$$f'(a) \approx \frac{f(a) - f(a-h)}{h} \tag{5}$$

и *центральную разделенную разность*

$$f'(a) \approx \frac{f(a-h) - f(a+h)}{2h} \tag{6}$$

Очевидно, что левая разделенная разность имеет первый порядок точности, как и правая. Центральная разделенная разность имеет второй порядок точности.

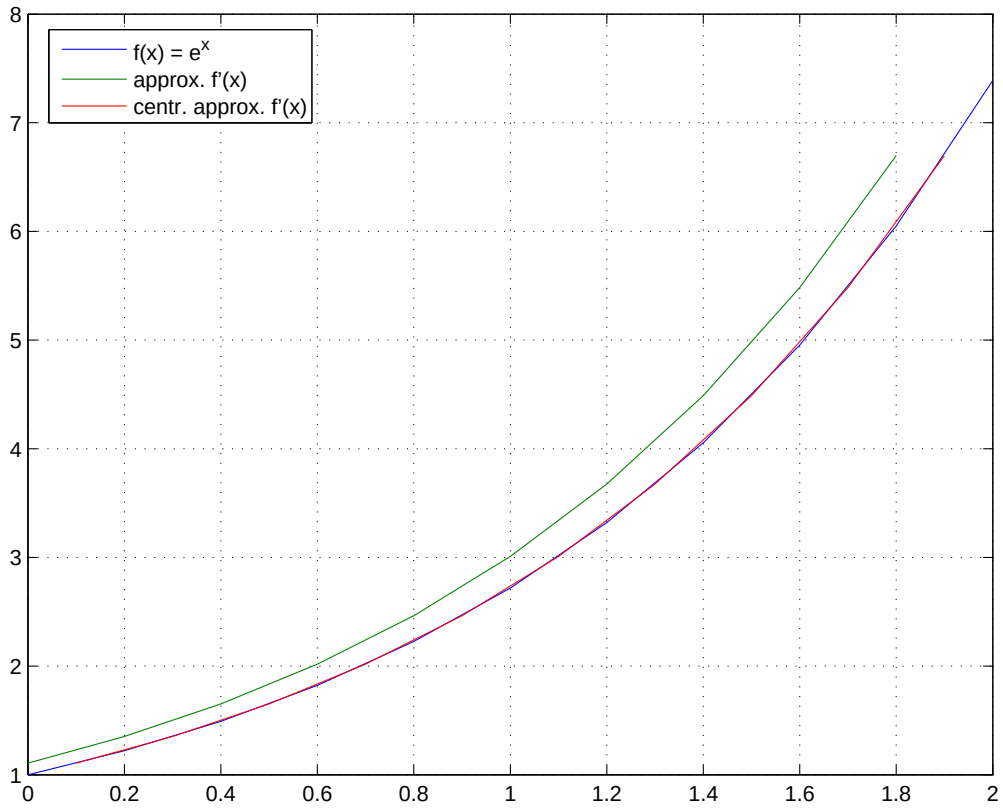


Рис. 4.15. Графики численно вычисленных производных функции e^x

Для численного нахождения производной функции, протабулированной в равномерной сетке, удобна функция *diff*. Если a — вектор длины n, то $diff(a)$ возвращает вектор длины $n-1$, i-я компонента которого равна $a(i+1)-a(i)$. Для матриц $diff(a)$ работает по столбцам.

Для примера вычислим производные функции e^x в наборе точек, используя правую и центрально симметричную разности и построим графики.

Листинг start/diffdiff.m

```
% Численное вычисление производной для набора точек
h = .2;
x = 0:h:2;
f = exp(x);
d = diff(f)/h;
plot(x, f, x(1:end - 1), d, x(1:end - 1) + h/2, d)
grid
legend('f(x) = e^x', 'approx. f''(x)', 'centr. approx. f''(x)', 2)
```

Функция $diff(a, n)$ находит n-ю разность, т. е. применяет *diff* к a рекурсивно n раз.

4.6. Линейная задача наименьших квадратов

Задача наименьших квадратов возникает при *аппроксимации* данных. Предположим, что имеются пары значений

$$(x_1, y_1),\ (x_2, y_2),\ \ldots,\ (x_m, y_m). \tag{7}$$

Пусть $f_1(x), f_2(x), \ldots, f_n(x)$ — заданные функции. Требуется найти функцию вида

$$f(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \cdots + \alpha_n f_n(x), \tag{8}$$

для которой $f(x_i) \approx y_i$ $(i = 1, 2, \ldots, m)$, т. е. функцию, приближающую данные (7). Более точно: требуется найти значения параметров $\alpha_1, \ldots, \alpha_n$, на которых достигается минимум

$$\min_{\alpha_1,\ldots,\alpha_n} \sum_{i=1}^{m} \Big(y_i - \alpha_1 f_1(x_i) - \cdots - \alpha_n f_n(x_i)\Big)^2.$$

Легко видеть, что набор этих параметров является псевдорешением системы

$$\begin{cases} \alpha_1 f_1(x_1) + \ \alpha_2 f_2(x_1) + \ldots + \ \alpha_n f_n(x_1) = y_1, \\ \alpha_1 f_1(x_2) + \ \alpha_2 f_2(x_2) + \ldots + \ \alpha_n f_n(x_2) = y_2, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \alpha_1 f_1(x_m) + \alpha_2 f_2(x_m) + \ldots + \alpha_n f_n(x_m) = y_m. \end{cases}$$

И наоборот, если $\alpha_1, \ldots, \alpha_n$ — псевдорешение указанной системы, то функция $f(x)$, определяемая по формуле (8), является решением задачи аппроксимации.

Часто в качестве функций $f_1(x), f_2(x), \ldots, f_n(x)$ рассматриваются многочлены $1, x, \ldots, x^{n-1}$. Таким образом, речь идет об аппроксимации данных полиномами.

В MATLAB'е для решения линейной задачи наименьших квадратов можно использовать команду $A\backslash y$. Для аппроксимации полиномами также можно использовать функцию *polyfit*.

Предположим, имеются данные

x = (1:10)';
y = [6; 1; 24; 37; 75; 107; 137; 150; 235; 305];

Приблизим их многочленом 2-й степени:

f = *polyfit*(x, y, 2)

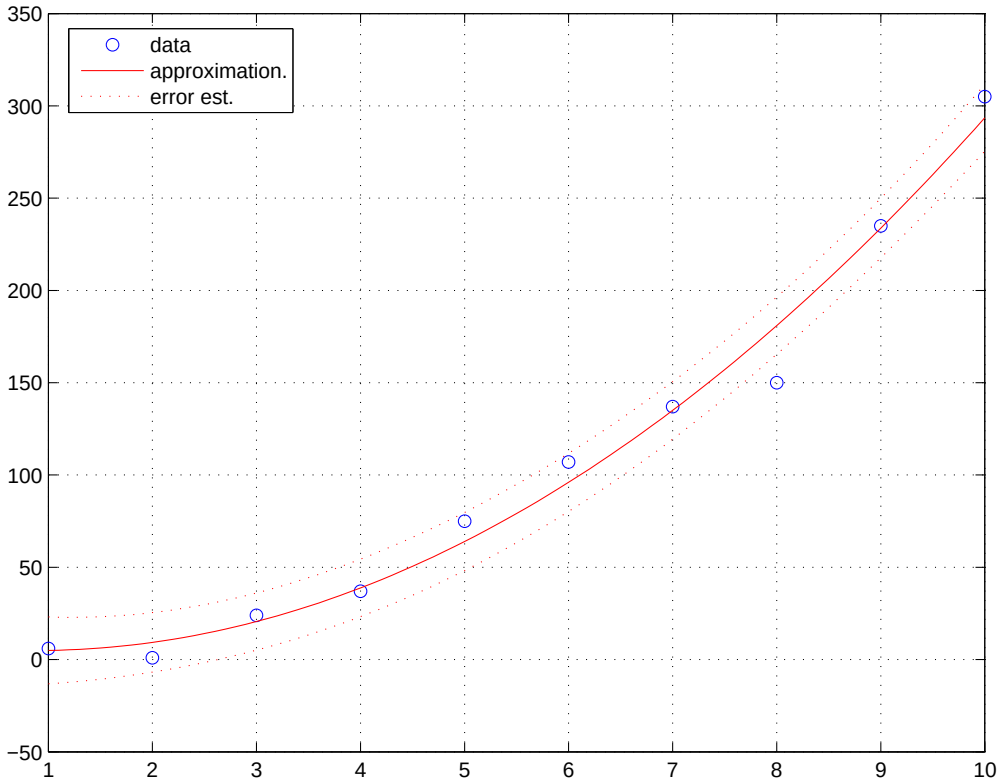


Рис. 4.16. Аппроксимация данных методом наименьших квадратов

Получим коэффициенты 3.4621, −6.0167, 7.5000. Таким образом, аппроксимирующий многочлен равен

$$f(x) = 3.4621x^2 - 6.0167x + 7.5000.$$

Коэффициенты многочлена можно было найти с помощью команды \:

$$A = [x.\hat{}2,\ x,\ ones(size(x))];$$
$$f = A \backslash y$$

Получим тот же многочлен $f(x)$. Вычислим ошибку и построим график аппроксимирующего многочлена:

$$err = norm(polyval(f,\ x)\ -\ y)$$
$$xx = 1{:}0.1{:}10;$$
$$yy = plot(x,\ y,\ 'o',\ xx,\ yy,\ 'r');$$

Ошибка составит 37.6958.

Для улучшения численных характеристик данных, можно провести их предварительное центрирование и масштабирование, воспользовавшись следующим вариантом

вызова функции *polyfit*:

$[f,\ S,\ q] = polyfit(x,\ y,\ n);$

В этом случае выдаются коэффициенты аппроксимирующего многочлена $\widehat{f}(x)$ для центрированных и масштабированных данных. Связь между $\widehat{f}(x)$ и $f(x)$ задается формулой:

$$\widehat{f}\left(\frac{x-\mu}{\sigma}\right) = f(x),$$

где μ — среднее значение, а σ — стандартное отклонение данных в x:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i, \qquad \sigma = \sqrt{\frac{1}{m-1}\sum_{i=1}^{m}(x_i-\mu)^2}.$$

Величины μ и σ возвращаются в векторе q. Структура S содержит информацию, которую можно использовать в расширенном варианте функции *polyval*:

$[yy,\ delta] = polyval(f,\ xx,\ S,\ q);$

Здесь *delta* — вектор такой же длины, что и xx. Он характеризует оценку ошибки. Если ошибки в данных, содержащихся в векторе y, независимы и распределены по нормальному закону с математическим ожиданием 0 и постоянным среднеквадратическим отклонением, то интервал $[yy - delta, yy + delta]$ содержит 50% предсказаний. Вычислим и изобразим на графике эти границы:

```
[f, S, q] = polyfit(x, y, 2);
[yy, delta] = polyval(f, xx, S, q);
plot(x, y, 'o', xx, yy, 'r', xx, [yy - delta; yy + delta]', 'r:');
grid;
legend('data', 'approximation.', 'error est.', 2)
```

Полученный результат приведен на рис. 4.16.

Заметим, что центрирование и масштабирование данных можно провести вручную, если воспользоваться функциями $mean(y)$ и $std(y)$, возвращающими среднее значение и стандартное отклонение соответственно.

4.7. Дискретное преобразование Фурье

Пусть $x = (x_0, x_1, \ldots, x_{n-1})$ — вещественный или комплексный вектор длины n. Его *дискретным пробразованием Фурье* называется комплексный вектор $X = (X_0, X_1, \ldots, X_{n-1})$,

компоненты которого определяются по формулам:

$$X_p = \sum_{k=0}^{n-1} x_k e^{-2\pi ikp/n}, \quad (p = 0, \ldots, n-1).$$

По вектору X определить вектор x можно с помощью *обратного преобразования Фурье*:

$$x_k = \frac{1}{n} \sum_{p=0}^{n-1} X_p e^{2\pi ikp/n}, \quad (k = 0, \ldots, n-1).$$

Алгоритм дискретного преобразования Фурье в MATLAB'е осуществляет функция *fft*(x). Для обратного преобразования есть функция *ifft*(X).

Приведем небольшой пример, показывающий использование преобразования фурье для частотного анализа данных. Рассмотрим функцию

$$x(t) = \sin 2\pi\omega_1 t + \sin 2\pi\omega_2 t, \qquad \omega_1 = 770, \qquad \omega_2 = 1477,$$

представляющую собой сумму двух гармонических колебаний с частотами ω_1, ω_2. На отрезке $[0, 0.25]$ рассмотрим сетку с частотой (дисретизации) *fs*:

```
T = 0.25
fs = 8192;
t = 0:1/fs:T;
```

и вычислим значения $x(t)$ в узлах этой сетки:

```
w1 = 770;
w2 = 1477;
x1 = sin(2*pi*w1*t);
x2 = sin(2*pi*w2*t);
x = x1 + x2;
plot(t, x);
```

Получим дискретный сигнал. Его можно озвучить с помощью функции *sound*:

```
sound(x1, fs)
sound(x2, fs)
sound([x1; x2]', fs)
sound(x, fs)
```

Вычислим дискретное преобразование Фурье от x:

```
X = fft(x);
```

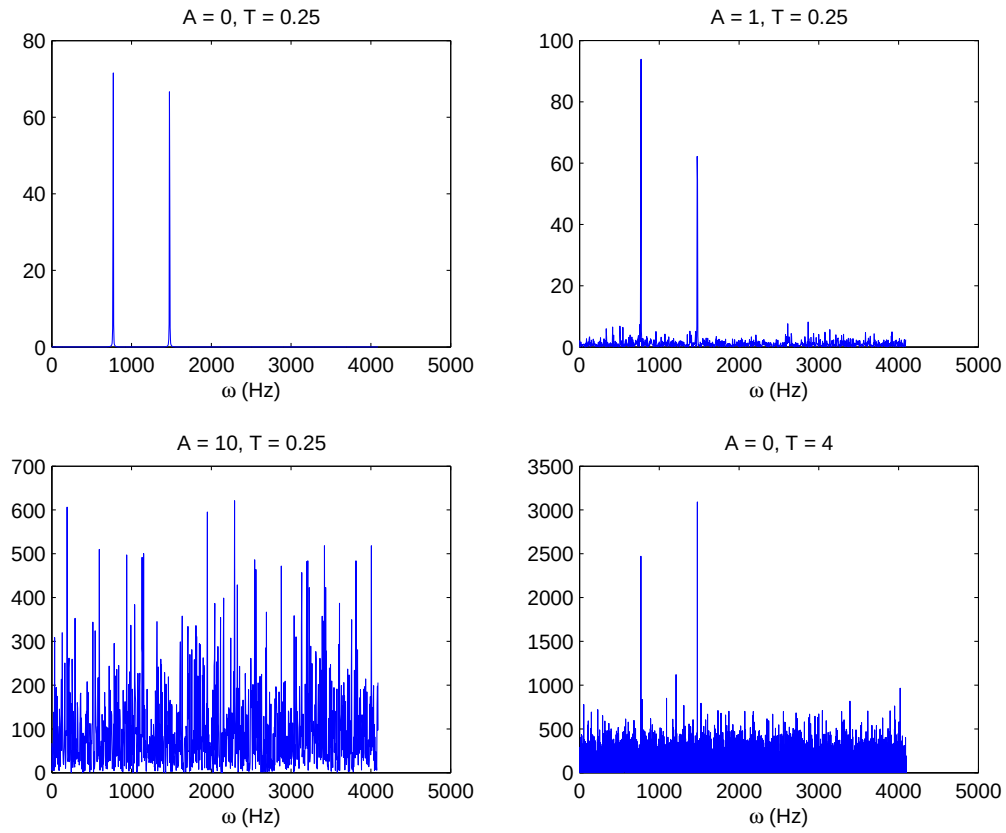


Рис. 4.17. Спектр мощности при различном зашумлении и различной длине выборки

Спектром мощности сигнала x называется функция

$$P(\omega) = \frac{1}{n}\sum_{p=0}^{n-1} |X_p|^2.$$

Она характеризует присутствие частот в сигнале. Чтобы проиллюстрировать это, построим график $P(\omega)$ для $0 < \omega \leq fs$:

```
n = length(x);
P = X .* conj(X) / n;
w = fs * (0:(n/2 - 1))/n;
plot(w, P(1:n/2))
title('Frequency content')
xlabel('\omega (Hz)')
```

На графике мы увидим два пика, соответствующие частотам ω_1, ω_2.

Добавим в сигнал шум:

```
A = .5;
x = x + A*randn(size(x));
sound(x, fs);
```

и также построим график спектра мощности. На нем по-прежнему будут хорошо видны два пика, соответсвующие исходным частотам, но появится случайный шум на всем спектре. Увеличивая A, заметим, что мощность шума растет (см. рис. 4.17).

Увеличивая длину выборки, например, положив

$t = 0{:}1/fs{:}4;$

удается преодолеть более сильный шум.

4.8. Оптимизация

4.8.1. Одномерная оптимизация

Функция $f(x)$ одного аргумента x называется *унимодальной* на отрезке $[a, b]$, если на отрезке нем найдется такая точка x_0 (точка минимума), что $f(x)$ монотонно убывает при $a \le x \le x_0$ и монотонно возрастает при $x_0 \le x \le b$. Для минимизации унимодальной функции, заданной на отрезке MATLAB предоставляет функцию *fminbnd*. Ее можно применять и для минимизации функций, не являющихся унимодальными. В этом случае будет найден локальный минимум.

fminbnd использует комбинацию методов золотого сечения и последовательной параболической интерполяции. Реализована функция в виде m-файла, поэтому желающие могут ознакомиться с деталями по исходному коду.

Простейший вариант вызова *fminbnd* имеет вид

$x = fminbnd(f,\ a,\ b)$

Здесь f — либо символьная строка, содержащая математическую запись выражения, либо указатель на функцию вида @*fun*, где *fun* — имя функции, например, имя m-файла. В первом случае именем независимой переменной может быть только x.

Рассмотрим функцию

$$y = x^6 + 7x^4 - 2x, \quad x \in [-2, 2].$$

Сперва построим ее график:

```
ezplot('x^6 + 7*x^4 - 2*x', -2, 2)
grid
```

Функция

```
fminbnd('x^6 + 7*x^4 - 2*x', -2, 2)
```

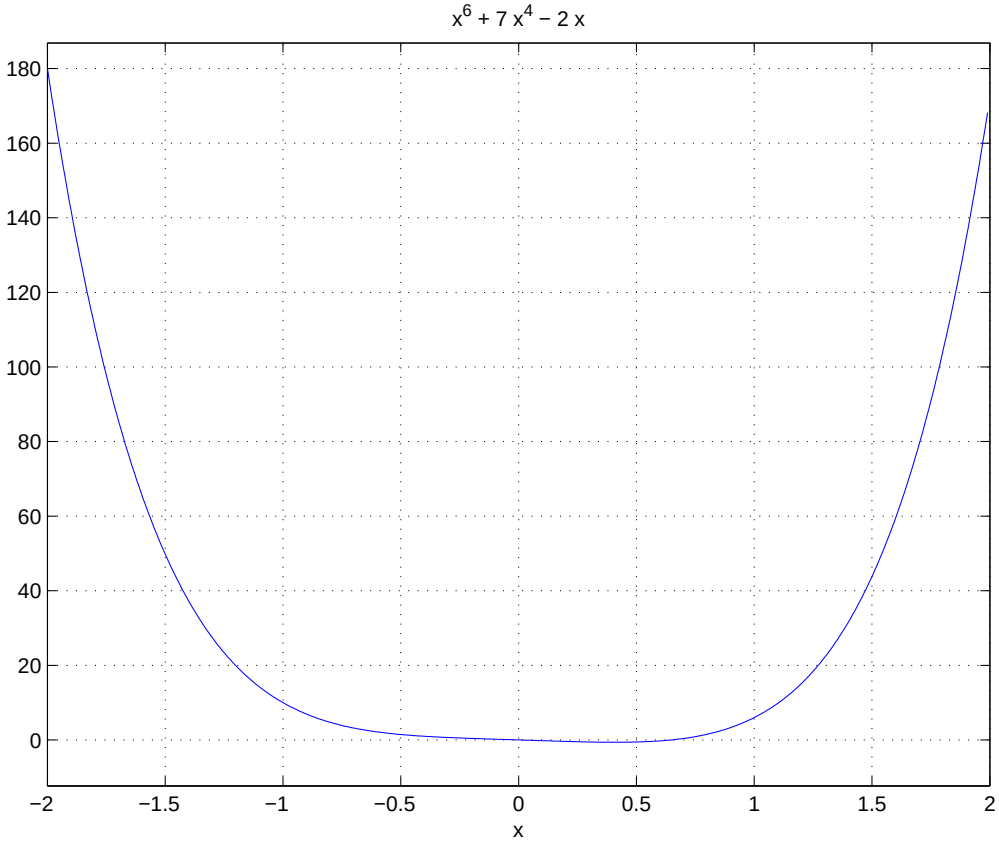


Рис. 4.18. Тестовая функция для метода fminbnd

найдет точку минимума $x = 0.4100$.

Можно оформить целевую функцию в виде m-файла или разместить ее как подфункцию в другой m-файл:

function $y = myfun(x)$

$y = x\hat{}6 + 7*x\hat{}4 - 2*x;$

Теперь вызвать *fminbnd* можно так:

$fminbnd(@myfun, -2, 2)$

Функцию *fminbnd* можно вызывать с разным количеством входных и выходных аргументов:

$x = fminbnd(f, a, b)$
$x = fminbnd(f, a, b, options)$
$x = fminbnd(f, a, b, options, P1, P2, \ldots, Pn)$
$[x, fval] = fminbnd(\ldots)$
$[x, fval, exitflag] = fminbnd(\ldots)$
$[x, fval, exitflag, output] = fminbnd(\ldots)$

options — это структура с дополнительными параметрами (опциями). Перечислим возможные опции:

Display: значение '*Off*' запрещает любые сообщения о ходе работы алгоритма; '*Iter*' сообщает о ходе вычислений на каждой итерации; '*Final*' выдает информацию только по окончании работы; сообщает о ходе вычислений на каждой итерации; '*Converge*' выдает информацию, только в случае, если алгоритм не сошелся (по умолчанию).

FunValCheck: если значение '*On*', то выдает предупреждение вскяий раз, когда значение функции — комплексное число или *NaN*; если значение опции '*Off*', то предупреждения не выдаются.

MaxFunEvals задает максимальное количество вычислений значений целевой функции.

MaxIter задает максимальное число итераций.

OutputFcn определяет пользовательскую функцию, которую MATLAB будет вызывать на каждой итерации.

TolX — допуск на ошибку в независимой переменной.

Определить поля структуры *options* можно с помощью функции *optimset*. Например,

options = *optimset*('*Display*', '*Iter*', '*OutputFcn*', @*OutputFcn*);
x = *fminbnd*(@*fun*, *x1*, *x2*, *options*);

fminbnd(*f*, *a*, *b*, *options*, *P1*, *P2*, ..., *Pn*) позволяет задать значения дополнительных параметров целевой функции. Например, если целевая функция задана так:

function *y* = *myfun*(*x*, *a*, *b*, *c*)

y = *a***x*^6 + *b***x*^4 + *c***x*;

то вызвать метод *fminbnd* можно командой:

fminbnd(@*myfun*, −2, 2, 1, 7, −2)

Всякий раз, когда алгоритму будет необходимо вычислить значение целевой функции, параметры *a*, *b*, *c* примут соответственно значения 1, 7, −2.

[*x*, *fval*] = *fminbnd*(...) кроме найденной точки минимума x, возвращает также само значение $f(x)$ в найденной точке x.

[*x*, *fval*, *exitflag*] = *fminbnd*(...) возвращает флаг окончания *exitflag*. Его значение равно 1, если минимум с допуском, определяемым параметром *options.TolX*, найден. Флаг равен 0, если количество итераций или количество вычисленных значений функции превысили величины *options.MaxIter* и *options.MaxFunEvals* соответственно. Флаг равен −1, если алгоритм был остановлен пользовательской функцией *options.OutputFcn*. И, наконец, *exitflag* равен −2, если $a > b$.

[*x*, *fval*, *exitflag*, *output*] = *fminbnd*(...) кроме того возвращает структуру *output*, в которой *output.algorithm* — символьная строка, содержащая название используемого алгоритма, *output.funcCount* — количество вычислений значений целевой функции, *output.iterations* — общее число итераций.

Для нашего примера

[*x*, *fval*, *flag*, *output*] = *fminbnd*('*x*^6 + 7**x*^4 − 2**x*', −2, 2)

найдет $x = 0.4100$, *fval* = −0.6174, *flag* = 1, *output.iterations* = 11, *output.funcCount* = 12, *output.algorithm* = 'golden section search, parabolic interpolation' *output.message* = 'Optimization terminated: the current x satisfies the termination criteria using *options.TolX* of 1.000000e−004'

4.8.2. Безусловная многомерная оптимизация

Рассмотрим задачу поиска локального минимума целевой функции $f(x)$, зависящей от векторного аргумента x:

$$\min_x f(x), \quad \text{где } x \in \mathbf{R}^n$$

Симплексный алгоритм Нелдера–Мида

Один из алгоритмов минимизации — симплексный алгоритм Нелдера–Мида. Его идея заключается в следующем. На предварительном шаге выбирается $n+1$ точек, не расположенных в одной гиперплоскости. Эти точки являются вершинами симплекса, откуда название алгоритма. Точка, в которой значение функции максимально, удаляется и вместо нее по определенным правилам выбирается другая. Итерации продолжаются до тех пор, пока симплекс не станет достаточно малым. Алгоритм Нелдера–Мида можно использовать для минимизации негладких и даже разрывных функций.

Функция

x = fminsearch(f, x0)

реализует алгоритм Нелдера–Мида. Здесь *f* — указатель на функцию, *x0* — стартовая точка, *x* — найденная точка локального минимума. Можно получить также значение *fval* целевой функции в найденной точке:

[x, fval] = fminsearch(f, x0)

Рассмотрим функцию

$$f(x) = 2x_1^2 - 1.05x_1^4 + \frac{1}{6}x_1^6 + x_1x_2 + x_2^2$$

Ее график можно получить командами

```
ezsurf('2*x^2 - 1.05*x^4 + x^6/6 + x*y + y^2', [-2, 2, -2, 2]);
shading interp
view(57, 69)
```

Испытаем несколько стартовых точек:

```
fun = '2*x(1)^2-1.05*x(1)^4+x(1)^6/6+x(1)*x(2)+x(2)^2';
[x, fval] = fminsearch(fun, [0, 0])
[x, fval] = fminsearch(fun, [-1.5, 1])
[x, fval] = fminsearch(fun, [2, -1])
[x, fval] = fminsearch(fun, [2, 2])
```

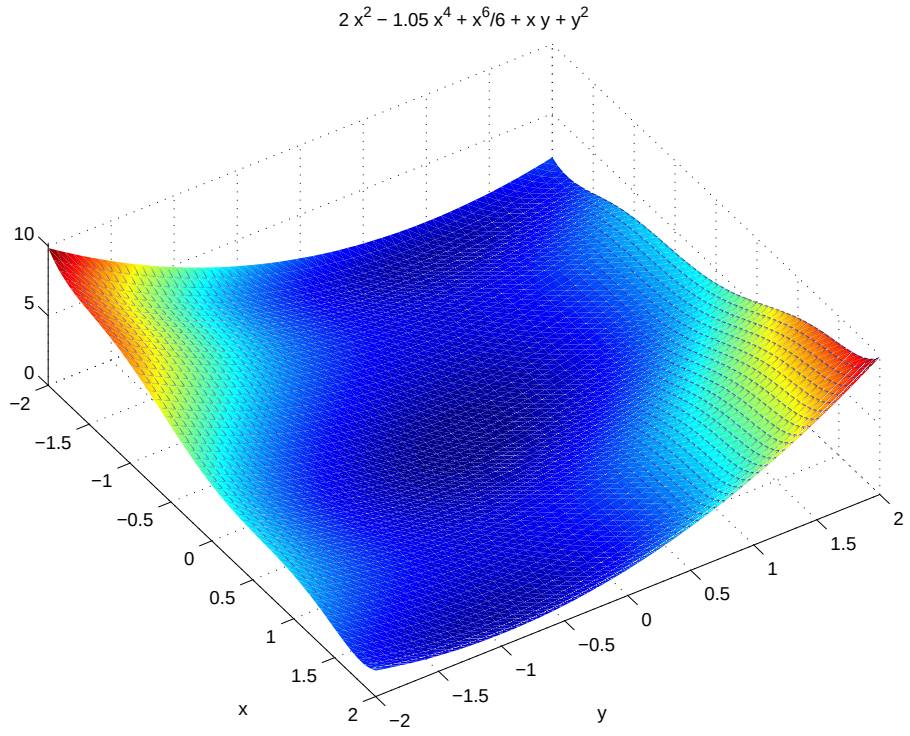


Рис. 4.19. Тестовая функция

Получим соответственно:

$$x =$$
$$0 \quad 0$$
$$fval =$$
$$0$$

$$x =$$
$$-1.7475 \quad 0.8738$$
$$fval =$$
$$0.2986$$

$$x =$$
$$1.7475 \quad -0.8738$$
$$fval =$$
$$0.2986$$

$x =$

1.0e−004 *

−0.3842 0.2342

$fval =$

2.6003e−009

Функцию *fminsearch* можно вызывать с разным количеством входных и выходных аргументов:

$x = fminsearch(fun,\ x0)$
$x = fminsearch(fun,\ x0,\ options)$
$x = fminsearch(fun,\ x0,\ options,\ P1,\ P2,\ \ldots,\ Pn)$
$[x,\ fval] = fminsearch(\ldots)$
$[x,\ fval,\ exitflag] = fminsearch(\ldots)$
$[x,\ fval,\ exitflag,\ output] = fminsearch(\ldots)$

Эти способы, по-существу, повторяют соответсвующие способы вызова функции *fminbnd*. Единственное отличие заключается в том, что возможные значения флага *exitflag* теперь — только 1, 0, −1.

Гладкая оптимизация

Функция *fminunc* из пакета Optimization Toolbox реализует следующие методы гладкой безусловной оптимизации:

- метод наискорейшего спуска (Steepest Descent method),
- квази-Ньютоновский BFGS-метод (Broyden–Fletcher–Goldfarb–Shanno),
- квази-Ньютоновский DFP-метод (Davidon–Fletcher–Powell),
- методы, основанные на построении доверительных двумерных областей (trust region).

Первые три из них отнесены к «medium-scale»-методам и предназначены для решения задач оптимизации средней размерности (например, не больше 100 переменных), последний — к «large-scale»-методам. По умолчанию, если пользователь задает аналитическое выражение для градиента, запускается метод доверительного интервала. В противном случае используется BFGS-метод, но есть возможность переключиться и на другие. Метод наискорейшего спуска, как правило, очень медлителен и его не рекомендуется использовать для решения практических задач. В MATLAB'е он присутствует

только для иллюстративных целей. Функция *fminunc* вызывается одним из следующих способов:

$x = fminunc(f,\ x0)$
$x = fminunc(f,\ x0,\ options)$
$x = fminunc(f,\ x0,\ options,\ P1,\ P2,\ \ldots,\ Pn)$
$[x,\ fval] = fminunc(\ldots)$
$[x,\ fval,\ exitflag] = fminunc(\ldots)$
$[x,\ fval,\ exitflag,\ output] = fminunc(\ldots)$
$[x,\ fval,\ exitflag,\ output,\ grad] = fminunc(\ldots)$
$[x,\ fval,\ exitflag,\ output,\ grad,\ hessian] = fminunc(\ldots)$

в целом, соответсвующих аналогичным способам вызова *fminsearch*, однако со значительно расширенным списком возможных параметров *options*, новыми значениями флага *exitflag* и новыми полями структуры *output*. Кроме того, появилась возможность получить в финальной точке градиент *grad* и гессиан *hessian*.

Флаг *exitflag* теперь может принимать значения от -2 до 3. значения 1, 2, 3 означают, что итерации завершены успешно: норма градиента в текущей точке, последний шаг или изменения значения целевой функции стали меньше соответсвующего допуска. Значение 0 означает, что количество итераций или количество вычислений целевой функции првысили *options.MaxIter* или *options.FunEvals*. Значение -1 означает, что алгоритм был прерван пользовательской функцией *options.OutputFcn*. Значение -2 свидетельствует о том, что на вычисленном текущем направлении примлемая точка не найдена (только для «medium-scale»-алгоритмов).

Структура *output* теперь содержит следующие поля: *output.algorithm* — символьная строка, содержащая название используемого алгоритма, *output.funcCount* — количество вычислений значений целевой функции, *output.iterations* — общее число итераций, *cgiterations* — число итераций метода сопряженных градиентов (для «large-scale»-алгоритмов), *stepsize* — величина последнего шага (для «medium-scale»-алгоритмов).

В качестве примера рассмотрим минимизацию известной тестовой «банановой» функции Розенброка:

$$f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

Минимум функции Розенброка равен нулю и достигается в точке $(1, 1)$. Получить графическое изображение линий уровня можно с помощью функции *bananadraw*.

Листинг optim/bananadraw.m

Функция рисует «банан» Розенброка $f(x_1, x_2) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$

```
function bananadraw(x0)

    h = 0.025;
    x = -3:h:3;
    y = -4:h:5;
    [X, Y] = meshgrid(x, y);
    F = 100*(Y - X.^2).^2 + (1 - X).^2 + 100;
    clf
    shg
    map = hsv(64);
    map = map([8:end, 1:7], :);
    colormap(map);
    surf(X, Y, F);
    shading interp
    view(18, 55);
    alpha 0.9
    hold on
```

Теперь напишем функцию, вычисляющую значение $f(x)$:

Листинг optim/banana.m

```
function f = banana(x)

    f = 100*(x(2) - x(1).^2).^2 + (1 - x(1)).^2;
```

Запускаем алгоритм:

```
[x, fval, exitflag, output] = fminunc(@banana, [-1, 2])
```

MATLAB выдает предупреждение: «Warning: Gradient must be provided for trust-region method; using line-search method instead», смысл которого заключается в следующем. По умолчанию, процедура *fminunc* запускает «large-scale»-метод, основанный на построении доверительного интервала, однако этот метод требует наличия аналитически вычисленного градиента функции. Так как мы не указали MATLAB'у процедуру, вычисляющую этот градиент, MATLAB переключился на «medium-scale» BFGS-алгоритм. В конечном итоге *fminunc* удачно завершает исполнение ($exitflag = 1$) и выдает точку

$x = (1.0000, 1.0000)$ и значение функции в ней $fval = 1.2262\text{e}{-010}$. Алгоритму потребовалось $options.iterations = 39$ итераций и $funcCount = 150$ вычислений значений функции.

Для того чтобы в ходе работы алгоритма отображались точки, в которых он вычисляет значения $f(x)$, напишем следующую функцию

Листинг optim/bananaoutputfcn.m

```
function stop = bananaoutputfcn(x, vals, state)

  switch state
    case 'init'
        clf
        shg
        bananadraw
        plot3(x(1), x(2), vals.fval(1) + 1000, 'm.', 'MarkerSize', 20);
        plot3(x(1), x(2), vals.fval(1) + 1000, 'ko', 'MarkerSize', 10);
        text(x(1), x(2), vals.fval(1) + 1000, 'Start point ', ...
            'HorizontalAlignment', 'Right')
        plot3(1, 1, 1000, 'm.', 'MarkerSize', 20);
        plot3(1, 1, 1000, 'ko', 'MarkerSize', 10);
        text(1, 1, 1000, 'Optimum ', ...
            'HorizontalAlignment', 'Right', ...
            'VerticalAlignment', 'Bottom')
    case 'iter'
        plot3(x(1), x(2), vals.fval + 1000, 'r.', 'MarkerSize', 20);
        pause(0.15);
    case 'done'
        plot3(x(1), x(2), vals.fval + 1000, 'r.', 'MarkerSize', 20);
        plot3(x(1), x(2), vals.fval(1) + 1000, 'ko', 'MarkerSize', 10);
        text(x(1), x(2), vals.fval(1) + 1000, '  Final point', ...
            'HorizontalAlignment', 'Left', 'VerticalAlignment', 'Bottom')
  end;

  stop = 0;
```

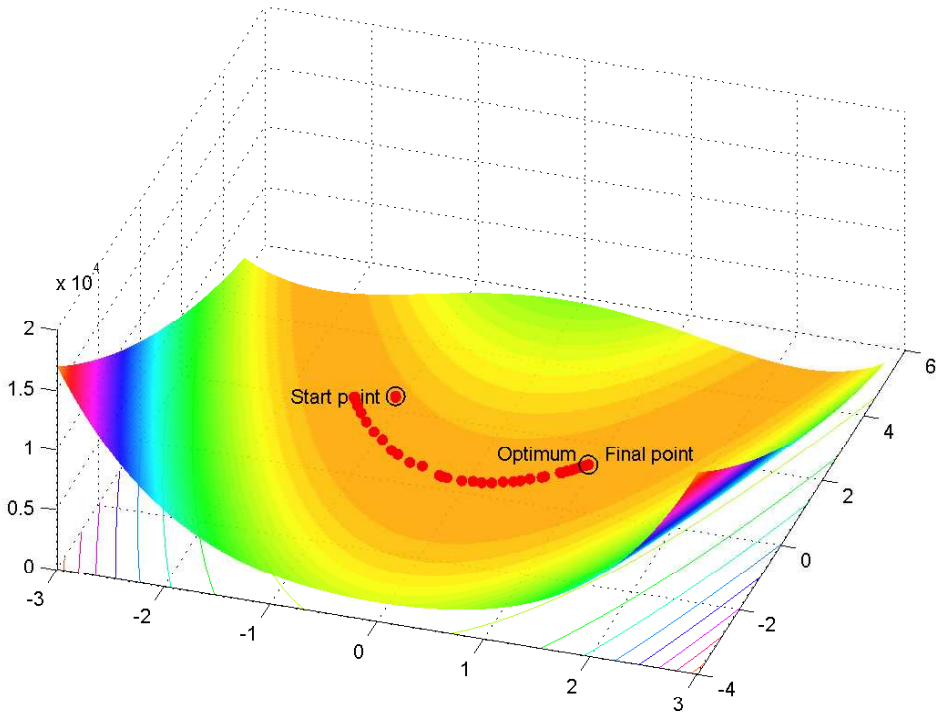


Рис. 4.20. Минимизация функции Розенброка. Метод BFGH

Листинг optim/drawpoint.m

```
function drawpoint(x, varargin);

  plot(x(1), x(2), varargin{:})
```

и укажем на нее методу *fminunc*, задав значение опции *options.OutputFcn*:

```
options = optimset('OutputFcn', @bananaoutputfcn);
fminunc(@banana, [-1, 2], options);
```

Графический вывод представлен на рис. 4.20.

Методу *fminunc* можно указать на процедуру, вычисляющую градиент целевой функции. Покажем как это сделать на примере функции Розенброка. Ее градиент равен:

$$\nabla f = (100(4x_1^3 - 4x_1x_2) + 2x_1 - 2;\ 100(2x_2 - 2x_1^2)).$$

Во-первых, напишем функцию, возвращающую значения $f(x)$ и $\nabla f(x)$:

Листинг optim/bananagrad.m

```
function [f, grad] = bananagrad(x)
```

```
f = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;

if nargout > 1
    grad = [100*(4*x(1)^3-4*x(1)*x(2))+2*x(1)-2; 100*(2*x(2)-2*x(1)^2)];
end;
```

Теперь сообщим методу *fminunc*, что *bananagrad* вычисляет не только значение функции, но и значение градиента. Для этого установим значение опции *options.GradObj* равным *'On'*:

```
options = optimset(options, 'GradObj', 'On')
fminunc(@bananagrad, [-1, 2], options)
```

Аналогичным образом методу *fminunc* сообщается информация о гессиане целевой функции. Для функции Розенброка гессиан равен

$$H = \begin{bmatrix} 1200x_1^2 - 400x_2 + 2 & -400x_1 \\ -400x_1 & 200 \end{bmatrix}$$

Напишем функцию, возвращающую значения $f(x)$, $\nabla f(x)$ и H:

Листинг optim/bananagradhess.m

```
function [f, grad, hes] = bananagradhess(x)

f = 100*(x(2) - x(1)^2)^2 + (1 - x(1))^2;

if nargout > 1
    grad = [100*(4*x(1)^3-4*x(1)*x(2))+2*x(1)-2; 100*(2*x(2)-2*x(1)^2)];
end;

if nargout > 2
    hes = [1200*x(1)^2 - 400*x(2)+2, -400*x(1); -400*x(1), 200];
end;
```

Теперь сообщим методу *fminunc*, что *bananagradhess* вычисляет значение функции, градиент и гессиан. Для этого установим значение опций *options.GradObj* и *options.Hessian*

равными '*On*':

options = *optimset*(*options*, '*GradObj*', '*On*', '*Hessian*', '*On*')
fminunc(@*bananagrad*, [−1, 2], *options*)

В следующей таблице собрана информация об алгоритмах, реализованых в *fminunc*, и соответствующих опциях (в фигурных скобках стоят значения по умолчанию):

Алгоритм	*Опции*			
	LargeScale	*HessUpdate*	*GradObj*	*Hessian*
Steepest Descent	'*Off*'	'*SteepDesc*'	{'*Off*'}/'*On*'	
BFGS	'*Off*'	{'*BFGS*'}	{'*Off*'}/'*On*'	
DFP	'*Off*'	'*DFP*'	{'*Off*'}/'*On*'	
Trust Region	{'*On*'}		'*On*'	{'*Off*'}/'*On*'

4.8.3. Нелинейный метод наименьших квадратов

MATLAB'овский метод *lsqnonlin* решает задачу минимизации функции, являющейся суммой квадратов:

$$\min f(x), \quad \text{где } f(x) = f_1(x)^2 + f_2(x)^2 + \cdots + f_m(x)^2$$

Здесь $f_1(x), f_2(x), \ldots, f_m(x)$ — скалярные функции векторного аргумента x. Рассмотрев векторную функцию

$$F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_m(x), \end{bmatrix}$$

данную задачу можно записать в виде

$$\min \|F(x)\|^2,$$

где $\|\cdot\|$ — евклидова норма.

В *lsqnonlin* реализованы LM-алгоритм (Levenberg–Marquardt), метод Гаусса–Ньютона и методы, основанные на вычислении доверительных областей.

Возможны несколько форм обращения к функции *lsqnonlin*:

$x = lsqnonlin(F,\ x0)$
$x = lsqnonlin(F,\ x0,\ l,\ u)$
$x = lsqnonlin(F,\ x0,\ l,\ u,\ options)$
$x = lsqnonlin(F,\ x0,\ l,\ u,\ options,\ P1,\ P2,\ \ldots,\ Ps)$
$[x,\ fval] = lsqnonlin(\ldots)$
$[x,\ fval,\ Fval] = lsqnonlin(\ldots)$
$[x,\ fval,\ Fval,\ exitflag] = lsqnonlin(\ldots)$
$[x,\ fval,\ Fval,\ exitflag,\ output] = lsqnonlin(\ldots)$
$[x,\ fval,\ Fval,\ exitflag,\ output,\ lambda] = lsqnonlin(\ldots)$
$[x,\ fval,\ Fval,\ exitflag,\ output,\ lambda,\ jacobian] = lsqnonlin(\ldots)$

Входные параметры: F — указатель на процедуру, вычисляющую векторную функцию $F(x)$; $x0$ — начальная точка; l, u — векторы, определяющие границы для компонент вектора x: компоненты оптимальной точки должны удовлетворять неравенствам $l \le x \le u$. Выходные параметры: x — найденная оптимальная точка, *fval* — значение функции $f(x)$ в точке x; *Fval* — вектор значений функций $f_j(x)$ в точке x; *exitflag* — код завершения работы; *lambda* — массив структур с двумя полями *lower* и *upper*, содержащих значения множителей Лагранжа в точке x для левого (*lambda.lower*) и правого (*lambda.upper*) ограничений $l \le x \le u$; *jacobian* — матрица Якоби в финальной точке x.

Функция Розенброка является суммой двух квадратов, поэтому для ее минимизации можно воспользоваться *lsqnonlin*. Имеем

$$F(x) = \begin{bmatrix} 10(x_2 - x_1^2) \\ 1 - x_1 \end{bmatrix}, \qquad J = \begin{bmatrix} -20x_1 & 10 \\ -1 & 0 \end{bmatrix}.$$

Вначале напишем функцию, возвращающую значения $F(x)$ и матрицу Якоби:

Листинг optim/bananalsqjac.m

```
function [f, jac] = bananalsqjac(x)

  f = [10*(x(2) - x(1)^2), 1 - x(1)];

  if nargout > 1
    jac = [-20*x(1), 10; -1, 0];
  end;
```

Теперь можно вызвать *lsqnonlin*

```
options = optimset('Jacobian', 'On');
[x, fval] = lsqnonlin(@bananalsqjac, [−1, 2], options);
```

Основное назначение функции *lsqnonlin* — аппроксимация данных. Рассмотрим пример. Пусть требуется аппроксимировать данные [Lanczos]

```
x = [2.5134; 2.0443; 1.6684; 1.3664; 1.1232; 0.9269; 0.7679; 0.6389; 0.5338;
     0.4479; 0.3776; 0.3197; 0.2720; 0.2325; 0.1997; 0.1723; 0.1493; 0.1301;
     0.1138; 0.1000; 0.0883; 0.0783; 0.0698; 0.0624];
y = [0; 0.05; 0.10; 0.15; 0.20; 0.25; 0.30; 0.35; 0.40; 0.45; 0.50; 0.55;
     0.60; 0.65; 0.70; 0.75; 0.80; 0.85; 0.90; 0.95; 1.00; 1.05; 1.10; 1.15];
```

функцией вида

$$f(x) = f(x; \beta_1, \beta_2, \lambda_1, \lambda_2) = \beta_1 e^{-\lambda_1 x} + \beta_2 e^{-\lambda_2 x},$$

так, чтобы норма невязки

$$\sum_{j=1}^{m} \Big(f(x_j; \beta_1, \beta_2, \lambda_1, \lambda_2) - y_j \Big)^2$$

была минимальной (минимум берется по всем $\beta_1, \beta_2, \lambda_1, \lambda_2$). Функция имеет два линейных параметра β_1 и β_2 и два нелинейных параметра λ_1 и λ_2. Задача сводится к отысканию минимума

$$\min_{\lambda_1, \lambda_2} \left(\min_{\beta_1, \beta_2} \sum_{j=1}^{m} \Big(f(x_j; \beta_1, \beta_2, \lambda_1, \lambda_2) - y_j \Big)^2 \right).$$

При заданных значениях λ_1 и λ_2 внутренний минимум легко можно найти с помощью решения линейной задачи наименьших квадратов (оператор \):

Листинг lsqnonlin/lanczos.m

```
function f = lanczols(lambda, x, y)

A = [exp(−lambda(1)*x), exp(−lambda(2)*x)];
beta = A\y;
f = y − A*beta;
```

Для нахождения внешнего минимума воспользуемся функцией `lsqnonlin`.

$[lambda,\ resnorm] = lsqnonlin(@lanczos,\ [1,\ 2],\ [\,],\ [\,],\ [\,],\ x,\ y)$

Получим значения $\lambda_1 = 1.2905$, $\lambda_2 = 12.6962$. Норма невязки составила $resnorm = 0.0018$. Вычислим β_1, β_2, соответсвующее найденному оптимуму:

$beta = [exp(-lambda(1)*x),\ exp(-lambda(2)*x)]\backslash y$

Получим $\beta_1 = 0.8087$, $\beta_2 = 0.8725$. Построим график полученной функции bи нанесем на график данные:

```
xx = 0:0.01:3;
yy = beta(1)*exp(-lambda(1)*xx) + beta(2)*exp(-lambda(2)*xx);
plot(x, y, '.', xx, yy);
legend('data', 'approximation')
title('\beta_1 e^{-\lambda_1 x} + \beta_2 e^{-\lambda_2 x}')
grid on;
```

Результат см. на рис. 4.21.

В качестве начальной точки для метода *lsqnonlin* мы взяли [1, 2]. В данном примере этот выбор не оказывает большого влияния на ответ, однако в общем случае от него может многое зависеть. Рассмотрите, например, задачу с теми же данными, но с аппроксимацией суммой трех экспонент.

В следующей таблице собрана информация об алгоритмах, реализованых в *lsqnonlin*, и соответствующих опциях (в фигурных скобках стоят значения по умолчанию):

Алгоритм	*Опции*		
	LargeScale	*Jacobian*	*LevenbergMarquardt*
LM	{'*Off*'}	{'*Off*'}/'*On*'	{'*On*'}
Gauss–Newton	{'*Off*'}	{'*Off*'}/'*On*'	'*Off*'
Trust Region	'*On*'	{'*Off*'}/'*On*'	

4.8.4. Условная оптимизация

Функция *fmincon* позволяет решать задачи условной локальной оптимизации:

$\min f(x)$

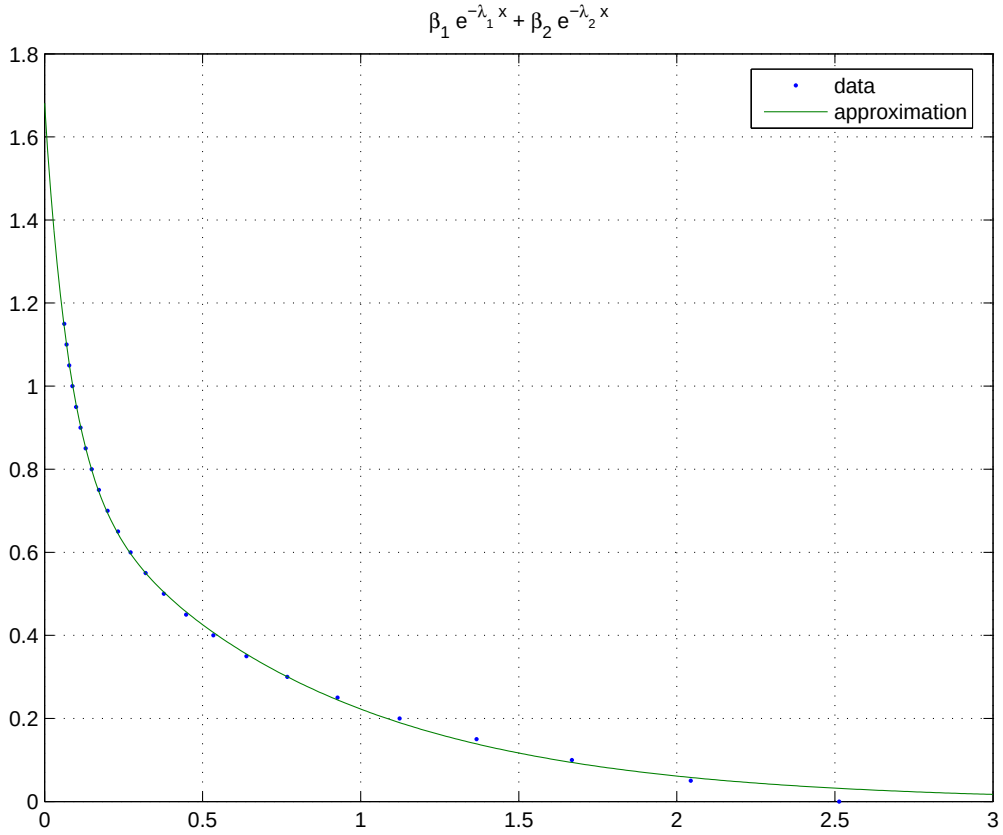


Рис. 4.21. Аппроксимация данных с помощью нелинейного метода наименьших квадратов

при ограничениях

$$g(x) \le 0, h(x) = 0, Ax \le b, Cx = d, l \le x \le u, \tag{9}$$

где $f(x)$ — скалярная функция векторного аргумента $x \in \mathbf{R}^n$, $g(x)$, $h(x)$ — векторные функции векторного аргумента x, A, C — матрицы, b, d, l, u — векторы-столбцы.

Возможны следующие формы обращения к функции *fmincon*:

$x = fmincon(f,\ x0,\ A,\ b)$
$x = fmincon(f,\ x0,\ A,\ b,\ C,\ d)$
$x = fmincon(f,\ x0,\ A,\ b,\ C,\ d,\ l,\ u)$
$x = fmincon(f,\ x0,\ A,\ b,\ C,\ d,\ l,\ u,\ nonlcon)$
$x = fmincon(f,\ x0,\ A,\ b,\ C,\ d,\ l,\ u,\ nonlcon,\ options)$
$x = fmincon(f,\ x0,\ A,\ b,\ C,\ d,\ l,\ u,\ nonlcon,\ options,\ P1,\ P2,\ \ldots,\ Pn)$
$[x,\ fval] = fmincon(\ldots)$
$[x,\ fval,\ exitflag] = fmincon(\ldots)$
$[x,\ fval,\ exitflag,\ output] = fmincon(\ldots)$
$[x,\ fval,\ exitflag,\ output,\ lambda] = fmincon(\ldots)$
$[x,\ fval,\ exitflag,\ output,\ lambda,\ grad] = fmincon(\ldots)$
$[x,\ fval,\ exitflag,\ output,\ lambda,\ grad,\ hessian] = fmincon(\ldots)$

Параметры f, *x0*, *options*, *P1*, ..., *Pn*, x, *fval*, *exitflag*, *output*, *grad*, *hessian* аналогичны соответствующим параметрам функции *fminunc*. Входной параметр *nonlcon* — указатель на функцию, вычисляющую $g(x)$ и $h(x)$. Выходной параметр *lambda* — значение модифицированных множителей Лагранжа в финальной точке. Значения остальных аргументов функции ясно из формулировки задачи (9).

В качестве примера рассмотрим задачу поиска минимума функции Розенброка при ограничении

$$(x+1)^2 + y^2 \le 2.25.$$

Напишем функцию, вычисляющую правые части нелинейных ограничений:

Листинг optim/bananacon.m

function $[ineq,\ eq] = bananacon(x)$

$ineq = (x(1) + 1)\text{^}2 + x(2)\text{^}2 - 2.25;$
$eq = [\,];$

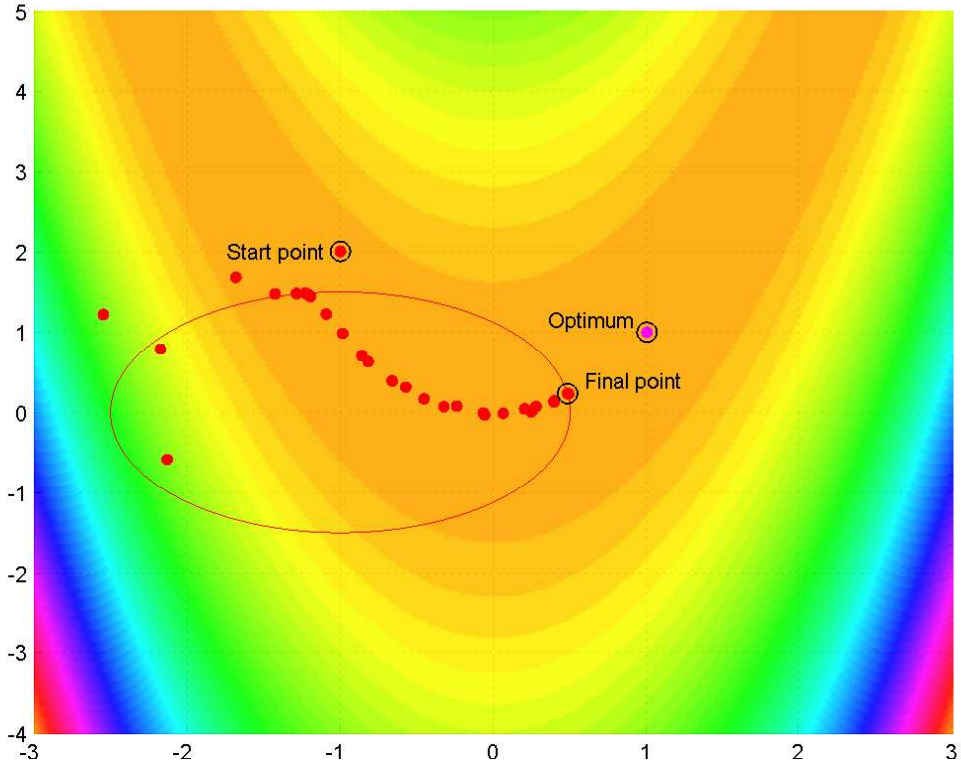


Рис. 4.22. Минимизация функции Розенброка с ограничением

Теперь можно запускать алгоритм на выполнение:

$$[x,\ fval] = fmincon(@banana,\ [-1,\ 2],\ [],\ [],\ [],\ [],\ [],\ [],\ @bananacon)$$

Линейные ограничения в нашей задаче отсутствуют, поэтому мы вставляем пустые массивы на месте аргументов A, b, C, d, l, u. Получим точку $x = (0.4820, 0.2316)$ и значение функции в ней $fval = 0.2684$. Графический вывод представлен на рис. 4.22.

4.9. Решение систем нелинейных уравнений

4.9.1. Численное решение нелинейного уравнения

Функция *fzero* предназначена для поиска корня уравнения $f(x) = 0$. Используется комбинация методов деления пополам, секущих и обратная квадратичная интерполяция. Функция реализована в виде m-файла, поэтому желающие могут изучить подробности.

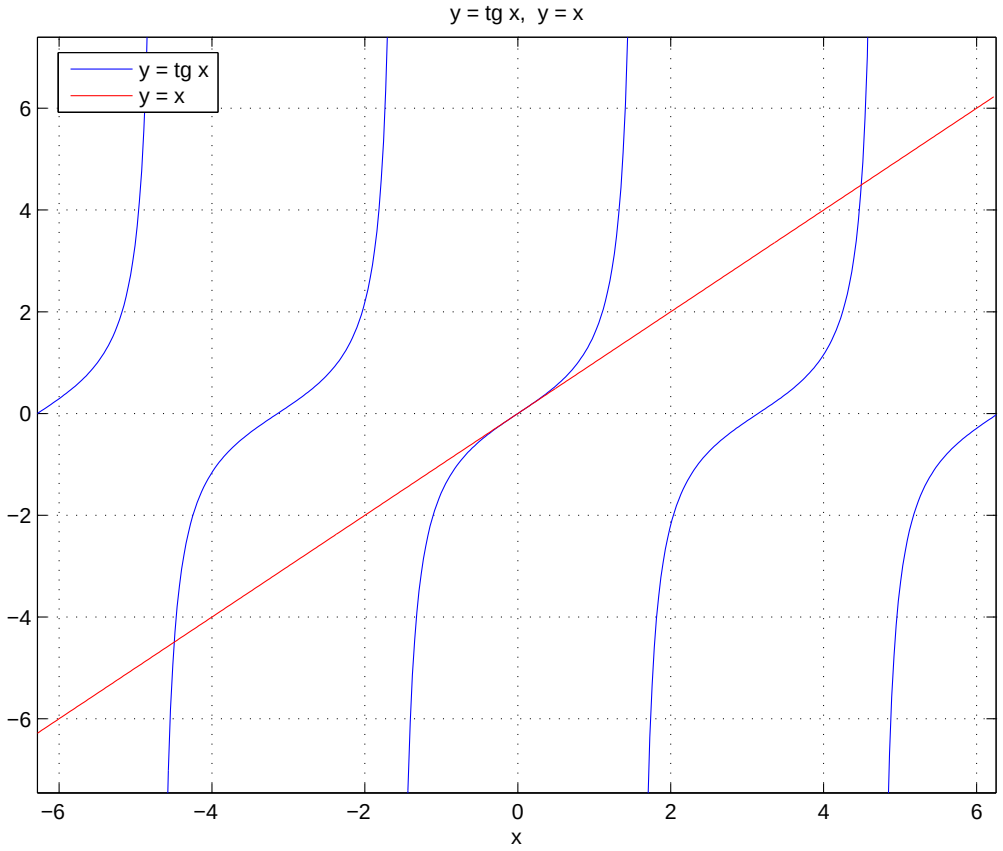


Рис. 4.23. Точки пересечения графиков соответствуют корням уравнения $\operatorname{tg} x = x$

Возможны следующие варианты вызова функции *fzero*:

$x = fzero(f,\ x0)$
$x = fzero(f,\ x0,\ options)$
$x = fzero(f,\ x0,\ options,\ P1,\ P2,\ \ldots,\ Pn)$
$[x,\ fval] = fzero(\ldots)$
$[x,\ fval,\ exitflag] = fzero(\ldots)$
$[x,\ fval,\ exitflag,\ output] = fzero(\ldots)$

Здесь f — указатель на функцию или строка символов, содержащая выражение. В последнем случае в качестве имени аргумента может использоваться только x (никакая другая буква). Если $x0$ — скаляр, то *fzero* находит корень уравнения $f(x) = 0$ вблизи $x0$. Если $x0 = [a, b]$ — вектор из двух компонент, то *fzero* находит корень на отрезке $[a, b]$. В этом случае $f(a)$ и $f(b)$ должны иметь разные знаки, иначе выдается сообщение об ошибке.

Рассмотрим уравнение $\operatorname{tg} x = x$. Получить представление о его корнях можно по

графикам функций $y = \operatorname{tg} x$, $y = x$:

$ezplot('tan(x)')$
$hold\ \ on$
$ezplot('x')$
$grid$
$title('y\ =\ tg\ \ x,\ \ \ y\ =\ x')$

Найдем ближайший к нулю положительный корень уравнения $\operatorname{tg} x = x$:

$[x,\ fval]\ =\ fzero('tan(x)\ -\ x',\ 4)$

Получим $x = 4.4934$, $fval = 8.8818 \times 10^{-16}$

4.9.2. Системы нелинейных уравнений

Функция *fsolve* из пакета Optimization Toolbox решает численно систему нелинейных уравнений

$$f(x) = 0,$$

где x — вектор длины n, а f — векторная функция. Задача сводится к минимизации суммы квадратов левых частей уравнений. Для решения этой задачи вызывается *lsqnonlin*. Возможны следующие варианты обращения к *fsolve*:

$x\ =\ fsolve(f,\ x0)$
$x\ =\ fsolve(f,\ x0,\ options)$
$x\ =\ fsolve(f,\ x0,\ options,\ P1,\ P2,\ \ldots,\ Pn)$
$[x,\ fval]\ =\ fsolve(\ldots)$
$[x,\ fval,\ exitflag]\ =\ fsolve(\ldots)$
$[x,\ fval,\ exitflag,\ output]\ =\ fsolve(\ldots)$
$[x,\ fval,\ exitflag,\ output,\ jacobian]\ =\ fsolve(\ldots)$

Здесь f — указатель на функцию, вычисляющую левую часть системы уравнений, а $x0$ — начальная точка, возле которой MATLAB будет искать решение системы. Выходной аргумент *jacobian* равен якобиану в найденной точке x. Остальные параметры аналогичны аргументам функции $fminunc$.

Рассмотрим систему

$$\begin{cases} \sin x + y^2 = 3, \\ 6x^2 + 2^y = 6 \end{cases}$$

Сперва напишем функцию, вычисляющую левую часть системы:

Листинг optim/fsolvedemo.m

```
function f = fsolvedemo(x)
    f = [sin(x(1)) + x(2)^2 - 3; 6*x(1)^2 + 2^x(2) - 6];
```

Запуская на выполнение *fsolve* с различными начальными точками:

```
x = fsolve(@fsolvedemo, [0 0])
x = fsolve(@fsolvedemo, [1 0])
x = fsolve(@fsolvedemo, [-1 0])
x = fsolve(@fsolvedemo, [-1 -2])
```

получим соответственно решения

$$\begin{matrix} 0.7202 & 1.5298 \\ 0.9696 & -1.4749 \\ -0.6176 & 1.8919 \\ -0.9783 & -1.9569 \end{matrix}$$

Методу *fsolve* можно указать процедуру, вычисляющую матрицу Якоби. Для нашего примера вычислим аналитически матрицу Якоби:

$$J = \begin{bmatrix} \cos x & 2y \\ 12x & 2^y \ln 2 \end{bmatrix}$$

Листинг optim/fsolvedemograd.m

```
function [f, Jac] = fsolvedemo(x)
    f = [sin(x(1)) + x(2)^2 - 3; 6*x(1)^2 + 2^x(2) - 6];

    if nargout > 1
        Jac = [cos(x), 2*y; 12*x, 2^y*log(2)];
    end;
```

и укажем методу *fsolve* использовать аналитически вычисленную матрицу:

```
options = optimset('Jacobian', 'On');
x = fsolve(@fsolvedemograd, [-1 -2])
```

В следующей таблице собрана информация об алгоритмах, реализованых в *fsolve*, и соответствующих опциях (в фигурных скобках стоят значения по умолчанию):

Алгоритм	*Опции*		
	LargeScale	*Jacobian*	*NonlEqnAlgorithm*
DogLeg	{'*Off*'}	{'*Off*'}/'*On*'	{'*dogleg*'}
LM	{'*Off*'}	{'*Off*'}/'*On*'	'*lm*'
Gauss–Newton	{'*Off*'}	{'*Off*'}/'*On*'	'*gn*'
Trust Region	'*On*'	{'*Off*'}/'*On*'	

4.10. Обыкновенные дифференциальные уравнения

4.10.1. Задача Коши

В MATLAB'е есть 7 функций — «решателей» задачи Коши для систем обыкновенных дифференциальных уравнений:

$$y' = f(t, y), \qquad y(t_0) = y_0.$$

Вот их имена:

ode45, *ode23*, *ode113*, *ode15s*, *ode23s*, *ode23t*, *ode23tb*.

Решатели используют различные методы. Шаг интегрирования выбирается автоматически и поэтому пользователю, как правило, не нужно задавать его. С другой стороны, пользователь может задать относительную и/или абсолютную погрешности, если значения по умолчанию (10^3 и 10^6 соответственно) его не устраивают. Функции *ode45*, *ode23*, *ode113* предназначены для решения нежестких задач, а *ode15s*, *ode23s*, *ode23t*, *ode23tb* — для жестких.

Один из форматов вызова функций следующий:

[*t*, *y*] = *ode****(*fun*, *tspan*, *y0*)

где *ode**** — любое из имен, перечисленных выше.

fun — указатель на функцию вычисляющую правую часть дифференциального уравнения. Функция должна иметь не менее двух входных аргументов, например, t и y, соответствующих независимой и зависимой переменным соответственно. Вместо указателя на функцию можно использовать имя *inline*-функции, но это не всегда работает.

tspan задает промежуток интегрирования. В простейшем случае *tspan* — это вектор $[t_0, T]$, где t_0 — начальный момент времени, T — конечный. Также в качестве *tspan* можно задать вектор из большего числа компонент. В этом случае решение задачи Коши будет найдено во всех указанных точках.

y0 — вектор, содержащий значения искомых функции в момент времени t_0.

На выходе t — столбец со значениям времени, y — матрица, каждая строка которой соответствует вектору значений искомого решения в момент времени, записанный в соответствующей строке вектора t. Если *tspan* содержад более чем 2 компоненты, то $t = tspan$, в противном случае каждая компонента t соответствует одному шагу интегрирования. Если выходные параметры не заданы, то MATLAB рисует графики найденных решений.

В качестве примера рассмотрим задачу Коши

$$y' = -10ty, \qquad y(0.01) = 0.05.$$

Правую часть уравнения реализует следующая функция.

Листинг ode/odefn.m

```
function dydt = odefn(t,y)

dydt=-10*t.*y;
```

Вызовем решатель:

```
[t, y] = ode45(@odefn, [-1, 1], .05);
plot(t, y)
```

Полученный график приведен на рис. 4.24.

Дополнительные параметры

Для того, чтобы задать дополнительные параметры (опции) алгоритма необходимо воспользоваться следующим вариантом вызова функций:

```
[t, Y] = ode***(fun, tspan, y0, options)
```

Здесь *options* — это структура, поля которой нужно заполнить заранее с помощью вызова функции *odeset*.

```
options = odeset('name1', value1, 'name2', value2,...)
options = odeset(oldopts, 'name1', value1,...)
```

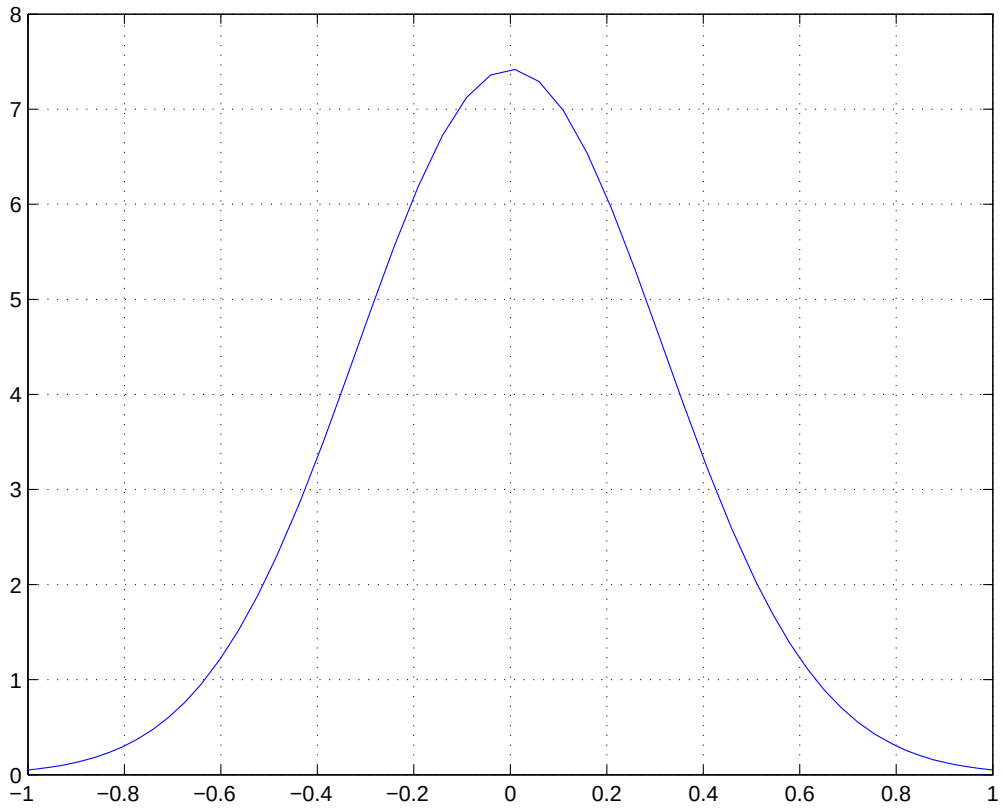


Рис. 4.24. Задача Коши $y' = -10ty$, $y(0.01) = 0.05$

Всего можно задать 20 различных опций. Некоторые из решателей используют только часть из них.

RelTol определяет допустимую относительную ошибку в компонентах решения, по умолчанию 10^{-3}. Ожидается, что решатель определяет значения искомых функций с ошибкой, не превышающей *RelTol*.

AbsTol определяет допустимую абсолютную ошибку в компонентах решения, по умолчанию 10^{-6}. Ожидается, что решатель определяет значения искомых функций с ошибкой, не превышающей *AbsTol*. Этот параметр может быть либо скаляром, либо вектором. Последний вариант задает абсолютную ошибку в каждой компоненте решения.

InitialStep — величина начального шага интегрирования. По умполчанию, определяется автоматически.

Mass позволяет определить квадратную матрицу $M = M(t, y)$, на которую домножается y' в левой части дифференциального уравнения $M(t, y)y' = f(t, y)$. Если M не зависит от t, то *Mass* — это сама матрица M, в противном случае *Mass* — это указатель на функцию от двух входных аргументов t, y, вычисляющую $M(t, y)$. По умолчанию, *Mass* — единичная матрица.

Если правая часть дифференциального уравнения зависит от параметров:

$y' = f(t, y, p_1, \ldots, p_s)$,

то эти параметры также можно передать соответсвующей функции, вычисляющей правую часть:

$[t,\ Y] = ode***(@fun,\ tspan,\ y0,\ options,\ p1,\ p2,\ \ldots)$

При этом заголовок функции *fun* также должен содержать из в списке параметров:

function $dfdy = fun(t,\ y,\ p1,\ p2,\ \ldots)$

События

События связываются с обращением в нуль некоторых индикаторных функций. Эти функции могут зависеть от фазовых переменных, независимой переменной и др. В процессе интегрирования уравнения MATLAB выявляет события и вызывает пользовательский обработчик. Чтобы описать событие и реакцию на него, пользователь должен создать функцию со следующими входными и выходными аргументами:

function $[value,\ isterminal,\ direction] = eventsfunction(t,\ y)$

Здесь *value*, *isterminal*, *direction* — векторы длины m, где m — число событий.

в $value(j)$ должно возвращаться значение j-го индикаторного выражения

в $isterminal(j)$ должна возвращаться 1, если при наступлении j-го события процесс нахождения решения нужно прекратить, и 0 в противном случае

в $direction(j)$ должна возвращаться 1, -1 и 0, если событие наступает соответственно только при возрастании значения индикаторного выражения, его убывании или в любом случае

Далее необходимо передать решателю указатель на функцию *eventsfunction*. Делается это с помощью *odeset*:

$options = odeset('Events',\ eventsfunction);$

Теперь решатель можно вызывать с 5 выходными параметрами:

$[t,\ Y,\ TE,\ YE,\ IE] = ode***(odefun,\ tspan,\ y0,\ options);$

где TE, YE, IE — время, значения фазовых переменных и номера наступивших событий.

В качестве примера рассмотрим задачу Коши, описывающую движение тела, брошенного с начальной скоростью v_0 под углом к горизонту α:

$$x'' = -kx'\sqrt{(x')^2 + (y')^2},$$
$$y'' = -ky'\sqrt{(x')^2 + (y')^2} - g,$$
$$x(0) = 0,$$
$$y(0) = h,$$
$$x'(0) = v\cos\alpha,$$
$$y'(0) = v\sin\alpha.$$

где k — коэффициент сопротивления воздуха, а g — ускорение свободного падения. Положим

$$y(1) = x, \quad y(2) = x', \quad y(3) = y, \quad y(4) = y'.$$

Мы будем интересоваться двумя событиями. Первое — тело упало на землю. В этом случае $y(3) = 0$, и в момент прохождения 0 фазовая переменная $y(3)$ уменьшается. В случае наступления этого события интегрирование прекращается. Второе событие — тело находится на максимальной высоте. В этом случае $y(4) = 0$ (и фазовая переменная $y(4)$ уменьшается, однако это последнее условие можно опустить: из физического смысла задачи ясно, что $y(4)$ равно 0 лишь в единственной точке — в вершине траектории полета тела).

Листинг ode/air.m

```
function [t, y] = air(alpha, k, m, v0, y0, varargin)

  if nargin < 1
    shg
    ylim([0, Inf]);
    daspect([1, 1, 1]);
    hold on;

    xlabel('x');
    ylabel('y');
    grid
```

```
  air(pi/4, [], [], [], [], 'b:');
  air(pi/6, [], [], [], [], 'm:');
  air(pi/3, [], [], [], [], 'k:');
  air(pi/4, 1e-3, [], [], [], 'b');
  air(pi/6, 1e-3, [], [], [], 'm');
  air(pi/3, 1e-3, [], [], [], 'k');

  legend('\alpha = \pi/4, k = 0', ...
         '\alpha = \pi/6, k = 0', ...
         '\alpha = \pi/3, k = 0', ...
         '\alpha = \pi/4, k = 10^{-3}', ...
         '\alpha = \pi/6, k = 10^{-3}', ...
         '\alpha = \pi/3, k = 10^{-3}')
  return
end;
if nargin < 1 || isempty(alpha)
  alpha = pi/4;
end;
if nargin < 2 || isempty(k)
  k = 0;
end;
if nargin < 3 || isempty(m)
  m = 1;
end;
if nargin < 4 || isempty(v0)
  v0 = 20;
end;
if nargin < 5 || isempty(y0)
  y0 = 0;
end;

options = odeset('Events', @events);
[t, y, te, ye, ie] = ode45(@airfn, [0; Inf], ...
    [0, v0*cos(alpha), y0, v0*sin(alpha)], ...
    options, k, m);
```

```
    plot(y(:, 1), y(:, 3), varargin{:})

    disp('*************************')
    disp(['alpha = ' num2str(alpha) '; k = ' num2str(k)]);

    len = ye(find(ie == 1),1);
    tm = te(find(ie == 1));
    height = ye(find(ie == 2),3);
    tmup = te(find(ie == 2));

    disp(['Maximum length = ', num2str(len)]);
    disp(['Full time      = ', num2str(tm)]);
    disp(['Maximum height = ', num2str(height)]);
    disp(['Time up        = ', num2str(tmup)]);
    disp(['Time down      = ', num2str(tm - tmup)]);

function dydt = airfn(t, y, k, m)
    g = 9.8;
    dydt = [ ...
      y(2); ...
      -k*sqrt(y(2)^2+y(4)^2)*y(2)/m; ...
      y(4); ...
      -k*sqrt(y(2)^2+y(4)^2)*y(4)/m-g];

function [value, isterminal, direction] = events(t, y, k, m)

    % без входного параметра k не работает, так как ode45 ее вызывает
    % 1-ое событие: тело упало на землю
    % 2-ое событие: тело достигло максимальной высоты (скорость по y равно 0)

      value =      [y(3), y(4)];
      isterminal = [  1,     0];
      direction  = [ -1,     0];
```

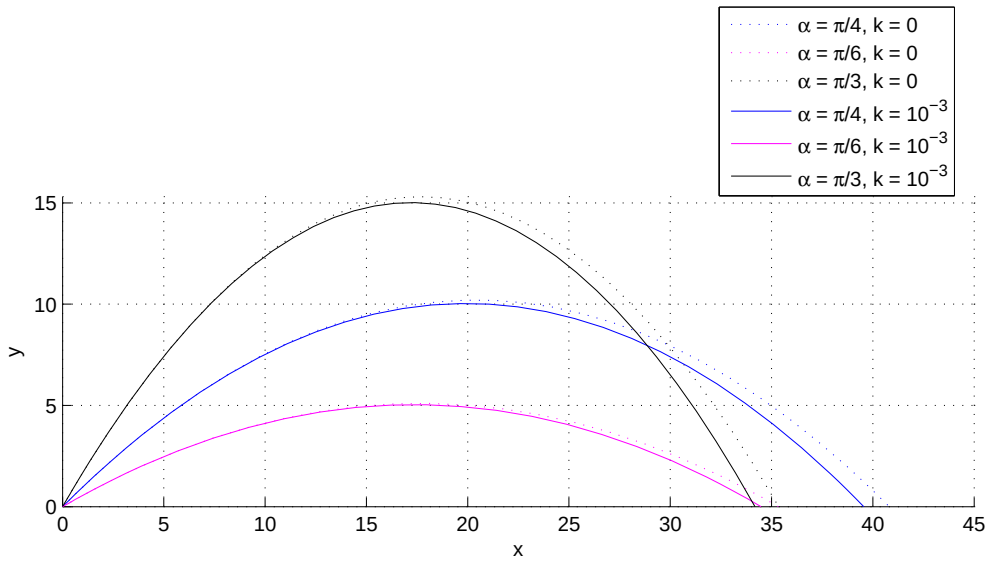


Рис. 4.25. Падение тела с учетом сопротивления воздуха

4.10.2. Краевая задача

Краевая задача для системы дифференциальных уравнений первого порядка

$$\begin{cases} u_1' = f_1(x, u_1, u_2, \ldots, u_n), \\ u_2' = f_2(x, u_1, u_2, \ldots, u_n), \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ u_n' = f_n(x, u_1, u_2, \ldots, u_n), \end{cases} \qquad x \in [a, b] \tag{10}$$

заключается в отыскании на отрезке $[a, b]$ таких функций $u_1(x), \ldots, u_n(x)$, удовлетворяющих этим уравнениям и n *граничным условиям*:

$$\begin{aligned} &g_1(u_1(a), \ldots, u_n(a), u_1(b), \ldots, u_n(b)) = 0, \\ &g_2(u_1(a), \ldots, u_n(a), u_1(b), \ldots, u_n(b)) = 0, \\ &\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ &g_n(u_1(a), \ldots, u_n(a), u_1(b), \ldots, u_n(b)) = 0, \end{aligned} \tag{11}$$

где $g_1, g_2, \ldots, g_n$ — известные функции. Заметим, что при постановке краевой задачи возникают непростые вопросы о существовании и единственности ее решения.

MATLAB'овская функция *bvp4c* для решения краевых задач вида (10), (11) использует метод сеток.

Обычная схема решения краевой задачи с помощью *bvp4c* следующая:

```
sol_init = bvpinit(x_init, u_init);
sol = bvp4c(@lhs, @border, sol_init);
u = deval(sol, x)
```

Функция *bvp_init* задают некоторую «подсказку»: *x_init* должен быть вектором-строкой, задающим сетку по переменной x, а *u_init* — либо матрицей размера $s \times n$, где s — число узлов сетки, либо столбцом высоты n. Элементы матрицы *u_init* задают значения функций $u_1, \ldots, u_n$ в узлах сетки. Если *u_init* — вектор-столбец, то эти значения для каждой функции $u_j(x)$ постоянны.

Первый входной параметр в *bvp4c* задает указатель на функцию, вычисляющую правую часть системы дифференциальных уравнений. Функция *lhs* должна быть описана следующим образом:

```
function dudx = lhs(x, u)
```

Здесь u — вектор, содержащий значения функций $u_1, \ldots, u_n$ в точке x. Выходной параметр *dudx* должен быть вектором, составленным из значения функций $f_1(u), \ldots, f_n(u)$.

Второй входной параметр в *bvp4c* задает указатель на функцию, определяющую граничные условия. Функция *border* должна быть описана следующим образом:

```
function bc = border(ua, ub)
```

Здесь *ua*, *ub* — векторы, содержащие значения функций $u_1, \ldots, u_n$ в левом и правом концах отрезка $[a, b]$ соответственно. Выходной параметр *bc* должен быть вектором, составленным из значения функций $g_1(u), \ldots, g_n(u)$.

На выходе функции *bvp4c* — структура *sol*, описывающая найденное решение. Функция *deval* вычисляет значение найденного решения $u_1(x), \ldots, u_n(x)$ в точках x.

Листинг ode/bvp.m

Решение краевой задачи $u'' + u = \cos x$, $u(0) + u'(0) = 0$, $u(\pi) = 1$

```
function sol = bvp;


  a = 0;
  b = pi;


  x_init = linspace(a, b, 5);
```

```
u_init = [-1; -1];

sol_init = bvpinit(x_init, u_init);

sol = bvp4c(@lhs, @bc, sol_init);

x = linspace(a, b, 100);
u = deval(sol, x);

plot(x, u);
legend('u(x)', 'u''(x)', 0);
grid;

function Du = lhs(x, u)

Du = [u(2); -u(1) + cos(x)];

function border = bc(ua, ub)

border = [ua(1) + ua(2); ub(1) - 1];
```

Листинг ode/bvp2.m

Решение краевой задачи $u'' + |u| = 0$, $u(0) = 0$, $u(4) = -1$

```
function [sol1, sol2] = bvp2;

a = 0;
b = 4;

x_init = linspace(a, b, 5);

u_init = [1; -1];
sol_init = bvpinit(x_init, u_init);
sol1 = bvp4c(@lhs, @bc, sol_init);

u_init = [-1; -1];
```

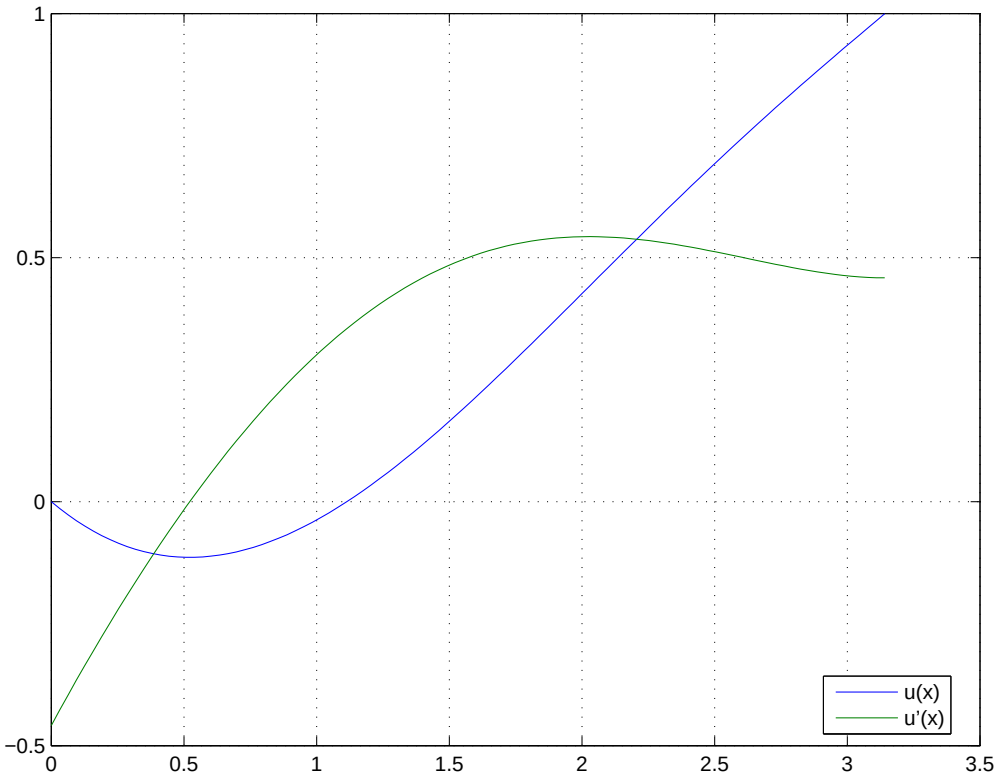


Рис. 4.26. Решение краевой задачи $u'' + u = \cos x$, $u(0) + u'(0) = 0$, $u(\pi) = 1$

```
sol_init = bvpinit(x_init, u_init);
sol2 = bvp4c(@lhs, @bc, sol_init);

x = linspace(a, b, 100);
u1 = deval(sol1, x);
u2 = deval(sol2, x);

shg;
plot(x, u1(1, :), x, u2(1, :));
grid;

function Du = lhs(x, u)

    Du = [u(2); -abs(u(1))];

function border = bc(ua, ub)

    border = [ua(1); ub(1) + 1];
```

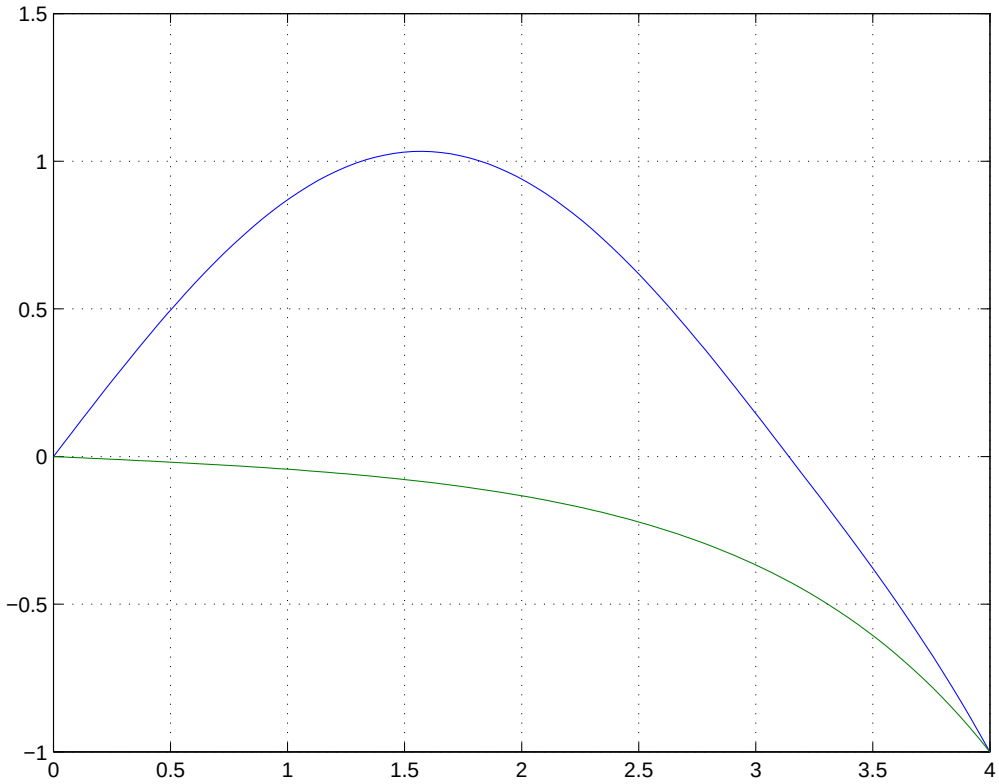


Рис. 4.27. Два решения задачи $u'' + |u| = 0$, $u(0) = 0$, $u(4) = -1$

Рассмотрим еще один пример. Профиль скорости несжимаемого газа при обтекании плоской пластины описывает уравнение Блазиуса:

$$2f''' + ff'' + \beta\left(1 - (f')^2\right) = 0.$$

β — коэффициент вязкости.

$$f(0) = f'(0) = 0, \qquad f'(\eta) \to 1 \qquad \text{при } \eta \to \infty.$$

Вместо того, чтобы рассматривать бесконечный интервал $[0, \infty)$, мы рассмотрим конечный отрезок $[0, b]$.

Листинг ode/blasius.m

```
function [x, y] = blasius(N, beta);

if nargin < 1 | isempty(N)
    N = 6;
end;
if nargin < 2 | isempty(beta)
    beta = 0;
```

```
end;

a = 0;
b = N;

x_init = linspace(a, b, 5);
y_init = [0; 1; 0];

sol_init = bvpinit(x_init, y_init);

sol = bvp4c(@lhs, @border, sol_init, [], beta);

x = linspace(a, b, 100);
y = deval(sol, x);

shg;
subplot(3,1,1)
plot(x, y(1,:));
xlabel('\eta');
ylabel('y(\eta)');
grid;

subplot(3,1,2)
plot(x, y(2,:));
xlabel('\eta');
ylabel('dy/d\eta');
grid;

subplot(3,1,3)
plot(x, y(3,:));
xlabel('\eta');
ylabel('d^2y/d\eta^2');
grid;

function Dy = lhs(x, y, beta)
```

```
  Dy = [y(2); y(3); -y(1)*y(3)/2 - beta*(1-y(2)^2)/2];

function border = border(ya, yb, beta)

  border = [ya(1); ya(2); yb(2) - 1];
```

Листинг ode/xblasius.m

```
N = 6;

[x1, y1] = blasius(N, -0.2);
[x2, y2] = blasius(N, 0);
[x3, y3] = blasius(N, 0.5);

shg;
subplot(3,1,1)
plot(x1, y1(1,:), x2, y2(1,:), x3, y3(1,:));
xlabel('\eta');
ylabel('f(\eta)');
grid;
legend('\mu = -0.2', '\mu = 0', '\mu = 0.5', 2);

subplot(3,1,2)
plot(x1, y1(2,:), x2, y2(2,:), x3, y3(2,:));
xlabel('\eta');
ylabel('df/d\eta');
grid;

subplot(3,1,3)
plot(x1, y1(3,:), x2, y2(3,:), x3, y3(3,:));
xlabel('\eta');
ylabel('d^2f/d\eta^2');
grid;
```

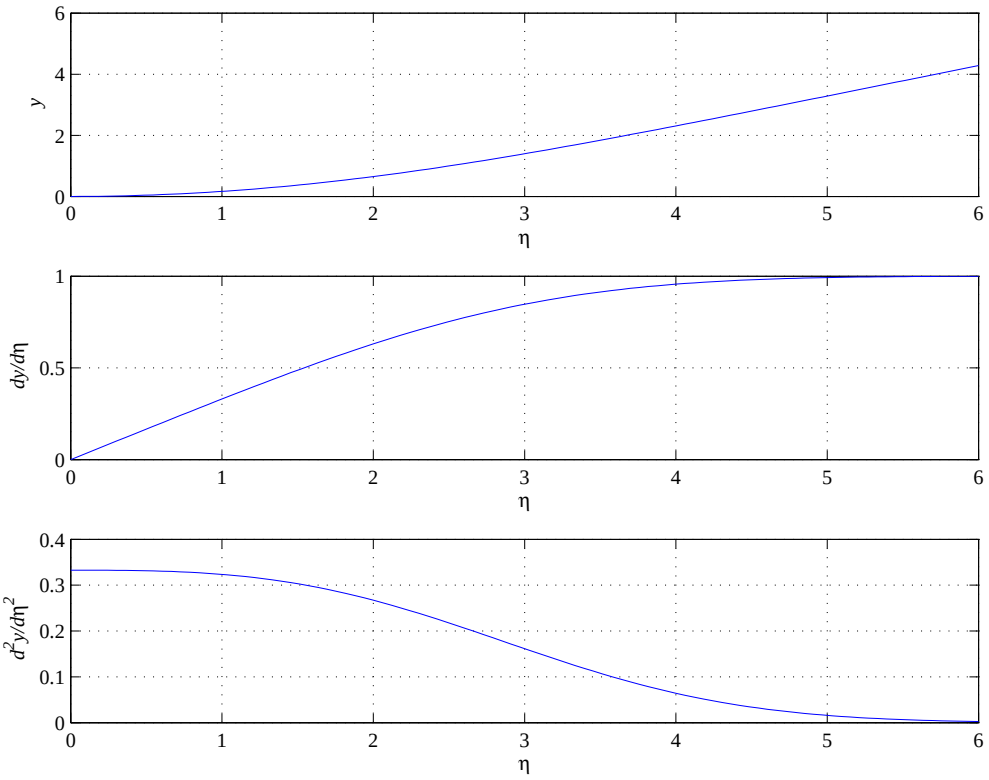


Рис. 4.28. Профиль скорости несжимаемого газа при обтекании плоской пластины

4.11. Разностные методы для уравнений в частных производных

В этом разделе мы рассмотрим разностные методы для решения уравнений в частных производных.

В MATLAB'е легко реализовать разностные методы (методы сеток) решения уравнений с частными производными. Основными инструментами для этого являются разреженные матрицы, функции *numgrid*, *delsq* и решатель систем алгебраических уравнений \ (или итерационные решатели). Вариационно-разностные и проекционно-разностные методы (методы конечных элементов) реализованы в пакете Partial Differential Toolbox и здесь не рассматриваются.

4.11.1. Задача Дирихле

Напомним, что *уравнением Пуассона* называется уравнение (эллиптического типа)

$$\Delta u \equiv \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y), \qquad (x,y) \in G$$

На решение $u = u(x,y)$ накладываются краевые уловия:

$$\left(a(x,y)u - b(c,y)\frac{\partial u}{\partial n} \right)\Big|_{\partial G} = \psi(x,y)$$

В частности, условия

$$u\,|_{\partial G} = \psi(x,y)$$

соответствуют (краевой) задаче Дирихле, а

$$\left.\frac{\partial u}{\partial n}\right|_{\partial G} = \psi(x,y)$$

соответствуют (краевой) задаче Неймана.

Если решение меняется более или менее равномерно на всей области G и область не содержит узких перешейков, то можно взять равномерную сетку

$$\big\{(x_i, y_j): \quad i = 0,1,\ldots,m;\ j = 0,1,\ldots,n\big\},$$

где

$$x_i = x_0 + ih, \qquad y_j = y_0 + jh.$$

Отбросим те узлы, которые не попали в область G.

Каждой функции $f(x,y)$, заданной на G, поставим в соответствии *сеточную функцию*, заданную только в узлах сетки, и совпадающую в этих узлах с f. Будем обозначать

$$f_{ij} = f(x_i, y_j).$$

Занумеруем точки сетки, попавшие внутрь и на границу области G, в каком-либо порядке. Тогда каждой сеточной функции f можно поставить во взаимно-однозначное соответсвие вектор-столбец $\widehat{f}$, составленный из значений функции $f(x,y)$ в узлах сетки, записанных в том же порядке. Например,

$$\widehat{f} = \begin{bmatrix} f_{11}, \\ f_{21}, \\ f_{31}, \\ \ldots, \\ f_{mn} \end{bmatrix}$$

(отбросив те пары i, j, для которых (x_i, y_j) не попали в G). Будем аппроксимировать решение $u(x,y)$ сеточной функцией

$$u(x_i, y_j) \approx u_{ij}$$

Производные аппроксимируем конечными разделенными разностями:

$$\left.\frac{\partial u}{\partial x}\right|_{(x_i,y_j)} \approx \frac{u(x_{i+1},y_j)-u(x_i,y_j)}{h}, \qquad \left.\frac{\partial u}{\partial y}\right|_{(x_i,y_j)} \approx \frac{u(x_i,y_{j+1})-u(x_i,y_j)}{h},$$

$$\left.\frac{\partial^2 u}{\partial x^2}\right|_{(x_i,y_j)} \approx \frac{u(x_{i+1},y_j)+u(x_{i-1},y_j)-2u(x_i,y_j)}{h^2},$$

$$\left.\frac{\partial^2 u}{\partial y^2}\right|_{(x_i,y_j)} \approx \frac{u(x_i,y_{j+1})+u(x_i,y_{j-1})-2u(x_i,y_j)}{h^2},$$

Получаем аппроксимацию оператора Лапласа — разностный оператор Лапласа:

$$\Delta u\Big|_{(x_i,y_j)} \approx \frac{u(x_{i+1},y_j)+u(x_{i-1},y_j)+u(x_i,y_{j+1})+u(x_i,y_{j-1})-4u(x_i,y_j)}{h^2}$$

Разностный оператор Лапласа можно представить в виде

$$-\frac{1}{h^2}A\widehat{u},$$

где A — квадратная матрица.

Рассмотрим, например, квадратную область G, покрытую сеткой 6×6. Занумеруем точки следующим образом:

·	·	·	·	·	·
·	1	5	9	13	·
·	2	6	10	14	·
·	3	7	11	15	·
·	4	8	12	16	·
·	·	·	·	·	·

Точки, попавшие на границу, отмечены знаком «·». Тогда

$$A = \begin{bmatrix}
4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 4 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4
\end{bmatrix}$$

Подобные разреженные матрицы (для разных областей) генерирует MATLAB'овская функция *numgrid*(*Domain*, *n*). Здесь *Domain* — символ, заключенный в одинарные кавычки, описывающий вид области G:

- 'S' — весь квадрат;
- 'N' — другое упорядочение точек квадрата;
- 'L' — L-образная область;
- 'C' — L-образная область со скругленной границей;
- 'D' — круг;
- 'A' — кольцо;
- 'H' — область, ограниченная кардиоидой;

- 'B' — внешняя часть «бабочки».

Получаем систему линейных уравнений

$$A\widehat{u} = -h^2\widehat{f}$$

В этой же системе также можно учесть и краевые условия. Для этого напишем функцию *numgridborder*.

Листинг pde/numgridborder.m

```
function B = regionborder(G)

  % G - матрица, возвращаемая функцией numgrid
  % B - то, что останется от G, если убрать внутренние точки

  [m, n] = size(G);

  inner = find(G);
  border = []; % номера граничных точек
  for k = [-1, 1, -m, m, -1-m, -1+m, 1-m, 1+m]
    % точка - граничная,
    % если по крайней мере одна из восьми соседних
    % не принадлежит области
    % for k = [-1, 1, -m, m]
    % точка - граничная,
    % если справа или слева или снизу или сверху есть точка,
    % не принадлежащая области
    border = [border; inner(G(inner + k) == 0)];
  end;

  B = zeros(size(G));
  B(border) = G(border);
```

Решение задачи Дирихле реализует следующая функция.

Листинг pde/dirichlet.m

```
  Задача Дирихле для уравнения Пуассона Δu = fn(x,y), u|∂Γ = fnborder
function [X, Y, U] = dirichlet(h, Region, fn, fnborder, varargin)
```

```
if nargin < 1 || isempty(h)
    h = 0.02;
end;
if nargin < 2 || isempty(Region)
    Region = 'L';
end;
if nargin < 3 || isempty(fn)
    fn = @fun;
end;
if nargin < 4 || isempty(fnborder)
    fnborder = @funborder;
end;

x0 = 0;
x1 = 1;
n = (x1 - x0)/h + 3;
[X, Y] = meshgrid(linspace(x0 - h, x1 + h, n));
U = zeros(n);
U(:) = NaN;
R = numgrid(Region, n); % нумерация узлов области
B = numgridborder(R);   % узлы, лежащие на границе
Region = R > 0;         % шаблон из нулей и единиц для все узлов области
Border = B > 0;         % шаблон из нулей и единиц для узлов на границе
region = R(Region);     % номера всех узлов в определенном порядке
border = R(Border);     % номера узлов, лежащих на границе
nregion = length(region);
nborder = length(border);

A = -delsq(R)/h^2;

% правая часть уравнения
f = zeros(nregion, 1);
f(region) = feval(fn, X(Region), Y(Region), varargin{:});
```

```
% граничные условия
A(border, :) = sparse(1:nborder, border, 1, nborder, nregion);
f(border) = feval(fnborder, X(Border), Y(Border), varargin{:});

u = A\f;
U(Region) = u(region);

X = X(2 : n - 1, 2 : n - 1);
Y = Y(2 : n - 1, 2 : n - 1);
U = U(2 : n - 1, 2 : n - 1);

clf
shg
surfc(X, Y, U);
shading interp;
view(-63, 17)

function f = fun(x, y)
    f = 50*(y - x);

function f = funborder(x, y)
    f = 0;
    % f = x - y;
    % f = sin(pi*x) - sin(pi*y);
```

4.11.2. Уравнение теплопроводности

Уравнением теплопроводности называется уравнение (параболического типа)

$$\frac{\partial u}{\partial t} = a^2 \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) + f(x, y), \qquad t \geq 0, \qquad (x, y) \in G$$

Начально-краевая задача заключается в нахождении решения $u = u(t, x, y)$ этого уравнения, удовлетворяющего к тому же начальным условиям:

$$u(0, x, y) = \alpha(x, y), \qquad (x, y) \in G$$

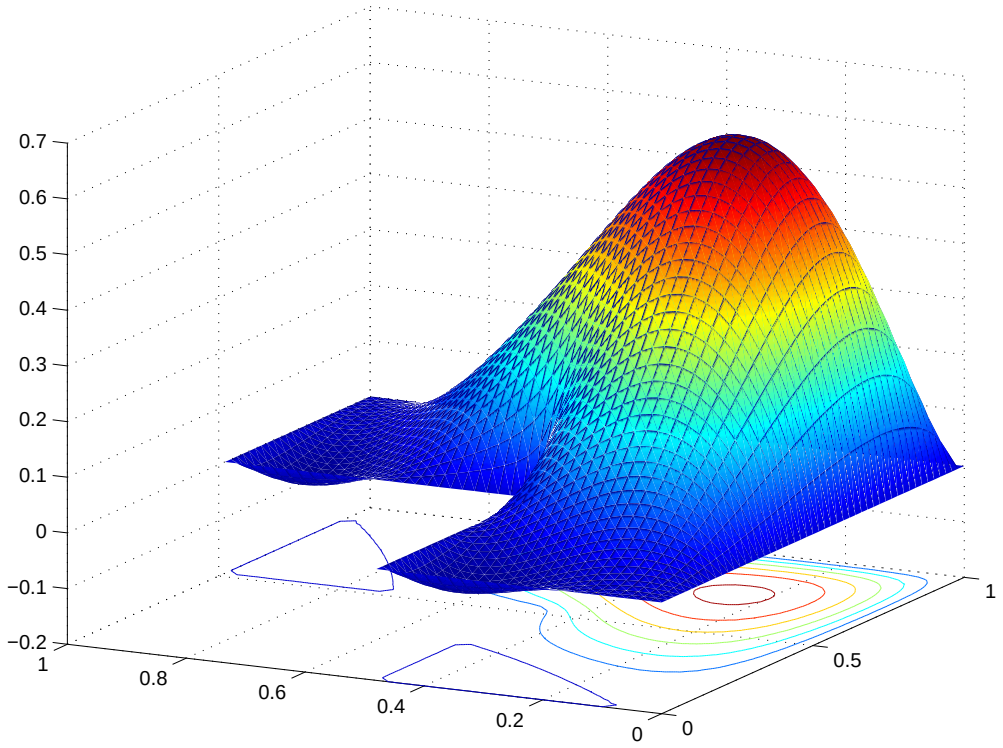


Рис. 4.29. Решение задачи Дирихле для уравнения Пуассона

и краевым условиям:

$$\left(a(t,x,y)u - b(t,x,y)\frac{\partial u}{\partial n}\right)\Big|_{\partial G} = \psi(t,x,y)$$

Помимо сетки по пространственным переменным x и y введем сетку на временной оси:

$$\{t_k = k\tau : \quad k = 0,\ 1,\ 2, \dots\}$$

Аппроксимируем неизвестную функцию:

$$u(x_i, y_j, t_k) \approx u_{ijk}.$$

и производную:

$$\frac{\partial u}{\partial t}\Big|_{(x_i,y_j,t_k)} \approx \frac{u_{i,j,k+1} - u_{i,j,k}}{\tau}$$

Явная схема заключается в поиске $u_{i,j,k+1}$ для $(k+1)$-го временного шага по формуле:

$$u_{i,j,k+1} = u_{ijk} + \tau\frac{u_{i+1,j,k} + u_{i-1,j,k} + u_{i,j+1,k} + u_{i,j-1,k} - 4u_{ijk}}{h^2} + \tau f_{ijk}$$

Явная схема реализуется с помощью умножения разреженной матрица на столбец.

Неявная схема заключается в поиске значений $u_{i,j,k+1}$ на $(k+1)$-м временном шаге из системы линейных уравнений:

$$\frac{u_{i,j,k+1}-u_{i,j,k}}{\tau}=\frac{u_{i+1,j,k+1}+u_{i-1,j,k+1}+u_{i,j+1,k+1}+u_{i,j-1,k+1}-4u_{i,j,k+1}}{h^2}+f_{ijk}$$

Реализуется с помощью решения (разреженной) системы линейных алгебраических уравнений (на каждом временном шаге).

4.11.3. Волновое уравнение

Волновым уравнением называется уравнение (гиперболического типа)

$$\frac{\partial^2 u}{\partial t^2}=a^2\left(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}\right)+f(x,y),\qquad t\geq 0,\qquad (x,y)\in G$$

Начально-краевая задача заключается в нахождении решения $u=u(t,x,y)$ этого уравнения, удовлетворяющего к тому же начальным условиям:

$$u(t,x,y)|_{t=0}=\alpha(x,y),\qquad \left.\frac{\partial u(t,x,y)}{\partial t}\right|_{t=0}=\beta(x,y),\qquad (x,y)\in G$$

и краевым условиям:

$$\left.\left(a(t,x,y)u-b(t,x,y)\frac{\partial u}{\partial n}\right)\right|_{\partial G}=\psi(t,x,y)$$

Аппроксимируем производную:

$$\left.\frac{\partial^2 u}{\partial t^2}\right|_{(x_i,y_j,t_k)}\approx\frac{u_{i,j,k+1}+u_{i,j,k-1}-2ui,j,k}{\tau^2}$$

Явная схема заключается в поиске значений $u_{i,j,k+1}$ на $(k+1)$-м временном шаге по формулам:

$$u_{i,j,k+1}=2u_{ijk}-u_{i,j,k-1}+\tau^2\frac{u_{i+1,j,k}+u_{i-1,j,k}+u_{i,j+1,k}+u_{i,j-1,k}-4u_{ijk}}{h^2}+\tau^2 f_{ijk}$$

Реализуется с помощью умножения разреженной матрица на столбец.

Неявная схема заключается в поиске значений $u_{i,j,k+1}$ на $(k+1)$-м временном шаге из системы линейных уравнений:

$$\frac{u_{i,j,k+1}+u_{i,j,k-1}-2u_{ijk}}{\tau^2}=\frac{u_{i+1,j,k+1}+u_{i-1,j,k+1}+u_{i,j+1,k+1}+u_{i,j-1,k+1}-4u_{i,j,k+1}}{h^2}+f_{ijk}$$

Реализуется с помощью решения (разреженной) системы линейных алгебраических уравнений (на каждом временном шаге).

Листинг pde/wave.m

Начально-краевая задача для волнового уравнения разностным методом $\frac{\partial^2 u}{\partial t^2} = \Delta u + fn(t,x,y)$, $(x,y) \in G$ $u(t,x,y) = fnborder(t,x,y)$, $(x,y) \in \partial G$ $u(0,x,y) = fninitial(x,y)$ $\frac{\partial u(0,x,y)}{\partial t} = fndinitial$ h — шаг по пространственным координатам x, y tau — шаг по t $explicit = 1$ — явная схема $explicit = 0$ — неявная схема $Region$ — форма области (см. $numgrid$)

```
function wave(h, tau, Region, fn, fnborder, fninitial, fndinitial, explicit, varargin)

    if nargin < 1 || isempty(h)
        h = 0.05;
    end;
    if nargin < 2 || isempty(tau)
        tau = 0.01;
    end;
    if nargin < 3 || isempty(Region)
        Region = 'L';
    end;
    if nargin < 4 || isempty(fn)
        fn = @fun;
    end;
    if nargin < 5 || isempty(fnborder)
        fnborder = @funborder;
    end;
    if nargin < 6 || isempty(fninitial)
        fninitial = @funinitial;
    end;
    if nargin < 7 || isempty(fndinitial)
        fndinitial = @fundinitial;
    end;
    if nargin < 8 || isempty(explicit)
        explicit = 1;
    end;

    stop = uicontrol('style', 'togglebutton', 'string', 'stop');

    x0 = 0;
```

```
x1 = 1;
n = (x1 - x0)/h + 3;
[X, Y] = meshgrid(linspace(x0 - h, x1 + h, n));
U = zeros(n);
U(:) = NaN;

R = numgrid(Region, n); % region grid
B = numgridborder(R);   % border grid
A = -delsq(R);          % разностный оператор Лапласа

Region = R > 0;         % шаблон из нулей и единиц
region = R(Region);     % номера элементов, содержащих 1
Border = B > 0;
border = B(Border);

len = max(max(R));

u_cur = zeros(len, 1);
dudt = zeros(len, 1);

u_cur(region) = feval(fninitial, X(Region), Y(Region)); % U — матрица, u — вектор
dudt(region) = feval(fndinitial, X(Region), Y(Region));
u_prev = u_cur - tau*dudt; % для начальных условий

shg;
graph = surf(X, Y, U);
axis([x0 x1 x0 x1 -30 30]);

t = 0;

while ~get(stop, 'value')
  U(Region) = u_cur(region);
  set(graph, 'ZData', U);
  drawnow;
```

```
    t = t + tau;

    f = feval(fn, X(Region), Y(Region), t);

    if explicit
      u_new = 2*u_cur - u_prev + tau^2/h^2 * A * u_cur + tau^2*f;
    else
      u_new = (speye(size(A)) - tau^2/h^2 * A ) \ (2*u_cur - u_prev + tau^2*f);
    end;
    u_prev = u_cur;
    u_cur = u_new;

    u_cur(border) = feval(fnborder, X(Border), Y(Border), t);
  end;

function f = fun(x, y, t)
  f = 500*(x - y);

function f = funborder(x, y, t)
  f = 5*(x - y)*sin(2*pi*t);

function f = funinitial(x, y)
  f = 0;

function f = fundinitial(x, y)
  f = 0;
```

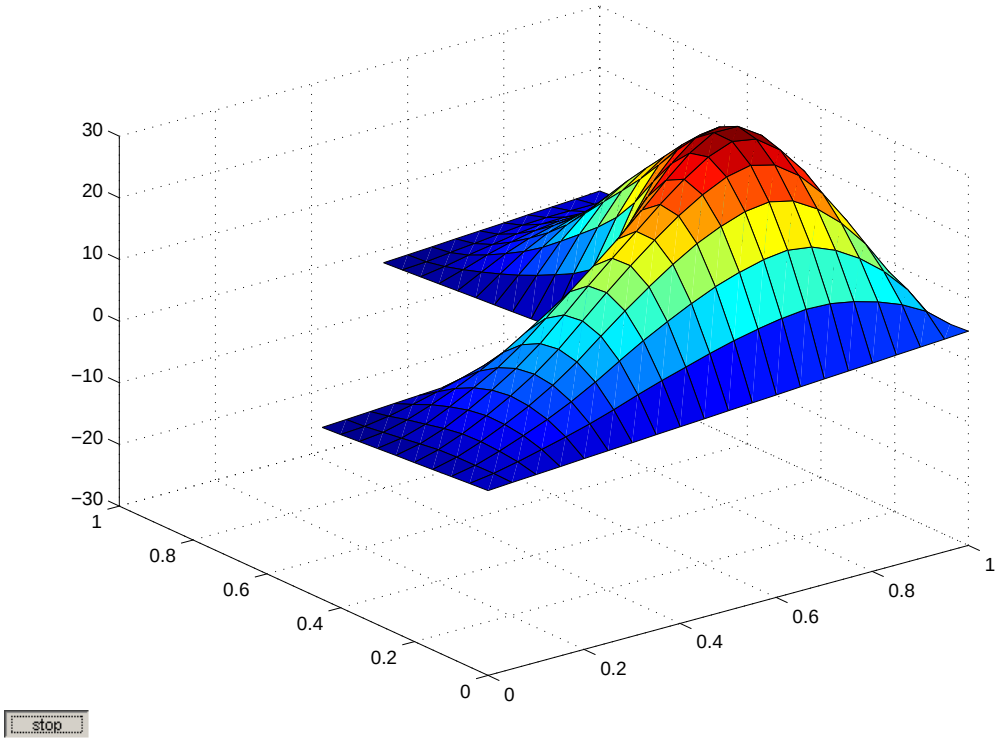


Рис. 4.30. Начально краевая задача для волнового уравнения

Литература

1. *Ануфриев И.Е., Смирнов А.Б., Смирнова Е.Н.* MATLAB 7.0 СПб.: BHV, 2005.

2. *Мартынов Н.Н.* Matlab 7. Элементарное введение. М.: Кудиц-образ, 2005.

3. *Кетков Ю.Л., Кетков А.Ю., Шульц М.М.* MATLAB 6.X: программирование численных методов. СПб.: BHV, 2003. MATLAB 7.0: программирование, численные методы. СПб.: BHV, 2005.

4. *Чен К., Джиблин К., Ирвинг А.* MATLAB в математических исследованиях. М.: Мир. 2001.

5. *Кондрашов В.Е., Королев С.Б.* MATLAB как система программирования научно-технических расчетов. М.: Мир. 2002.

6. *Moler C.B.* Numerical Computing with MATLAB
www.mathworks.com/moler

7. *Van Loan C.F.* Introduction to scientific computing: a matrix-vector approach using MATLAB. N. J.: Prentice Hall, 1997

[-]