О.В.Бартеньев

ФОРТРАН ДЛЯ ПРОФЕССИОНАЛОВ. Математическая библиотека IMSL. Выпуск 2

Излагаются средства математической библиотеки IMSL, входящей в состав профессиональных версий Фортрана фирм Microsoft и Compaq, позволяющие выполнять преобразования Фурье, решать нелинейные уравнения и задачи оптимизации, выполнять сортировку векторов. Представляемый материал иллюстрируется большим числом примеров.

Предназначено для научных работников, инженеров, преподавателей, студентов и аспирантов вузов.

Содержание

Предметный указатель

ПРЕДИСЛОВИЕ

Пособие продолжает начатое в [5] рассмотрение математической библиотеки IMSL фирмы Visual Numerics, Inc., являющейся неотъемлемой частью профессиональных версий Fortran PowerStation (FPS) 4.0 и Compaq Visual Fortran (CVF).

Пособие содержит описание процедур библиотеки, осуществляющих прямые и обратные дискретные преобразования Фурье, находящих корни нелинейных уравнений и их систем, решающих разнообразные задачи оптимизации и сортирующих векторы. Наряду с процедурами, реализованными средствами прежнего стандарта Фортрана, в пособие включены процедуры библиотеки IMSL 90 MP, которые написаны в стиле стандарта Фортран 90.

Кроме процедур библиотеки IMSL пособие имеет приложения. Первое описывает возможности отображателя массивов - программы, дополняющей CVF и осуществляющей графическое воспроизведение массивов. Отображатель содержит в качестве ядра графическую библиотеку OpenGL®. Во втором приложении описываются процедуры вывода графиков функций одной переменной и поверхностей. Приводимые в нем программы также используют средства OpenGL®. Изложенные в приложениях средства использованы в настоящем пособии (а также в следующей, третьей части книги) для графического представления функций и результатов вычислений. Два последних приложения являются небольшими дополнениями к ранее изданному пособию [2]. Они появились в результате общения с читателями, указавшими на проблемы, возникшие при употреблении расширенных (по отношению в стандарту) средств Фортрана и не получившие в [2] должного освещения.

Одновременно со второй частью выходит третья, завершающая описание математической библиотеки IMSL часть книги. Она включает процедуры, строящие сплайны, выполняющие численное интегрирование и дифференцирование и решающие дифференциальные уравнения. Заметим, что во второй и третьей частях пришлось отказаться от первоначального замысла предварить описание процедур библиотеки IMSL изложением методов, лежащих в основе этих процедур. Это позволило существенно сократить объем пособий, а значит, и их стоимость, а также сроки их подготовки. В то же время отсутствие подобных материалов вряд ли можно классифицировать как недостаток, поскольку задачи, требующие применения численных методов, решают, как правило, специалисты, владеющие необходимой математической подготовкой. Недостающие сведения можно получить из источников, список которых дается в конце пособия.

По своему характеру описание процедур IMSL максимально приближено к тексту, имеющемуся в поставляемой с Фортраном документации. Сохранены, как правило, и сопровождающие описание процедур примеры. Как и в первой части, с целью сокращения объёма пособия, как правило, не приводятся процедуры второго уровня, кроме тех случаев, когда подобные процедуры предоставляют пользователю дополнительные по сравнению с базовыми процедурами возможности. Предполагается, что читатель знаком с первой частью и, в частности, осведомлен, что процедуры библиотеки IMSL 77, имеющие префикс D, работают с двойной точностью, т. е. с вещественными данными типа REAL(8) или комплексными типа COMPLEX(8), и помнит, что процедуры библиотеки IMSL 90 обладают родовым интерфейсом и могут принимать вещественные или в некоторых случаях комплексные данные как одинарной, так и двойной точности, возвращая той же точности и того же типа результат.

В заключение хотелось бы выразить признательность сотрудникам издательства "ДИАЛОГ-МИФИ", неустанно работающим над повышением качества представляемого им материала, а также читателям, нашедшим возможность прислать свои, как правило благожелательные, отклики на ранее выпущенные книги. Для общения читатели использовали электронный адрес BartenyevOV@mpei.ru .

1. ПРЕОБРАЗОВАНИЯ ФУРЬЕ И ЛАПЛАСА

1.1. ВВЕДЕНИЕ

1.1.1. ДИСКРЕТНОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ

Известно, что функция f, заданная в конечном числе точек $x_j = j/N$, $j = 0, \ldots, N-1$ может быть представлена в виде разложения

$$f(x_j) = \sum_{k=0}^{N-1} c_k e^{2\pi i k x_j}, \quad 0 \le j < N,$$

где i - мнимая единица. Коэффициенты c_k, называемые *коэффициентами Фурье*, определяются по формуле

$$c_k = \frac{1}{N} \sum_{l=0}^{N-1} f(x_l) e^{-2\pi i k x_l}, \quad 0 \le k < N.$$

Операцию преобразования набора значений $f(x_0), f(x_1), \ldots, f(x_{N-1})$ в набор коэффициентов $c_0, c_1, \ldots, c_{N-1}$ называют *прямым дискретным преобразованием Фурье*, а обратную операцию - *обратным дискретным преобразованием Фурье*.

Дискретные преобразования Фурье можно выполнить при помощи эффективного вычислительного метода, называемого *быстрым преобразованием Фурье* (БПФ).

1.1.2. БЫСТРЫЕ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

Быстрые дискретные преобразования Фурье гораздо эффективнее непосредственных преобразований Фурье. Так, последние требуют примерно N^2 операций, где N - число точек в преобразовании, в то время как БПФ для того же набора данных будут выполнены примерно за $N \log N$ операций. Алгоритм БПФ, используемый процедурами библиотеки IMSL, основан на алгоритме [10], в котором вычислительные затраты снижаются, если N (или в некоторых случаях $N + 1$ или $N - 1$) является произведением небольших простых чисел.

Процедуры библиотеки IMSL, вычисляющие БПФ, принимают вектор x размера N и возвращают вектор $\hat{x}$ с элементами

$$\hat{x}_m = \sum_{n=1}^{N} x_n \omega_{nm},$$

где ω_{nm} - одна из формул табл. 1.1, определяющая вид БПФ. В табл. 1.1 также указывается и имя подпрограммы, реализующей соответствующее преобразование.

Таблица 1.1. Формулы для преобразований Фурье

ω_{nm}	*Подпрограммы*
$\cos$ или $\sin\dfrac{(m-1)(n-1)2\pi}{N}$	FFTRF, FFTRB
$e^{-2\pi i(n-1)(m-1)/N}$	FFTCF
$e^{2\pi i(n-1)(m-1)/N}$	FFTCB
$\sin\dfrac{nm\pi}{N+1}$	FSINT
$\cos\dfrac{(n-1)(m-1)\pi}{N-1}$	FCOST
$2\sin\dfrac{(2m-1)n\pi}{2N}$	QSINF
$4\sin\dfrac{(2n-1)m\pi}{2N}$	QSINB
$2\cos\dfrac{(2m-1)(n-1)\pi}{2N}$	QCOSF
$4\cos\dfrac{(2n-1)(m-1)\pi}{2N}$	QCOSB

Для многих из перечисленных в табл. 1.1 процедур существует процедура инициализации, имя которой оканчивается на букву I. Эти процедуры целесообразно использовать лишь при повторных преобразованиях последовательностей одинаковой длины. В таких случаях процедура инициализации вычисляет начальную последовательность, которая затем неоднократно используется соответствующей процедурой второго уровня. Понятно, что такой механизм приводит к существенному снижению вычислительных затрат.

Наряду с одномерными преобразованиями, описанными в табл. 1.1, библиотека IMSL содержит процедуры комплексного двумерного и трехмерного БПФ (прямого и обратного), основанные либо на подпрограмме FFTCF, либо на подпрограмме FFTCB. Идею выполнения преобразований большей размерности можно почерпнуть из примера для подпрограммы FFTCI, содержащего основы стратегии подобных действий.

1.1.3. НЕПРЕРЫВНЫЕ И ДИСКРЕТНЫЕ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

Между непрерывными (интегральными) и дискретными преобразованиями Фурье есть, разумеется, тесная связь. Напомним, что непрерывные преобразования определяются формулой

$$\hat{f}(\omega) = \int_{-\infty}^{\infty} f(t)e^{-2\pi i\omega t}dt .$$

Чтобы получить из формулы непрерывных формулу дискретных преобразований, выполним следующие операции:

$$\hat{f}(\omega) = \int_{-T/2}^{T/2} f(t)e^{-2\pi i\omega t}dt = \int_{0}^{T} f(t-T/2)e^{-2\pi i\omega(t-T/2)}dt =$$

$$= e^{\pi i\omega T}\int_{0}^{T} f(t-T/2)e^{-2\pi i\omega t}dt$$

и, аппроксимировав последний интеграл суммой, используя правило прямоугольника с шагом $h = T/N$, получим

$$\hat{f}(\omega) = e^{\pi i\omega T}h\sum_{k=0}^{N-1} e^{-2\pi i\omega kh} f(kh-T/2).$$

Полагая теперь $\omega = j/T$ для $j = 0, \ldots, N-1$, найдем

$$\hat{f}(j/T) = e^{\pi ij}h\sum_{k=0}^{N-1} e^{-2\pi ijk/N} f(kh-T/2) = (-1)^j h\sum_{k=0}^{N-1} e^{-2\pi ijk/N} f_k^h ,$$

где вектор $f^h = (f(-T/2), \ldots, f((N-1)h - T/2))$. После масштабирования компонентов вектора f^h на $(-1)^j h$ получим формулу, по которой подпрограмма FFTCF вычисляет дискретное преобразование Фурье.

Связь между непрерывным и дискретным преобразованиями Фурье станет более наглядной, если в последней сумме выполнить следующую замену переменных:

$$k = n+1,\ m = j+1 \text{ и } f_k^h = x_n.$$

Тогда для $m = 1, \ldots, N$ имеем

$$\hat{f}((m-1)/T) = -(-1)^m h\hat{x}_m = -(-1)^m h\sum_{n=1}^{N} e^{-2\pi i(m-1)(n-1)/N} x_n .$$

Если функция f представлена процедурой Фортрана, то непрерывное преобразование Фурье $\hat{f}$ может быть выполнено подпрограммой QDAWF библиотеки IMSL.

Преобразования Фурье используются для анализа циклических данных во многих областях науки и техники, включая теорию электрических цепей, теорию механических и колебательных систем, оптику, акустику и термодинамику. Причем в некоторых приложениях, например в оптике, применяются интегральные преобразования Фурье. В других ситуациях, например при цифровой обработке сигналов, наблюдаемые величины являются дискретными и вместо интегральных употребляются дискретные преобразования Фурье.

1.1.4. ПРЕОБРАЗОВАНИЕ ЛАПЛАСА

Преобразование Лапласа комплексной функции выполняется приведенными в конце главы подпрограммами INLAP и SINLP.

1.2. ОДНОМЕРНЫЕ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

1.2.1. ПЕРЕЧЕНЬ ПОДПРОГРАММ И ПАРАМЕТРОВ

Перечень подпрограмм, осуществляющих одномерные прямые и обратные преобразования Фурье, а также процедур инициализации, приведен в табл. 1.1. Их параметры представлены в табл. 1.2. Все сведения даны для подпрограмм, работающих с одинарной точностью. (Напомним, что программы, использующие двойную точность, имеют префикс D.)

Таблица 1.2. Подпрограммы одномерных преобразований Фурье

Подпрограмма	*Назначение*	*Автоматически выделяемая память (в байтах)*
Вещественные быстрые преобразования Фурье		
FFTRF	Выполняет прямое преобразование Фурье	$2N + 15$
FFTRB	Выполняет обратное преобразование Фурье	$2N + 15$
FFTRI	Вычисляет инициализирующие параметры для подпрограмм FFTRF и FFTRB	-
Комплексные быстрые преобразования Фурье		
FFTCF	Вычисляет прямое преобразование Фурье	$6N + 15$
FFTCB	Выполняет обратное преобразование Фурье	$6N + 15$
FFTCI	Вычисляет инициализирующие параметры для подпрограмм FFTCF, FFTCB, FFT2D, FFT2B, FFT3D и FFT3B	-

Вещественные быстрые синус- и косинус-преобразования Фурье		
FSINT	Вычисляет прямое и обратное дискретное синус-преобразование Фурье	INT(2.5*N* + 15)
FSINI	Вычисляет инициализирующие параметры для подпрограммы FSINT	-
FCOST	Вычисляет прямое и обратное дискретное косинус-преобразование Фурье	
FCOSI	Вычисляет инициализирующие параметры для подпрограммы FCOST	
Вещественные четвертьбыстрые синус- и косинус-преобразования Фурье		
QSINF	Вычисляет прямое дискретное синус-преобразование Фурье	3*N* + 15
QSINB	Вычисляет обратное дискретное синус-преобразование Фурье	3*N* + 15
QSINI	Вычисляет инициализирующие параметры для подпрограмм QSINF и QSINB	-
QCOSF	Вычисляет прямое дискретное косинус-преобразование Фурье	3*N* + 15
QCOSB	Вычисляет обратное дискретное косинус-преобразование Фурье	3*N* + 15
QCOSI	Вычисляет инициализирующие параметры для подпрограмм QCOSF и QCOSB	-

Таблица 1.3. Параметры подпрограмм из табл. 1.2

Имя	*Смысл/вид*	*Тип*
N	Размер преобразовываемой периодической последовательности / *входной*	INTEGER(4)
coef	Вектор размера *n*, содержащий коэффициенты Фурье / *выходной* в подпрограммах, выполняющих прямое преобразование Фурье, и *входной* в подпрограммах, выполняющих обратное преобразование Фурье	REAL(4) в случае вещественных преобразований и COMPLEX(4) в случае комплексных
seq	Вектор размера *n*, содержащий периодическую последовательность / *входной* в подпрограммах, выполняющих прямое преобразование Фурье, и *выходной* в подпрограммах, выполняющих обратное преобразование Фурье	То же

Замечание. Смысл параметров, описание которых отличается от приведенного в табл. 1.3, дополнительно уточняется при рассмотрении подпрограмм.

Для всех подпрограмм, выполняющих одномерные преобразования Фурье, справедливы следующие положения:

- для подпрограмм, употребляющих двойную точность, автоматически предоставляется в 2 раза больше памяти, чем для соответствующих подпрограмм, работающих с одинарной точностью;
- подпрограммы используют вариант алгоритма Кули-Туки, который наиболее эффективен, когда N является произведением небольших простых чисел. В этом случае время вычислений пропорционально $N\log N$;
- векторы *seq* и *coef* могут совпадать;
- если преобразовывающая подпрограмма, например FFTRF или FFTRB, используется несколько раз подряд с одним и тем же значением N, то для снижения вычислительных затрат следует прежде употреблять соответствующую подпрограмму инициализации (для FFTRF и FFTRB это FFTRI), а затем нужное число раз вызывать соответствующую подпрограмму второго уровня (для FFTRF и FFTRB это соответственно F2TRF и F2TRB);
- все подпрограммы основаны на процедурах пакета FFTPACK, созданного в Национальном центре атмосферных исследований США

1.2.2. ВЕЩЕСТВЕННЫЕ БЫСТРЫЕ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

1.2.2.1. Подпрограмма FFTRF (DFFTRF)

Выполняет прямое преобразование Фурье, т. е. вычисляет коэффициенты Фурье вещественной периодической последовательности. Имеет вызов

CALL FFTRF(*N, seq, coef*)

Описание параметров подпрограммы FFTRF см. в табл. 1.3.

Подпрограмма второго уровня F2TRF (DF2TRF) имеет вызов

CALL F2TRF(*N, seq, coef, wfftr*)

Дополнительный параметр подпрограммы F2TRF:

wfftr (*входной*) - вектор размера $2N + 15$, инициализируемый рассматриваемой ниже подпрограммой FFTRI.

Описание:

Подпрограмма FFTRF находит дискретное преобразование Фурье вещественного вектора размера N. Используется разновидность алгоритма Кули-Туки, который наиболее эффективен, когда N является произведением

небольших простых чисел. Если N удовлетворяет этим условиям, то время вычислений пропорционально $N\log N$.

Если N четно, то FFTRF возвращает в *coef* коэффициенты

$$c_{2m-2} = \sum_{n=1}^{N} s_n \cos\frac{(m-1)(n-1)2\pi}{N}, \; m = 2, \ldots, N/2 + 1,$$

$$c_{2m-1} = \sum_{n=1}^{N} s_n \sin\frac{(m-1)(n-1)2\pi}{N}, \; m = 2, \ldots, N,$$

$$c_1 = \sum_{n=1}^{N} s_n ,$$

где $s = seq$.

Если N четно, то коэффициенты c_m определяются по тем же формулам для $m = 2, \ldots, (N + 1)/2$.

Рассмотрим наиболее общий случай употребления подпрограммы FFTRF. Пусть f - вещественная функция, зависящая от времени. Пусть начиная с точки t_0 N точек функции f равномерно расположены на временном интервале длиной Δ секунд. То есть мы имеем последовательность, каждый элемент которой равен

$$seq_i = f(t_0 + (i - 1)\Delta), \; i = 1, 2, \ldots, N.$$

Подпрограмма FFTRF рассматривает эту последовательность как периодическую с периодом $N\Delta$. То есть она считает, что $f(t_0) = f(t_0 + N\Delta)$.

Далее, принимая вектор *seq*, подпрограмма FFTRF возвращает вектор коэффициентов $c = coef$, такой, что

$$seq_i = \frac{1}{N}\left(c_1 + 2\sum_{n=2}^{(N+1)/2} c_{2n-2} \cos\frac{2\pi(n-1)(i-1)}{2} - \right.$$
$$\left. - 2\sum_{n=2}^{(N+1)/2} c_{2n-1} \sin\frac{2\pi(n-1)(i-1)}{2} \right),$$

где N - нечетное число.

Коэффициенты, возвращаемые FFTRF, можно рассматривать как коэффициенты вычисленного по входным данным *seq* интерполяционного тригонометрического многочлена

$$g(t) = \frac{1}{N}\left(c_1 + 2\sum_{n=2}^{(N+1)/2} c_{2n-2}\cos\frac{2\pi(n-1)(t-t_0)}{T} - \right.$$
$$\left. - 2\sum_{n=2}^{(N+1)/2} c_{2n-1}\sin\frac{2\pi(n-1)(t-t_0)}{T} \right),$$

где период $T = N\Delta$. То есть мы имеем

$$f(t_0 + (i - 1)\Delta) = g(t_0 + (i - 1)\Delta).$$

Чтобы теперь выделить доминантную частоту, сформируем вектор p размера $N/2$, такой, что

$$p_1 = |c_1|,$$

$$p_k = \sqrt{c_{2k-2}^2 + c_{2k-1}^2}\ , k = 2, 3, \ldots, (N + 1)/2.$$

Эти величины соответствуют энергии в спектре сигнала. В частности, p_k соответствует энергетическому уровню на частоте

$$(k - 1)/T = (k - 1)/N\Delta, k = 1, 2, \ldots, (N + 1)/2.$$

Приведенные соотношения позволяют оценить вклад каждой гармоники в полученном интерполяционном многочлене. Причем в разложении - интерполяционном тригонометрическом многочлене - при N наблюдениях будут присутствовать только $(N + 1)/2 \approx T/(2\Delta)$ частот.

Для четного N преобразования выполняются по аналогичным формулам.

Пример. В качестве входных данных используется набор, представляющий функцию $y = \cos(2\pi i/N)$, $i = 0, 1, \ldots, N - 1$. Результирующий вектор *coef* с коэффициентами Фурье имеет только один ненулевой коэффициент, отвечающий частоте входного сигнала.

```
program fftrfTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: coef(N), twopi, seq(N)
 twopi = 2.0 * const('pi')              ! Функция CONST('pi') вернет число π
 do i = 1, N                            ! Формируем входной вектор
  seq(i) = cos(float(i - 1) * twopi / float(N))
 end do
 call fftrf(N, seq, coef)               ! Преобразование Фурье последовательности seq
 write(*, "(9x, 'Index', 5x, 'Seq', 6x, 'Coef')")
 write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, seq(i), coef(i), i = 1, N)
end program fftrfTest
```

Результат:

Index	Seq	Coef
1	1.00	0.00
2	0.62	3.50
3	-0.22	0.00
4	-0.90	0.00
5	-0.90	0.00
6	-0.22	0.00
7	0.62	0.00

1.2.2.2. Подпрограмма FFTRB (DFFTRB)

Выполняет обратное преобразование Фурье, т. е. вычисляет вещественную периодическую последовательность по коэффициентам Фурье. Имеет вызов

CALL FFTRB(*N, coef, seq*)

Описание параметров подпрограммы FFTRB см. в табл. 1.3.

Подпрограмма второго уровня F2TRB (DF2TRB) имеет вызов

CALL F2TRB(*N, coef, seq, wfftr*)

Дополнительный параметр подпрограммы F2TRB:

wfftr (*входной*) - вектор размера $2N + 15$, инициализируемый рассматриваемой ниже подпрограммой FFTRI.

Описание:

Так же как и FFTRF, подпрограмма FFTRB, вычисляющая обратное преобразование Фурье вещественного вектора $c = coef$ размера N, наиболее эффективна, когда N является произведением небольших простых чисел. Подпрограмма возвращает в векторе $s = seq$ вещественную периодическую последовательность, элементы которой равны

$$s_m = c_1 + (-1)^{(m-1)} c_N + 2\sum_{n=2}^{N/2} c_{2n-2} \cos\frac{2\pi(n-1)(m-1)}{N} - 2\sum_{n=2}^{N/2} c_{2n-1} \sin\frac{2\pi(n-1)(m-1)}{N},$$

если N четно, и равны

$$s_m = c_1 + 2\sum_{n=2}^{(N+1)/2} c_{2n-2} \cos\frac{2\pi(n-1)(m-1)}{N} - 2\sum_{n=2}^{(N+1)/2} c_{2n-1} \sin\frac{2\pi(n-1)(m-1)}{N},$$

если N нечетно.

Пример. Первоначально вычисляется прямое вещественное преобразование Фурье вектора x с элементами $x_j = (-1)^j$, $j = 1, 2, \ldots, N$. Затем вектор коэффициентов Фурье подается подпрограмме FFTRB, возвращающей вектор $s = Nx$, в котором $s_j = (-1)^j N$, $j = 1, 2, \ldots, N$.

```
program fftrbTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: x(N), coef(N), seq(N)
 do i = 1, N                          ! Формируем входную последовательность
  x(i) = float((-1)**i)
 end do
 call fftrf(N, x, coef)               ! Прямое дискретное преобразование Фурье
 write(*, "(9x, 'Result after forward transform')")
 write(*, "(9x, 'Index', 5x, 'x', 8x, 'Coef')")
 write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, x(i), coef(i), i = 1, N)
 call fftrb(N, coef, seq)             ! Обратное дискретное преобразование Фурье
 write(*, "(/, 9x,'Result after backward transform')")
 write(*, "(9x, 'Index', 4x, 'Coef', 6x, 'Seq')")
 write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, coef(i), seq(i), i = 1, N)
end program fftrbTest
```

Результат:

Result after forward transform			Result after backward transform		
Index	x	Coef	Index	Coef	Seq
1	-1.00	-1.00	1	-1.00	-7.00
2	1.00	-1.00	2	-1.00	7.00
3	-1.00	-0.48	3	-0.48	-7.00
4	1.00	-1.00	4	-1.00	7.00
5	-1.00	-1.25	5	-1.25	-7.00
6	1.00	-1.00	6	-1.00	7.00
7	-1.00	-4.38	7	-4.38	-7.00

1.2.2.3. Подпрограмма FFTRI (DFFTRI)

Вычисляет инициализирующие параметры для подпрограмм FFTRF и FFTRB. Имеет вызов

```
CALL FFTRI(N, wfftr)          ! Смысл параметра N см. в табл. 1.3
```

wfftr (*выходной*) - вектор размера $2N + 15$, содержащий инициализирующие параметры для подпрограмм FFTRF и FFTRB. Тип *wfftr* - REAL(4) или REAL(8).

Описание:

Подпрограмма FFTRI позволяет увеличить эффективность работы с подпрограммами FFTRF и FFTRB в тех случаях, когда последние неоднократно вызываются для вычисления преобразований Фурье последовательностей одной длины. Правда, в этом случае вместо FFTRF употребляется F2TRF, а вместо FFTRB - F2TRB.

Пример. Вычисляются 3 БПФ различных последовательностей одинаковой длины. Для подготовки начальных данных 1 раз вызывается подпрограмма FFTRI, а затем 3 раза - подпрограмма F2TRF.

```
program fftriTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i, k
 real(4) :: coef(N), twopi, wfftr(2 * N + 15), seq(N)
 twopi = 2.0 * const('pi')                ! Функция CONST('pi') вернет число π
 call fftri(N, wfftr)                     ! Формируем рабочий вектор
 do k = 1, 3
  do i = 1, N                             ! Формируем входной вектор
   seq(i) = cos(float(k * (i - 1)) * twopi / float(N))
  end do
  ! Преобразование Фурье последовательности seq
  call f2trf(N, seq, coef, wfftr)
  write(*, "(1x, 'k = ', i1)") k
  write(*, "(9x, 'Index', 5x, 'Seq', 6x, 'Coef')")
  write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, seq(i), coef(i), i = 1, N)
 end do
end program fftriTest
```

Результат:

k = 1			k = 2			k = 3		
Index	Seq	Coef	Index	Seq	Coef	Index	Seq	Coef
1	1.00	0.00	1	1.00	0.00	1	1.00	0.00
2	0.62	3.50	2	-0.22	0.00	2	-0.90	0.00
3	-0.22	0.00	3	-0.90	0.00	3	0.62	0.00
4	-0.90	0.00	4	0.62	3.50	4	-0.22	0.00
5	-0.90	0.00	5	0.62	0.00	5	-0.22	0.00
6	-0.22	0.00	6	-0.90	0.00	6	0.62	3.50
7	0.62	0.00	7	-0.22	0.00	7	-0.90	0.00

1.2.3. КОМПЛЕКСНЫЕ БЫСТРЫЕ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

1.2.3.1. Подпрограмма FFTCF (DFFTCF)

Вычисляет коэффициенты Фурье комплексной периодической последовательности. Имеет вызов

CALL FFTCF(*N, seq, coef*)

Описание параметров подпрограммы FFTCF см. в табл. 1.3.

Подпрограмма второго уровня F2TCF (DF2TCF) имеет вызов

CALL F2TCF(*N, seq, coef, wfftc, cpy*)

Дополнительные параметры подпрограммы F2TCF:

wfftc (*входной*) - вещественный вектор размера $4N + 15$, данные в который заносятся подпрограммой FFTCI;

cpy (*рабочая область*) - вещественный вектор размера $2N$.

Описание:

Подпрограмма FFTCF осуществляет прямое дискретное преобразование Фурье комплексного вектора размера N. Элементы вектора коэффициентов $c = coef$ по заданному вектору $s = seq$ вычисляются в FFTCF по формуле

$$c_m = \sum_{n=1}^{N} s_n e^{-2\pi i(n-1)(m-1)/N} .$$

Коэффициенты нормализуются таким образом, что евклидова норма d вектора c оказывается равной $\sqrt{N}d$.

Пример. Входным является вектор с чистым экспоненциальным сигналом, имеющим частоту, кратную 3. На выходе - вектор, все компоненты которого равны нулю, кроме имеющего подходящую частоту. Заметим, что норма входного вектора - $\sqrt{N}$, а результирующего - N.

```
program fftcfTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: twopi
 complex(4) :: c = (0.0, 1.0), coef(N), h, seq(N)
 twopi = 2.0 * const('pi')              ! Функция CONST('pi') вернет число π
 h = (twopi * c / N) * 3.0              ! Вычисляем 3(2πi/N)
 ! Генерация входного вектора с чистым экспоненциальным сигналом
 ! с частотой, кратной 3
 do i=1, N; seq(i) = cexp((i - 1) * h); end do
 call fftcf(N, seq, coef)               ! Прямое преобразование Фурье для вектора seq
```

```
! Вывод результата
print '(7x, a, 15x, a, 14x, a)', 'index', 'seq', 'coef'
print "((1x, i11, 5x,'(',f5.2,',',f5.2,')', 5x,'(',f5.2,',',f5.2,')'))", (i, seq(i), coef(i), i = 1, N)
end program fftcfTest
```

Результат:

Index	Seq	Coef
1	(1.00, 0.00)	(0.00, 0.00)
2	(-0.90, 0.43)	(0.00, 0.00)
3	(0.62,-0.78)	(0.00, 0.00)
4	(-0.22, 0.97)	(7.00, 0.00)
5	(-0.22,-0.97)	(0.00, 0.00)
6	(0.62, 0.78)	(0.00, 0.00)
7	(-0.90,-0.43)	(0.00, 0.00)

1.2.3.2. Подпрограмма FFTCB (DFFTCB)

Вычисляет комплексную периодическую последовательность по известным коэффициентам Фурье. Имеет вызов

CALL FFTCB(*N, coef, seq*)

Описание параметров подпрограммы FFTCB см. в табл. 1.3.

Подпрограмма второго уровня F2TCB (DF2TCB) имеет вызов

CALL F2TCB(*N, coef, seq, wfftc, cpy*)

Дополнительные параметры подпрограммы F2TCB: *wfftc, cpy*. Их описание см. в предшествующем разделе.

Описание:

Обратное преобразование по известным коэффициентам $c = coef$ возвращается подпрограммой FFTCB в векторе $s = seq$, элементы которого вычисляются по формуле

$$s_m = \sum_{n=1}^{N} c_n e^{2\pi i(n-1)(m-1)/N}.$$

Коэффициенты нормализуются таким образом, что евклидова норма d вектора s оказывается равной $\sqrt{N}d$.

Пример. Первоначально выполняется преобразование Фурье вектора x, в котором $x_j = j$ для $j = 1, \ldots, N$. Норма вектора x равна

$$\sqrt{N(N+1)(2N+1)/6}.$$

Следовательно, норма преобразованного вектора равна

$$N\sqrt{(N+1)(2N+1)/6}\ .$$

Затем преобразованный вектор используется в качестве входного параметра подпрограммы FFTCB, выполняющей обратное преобразование Фурье. Его результатом является вектор $s = Nx$, в котором $s_j = jN$ для $j = 1, \ldots, N$.

```
program fftcbTest
! Текст модуля text_transfer см. в [5, прил. 1]
use text_transfer
use dfimsl
integer(4), parameter :: N = 7
integer(4) :: i
complex(4) :: coef(N), seq(N), x(N)
! Входная периодическая последовательность для FFTCF
do i=1, N; x(i) = cmplx(i, 0); end do
call fftcf(N, x, coef)                    ! Прямое преобразование Фурье
call fftcb(N, coef, seq)                  ! Обратное преобразование Фурье
! Текст подпрограммы ru_doswin см. в [5, прил. 1]
print '(2x, a, 12x, a, 6x, a, 3x, a)',                                    &
      trim(ru_doswin('Индекс', .false.)),                                 &
      trim(ru_doswin('Вход', .false.)),                                   &
      trim(ru_doswin('Прямое преобразование', .false.)),                  &
      trim(ru_doswin('Обратное преобразование ', .false.))
print "(1x, i5, 9x, '(', f5.2, ', ', f5.2, ')', 7x, '(', f5.2, ', ', f5.2, ')', 12x,  &
      '(', f5.2, ', ', f5.2 ,')')", (i, x(i), coef(i), seq(i), i = 1, N)
end program fftcbTest
```

Результат:

Индекс	Вход	Прямое преобразование	Обратное преобразование
1	(1.00, 0.00)	(28.00, 0.00)	(7.00, 0.00)
2	(2.00, 0.00)	(-3.50, 7.27)	(14.00, 0.00)
3	(3.00, 0.00)	(-3.50, 2.79)	(21.00, 0.00)
4	(4.00, 0.00)	(-3.50, 0.80)	(28.00, 0.00)
5	(5.00, 0.00)	(-3.50,-0.80)	(35.00, 0.00)
6	(6.00, 0.00)	(-3.50,-2.79)	(42.00, 0.00)
7	(7.00, 0.00)	(-3.50,-7.27)	(49.00, 0.00)

1.2.3.3. Подпрограмма FFTCI (DFFTCI)

Вычисляет параметры для подпрограмм FFTCF, FFTCB, FFT2D, FFT2B, FFT3D и FFT3B. Имеет вызов

```
CALL FFTCI(N, wfftc)          ! Смысл параметра N см. в табл. 1.3
```

wfftc (*выходной*) - вещественный вектор размера $4N + 15$, содержащий инициализирующие параметры для подпрограмм FFTCF и FFTCB. Тип *wfftc* - REAL(4) или REAL(8).

Пример. Вычисляется двумерное комплексное БПФ. Для подготовки начальных данных один раз вызывается подпрограмма FFTCI, а затем $2N$ раз - подпрограмма F2TCF.

```
program fftciTest
 use dfimsl
 integer(4), parameter :: N = 4
 integer(4) :: i, ir, is, j
 real(4) :: twopi, wfftc(4 * N + 15), cpy(2 * N)
 complex(4) :: coef(N, N), h, seq(N, N), temp
 twopi = 2.0 * const('pi')              ! Функция CONST('pi') вернет число π
 ir = 3; is = 1
 temp = cmplx(0.0, twopi / float(N))    ! Вычисляем exp(2πi/N)
 h = cexp(temp)
 do i = 1, N                            ! Формируем массив seq
  do j = 1, N; seq(i, j) = h**((i - 1) * (ir - 1) + (j - 1) * (is - 1)); end do
 end do
 ! Печатаем массив seq
 write(*, "(1x, 'The input matrix is below')")
 do i = 1, N; write(*, 1) (seq(i, j), j = 1, N); end do
 call fftci(N, wfftc)                   ! Формируем инициализирующий вектор
 do i = 1, N                            ! Преобразование столбца массива seq
  call f2tcf(N, seq(1, i), coef(1, i), wfftc, cpy)
 end do
 coef = transpose(coef)                 ! Транспонируем массив coef
 do i =1, N                             ! Преобразование столбца массива coef
  call f2tcf(N, coef(1, i), seq(1, i), wfftc, cpy)
 end do
 seq = transpose(seq)                   ! Транспонируем массив seq
 ! Вывод результата
 write(*, "(/, 1x, 'Result of two-dimensional transform'):")
 do i = 1, N; write(*, 1) (seq(i, j), j = 1, N); end do
 1 format(1x, 4(' (', f5.2, ', ', f5.2, ')'))
end program fftciTest
```

Результат:

```
The input matrix is below
    ( 1.00, 0.00)     ( 1.00, 0.00)     ( 1.00, 0.00)     ( 1.00, 0.00)
    (-1.00, 0.00)     (-1.00, 0.00)     (-1.00, 0.00)     (-1.00, 0.00)
    ( 1.00, 0.00)     ( 1.00, 0.00)     ( 1.00, 0.00)     ( 1.00, 0.00)
    (-1.00, 0.00)     (-1.00, 0.00)     (-1.00, 0.00)     (-1.00, 0.00)
```

Result of two-dimensional transform

(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)
(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)
(16.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)
(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)

1.2.4. ВЕЩЕСТВЕННЫЕ БЫСТРЫЕ СИНУС- И КОСИНУС-ПРЕОБРАЗОВАНИЯ ФУРЬЕ

1.2.4.1. Подпрограмма FSINT (DFSINT)

Вычисляет прямое и обратное дискретное синус-преобразование Фурье. Имеет вызов

CALL FSINT(*N, seq, coef*)

Описание параметров подпрограммы FSINT, кроме *coef*, см. в табл. 1.3.

coef - вектор размера $N + 1$, содержащий коэффициенты Фурье.

Подпрограмма второго уровня F2INT (DF2INT) имеет вызов

CALL F2INT(*N, seq, coef, wfsin*)

Дополнительный параметр подпрограммы F2INT:

wfsin (*входной*) - вектор размера INT(2.5N + 15), инициализируемый рассматриваемой ниже подпрограммой FSINI.

Комментарии:

1. Подпрограмма FSINT наиболее эффективна, когда $N + 1$ является произведением небольших простых чисел.
2. Подпрограмма FSINT также выполняет и обратные ненормализованные преобразования. Двукратное применение FSINT воспроизводит первоначальную последовательность, умноженную на $2(N + 1)$.
3. Векторы *seq* и *coef* могут совпадать, если размер *seq* не меньше чем $N + 1$.
4. Элемент *coef*($N + 1$) необходим как рабочее пространство.

Описание:

Подпрограмма FSINT для данного вектора s = *seq* размера N возвращает коэффициенты c = *coef*, вычисляемые по формуле

$$c_m = 2\sum_{n=1}^{N} s_n \sin\frac{mn\pi}{N+1}.$$

Она же вычисляет и обратное ненормализованное преобразование Фурье.

Пример. Вычисляется прямое синус-преобразование Фурье вектора *seq*, порожденного функцией $\sin(\pi x/(N + 1))$. Как и ожидалось, все полученные коэффициенты Фурье, кроме первого, равны нулю. Величина первого ко-

эффициента равна $N + 1$. Далее вычисляется обратное синус-преобразование Фурье.

```
program fsintTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: coef(N + 1), pi, seq(N)
 pi = const('pi')                          ! Функция CONST('pi') вернет число π
 do i = 1, N                               ! Формируем входной вектор seq
  seq(i) = sin(float(i) * pi / float(N + 1))
end do
! Прямое преобразование Фурье последовательности seq
call fsint(N, seq, coef)
write(*, "('Forward transform')")
write(*, "(9x, 'Index', 5x, 'Seq', 6x, 'Coef')")
write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, seq(i), coef(i), i = 1, N)
! Обратное преобразование Фурье последовательности seq
call fsint(N, coef, seq)
seq = seq / real(2 * (N + 1))              ! Для проверки обратного преобразования
write(*, "('Backward transform')")
write(*, "(9x, 'Index', 5x, 'Seq')")
write(*, "(1x, i11, 5x, f5.2)") (i, seq(i), i = 1, N)
end program fsintTest
```

Результат:

Forward transform			Backward transform	
Index	Seq	Coef	Index	Seq
1	0.38	8.00	1	0.38
2	0.71	0.00	2	0.71
3	0.92	0.00	3	0.92
4	1.00	0.00	4	1.00
5	0.92	0.00	5	0.92
6	0.71	0.00	6	0.71
7	0.38	0.00	7	0.38

1.2.4.2. Подпрограмма FSINI (DFSINI)

Вычисляет параметры для подпрограммы FSINT. Имеет вызов

CALL FSINI(*N, wfsin*)

Параметры подпрограммы FSINI:

N (*входной*) - размер преобразовываемой последовательности; должен быть больше единицы.

wfsin (*выходной*) - вектор размера INT(2.5*N* + 15), содержащий инициализирующие параметры для подпрограммы FSINT. Тип *wfftr* - REAL(4) или REAL(8).

Пример. Вычисляются 3 БПФ различных последовательностей одинаковой длины, порождаемые чистыми синусоидальными волнами. Для подготовки начальных данных 1 раз вызывается подпрограмма FSINI, а затем 3 раза - подпрограмма F2INT.

```
program fsiniTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i, k
 real(4) :: coef(N + 1), pi, wfsin(int(2.5 * N + 15)), seq(N)
 pi = const('pi')                    ! Функция CONST('pi') вернет число π
 call fsini(N, wfsin)                ! Формируем рабочий вектор wfsin
 do k = 1, 3
  do i = 1, N                        ! Формируем входной вектор
   seq(i) = sin(float(k * i) * pi / float(N + 1))
  end do
  ! Преобразование Фурье последовательности seq
  call f2int(N, seq, coef, wfsin)
  write(*, "(1x, 'k = ', i1)") k
  write(*, "(9x, 'Index', 5x, 'Seq', 6x, 'Coef')")
  write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, seq(i), coef(i), i = 1, N)
 end do
end program fsiniTest
```

Результат:

k = 1			k = 2			k = 3		
Index	Seq	Coef	Index	Seq	Coef	Index	Seq	Coef
1	0.38	8.00	1	0.71	0.00	1	0.92	0.00
2	0.71	0.00	2	1.00	8.00	2	0.71	0.00
3	0.92	0.00	3	0.71	0.00	3	-0.38	8.00
4	1.00	0.00	4	0.00	0.00	4	-1.00	0.00
5	0.92	0.00	5	-0.71	0.00	5	-0.38	0.00
6	0.71	0.00	6	-1.00	0.00	6	0.71	0.00
7	0.38	0.00	7	-0.71	0.00	7	0.92	0.00

1.2.4.3. Подпрограмма FCOST (DFCOST)

Вычисляет прямое и обратное дискретное косинус-преобразование Фурье. Имеет вызов

CALL FCOST(*N, seq, coef*)

Описание параметров подпрограммы FCOST, кроме *N*, см. в табл. 1.3.

N (*входной*) - размер преобразовываемой последовательности; должен быть больше единицы.

Подпрограмма второго уровня F2OST (DF2OST) имеет вызов

CALL F2OST(*N, seq, coef, wfcos*)

Дополнительный параметр подпрограммы F2OST:

wfcos (*входной*) - вектор размера $3N + 15$, инициализируемый рассматриваемой ниже подпрограммой FCOSI.

Комментарии:

1. Подпрограмма FCOST наиболее эффективна, когда $N - 1$ является произведением небольших простых чисел.
2. Подпрограмма FCOST также выполняет и обратные ненормализованные преобразования. Двукратное применение FCOST воспроизводит первоначальную последовательность, умноженную на $2(N - 1)$.

Описание:

Подпрограмма FCOST для данного вектора $s = seq$ размера N возвращает коэффициенты $c = coef$, вычисляемые по формуле

$$c_m = s_1 + 2\sum_{n=1}^{N-1} s_n \cos\frac{(m-1)(n-1)\pi}{N-1} + s_N(-1)^{m-1}.$$

Она же вычисляет и обратное ненормализованное преобразование Фурье.

Пример. Вычисляется прямое косинус-преобразование Фурье вектора *seq*, порожденного функцией $\cos(\pi(x - 1)/(N - 1))$. Как и ожидалось, все полученные коэффициенты Фурье, кроме второго, равны нулю. Величина второго коэффициента равна $N - 1$. Затем вычисляется обратное косинус-преобразование Фурье.

```
program fcostTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: coef(N), pi, seq(N)
 pi = const('pi')                          ! Функция CONST('pi') вернет число π
 do i = 1, N                               ! Формируем входной вектор
  seq(i) = cos(float(i - 1) * pi / float(N - 1))
 end do
 ! Прямое преобразование Фурье последовательности seq
 call fcost(N, seq, coef)
 write(*, "('Forward transform')")
 write(*, "(9x, 'Index', 5x, 'Seq', 6x, 'Coef')")
```

```
write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, seq(i), coef(i), i = 1, N)
! Обратное преобразование Фурье последовательности seq
call fcost(N, coef, seq)
seq = seq / real(2 * (N - 1))          ! Для проверки обратного преобразования
write(*, "('Backward transform')")
write(*, "(9x, 'Index', 5x, 'Seq')")
write(*, "(1x, i11, 5x, f5.2)") (i, seq(i), i = 1, N)
end program fcostTest
```

Результат:

Forward transform			Backward transform	
Index	Seq	Coef	Index	Seq
1	1.00	0.00	1	1.00
2	0.87	6.00	2	0.87
3	0.50	0.00	3	0.50
4	0.00	0.00	4	0.00
5	-0.50	0.00	5	-0.50
6	-0.87	0.00	6	-0.87
7	-1.00	0.00	7	-1.00

1.2.4.4. Подпрограмма FCOSI (DFCOSI)

Вычисляет параметры для подпрограммы FCOST. Имеет вызов

CALL FCOSI(*N, wfcos*)

Параметры подпрограммы FCOSI:

N (*входной*) - размер преобразовываемой последовательности; должен быть больше единицы.

wfcos (*выходной*) - вектор размера 3*N* + 15, содержащий инициализирующие параметры для подпрограммы FCOST. Тип *wfftr* - REAL(4) или REAL(8).

1.2.5. ВЕЩЕСТВЕННЫЕ ЧЕТВЕРТЬБЫСТРЫЕ СИНУС- И КОСИНУС-ПРЕОБРАЗОВАНИЯ ФУРЬЕ

1.2.5.1. Подпрограмма QSINF (DQSINF)

Вычисляет прямое дискретное синус-преобразование Фурье. Имеет вызов

CALL QSINF(*N, seq, coef*)

Описание параметров подпрограммы QSINF см. в табл. 1.3.

Подпрограмма второго уровня Q2INF (DQ2INF) имеет вызов

CALL Q2INF(*N, seq, coef, wqsin*)

Дополнительный параметр подпрограммы Q2INF:

wqsin (*входной*) - вектор размера $3N + 15$, инициализируемый рассматриваемой ниже подпрограммой QSINI.

Описание:

Подпрограмма QSINF для данного вектора $s = seq$ размера N возвращает коэффициенты $c = coef$, вычисляемые по формуле

$$c_m = 2\sum_{n=1}^{N-1} s_n \sin\frac{(2m-1)n\pi}{2N} + s_N(-1)^{m-1}.$$

Пример. Вычисляется четвертьбыстрое синус-преобразование Фурье вектора *seq*, порожденного функцией $\sin(0.5\pi x/N)$.

```
program qsinfTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: coef(N), pi, seq(N)
 pi = const('pi')                          ! Функция CONST('pi') вернет число π
 do i = 1, N                               ! Формируем входной вектор seq
  seq(i) = sin(float(i) * (pi / 2.0) / float(N))
 end do
 ! Прямое преобразование Фурье последовательности seq
 call qsinf(N, seq, coef)
 write(*, "(9x, 'Index', 5x, 'Seq', 6x, 'Coef')")
 write(*, "(1x, i11, 5x, f5.2, 5x, f5.2)") (i, seq(i), coef(i), i = 1, N)
end program qsinfTest
```

Результат:

Index	Seq	Coef
1	0.22	7.00
2	0.43	0.00
3	0.62	0.00
4	0.78	0.00
5	0.90	0.00
6	0.97	0.00
7	1.00	0.00

1.2.5.2. Подпрограмма QSINB (DQSINB)

Вычисляет обратное дискретное синус-преобразование Фурье. Имеет вызов

CALL QSINB(*N, coef, seq*)

Описание параметров подпрограммы QSINB см. в табл. 1.3.

Подпрограмма второго уровня DQ2INB (DQ2INB) имеет вызов

CALL Q2INB(*N, seq, coef, wqsin*)

Смысл дополнительного параметра *wqsin* подпрограммы Q2INB см. в предшествующем разделе.

Описание:

Подпрограмма QSINB для данного вектора коэффициентов Фурье c = = *coef* размера N возвращает последовательность s = *seq*, элементы которой вычисляются по формуле

$$s_m = 4\sum_{n=1}^{N} c_n \sin\frac{(2n-1)m\pi}{2N}.$$

Кроме того, вектор x размера N, преобразованный подпрограммой QSINF, а затем - QSINB, будет возвращен последней подпрограммой как $4Nx$.

Пример. Первоначально вычисляется прямое четвертьбыстрое синус-преобразование Фурье вектора x с элементами $x_j = j$, j = 1, 2, ..., N. Затем вектор коэффициентов Фурье подается подпрограмме QSINB, возвращающей вектор $s = 4Nx$.

```
program qsinbTest
 use dfimsl
 integer(4), parameter :: N = 7
 integer(4) :: i
 real(4) :: x(N), coef(N), seq(N)
 do i = 1, N; x(i) = float(i); end do      ! Формируем входную последовательность
 call qsinf(N, x, coef)                    ! Прямое дискретное преобразование Фурье
 call qsinb(N, coef, seq)                  ! Обратное дискретное преобразование Фурье
 write(*, "(5x, 'Input', 5x, 'Forward transform', 3x, 'Backward transform')")
 write(*, "(3x, f6.2, 10x, f6.2, 15x, f6.2)") (x(i), coef(i), seq(i), i = 1, N)
end program qsinbTest
```

Результат:

Input	Forward transform	Backward transform
1.00	39.88	28.00
2.00	-4.58	56.00
3.00	1.77	84.00
4.00	-1.00	112.00
5.00	0.70	140.00
6.00	-0.56	168.00
7.00	0.51	196.00

1.2.5.3. Подпрограмма QSINI (DQSINI)

Вычисляет инициализирующие параметры для подпрограмм QSINF и QSINB. Имеет вызов

CALL QSINI(*N, wqsin*) ! Смысл параметра *N* см. в табл. 1.3

wqsin (*выходной*) - вектор размера $3N + 15$, содержащий инициализирующие параметры для подпрограмм QSINF и QSINB. Тип *wqsin* - REAL(4) или REAL(8).

1.2.5.4. Подпрограмма QCOSF (DQCOSF)

Вычисляет прямое дискретное косинус-преобразование. Имеет вызов

CALL QCOSF(*N, seq, coef*)

Описание параметров подпрограммы QCOSF см. в табл. 1.3.

Подпрограмма второго уровня имеет вызов

CALL Q2OSF(*N, seq, coef, wqcos*)

Дополнительный параметр подпрограммы Q2OSF:

wqcos (*входной*) - вектор размера $3N + 15$, инициализируемый рассматриваемой ниже подпрограммой QCOSI.

Описание:

Подпрограмма QCOSF для данного вектора $s = seq$ размера N возвращает коэффициенты $c = coef$, вычисляемые по формуле

$$c_m = s_1 + 2\sum_{n=2}^{N} s_n \cos\frac{(2m-1)(n-1)\pi}{2N}.$$

1.2.5.5. Подпрограмма QCOSB (DQCOSB)

Вычисляет обратное дискретное косинус-преобразование Фурье. Имеет вызов

CALL QCOSB(*N, coef, seq*)

Описание параметров подпрограммы QCOSB см. в табл. 1.3.

Подпрограмма второго уровня имеет вызов

CALL Q2OSB(*N, coef, seq, wqcos*)

Смысл дополнительного параметра *wqcos* подпрограммы Q2OSB см. в предшествующем разделе.

Описание:

Подпрограмма QCOSB для данного вектора коэффициентов Фурье $c = coef$ размера N возвращает последовательность $s = seq$, элементы которой вычисляются по формуле

$$s_m = 4\sum_{n=1}^{N} c_n \cos\frac{(2n-1)(m-1)\pi}{2N}.$$

Кроме того, вектор *x* размера *N*, преобразованный подпрограммой QCOSF, а затем - QCOSB, будет возвращен последней подпрограммой как $4Nx$.

1.2.5.6. Подпрограмма QCOSI (DQCOSI)

Вычисляет инициализирующие параметры для подпрограмм QCOSF и QCOSB. Имеет вызов

CALL QCOSI(*N, wqcos*) ! Смысл параметра *N* см. в табл. 1.3

wqcos (*выходной*) - вектор размера $3N + 15$, содержащий инициализирующие параметры для подпрограмм QCOSF и QCOSB. Тип *wqcos* - REAL(4) или REAL(8).

1.3. ДВУМЕРНЫЕ И ТРЕХМЕРНЫЕ КОМПЛЕКСНЫЕ БЫСТРЫЕ ПРЕОБРАЗОВАНИЯ ФУРЬЕ

1.3.1. ПЕРЕЧЕНЬ ПОДПРОГРАММ

Приведен в табл. 1.4.

Таблица 1.4. Подпрограммы, выполняющие двумерные и трехмерные комплексные быстрые преобразования Фурье

Подпрограмма	*Назначение*
Двумерные комплексные быстрые преобразования Фурье	
FFT2D	Выполняет прямое преобразование Фурье комплексного двумерного *N*×*M*-массива
FFT2B	Выполняет обратное преобразование Фурье комплексного двумерного *N*×*M*-массива
Трехмерные комплексные быстрые преобразования Фурье	
FFT3D	Вычисляет прямое преобразование Фурье комплексного трехмерного *N*×*M*×*L*-массива
FFT3B	Выполняет обратное преобразование Фурье комплексного трехмерного *N*×*M*×*L*-массива

1.3.2. ПОДПРОГРАММА FFT2D (DFFT2D)

Вычисляет коэффициенты Фурье комплексного периодического двумерного массива *a*. Имеет вызов

CALL FFT2D(*nra, nca, a, Lda, coef, Ldcoef*)

Параметры подпрограммы FFT2D:
Входные: nra, nca, a, Lda, coef, Ldcoef.
Выходной: coef.

nra, nca - соответственно число строк и столбцов во входном наборе данных.

a - комплексный массив формы (*Lda, nca*), представляющий комплексный периодический двумерный *nra*×*nca*-набор данных.

Lda - ведущий размер массива *a*; *Lda* ≥ *nra*.

coef - комплексный массив формы (*Ldcoef, nca*), представляющий комплексный двумерный *nra*×*nca*-набор данных с коэффициентами Фурье.

Ldcoef - ведущий размер массива *coef*; *Ldcoef* ≥ *nra*.

Автоматически для решения предоставляется память:

- 4(*nra* + *nca*) + 32 + 2max(*nra, nca*) байт в случае FFT2D;
- 8(*nra* + *nca*) + 64 + 4max(*nra, nca*) байт в случае DFFT2D.

Память можно выделить явно, употребив F2T2D (DF2T2D):

CALL F2T2D(*nra, nca, a, Lda, coef, Ldcoef, wff*1, *wff*2, *cwk, cry*)

Дополнительные параметры подпрограммы F2T2D:

*wff*1, *wff*2 (*выходные*) - вещественные векторы, имеющие соответственно размеры 4*nra* + 15 и 4*nca* + 15; инициализируются подпрограммой FFTCI.

cwk (*рабочая область*) - комплексный вектор размера 1.

cry (*рабочая область*) - вещественный массив размера 2max(*nra, nca*).

Комментарии:

1. Подпрограмма FFT2D наиболее эффективна, когда *nra* и *nca* являются произведением небольших простых чисел. В этом случае время вычислений пропорционально $NM \log NM$, где $M = nca$ и $N = nra$.
2. Массивы *coef* и *a* могут совпадать.
3. Если FFT2D или FFT2B неоднократно вызываются с одинаковыми значениями *nra* и *nca*, то прежде следует использовать инициализирующую подпрограмму FFTCI для заполнения векторов *wff*1 и *wff*2, а затем нужное число раз вызвать подпрограмму второго уровня F2T2D или F2T2B. Такой подход более эффективен, чем многократный вызов FFT2D или FFT2B.

Описание:

Для заданного *N*×*M*-массива *a* подпрограмма возвращает массив *c* = *coef*, элементы которого вычисляются по формуле

$$c_{jk} = \sum_{n=1}^{N}\sum_{m=1}^{M} a_{nm} e^{-2\pi i(j-1)(n-1)/N} e^{-2\pi i(k-1)(m-1)/M}.$$

Коэффициенты нормализуются таким образом, что евклидова норма d вектора оказывается равной $\sqrt{NMd}$.

Пример. Вычисляется преобразование Фурье 5×4-массива, имеющего элементы

$$a_{nm} = e^{2\pi i(n-1)2/5} e^{2\pi i(m-1)3/4}, \quad 1 \le n \le 5 \text{ и } 1 \le m \le 4.$$

Результат имеет нули во всех позициях, кроме (3, 4).

```
program fft2dTest
 use dfimsl
 integer(4) :: i, ir, is, j, Lda, Ldcoef, nca, nra
 real(4) :: twopi
 complex(4) :: a(5, 4), c, coef(5, 4), h
 character(26) :: tit1 = 'The input matrix', tit2 = 'The output matrix'
 nra = 5; nca = 4; Lda = 5; Ldcoef = 5; ir = 3; is = 4
 ! Формируем начальные данные
 twopi = 2.0 * const('pi')                    ! Функция CONST('pi') вернет число π
 c = cmplx(0.0, 1.0); h = cexp(twopi * c)
 do i = 1, nra; do j = 1, nca
  a(i, j) = cexp(twopi * c * ((float((i - 1) * (ir - 1)) / float(nra) +          &
       float((j - 1) * (is - 1)) / float(nca))))
 end do; end do
call wrcrn(tit1, nra, nca, a, Lda, 0)
call fft2d(nra, nca, a, Lda, coef, Ldcoef)
call wrcrn(tit2, nra, nca, coef, Ldcoef, 0)
end program fft2dTest
```

Результат:

The input matrix

	1	2	3	4
1	(1.000, 0.000)	(0.000,-1.000)	(-1.000, 0.000)	(0.000, 1.000)
2	(-0.809, 0.588)	(0.588, 0.809)	(0.809,-0.588)	(-0.588,-0.809)
3	(0.309,-0.951)	(-0.951,-0.309)	(-0.309, 0.951)	(0.951, 0.309)
4	(0.309, 0.951)	(0.951,-0.309)	(-0.309,-0.951)	(-0.951, 0.309)
5	(-0.809,-0.588)	(-0.588, 0.809)	(0.809, 0.588)	(0.588,-0.809)

The output matrix

	1	2	3	4
1	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)
2	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)
3	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(20.00, 0.00)
4	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)
5	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)	(0.00, 0.00)

1.3.3. ПОДПРОГРАММА FFT2B (DFFT2B)

Вычисляет обратное преобразование Фурье комплексного периодического двумерного массива *coef*. Имеет вызов:

CALL FFT2B(*nrcoef, nccoef, coef, Ldcoef, a, Lda*)

Параметры подпрограммы FFT2B:
Входные: nrcoef, nccoef, coef, Ldcoef, Lda.
Выходной: a.

nrcoef, nccoef - соответственно число строк и столбцов во входном массиве с коэффициентами Фурье.

coef - комплексный массив формы (*Ldcoef, nccoef*), содержащий коэффициенты Фурье.

Ldcoef - ведущий размер массива *coef*; *Ldcoef* ≥ *nrcoef*.

a - комплексный массив формы (*Lda, nccoef*), содержащий обратное *nrcoef*×*nccoef*-преобразование Фурье.

Lda - ведущий размер массива *a*; *Lda* ≥ *nrcoef*.

Автоматически для решения предоставляется память:

- 4(*nrcoef* + *nccoef*) + 32 + 2max(*nrcoef, nccoef*) байт в случае FFT2B;
- 8(*nrcoef* + *nccoef*) + 64 + 4max(*nrcoef, nccoef*) байт в случае DFFT2B.

Память можно выделить явно, употребив F2T2B (DF2T2B):

CALL F2T2B(*nrcoef, nccoef, a, Lda, coef, Ldcoef, wff*1, *wff*2, *cwk, cpy*)

Дополнительные параметры подпрограммы F2T2B:

*wff*1, *wff*2 (*входные*) - вещественные векторы, имеющие соответственно размеры 4*nrcoef* + 15 и 4*nccoef* + 15; инициализируются подпрограммой FFTCI.

cwk (*рабочая область*) - комплексный вектор размера 1.

cpy (*рабочая область*) - вещественный массив размера 2max(*nrcoef, nccoef*).

Комментарии:

1. Подпрограмма FFT2B наиболее эффективна, когда *nrcoef* и *nccoef* являются произведением небольших простых чисел.
2. См. комментарии 2 и 3 в предшествующем разделе.

Описание:

Для заданного $N \times M$-массива коэффициентов c, где N = *nrcoef* и M = *nccoef*, подпрограмма возвращает двумерный массив a, элементы которого вычисляются по формуле

$$a_{jk} = \sum_{n=1}^{N} \sum_{m=1}^{M} c_{nm} e^{2\pi i(j-1)(n-1)/N} e^{2\pi i(k-1)(m-1)/M}.$$

Коэффициенты нормализуются таким образом, что евклидова норма d вектора оказывается равной $\sqrt{NMd}$.

Пример. Первоначально подпрограмма FFT2D вычисляет прямое преобразование Фурье 5×4-массива, имеющего элементы

$$x_{nm} = n + 5(m - 1), \quad 1 \le n \le 5 \text{ и } 1 \le m \le 4.$$

Затем подпрограмма FFT2B находит обратное преобразование. Заметьте, что результат $a = 5 * 4 * x = 20 * x$.

```
program fft2bTest
 use dfimsl
 integer(4) :: Lda, Ldcoef, m, n, nca, nra
 complex(4) :: x(5, 4), a(5, 4), coef(5, 4)
 character(26) :: tit1 = 'The input matrix', tit2 = 'After FFT2D', tit3 = 'After FFT2B'
 nra = 5; nca = 4; Lda = 5; Ldcoef = 5
 do n = 1, nra; do m = 1, nca              ! Формируем массив x
  x(n, m) = cmplx(float(n + 5 * m - 5), 0.0)
 end do; end do
 call wrcrn(tit1, nra, nca, x, Lda, 0)          ! Входная матрица
 call fft2d(nra, nca, x, Lda, coef, Ldcoef)     ! Прямое преобразование Фурье
 call wrcrn(tit2, nra, nca, coef, Ldcoef, 0)    ! После FFT2D
 call fft2b(nra, nca, coef, Ldcoef, a, Lda)     ! Обратное преобразование Фурье
 call wrcrn(tit3, nra, nca, a, Lda, 0)          ! После FFT2B
end program fft2bTest
```

Результат:

The input matrix

	1	2	3	4
1	(1.00, 0.00)	(6.00, 0.00)	(11.00, 0.00)	(16.00, 0.00)
2	(2.00, 0.00)	(7.00, 0.00)	(12.00, 0.00)	(17.00, 0.00)

3	(3.00, 0.00)	(8.00, 0.00)	(13.00, 0.00)	(18.00, 0.00)
4	(4.00, 0.00)	(9.00, 0.00)	(14.00, 0.00)	(19.00, 0.00)
5	(5.00, 0.00)	(10.00, 0.00)	(15.00, 0.00)	(20.00, 0.00)

After FFT2D

	1	2	3	4
1	(210.0, 0.0)	(-50.0, 50.0)	(-50.0, 0.0)	(-50.0, -50.0)
2	(-10.0, 13.8)	(0.0, 0.0)	(0.0, 0.0)	(0.0, 0.0)
3	(-10.0, 3.2)	(0.0, 0.0)	(0.0, 0.0)	(0.0, 0.0)
4	(-10.0, -3.2)	(0.0, 0.0)	(0.0, 0.0)	(0.0, 0.0)
5	(-10.0, -13.8)	(0.0, 0.0)	(0.0, 0.0)	(0.0, 0.0)

After FFT2B

	1	2	3	4
1	(20.0, 0.0)	(120.0, 0.0)	(220.0, 0.0)	(320.0, 0.0)
2	(40.0, 0.0)	(140.0, 0.0)	(240.0, 0.0)	(340.0, 0.0)
3	(60.0, 0.0)	(160.0, 0.0)	(260.0, 0.0)	(360.0, 0.0)
4	(80.0, 0.0)	(180.0, 0.0)	(280.0, 0.0)	(380.0, 0.0)
5	(100.0, 0.0)	(200.0, 0.0)	(300.0, 0.0)	(400.0, 0.0)

1.3.4. ПОДПРОГРАММА FFT3F (DFFT3F)

Вычисляет коэффициенты Фурье комплексного периодического трехмерного массива *a*. Имеет вызов

CALL FFT3F(*n*1, *n*2, *n*3, *a, Lda, mda, b, Ldb, mdb*)

Параметры подпрограммы FFT3F:

*Входные: n*1, *n*2, *n*3, *a, Lda, mda, Ldb, mdb.*

Выходной: b.

*n*1, *n*2, *n*3 - верхние границы индексов массивов *a* и *b* соответственно по первому, второму и третьему измерениям.

a - трехмерный комплексный массив формы (*Lda, mda, n*3), содержащий *n*1×*n*2×*n*3 подлежащих преобразованию данных.

Lda - ведущий размер массива *a*.

mda - средний размер массива *a*.

b - трехмерный комплексный массив формы (*Ldb, mdb, n*3), содержащий *n*1×*n*2×*n*3 коэффициентов Фурье.

Ldb - ведущий размер массива *b*.

mdb - средний размер массива *b*.

Автоматически для решения предоставляется память:

- $4(n1 + n2 + n3) + 2\max(n1, n2, n3) + 45$ байт в случае FFT3F;
- $8(n1 + n2 + n3) + 4\max(n1, n2, n3) + 90$ байт в случае DFFT3F.

Память можно выделить явно, употребив F2T3F (DF2T3F):

CALL F2T3F(*n*1, *n*2, *n*3, *a, Lda, mda, b, Ldb, mdb, wff*1, *wff*2, *wff*3, *cpy*)

Дополнительные параметры подпрограммы FFT3F:

*wff*1, *wff*2, *wff*3 (*выходные*) - вещественные векторы, имеющие соответственно размеры 4*n*1 + 15, 4*n*2 + 15 и 4*n*3 + 15; инициализируются подпрограммой FFTCI.

cpy (*рабочая область*) - вещественный массив размера 2max(*n*1, *n*2, *n*3).

Комментарии:

1. Подпрограммы FFT3F и FFT3B наиболее эффективны, когда *n*1, *n*2 и *n*3 являются произведением небольших простых чисел. В этом случае время вычислений пропорционально $NML \log NML$, где $L = n3$, $M = n2$ и $N = n1$.
2. Массивы *a* и *b* могут совпадать.
3. Если FFT3F или FFT3B неоднократно вызываются с одинаковыми значениями *n*1, *n*2 и *n*3, то прежде следует использовать инициализирующую подпрограмму FFTCI для заполнения векторов *wff*1, *wff*2 и *wff*3, а затем нужное число раз вызвать подпрограмму второго уровня F2T3F или F2T3B. Такой подход более эффективен, чем многократный вызов FFT3F или FFT3B.

Описание:

Для заданного комплексного периодического трехмерного $N \times M \times L$-массива *a* подпрограмма FFT3F возвращает комплексный массив коэффициентов Фурье *b*, вычисляемых по формуле

$$b_{jkl} = \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{l=1}^{L} a_{nml} e^{-2\pi i(j-1)(n-1)/N} e^{-2\pi i(k-1)(m-1)/M} e^{-2\pi i(k-1)(l-1)/L}$$

Коэффициенты нормализуются таким образом, что евклидова норма *d* вектора оказывается равной $\sqrt{NMLd}$.

Пример. Вычисляется преобразование Фурье 2×3×4-массива, имеющего элементы

$$a_{nml} = e^{2\pi i(n-1)1/2} e^{2\pi i(m-1)2/3} e^{2\pi i(m-1)2/4}, \quad 1 \le n \le 2,\ 1 \le m \le 3 \text{ и } 1 \le l \le 4.$$

Результат имеет нули во всех позициях, кроме (2, 3, 4).

```
program fft3fTest
 integer(4), parameter :: Lda = 2, Ldb = 2, mda = 3, mdb = 3, nda = 4, ndb = 4
 integer(4) :: i, j, k, L, m, n, n1, n2, n3
 real(4) :: pi
 complex(4) :: a(Lda, mda, nda), b(Ldb, mdb, ndb), c, h
```

```
n1 = 2; n2 = 3; n3 = 4; pi = const('pi'); c = cmplx(0.0, 2.0 * pi)
do n = 1, 2; do m = 1, 3; do L = 1, 4    ! Формируем массив a
 h = c * (n - 1) * 1 / 2 + c * (m - 1) * 2 / 3 + c * (L - 1) * 2 / 4
 a(n, m, L) = cexp(h)
end do; end do; end do
call fft3f(n1, n2, n3, a, Lda, mda, b, Ldb, mdb)
write(*, "(1x, 'The input for fft3f is')")
do i = 1, 2;  write(*,"(/, ' Face no. ', i1)") i; do j = 1, 3
  write(*,"(1x, 4('(', f6.2, ', ', f6.2, ')', 3x))") (a(i, j, k), k = 1, 4)
end do; end do
write(*, "(/, 1x, 'The results from fft3f are')")
do i = 1, 2; write(*, "(/, ' Face no. ', i1)") i; do j = 1, 3
  write(*, "(1x, 4('(', f6.2, ', ', f6.2, ')', 3x))") (b(i, j, k), k = 1, 4)
end do; end do
end program fft3fTest
```

Результат:

```
The input for FFT3F is
Face no. 1
    ( 1.00, 0.00)    ( -1.00, 0.00)    ( 1.00, 0.00)    ( -1.00, 0.00)
   ( -0.50, -0.87)   ( 0.50, 0.87)     ( -0.50, -0.87)  ( 0.50, 0.87)
    ( -0.50, 0.87)   ( 0.50, -0.87)    ( -0.50, 0.87)   ( 0.50, -0.87)
Face no. 2
    ( -1.00, 0.00)   ( 1.00, 0.00)     ( -1.00, 0.00)   ( 1.00, 0.00)
    ( 0.50, 0.87)    ( -0.50, -0.87)   ( 0.50, 0.87)    ( -0.50, -0.87)
    ( 0.50, -0.87)   ( -0.50, 0.87)    ( 0.50, -0.87)   ( -0.50, 0.87)

The results from FFT3F are
Face no. 1
    ( 0.00, 0.00)    ( 0.00, 0.00)     ( 0.00, 0.00)    ( 0.00, 0.00)
    ( 0.00, 0.00)    ( 0.00, 0.00)     ( 0.00, 0.00)    ( 0.00, 0.00)
    ( 0.00, 0.00)    ( 0.00, 0.00)     ( 0.00, 0.00)    ( 0.00, 0.00)
Face no. 2
    ( 0.00, 0.00)    ( 0.00, 0.00)     ( 0.00, 0.00)    ( 0.00, 0.00)
    ( 0.00, 0.00)    ( 0.00, 0.00)     ( 0.00, 0.00)    ( 0.00, 0.00)
    ( 0.00, 0.00)    ( 0.00, 0.00)     ( 24.00, 0.00)   ( 0.00, 0.00)
```

1.3.5. ПОДПРОГРАММА FFT3B (DFFT3B)

Вычисляет обратное преобразование Фурье комплексного периодического трехмерного массива *a*. Имеет вызов

CALL FFT3B(*n1, n2, n3, a, Lda, mda, b, Ldb, mdb*)

Параметры подпрограммы FFT3B:
Входные: n1, n2, n3, a, Lda, mda, Ldb, mdb.

Выходной: b.

Описание параметров подпрограммы FFT3B см. в предшествующем разделе.

Автоматически для решения предоставляется память:

- $4(n1 + n2 + n3) + 2\max(n1, n2, n3) + 45$ байт в случае F2T3B;
- $8(n1 + n2 + n3) + 4\max(n1, n2, n3) + 90$ байт в случае DF2T3B.

Память можно выделить явно, употребив F2T3B (DF2T3B):

CALL F2T3B(*n1, n2, n3, a, Lda, mda, b, Ldb, mdb, wff1, wff2, wff3, cpy*)

Описание *wff1, wff2, wff3, cpy - дополнительных параметров подпрограммы* F2T3B - см. в предшествующем разделе.

Комментарии см. в предшествующем разделе.

Описание:

Для заданного комплексного $N \times M \times L$-массива коэффициентов Фурье a подпрограмма FFT3B возвращает комплексный массив b с обратным преобразованием Фурье, элементы которого вычисляются по формуле

$$b_{jkl} = \sum_{n=1}^{N} \sum_{m=1}^{M} \sum_{l=1}^{L} a_{nml} e^{2\pi i(j-1)(n-1)/N} e^{2\pi i(k-1)(m-1)/M} e^{2\pi i(k-1)(l-1)/L}.$$

Коэффициенты нормализуются таким образом, что евклидова норма d вектора оказывается равной $\sqrt{NMLd}$.

Пример. Первоначально подпрограмма FFT3F вычисляет прямое преобразование Фурье 2×3×4-массива, имеющего элементы

$$x_{nml} = n + 2(m - 1) + 2 * 3(L - 1),\ 1 \le n \le 2,\ 1 \le m \le 3 \text{ и } 1 \le L \le 4.$$

Затем подпрограмма FFT3B находит обратное преобразование. Заметьте, что результат $a = 2 * 3 * 4 * x = 24 * x$.

```
program fft3bTest
 use dfimsl
 integer(4), parameter :: Lda = 2, Ldb = 2, mda = 3, mdb = 3, nda = 4, ndb = 4
 integer(4) :: i, j, k, L, m, n, n1, n2, n3
 complex(4) :: a(Lda, mda, nda), b(Ldb, mdb, ndb), x(Ldb, mdb, ndb)
 n1 = 2; n2 = 3; n3 = 4
 do n = 1, 2; do m = 1, 3; do L = 1, 4    ! Формируем массив x
  x(n, m, L) = n + 2 * (m - 1) + 2 * 3 * (L - 1)
 end do; end do; end do
 call fft3f(n1, n2, n3, x, Lda, mda, a, Ldb, mdb)
 call fft3b(n1, n2, n3, a, Lda, mda, b, Ldb, mdb)
 write(*, "(1x, 'The input for fft3f is')")
```

```
do i = 1, 2;  write(*,"(/, ' Face no. ', i1)") i; do j = 1, 3
  write(*,"(1x, 4('(', f6.2, ', ', f6.2, ')', 3x))") (x(i, j, k), k = 1, 4)
end do; end do
write(*, "(/, 1x, 'The results from fft3f are')")
do i = 1, 2; write(*, "(/, ' Face no. ', i1)") i; do j = 1, 3
  write(*, "(1x, 4('(', f6.2, ', ', f6.2, ')', 3x))") (a(i, j, k), k = 1, 4)
end do; end do
write(*, "(1x, 'The unnormalized inverse is')")
do i = 1, 2; write(*, "(/, ' Face no. ', i1)") i; do j = 1, 3
  write(*, "(1x, 4('(', f6.2, ', ', f6.2, ')', 3x))") (b(i, j, k), k = 1, 4)
end do; end do
end program fft3bTest
```

Результат:

```
The input for FFT3F is
Face no. 1
     ( 1.00, 0.00)     ( 7.00, 0.00)     ( 13.00, 0.00)     ( 19.00, 0.00)
     ( 3.00, 0.00)     ( 9.00, 0.00)     ( 15.00, 0.00)     ( 21.00, 0.00)
     ( 5.00, 0.00)     ( 11.00, 0.00)    ( 17.00, 0.00)     ( 23.00, 0.00)
Face no. 2
     ( 2.00, 0.00)     ( 8.00, 0.00)     ( 14.00, 0.00)     ( 20.00, 0.00)
     ( 4.00, 0.00)     ( 10.00, 0.00)    ( 16.00, 0.00)     ( 22.00, 0.00)
     ( 6.00, 0.00)     ( 12.00, 0.00)    ( 18.00, 0.00)     ( 24.00, 0.00)

The results from FFT3F are
Face no. 1
     (300.00, 0.00)    (-72.00, 72.00)   (-72.00, 0.00)     (-72.00,-72.00)
     (-24.00, 13.86)   ( 0.00, 0.00)     ( 0.00, 0.00)      ( 0.00, 0.00)
     (-24.00,-13.86)   ( 0.00, 0.00)     ( 0.00, 0.00)      ( 0.00, 0.00)
Face no. 2
     (-12.00, 0.00)    ( 0.00, 0.00)     ( 0.00, 0.00)      ( 0.00, 0.00)
     ( 0.00, 0.00)     ( 0.00, 0.00)     ( 0.00, 0.00)      ( 0.00, 0.00)
     ( 0.00, 0.00)     ( 0.00, 0.00)     ( 0.00, 0.00)      ( 0.00, 0.00)

The unnormalized inverse is
Face no. 1
     ( 24.00, 0.00)    (168.00, 0.00)    (312.00, 0.00)     (456.00, 0.00)
     ( 72.00, 0.00)    (216.00, 0.00)    (360.00, 0.00)     (504.00, 0.00)
     (120.00, 0.00)    (264.00, 0.00)    (408.00, 0.00)     (552.00, 0.00)
Face no. 2
     ( 48.00, 0.00)    (192.00, 0.00)    (336.00, 0.00)     (480.00, 0.00)
     ( 96.00, 0.00)    (240.00, 0.00)    (384.00, 0.00)     (528.00, 0.00)
     (144.00, 0.00)    (288.00, 0.00)    (432.00, 0.00)     (576.00, 0.00)
```

1.4. СВЕРТКА И КОРРЕЛЯЦИЯ ДВУХ ВЕКТОРОВ

1.4.1. ПЕРЕЧЕНЬ ПОДПРОГРАММ

Приведен в табл. 1.5.

Таблица 1.5. Подпрограммы, вычисляющие свертку и корреляцию векторов

Подпрограммы	*Назначение*
Свертка двух векторов	
VCONR, RCONV	Вычисляют свертку двух вещественных векторов
VCONC, CCONV	Вычисляют свертку двух комплексных векторов
Корреляция двух векторов	
RCORL	Вычисляет корреляцию двух вещественных векторов
CCORL	Вычисляет корреляцию двух комплексных векторов

1.4.2. ПОДПРОГРАММЫ VCONR (DVCONR) И VCONC (DVCONC)

Вычисляют свертки вещественных и комплексных векторов. Имеют вызовы

CALL VCONR(*nx*, *x*, *ny*, *y*, *nz*, *z*) ! Свертка вещественных векторов

CALL VCONC(*nx*, *x*, *ny*, *y*, *nz*, *z*) ! Свертка комплексных векторов

Параметры подпрограмм VCONR и VCONC:

Входные: nx, x, ny, y, nz.

Выходной: z.

nx, *ny*, *nz* - соответственно длины векторов *x*, *y* и *z*, причем должно выполняться неравенство $nz \geq nx + ny - 1$.

x, *y* - входные векторы (вектор *y* иногда называют *фильтром*).

z - вектор, содержащий свертку векторов *x* и *y*.

Описание:

Пусть $n_x = nx$, $n_y = ny$, $n_z = nz$. Элементы вектора *z* вычисляются по формуле

$$z_j = \sum_{k=1}^{n_x} x_{j-k+1} y_k \ , j = 1, 2, \ldots, n_z,$$

где $n_z = n_x + n_y - 1$. Если индекс $j - k + 1$ находится вне диапазона 1, 2, ..., n_z, то x_{j-k+1} берется равным нулю.

Для вычисления свертки используется быстрое преобразование Фурье. Пусть комплексные векторы *u* и *v* размера $n_z = n_x + n_y - 1$ определены так:

$u = (x_1, x_2, \ldots, x_{nx}, 0, \ldots, 0)$ и $v = (y_1, y_2, \ldots, y_{ny}, 0, \ldots, 0)$.

Тогда в соответствии с теоремой Фурье о свертке $\hat{w}_i = \hat{u}_i \cdot \hat{v}_i$, $i = 1, 2, \ldots, n_z$, где $\hat{u}_i$ и $\hat{v}_i$ - соответственно преобразования Фурье векторов u и v вычисляются процедурой FFTCF библиотеки IMSL. Процедура FFTCB, выполняющая обратное преобразование Фурье, находит по $\hat{w}$ вектор w. В случае поиска свертки вещественных векторов u и v вектор z определяется как вещественная часть w, в случае комплексных u и v вектор $z = w$.

Пример для VCONR. Вычисляется свертка вектора x размера 8 и вектора y размера 3. Результат помещается в вектор z размера 8 + 3 - 1 = 10.

```
program vconrTest
 use dfimsl
 integer(4), parameter :: nx = 8, ny = 3, nz = nx + ny - 1
 real(4) :: x(nx), y(ny), z(nz)
 real(4) :: z2(2 * nz), zhat(2 * nz)        ! Для подпрограммы RCONV
 x = (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0 /); y = (/ 0.0, 0.0, 1.0 /)
 call vconr(nx, x, ny, y, nz, z)            ! Вычисление свертки z = x (*) y
 call wrrrn('z = x (*) y', 1, nz, z, 1, 0)  ! Вывод результата
 ! Тот же результат даст RCONV, примененная для непериодических данных
 call rconv(0, nx, x, ny, y, 1, 2 * nz, z2, zhat)
 call wrrrn('z = x (*) y', 1, nz, z, 1, 0)  ! Вывод результата
end program vconrTest
```

Результат:

```
                          z = x (*) y
     1       2       3       4       5       6       7       8       9      10
 0.000   0.000   1.000   2.000   3.000   4.000   5.000   6.000   7.000   8.000
```

Пример для VCONC. Вычисляется свертка вектора x размера 4 и вектора y размера 3. Результат помещается в вектор z размера 4 + 3 - 1 = 6.

```
program vconcTest
 use dfimsl
 integer(4), parameter :: nx = 4, ny = 3, nz = nx + ny - 1
 complex(4) :: x(nx), y(ny), z(nz)
 complex(4) :: z2(2 * nz), zhat(2 * nz)          ! Для подпрограммы CCONV
 x = (/ (1.0,2.0), (3.0,4.0), (5.0,6.0), (7.0,8.0) /); y = (/ (0.0,0.0), (0.0,0.0), (1.0,1.0) /)
 call vconc(nx, x, ny, y, nz, z)                 ! Вычисление свертки z = x (*) y
 call wrcrn('z = x (*) y', 1, nz, z, 1, 0)       ! Вывод результата
 ! Тот же результат даст CCONV, примененная для непериодических данных
 call cconv(0, nx, x, ny, y, 1, 2 * nz, z2, zhat)
 call wrcrn('z = x (*) y', 1, nz, z, 1, 0)       ! Вывод результата
end program vconcTest
```

Результат:

```
                                z = x (*) y
      1             2             3             4             5             6
( 0.00,  0.00) ( 0.00, 0.00) (-1.00, 3.00) (-1.00, 7.00) (-1.00, 11.00) (-1.00, 15.00)
```

1.4.3. ПОДПРОГРАММА RCONV (DRCONV)

Вычисляет свертку двух вещественных векторов. Имеет вызов

CALL RCONV(*ido*, *nx*, *x*, *ny*, *y*, *ipad*, *nz*, *z*, *zhat*)

Параметры подпрограммы RCONV:
Входные: ido, *nx*, *x*, *ny*, *y*, *ipad*.
Входной/выходной: *nz*.
Выходные: z, *zhat*.
Тип параметров *x*, *y*, *z* и *zhat* - REAL(4), остальных - INTEGER(4).

ido - параметр, задающий способ употребления RCONV. Если *ido* = 0, то RCONV вызывается 1 раз. В противном случае RCONV вызывается неоднократно с одними и теми же параметрами *nx*, *ny* и *ipad*. Причем при многократном вызове RCONV *ido* должен быть равен:

- 1, если выполняется первый вызов;
- 2, если выполняются промежуточные вызовы;
- 3, если выполняется завершающий вызов.

nx, ny - соответственно размеры векторов *x* и *y*.

x, y - векторы, свертка которых вычисляется.

ipad - параметр, равный нулю при периодических данных и равный единице, если данные непериодические.

nz - размер векторов *z* и *zhat*. На входе, когда *ipad* = 0, $nz \geq nx + ny - 1$; когда *ipad* = 1, $nz \geq 2^{\alpha}3^{\beta}5^{\gamma}$, где $2^{\alpha}3^{\beta}5^{\gamma}$ - наименьшее число, большее или равное $nx + ny - 1$; α, β и γ - неотрицательные целые числа. На выходе *nz* содержит значение, использованное RCONV.

z - вектор, содержащий свертку векторов *x* и *y*.

zhat - вектор, содержащий дискретное преобразование Фурье вектора *z*.

Комментарий. Если размер вектора *z* меньше необходимой величины, то возникнет завершающая ошибка, сообщающая приемлемый размер *nz*. Например:

```
*** FATAL ERROR 1 from RCONV. The length of the vector Z must be at
***        least 10 while NZ = 8 is given, where NZ is based on NX = 8 and
***        NY = 3.
Stop - Program terminated.
```

Описание:

Пусть n_x - размер x, а n_y - размер y. Если вычисляется циклическая свертка (*ipad* = 0), то $nz \geq nx + ny - 1$ и более короткий вектор расширяется до размера более длинного за счет добавления завершающих нулей. Затем вычисляются элементы вектора z:

$$z_i = \sum_{j=1}^{n_z} x_{i-j+1} y_j \ , i = 1, 2, \ldots, n_z,$$

где индекс вектора x - это MODULO(j, n_z).

Метод вычисления z_i основан на том, что дискретное преобразование Фурье циклической свертки двух наборов чисел (векторов) равно произведению дискретных преобразований Фурье этих векторов:

$$\hat{z}_n = \hat{x}_n \hat{y}_n \ ,$$

где

$$\hat{z}_n = \sum_{m=1}^{n_z} z_m e^{-2\pi i(m-1)(n-1)/n_z} \ .$$

После вычисления произведения дискретных преобразований Фурье векторов x и y выполняется обратное преобразование Фурье этого произведения и таким образом находится искомая свертка. Важно, чтобы величина n_z равнялась произведению небольших простых чисел. Тогда вычислительная сложность будет пропорциональна $n_z \log n_z$.

Заметим, что в случае вещественных векторов выполняются не комплексные, а вещественные преобразования Фурье, что приводит к существенному сокращению вычислительных затрат и используемой памяти.

Пример. Вычисляются как периодические, так и непериодические свертки. Программа может быть использована для цифровой фильтрации зашумленных данных. Усредняющий оператор крайне прост; он использует 5 последовательных точек набора. В периодическом случае восстанавливается зашумленная синусоидальная функция путем усреднения пяти рядом расположенных значений. В непериодическом случае восстанавливаются значения зашумленной экспоненциальной функции. Большая ошибка для последнего значения объясняется тем, что для него в процессе вычисления свертки усредняются удлиняющие вектор элементы, а не значения функции. Вывод данных ограничивается 10 точками.

```
program filter
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
```

```
use text_transfer                    ! Для вывода русского текста
integer(4), parameter :: nfltr = 5, ny = 100
integer(4) :: i, k, nz
real(4) :: fltr(nfltr), total1, total2, twopi, y(ny), z(2 * (nfltr + ny - 1)),          &
    zhat(2*(nfltr + ny - 1))
real(4), external :: f1, f2
character(30) :: fmt3, fmt4; character(100) :: fmt5, fmt6, fmt8, fmt9, st
! Формируем строки для задания формата вывода
! Текст подпрограммы ru_doswin см. в [5, прил. 1]
fmt3 = ru_doswin('(" Периодический случай")', .false.)
fmt4 = ru_doswin('(" Непериодический случай")', .false.)
st = ru_doswin('"Начальная ошибка', 5x, 'Ошибка после фильтрации"', .false.)
fmt5 = trim("(8x, 't', 9x, 'sin(t)', 6x, ") // trim(st) // ")"
fmt6 = trim("(8x, 't', 9x, 'exp(t)', 6x, ") // trim(st) // ")"
fmt8 = ru_doswin('Абсолютное значение средней ошибки до фильтрации:', .false.)
fmt8 = '(" " // trim(fmt8) // "", f11.5)'
fmt9 = ru_doswin('Абсолютное значение средней ошибки после фильтрации:', .false.)
fmt9 = '(" " // trim(fmt9) // "", f11.5)'
call rnset(1234579)                  ! Инициализация датчика случайных чисел
twopi = 2.0 * const('pi')            ! 2π
fltr = 0.2                           ! Задаем фильтр (вектор x)
! Случай периодических данных
call fcon(f1, twopi, 0)              ! Создаем вектор y и вычисляем свертку
print fmt3; print fmt5               ! Вывод результата
total1 = 0.0; total2 = 0.0
do i = 1, ny                         ! Вычисляем ошибки
 k = i + 2                           ! Смещение для вектора z
 if(i >= ny - 1) k = k - ny
 call sumerr(f1, k, twopi)           ! Суммируем ошибки
end do
print fmt8, total1 / float(ny)       ! Вывод средних ошибок
print fmt9, total2 / float(ny)
! Случай непериодических данных
call fcon(f2, 1.0, 1)                ! Создаем вектор y и вычисляем свертку
print fmt4; print fmt6               ! Вывод результата
total1 = 0.0; total2 = 0.0
do i = 1, ny                         ! Вычисляем ошибки
 call sumerr(f2, i + 2, 1.0)         ! Суммируем ошибки
end do
print fmt8, total1 / float(ny)       ! Вывод средних ошибок
print fmt9, total2 / float(ny)

contains

 subroutine fcon(f, p, ipad)         ! Создаем вектор y и вычисляем свертку
  real(4) :: f, p, t                 ! (осуществляем фильтрацию данных)
```

```
 integer(4) :: ipad
 ! Задаем вектор y - зашумленную синусоиду для случая периодических данных
 ! и зашумленную экспоненту (при втором вызове)
 ! для случая непериодических данных
 do i = 1, ny                          ! RNUNF - возвращает псевдослучайное
  t = p * float(i - 1) / float(ny - 1)  ! число из диапазона [0, 1)
  y(i) = f(t) + 0.5 * rnunf( ) - 0.25
 end do
 nz = 2 * (nfltr + ny - 1)
 call rconv(0, nfltr, fltr, ny, y, ipad, nz, z, zhat)
end subroutine fcon

subroutine sumerr(f, k, p)             ! Производит суммирование ошибок
 integer(4) :: k                       ! (осуществляем фильтрацию данных)
 real(4) :: f, p, t, fltrer, origer
 t = p * float(i - 1) / float(ny - 1)
 origer = abs(y(i) - f(t))             ! Ошибка до фильтрации
 fltrer = abs(z(k) - f(t))             ! Ошибка после фильтрации
 ! Выводим точки 1, 12, 23, ..., 100
 if(mod(i, 11) == 1) print 7, t, f(t), origer, fltrer
 7 format(1x, f10.4, f13.4, 2f18.4)
 total1 = total1 + origer              ! Накапливаем ошибки до
 total2 = total2 + fltrer              ! и после фильтрации
end subroutine sumerr
end program filter

function f1(t)                         ! Задаем исследуемые функции
 real(4) :: f1, t
 f1 = sin(t)
end function f1

function f2(t)                         ! Задаем исследуемые функции
 real(4) :: f2, t
 f2 = exp(t)
end function f2
```

Результат:

	Периодический случай		
t	sin(t)	Начальная ошибка	Ошибка после фильтрации
0.0000	0.0000	0.0811	0.0587
0.6981	0.6428	0.0226	0.0781
1.3963	0.9848	0.1526	0.0529
2.0944	0.8660	0.0959	0.0125
2.7925	0.3420	0.1747	0.0292
3.4907	-0.3420	0.1035	0.0238

4.1888	-0.8660	0.0402	0.0595
4.8869	-0.9848	0.0673	0.0798
5.5851	-0.6428	0.1044	0.0074
6.2832	0.0000	0.0154	0.0018

Абсолютное значение средней ошибки до фильтрации: 0.12481
Абсолютное значение средней ошибки после фильтрации: 0.04778

	Непериодический случай		
t	exp(t)	Начальная ошибка	Ошибка после фильтрации
0.0000	1.0000	0.1476	0.3915
0.1111	1.1175	0.0537	0.0326
0.2222	1.2488	0.1278	0.0932
0.3333	1.3956	0.1136	0.0987
0.4444	1.5596	0.1617	0.0964
0.5556	1.7429	0.0071	0.0662
0.6667	1.9477	0.1248	0.0713
0.7778	2.1766	0.1556	0.0158
0.8889	2.4324	0.1529	0.0696
1.0000	2.7183	0.2124	1.0562

Абсолютное значение средней ошибки до фильтрации: 0.12538
Абсолютное значение средней ошибки после фильтрации: 0.07764

1.4.4. ПОДПРОГРАММА CCONV (DCCONV)

Вычисляет свертку двух комплексных векторов. Имеет вызов

CALL CCONV(*ido, nx, x, ny, y, ipad, nz, z, zhat*)

Параметры подпрограммы CCONV:
Входные: ido, nx, x, ny, y, ipad.
Входной/выходной: nz.
Выходные: z, zhat.

Тип параметров *x*, *y*, *z* и *zhat* - COMPLEX(4), остальных - INTEGER(4). Описание параметров, завершающей ошибки и алгоритма см. в предшествующем разделе.

Пример. Решаются задачи, аналогичные рассмотренным в вышеприведенном примере. Периодические данные задаются комплексной функцией $f_1(x) = \cos(x) + i\sin(x)$, непериодические - комплексной функцией $f_2(x) = \exp(xf_1(x))$.

```
program filter2
 use dfimsl
```

```
! Текст модуля text_transfer см. в [5, прил. 1]
use text_transfer                    ! Для вывода русского текста
integer(4), parameter :: nfltr = 5, ny = 100
integer(4) :: i, k, nz
real(4) :: total1, total2, twopi
complex(4), external :: f1, f2
complex(4) :: fltr(nfltr), y(ny), z(2 * (nfltr + ny - 1)), zhat(2 * (nfltr + ny - 1))
character(30) :: fmt3, fmt4; character(100) :: fmt5, fmt6, fmt8, fmt9, st
! Формируем строки для задания форматов вывода (см. программу filter)
...  ! Формирование fmt3, fmt4, fmt5, fmt6, fmt8 и fmt9
call rnset(1234579)                  ! Инициализация датчика случайных чисел
twopi = 2.0 * const('pi')            ! 2π
fltr = (0.2, 0.0)                    ! Задаем фильтр (вектор x)
! Случай периодических данных
call fcon(f1, twopi, 0)              ! Создаем вектор y и вычисляем свертку
print fmt3; print fmt5               ! Вывод результата
total1 = 0.0; total2 = 0.0
do i = 1, ny                         ! Вычисляем ошибки
 k = i + 2                           ! Смещение для вектора z
 if(i >= ny - 1) k = k - ny
 call sumerr(f1, k, twopi)           ! Суммируем ошибки
end do
print fmt8, total1 / float(ny)       ! Вывод средних ошибок
print fmt9, total2 / float(ny)
! Случай непериодических данных
call fcon(f2, 1.0, 1)                ! Создаем вектор y и вычисляем свертку
print fmt4; print fmt6               ! Вывод результата
total1 = 0.0; total2 = 0.0
do i = 1, ny                         ! Вычисляем ошибки
 call sumerr(f2, i + 2, 1.0)         ! Суммируем ошибки
end do
print fmt8, total1 / float(ny)       ! Вывод средних ошибок
print fmt9, total2 / float(ny)

contains

subroutine fcon(f, p, ipad)          ! Создаем вектор y и вычисляем свертку
 complex(4) :: f                     ! (осуществляем фильтрацию данных)
 real(4) :: p, t
 integer(4) :: ipad
 ! Задаем вектор y - зашумленную функцию f1 для случая периодических данных
 ! и зашумленную функцию f2 (при втором вызове)
 ! для случая непериодических данных
 do i = 1, ny                        ! RNUNF - возвращает псевдослучайное число
  t = p * float(i - 1) / float(ny - 1)   ! из диапазона [0, 1)
  y(i) = f(t) + cmplx(0.5 * rnunf( ) - 0.25, 0.5 * rnunf( ) - 0.25)
```

```
 end do
 nz = 2 * (nfltr + ny - 1)
 call cconv(0, nfltr, fltr, ny, y, ipad, nz, z, zhat)
end subroutine fcon

subroutine sumerr(f, k, p)                ! Производит суммирование ошибок
 integer(4) :: k                          ! (осуществляем фильтрацию данных)
 complex(4) :: f
 real(4) :: p, t, fltrer, origer
 t = p * float(i - 1) / float(ny - 1)
 origer = abs(y(i) - f(t))                ! Ошибка до фильтрации
 fltrer = abs(z(k) - f(t))                ! Ошибка после фильтрации
 ! Выводим точки 1, 12, 23, ..., 100
 if(mod(i, 11) == 1) print 7, t, f(t), origer, fltrer
 7 format(1x, f10.4, 5x, '(', f7.4, ',', f8.4, ' )', 5x, f8.4, 10x, f8.4)
 total1 = total1 + origer                 ! Накапливаем ошибки до и после фильтрации
 total2 = total2 + fltrer
end subroutine sumerr
end program filter2

complex(4) function f1(t)                 ! Задаем исследуемые функции
 real(4) :: t
 f1 = cmplx(cos(t), sin(t))
end function f1

complex(4) function f2(t)                 ! Задаем исследуемые функции
 real(4) :: t
 f2 = exp(t) * cmplx(cos(t), sin(t))
end function f2
```

Результат:

Периодический случай

t	f1(t)	Начальная ошибка	Ошибка после фильтрации
0.0000	(1.0000, 0.0000)	0.1666	0.0773
0.6981	(0.7660, 0.0000)	0.1685	0.1399
1.3963	(0.1736, 0.0000)	0.1756	0.0368
2.0944	(-.5000, 0.0000)	0.2171	0.0142
2.7925	(-.9397, 0.0000)	0.1147	0.0200
3.4907	(-.9397, 0.0000)	0.0998	0.0331
4.1888	(-.5000, 0.0000)	0.1137	0.0586
4.8869	(0.1736, 0.0000)	0.2217	0.0843
5.5851	(0.7660, 0.0000)	0.1831	0.0744
6.2832	(1.0000, 0.0000)	0.3234	0.0893

Абсолютное значение средней ошибки до фильтрации: 0.19315

Абсолютное значение средней ошибки после фильтрации: 0.08296

Непериодический случай

t	f2(t)	Начальная ошибка	Ошибка после фильтрации
0.0000	(1.0000, 0.0000)	0.0783	0.4336
0.1111	(1.1106, 0.1239)	0.2434	0.0477
0.2222	(1.2181, 0.2752)	0.1819	0.0584
0.3333	(1.3188, 0.4566)	0.0703	0.1267
0.4444	(1.4081, 0.6706)	0.1458	0.0868
0.5556	(1.4808, 0.9192)	0.1946	0.0930
0.6667	(1.5307, 1.2044)	0.1458	0.0734
0.7778	(1.5508, 1.5273)	0.1815	0.0690
0.8889	(1.5331, 1.8885)	0.0805	0.0193
1.0000	(1.4687, 2.2874)	0.2396	1.1708

Абсолютное значение средней ошибки до фильтрации: 0.18549
Абсолютное значение средней ошибки после фильтрации: 0.09636

1.4.5. ПОДПРОГРАММА RCORL (DRCORL)

Вычисляет корреляцию двух вещественных векторов. Имеет вызов

CALL RCORL(*ido, n, x, y, ipad, nz, z, zhat*)

Параметры подпрограммы RCORL:
Входные: ido, n, x, y, ipad.
Входной/выходной: nz.
Выходные: z, zhat.
Тип параметров *x, y, z* и *zhat* - REAL(4), остальных - INTEGER(4).

ido - параметр, задающий способ употребления RCORL. Если *ido* = 0, то RCORL вызывается 1 раз. В противном случае RCORL вызывается неоднократно с одними и теми же параметрами *n* и *ipad*. Причем при многократном вызове RCORL *ido* должен быть равен:

- 1, если выполняется первый вызов;
- 2, если выполняются промежуточные вызовы;
- 3, если выполняется завершающий вызов.

n - размер векторов *x* и *y*.
x, y - вещественные векторы размера *n*.
ipad - принимает следующие значения:

- 0, если данные периодические и векторы *x* и *y* различаются;
- 1, если данные непериодические и векторы *x* и *y* различаются;
- 2, если данные периодические и векторы *x* и *y* совпадают;

- 3, если данные непериодические и векторы x и y совпадают.

nz - размер вектора *z*. На входе, когда *ipad* = 0 или *ipad* = 2, *nz* должен быть не меньше (2*n* - 1); когда *ipad* = 1 или *ipad* = 3, *nz* должен быть больше или равен наименьшему целому, которое больше или равно (2*n* - 1) и имеет вид $2^{\alpha}3^{\beta}5^{\gamma}$, где α, β и γ - неотрицательные целые числа. На выходе *nz* равен использованному подпрограммой RCORL значению.

z - вещественный вектор размера *nz*, содержащий корреляцию *x* и *y*.

zhat - вещественный вектор размера *nz*, содержащий дискретное преобразование Фурье вектора *z*.

Комментарий. При работе с RCORL может возникнуть информационная ошибка типа 4 с кодом 1, означающая, что длина вектора *z* недостаточна для размещения результата. Приемлемая длина возвращается в *nz*.

Описание:

После определения *nz* по изложенным выше правилам входные векторы расширяются за счет добавления завершающих нулей. Затем вычисляются элементы вектора *z:*

$$z_i = \sum_{j=1}^{n_z} x_{i+j-1}\bar{y}_j \ , i = 1, 2, \ldots, n_z,$$

где индекс вектора *x* - это MODULO(*j*, n_z). Это означает, что z_{n_z-k} содержит корреляцию *x*(* - k - 1) и *y*, *k* = 0, 1, ..., n_z/2. Таким образом, если *x*(*k* - 1) = *y*(*k*) для всех *k*, z_{n_z} будет наибольшим компонентом вектора *z*.

Механизм вычисления z_i основан на том, что комплексное дискретное преобразование Фурье сводит процесс вычисления корреляции к умножению: вычисляются дискретное преобразование Фурье вектора *x* и комплексно-сопряженное дискретное преобразование Фурье вектора *y*, результаты поэлементно перемножаются и затем находится обратное преобразование Фурье произведения. Важно, чтобы значение n_z было равно произведению небольших простых чисел. Тогда вычислительная сложность будет пропорциональна $n_z \log n_z$.

Заметим, что в случае вещественных векторов выполняются не комплексные, а вещественные преобразования Фурье, что приводит к сокращению вычислительных затрат в 6 раз.

Пример. Вычисляются как периодические, так и непериодические корреляции двух различных сигналов *x* и *y*. Первоначально генерируются 100 равномерно размещенных значений на отрезке [0, 2π] функции $f_1(x) = \sin(x)$. При этом *x* и *y* определяются следующим образом:

$$x_i = f_1\left(2\pi\frac{i-1}{n-1}\right), i = 1, \ldots, n,$$

$$y_i = f_1\left(2\pi\frac{i-1}{n-1}+\frac{\pi}{2}\right), i = 1, \ldots, n.$$

Заметим, что максимальное значение вектора z (корреляции x и y) наблюдается при $i = 26$.

В непериодическом случае берется функция $f_2(x) = \sin(x^2)$ и два входных сигнала формируются на отрезке $[0, 4\pi]$, причем

$$x_i = f_2\left(4\pi\frac{i-1}{n-1}\right), i = 1, \ldots, n,$$

$$y_i = f_2\left(4\pi\frac{i-1}{n-1}+\pi\right), i = 1, \ldots, n.$$

Максимум вновь наблюдается при $i = 26$.

```
program rcorlTest
 use dfimsl
 integer(4), parameter :: n = 100
 integer(4) :: i, k(1), nz
 real(4) :: a, f1, f2, pi, x(n), xnorm, y(n), ynorm, z(4 * n), zhat(4 * n)
 f1(a) = sin(a); f2(a) = sin(a * a)
 pi = const('pi')                          ! Число π
 do i = 1, n                               ! Формируем данные для периодического случая
  x(i) = f1(2.0 * pi * float(i - 1) / float(n - 1))
  y(i) = f1(2.0 * pi * float(i - 1) / float(n - 1) + pi / 2.0)
 end do
 nz = 2 * n
 call rcorl(0, n, x, y, 0, nz, z, zhat)    ! Периодический случай
 ! Нормализация вектора z
 xnorm = snrm2(n, x, 1); ynorm = snrm2(n, y, 1); z = z / (xnorm * ynorm)
 k = maxloc(z)                             ! Индекс максимального элемента вектора z
 write(*, "(' Case #1: periodic data')")   ! Вывод результата для периодического случая
 write(*, "(1x, 28('-'))"); write(*, 99998) k; write(*, 99999) k, z(k(1))
 do i = 1, n                               ! Задаем вектор для непериодического случая
  x(i) = f2(4.0 * pi * float(i - 1) / float(n - 1))
  y(i) = f2(4.0 * pi * float(i - 1) / float(n - 1) + pi)
 end do
 nz = 4*n
 call rcorl(0, n, x, y, 1, nz, z, zhat)    ! Непериодический случай
 ! Нормализация вектора z
```

```
xnorm = snrm2(n, x, 1); ynorm = snrm2(n, y, 1); z = z / (xnorm * ynorm)
k = maxloc(z)                          ! Индекс максимального элемента вектора z
! Вывод результата для непериодического случая
write(*, "(/, ' Case #2: nonperiodic data')")
write(*, "(1x, 28('-'))"); write(*, 99998) k; write(*, 99999) k, z(k(1))
99998 format(' The element of z with the largest normalized value is z(', i2, ')')
99999 format(' The normalized value of z(', i2, ') is', f6.3)
end program rcorlTest
```

Результат:

```
Example #1: Periodic case
----------------------------
The element of z with the largest normalized value is z(26)
The normalized value of z(26) is 1.000

Example #2: Nonperiodic case
----------------------------
The element of z with the largest normalized value is z(26)
The normalized value of z(26) is 0.661
```

1.4.6. ПОДПРОГРАММА CCORL (DCCORL)

Вычисляет корреляцию двух комплексных векторов. Имеет вызов

CALL CCORL(*ido, n, x, y, ipad, nz, z, zhat*)

Описание параметров, алгоритма и комментарий см. в предшествующем разделе, принимая во внимание, что векторы *x, y, z* и *zhat* имеют комплексный тип.

Пример. Вычисляются как периодические, так и непериодические корреляции двух различных сигналов x и y. Первоначально генерируются 100 равномерно размещенных значений на отрезке $[0, 2\pi]$ функции $f_1(x) = \cos(x) + i\sin(x)$. В непериодическом случае берется функция $f_2(x) = \cos(x^2) + i\sin(x^2)$ и два входных сигнала формируются на отрезке $[0, 4\pi]$. Формулы формирования x и y см. в предшествующем разделе в примере для RCORL. Максимальное значение вектора z (корреляции x и y) наблюдается и в периодическом и в непериодическом случаях при $i = 26$.

```
program ccorlTest
 use dfimsl
 integer(4), parameter :: n = 100
 integer(4) :: i, k(1), nz
 real(4) :: a, pi, xnorm, ynorm, zreal2(4 * n)
 complex(4) :: x(n), y(n), z(4 * n), zhat(4 * n), f1, f2
 f1(a) = cmplx(cos(a), sin(a))
```

```
f2(a) = cmplx(cos(a * a), sin(a * a))
pi = const('pi')                              ! Число π
! Формируем данные для периодического случая
do i = 1, n
 x(i) = f1(2.0 * pi * float(i - 1) / float(n - 1))
 y(i) = f1(2.0 * pi * float(i - 1) / float(n - 1) + pi / 2.0)
end do
nz = 2 * n
call ccorl(0, n, x, y, 0, nz, z, zhat)        ! Периодический случай
! Нормализация вещественной части вектора z
xnorm = scnrm2(n, x, 1); ynorm = scnrm2(n, y, 1); zreal2 = real(z) / (xnorm * ynorm)
! Индекс максимального элемента вектора zreal2
k = maxloc(zreal2)
write(*, "(' Case #1: periodic data')")  ! Вывод результата для периодического случая
write(*, "(1x, 28('-'))"); write(*, 99998) k; write(*, 99999) k, zreal2(k(1))
do i = 1, n                                   ! Задаем вектор для непериодического случая
 x(i) = f2(4.0 * pi * float(i - 1) / float(n - 1))
 y(i) = f2(4.0 * pi * float(i - 1) / float(n - 1) + pi)
end do
nz = 4 * n
call ccorl(0, n, x, y, 1, nz, z, zhat)        ! Непериодический случай
! Нормализация вещественной части вектора z
xnorm = scnrm2(n, x, 1); ynorm = scnrm2(n, y, 1); zreal2 = real(z) / (xnorm * ynorm)
k = maxloc(zreal2)                    ! Индекс максимального элемента вектора zreal2
! Вывод результата для непериодического случая
write(*, "(/, ' Case #2: nonperiodic data')")
write(*, "(1x, 28('-'))"); write(*, 99998) k; write(*, 99999) k, zreal2(k(1))
99998 format(' The element of z with the largest normalized real part is z(', i2, ')')
99999 format(' The normalized value of real(z(', i2, ')) is', f6.3)
end program ccorlTest
```

Результат:

```
Example #1: periodic case
---------------------------
The element of z with the largest normalized real part is z(26).
The normalized value of real(z(26)) is 1.000

Example #2: nonperiodic case
---------------------------
The element of z with the largest normalized real part is z(26).
The normalized value of real(z(26)) is 0.649
```

1.5. ВЫЧИСЛЕНИЕ ОБРАТНОГО ПРЕОБРАЗОВАНИЯ ЛАПЛАСА

1.5.1. ПОДПРОГРАММА INLAP (DINLAP)

Вычисляет обратное преобразование Лапласа комплексной функции. Имеет вызов

CALL INLAP(*f, n, t, alpha, relerr, kmax, finv*)

Параметры подпрограммы INLAP:
Пользовательская функция: f.
Входные: n, t, alpha, relerr, kmax.
Выходной: finv.

f - комплексная пользовательская функция, для которой вычисляется обратное преобразование Лапласа. Имеет вид *f*(*z*), где *z* - *входной* комплексный параметр. Функция *f* должна получить атрибут EXTERNAL в вызывающей программной единице.

n - число точек, в которых следует вычислить обратное преобразование Лапласа.

t - вектор размера *n*, содержащий точки, в которых следует вычислить обратное преобразование Лапласа. Все элементы вектора должны быть больше нуля.

alpha - оценка максимального значения вещественных частей особенных точек функции *f*. Если оценка неизвестна, то положите *alpha* = 0.

relerr - желаемая относительная точность результата.

kmax - допустимое число оценок функции для каждого элемента вектора *t*.

finv - вектор размера *n*, *i*-й элемент которого содержит приблизительное значение преобразования Лапласа в точке *t*(*i*).

Комментарий. При работе с INLAP могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
4	1	Не удается достигнуть заданную точность за *kmax* оценок функции для некоторого элемента вектора *t*
4	2	Наблюдается переполнение для некоторого элемента вектора *t*

Описание:

Напомним, если *f* - это функция, равная нулю на отрицательной вещественной полуоси, то можно определить преобразование Лапласа функции *f*:

$$L[f](s) = \int_0^\infty e^{-sx} f(x)dx.$$

Предполагается, что при некоторых значениях s интеграл является абсолютно сходящимся.

Вычисление обратного преобразования Лапласа основано на эпсилон-алгоритме для комплексных коэффициентов Фурье, получаемых как дискретное приближение инверсионного интеграла. Первоначально алгоритм был предложен в [11], затем он был существенно улучшен и представлен в [12].

По известному преобразованию Лапласа $F(s) = L[f](s)$ комплексной функции, аппроксимация обратного преобразования

$$g(t) = \frac{e^{\alpha t}}{T}\Re\left\{\frac{1}{2}f(\alpha) + \sum_{k=1}^{\infty} f\left(\alpha + \frac{ik\pi}{T}\right)\exp\left(\frac{ik\pi t}{T}\right)\right\}$$

находится по правилу трапеций. Функция $g(t)$ является вещественной частью комплексных коэффициентов, определяемых $z = \exp(i\pi t/T)$, и алгоритм ускоряет сходимость частичных сумм этих коэффициентов, используя эпсилон-алгоритм для вычисления соответствующей диагональной аппроксимации Паде. Алгоритм пытается выбрать такой порядок аппроксимации Паде, чтобы получить заданную относительную точность, не превышая максимально допустимое число оценок функции. Параметр α является оценкой максимального значения вещественных частей особенных точек функции F, и неверный выбор этого параметра может привести к ложной сходимости. Если оценка неизвестна, то следует задать α равным нулю. Тогда алгоритм попытается найти для α подходящее значение. В случае удовлетворительной сходимости ошибка дискредитации $E = g - f$ равна

$$E = \sum_{n=1}^{\infty} e^{-2n\alpha T} f(2nT + t).$$

Следовательно, если $|f(t)| \le Me^{\beta t}$, то можно оценить вышеприведенное выражение и получить (для $0 \le t \le 2T$)

$$E \le Me^{\alpha t} / \left(e^{2T(\alpha-\beta)} - 1\right).$$

Пример. Вычисляется обратное преобразование Лапласа функции $(s - 1)^2$. Правильным результатом является xe^x. Для проверки выводится разница между точным и полученным преобразованиями.

```
program inlapTest
 use dfimsl
 integer(4) :: i, kmax, n
 real(4) :: alpha, diff2(5), finv(5), relerr, t(5), true(5)
 complex(4), external :: f
 do i = 1, 5; t(i) = float(i) - 0.5; end do
```

```
n = 5; alpha = 1.0e0; kmax = 500; relerr = 5.0e-4
call inlap(f, n, t, alpha, relerr, kmax, finv)
true = t * exp(t); diff2 = true - finv      ! Оцениваем ошибку и выводим результат
write(*, 99999) (t(i), finv(i), true(i), diff2(i), i = 1, 5)
99999 format(7x, 't', 8x, 'finv', 9x, 'true', 9x, 'diff2', /,                    &
      5(1x, e9.1, 3x, 1pe10.3, 3x, 1pe10.3, 3x, 1pe10.3, /))
end program inlapTest

complex function f(s)
 complex(4) :: s
 f = 1.0 / (s - 1.0)**2
end function f
```

Результат:

t	finv	true	diff2
0.5E+00	8.244E-01	8.244E-01	-4.947-06
1.5E+00	6.723E+00	6.723E+00	-3.624E-05
2.5E+00	3.046E+01	3.046E+01	-1.564E-04
3.5E+00	1.159E+02	1.159E+02	-6.409E-04
4.5E+00	4.051E+02	4.051E+02	-2.014E-03

1.5.2. ПОДПРОГРАММА SINLP (DSINLP)

Вычисляет, так же как и подпрограмма INLAP, обратное преобразование Лапласа комплексной функции. Имеет вызов

CALL SINLP(*f, n, t, sigma0, epstol, errvec, finv*)

Параметры подпрограммы INLAP:
Пользовательская функция: f.
Входные: n, t, sigma0, epstol,.
Выходные: errvec, finv.
Описание параметров *f, n* и *t* см. в предшествующем разделе.
sigma0 - то же, что параметр *alpha* для подпрограммы INLAP.
epstol - требуемая абсолютная однородная псевдоошибка для коэффициентов и значений обратного преобразования Лапласа.
errvec - вектор размера 8, содержащий диагностическую информацию. Его компоненты зависят от генерируемых промежуточных коэффициентов Лагерра. В элементы вектора заносятся следующие данные:

- *errvec*(1) - общая оценка псевдоошибки, равная *errvec*(2) + *errvec*(3) + + *errvec*(4); псевдоошибка = абсолютная ошибка/exp(*sigma* * *tvalue*) (понятие псевдоошибки см. ниже).
- *errvec*(2) - оценка псевдоошибки дискредитации.
- *errvec*(3) - оценка псевдоошибки отсечения.

- *errvec*(4) - оценка псевдоошибки минимального уровня шума значений функции.
- *errvec*(5) = *k* - коэффициент функции распада для *acoef* - коэффициентов разложения Лагерра.
- *errvec*(6) = *r* - основание функции распада для *acoef*; здесь ABS(*acoef*(*j* + + 1)) ≤ *k*/*r****j* для *j* ≥ *mact*/2, где *mact* - число реально вычисленных коэффициентов Лагерра.
- *errvec*(7) = *alpha* - логарифм наибольшего элемента *acoef*.
- *errvec*(8) = *beta* - логарифм наименьшего ненулевого элемента *acoef*.

finv - вектор размера *n*, *i*-й компонент которого содержит приблизительное значение обратного преобразования Лапласа в точке *t*(*i*).

Автоматически для решения предоставляется память:

- 9*mtop*/4 + *n* байт в случае SINLP;
- 9*mtop*/2 + *n* байт в случае DSINLP.

Память можно выделить явно, употребив S2NLP (DS2NLP):

```
CALL S2NLP(f, n, t, sigma0, epstol, errvec, finv, sigma,          &
     bvalue, mtop, wk, iflovc)
```

Дополнительные параметры подпрограммы S2NLP:

sigma (*входной*) - первый параметр разложения Лагерра. Если *sigma* не больше *sigma*0, то значение *sigma* устанавливается равным *sigma*0 + 0.7.

bvalue (*входной*) - первый параметр разложения Лагерра. Если *bvalue* меньше 2.0(*sigma* - *sigma*0), то значение *bvalue* устанавливается равным 2.5(*sigma* - *sigma*0).

mtop (*входной*) - верхняя граница числа вычисляемых коэффициентов в разложении Лагерра; величина *mtop* должна быть кратна числу 4. Заметим, что максимальное число оценок преобразования Лапласа равно *mtop*/2 + 2. Значение по умолчанию - 1024.

wk - *рабочий вектор* размера 9*mtop*/4.

iflovc (*выходной*) - целочисленный вектор размера *n*, содержащий флаги переполнения/исчезновения для элементов вектора *finv*. Элементы вектора принимают следующие значения:

- *iflovc*(*i*) = 0 - нормальное завершение;
- *iflovc*(*i*) = 1 - величина обратного преобразования Лапласа слишком велика для представления в ЭВМ; значение *finv*(*i*) установлено равным AMACH(6);
- *iflovc*(*i*) = -1 - величина обратного преобразования Лапласа слишком мала для представления в ЭВМ; значение *finv*(*i*) установлено равным 0.0;

- *iflovc*(*i*) = 2 - величина обратного преобразования Лапласа слишком велика для представления в ЭВМ даже до выполнения разложения; значение *finv*(*i*) установлено равным AMACH(6);
- *iflovc*(*i*) = -2 - величина обратного преобразования Лапласа слишком мала для представления в ЭВМ даже до выполнения разложения; значение *finv*(*i*) установлено равным 0.0.

Комментарий. При работе с INLAP могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
1	1	Нормальное завершение, но оценка ошибки несколько больше, чем *epstol*. Заметьте, однако, что реальная ошибка конечного результата может быть меньше *epstol*, поскольку оценка ошибки является пессимистической
3	2	Вычисления завершены из-за того, что ошибки округления приводят к превышению оценки ошибки заданной точности
4	3	Уровень снижения значений коэффициентов недостаточен. Результат можно улучшить, применяя S2NLP с большим значением параметра *mtop*
4	4	Уровень снижения значений коэффициентов недостаточен. В дополнение ошибки округления не позволяют достигнуть заданную точность
4	5	Оценки ошибок не возвращаются, поскольку поведение коэффициентов не позволяет выполнять прогнозирование. Результат, возможно, неверен. Проверьте значение *sigma*0. В этой ситуации значения элементов *errvec*(1:5) устанавливаются равными -1.0

Описание:

Вычисление обратного преобразования Лапласа основано на модифицированном методе Вика (см. [47]), предложенном в [12]. Метод применим, когда функция f имеет непрерывные производные всех порядков на интервале $[0, \infty)$. В этом случае подпрограмму SINLP следует употреблять взамен подпрограммы INLAP. Такая замена особенно эффективна, когда необходимо вычислит преобразование в большом числе точек. В частности, для заданной комплексной функции $F(s) = L[f](s)$ можно представить f в виде разложения Лагерра, коэффициенты которого определяются F. Полностью метод описан в [12] и [26].

Алгоритм пытается найти функцию $g(t)$, аппроксимирующую $f(t)$ и удовлетворяющую неравенству

$$\left|\frac{g(t)-f(t)}{e^{\sigma t}} < \varepsilon\right|,$$

где ε = *epstol* и σ := *sigma* > *sigma*0. Выражение в левой части приведенного неравенства называется *псевдоошибкой*. Ее оценка содержится в *errvec*(1).

На первом шаге метода *F* преобразовывается в ϕ, где

$$\phi(z) = \frac{b}{1-z} F\left(\frac{b}{1-z} - \frac{b}{2} + \sigma\right).$$

Известно, что, если *f* - гладкая функция, ϕ является аналитической в единичном круге комплексной плоскости и, следовательно, представима в виде разложения Тейлора

$$\phi(z) = \sum_{s=0}^{\infty} a_s z^s ,$$

которое сходится для всех *z*, абсолютная величина которых меньше, чем радиус сходимости r_c. Это число оценивается в *errvec*(6). В *errvec*(5) оценивается минимальная величина *k*, удовлетворяющая неравенству

$$|a_s| < \frac{k}{r_s}$$

для всех $r < r_c$.

Коэффициенты ряда Тейлора функции *f* могут быть использованы для расширения *f* в разложении Лагерра

$$f(t) = e^{\sigma t} \sum_{s=0}^{\infty} a_s e^{-bt/2} L_s(bt).$$

Пример. Решается та же, что и в примере предшествующего раздела, задача.

```
program sinlpTest
use dfimsl
integer(4) :: i, n
real(4) :: diff(5), errvec(8), exp, finv(5), float, relerr, sigma0, t(5), true(5)
complex(4), external :: f
do i = 1, 5; t(i) = float(i) - 0.5; end do
n = 5; sigma0 = 1.0e0; relerr = 5.0e-4
call sinlp(f, n, t, sigma0, relerr, errvec, finv)
true = t * exp(t); diff = true - finv        ! Оцениваем ошибку и выводим результат
write(*, 99999) (t(i), finv(i), true(i), diff(i), i = 1, 5)
99999 format(7x, 't', 8x, 'finv', 9x, 'true', 9x, 'diff', /,                    &
      5(1x, e9.1, 3x, 1pe10.3, 3x, 1pe10.3, 3x, 1pe10.3, /))
```

```
end program sinlpTest

complex function f(s)
 complex(4) :: s
 f = 1.0 / (s - 1.0)**2
end function f
```

Результат:

t	finv	true	diff
0.5E+00	8.244E-01	8.244E-01	-2.205E-06
1.5E+00	6.723E+00	6.723E+00	-7.153E-06
2.5E+00	3.046E+01	3.046E+01	-1.335E+00
3.5E+00	1.159E+02	1.159E+02	3.052E-05
4.5E+00	4.051E+02	4.051E+02	-4.272E-04

2. ПРОЦЕДУРЫ БИБЛИОТЕКИ IMSL 90 MP ДЛЯ БЫСТРЫХ ПРЕОБРАЗОВАНИЙ ФРЬЕ

2.1. ПЕРЕЧЕНЬ И ПАРАМЕТРЫ ПОДПРОГРАММ

Библиотека IMSL 90 MP содержит подпрограммы FAST_DFT, FAST_2DFT и FAST_3DFT, выполняющие соответственно быстрые преобразования Фурье комплексных массивов ранга 1, ранга 2 и ранга 3. (При работе с одинарной точностью подпрограммы имеют префикс C_, а при работе с двойной - D_.)

Чтобы обратиться к названным подпрограммам, в вызывающей процедуре необходимо сделать ссылки

```
use fast_dft_int          ! Для вызова fast_dft
use fast_2dft_int         ! Для вызова fast_2dft
use fast_3dft_int         ! Для вызова fast_3dft
```

Общие параметры процедур приведены в табл. 2.1.

Таблица 2.1. Параметры подпрограмм FAST_DFT, FAST_2DFT и FAST_3DFT

Имя	*Смысл/вид*	*Тип*
forward_in = x	Подлежащий прямому преобразованию комплексный массив ранга 1, 2 или 3 / *входной*	COMPLEX(4) или COMPLEX(8)
forward_out = y	Комплексный массив ранга 1, 2 или 3 - результат прямых преобразований Фурье / *выходной*	То же
inverse_in = y	Подлежащий обратному преобразованию комплексный массив ранга 1, 2 или 3 / *входной*	"
inverse_out = x	Комплексный массив ранга 1, 2 или 3 - результат обратных преобразований Фурье / *выходной*	"
ido = ido	Целочисленный флаг, управляющий вычислениями. Обычно этот параметр используется, когда рабочие переменные, нужные для прямого и обратного преобразований, сохраняются в вызывающей программной единице. Вычисление рабочих переменных и их запись во внутренние массивы осуществляются по умолчанию. Это инициализирующий шаг весьма трудоемок. Вычисления, однако, можно проводить в два этапа (см., в частности, примеры 3 для FAST_DFT и FAST_2DFT). Общий порядок	INTEGER(4)

	вычислений следующий: первоначально FAST_DFT вызывается с *ido* = 0, в результате FAST_DFT завершается с *ido* < 0. Необязательный комплексный вектор *w*(:) с SIZE(*w*) ≥ -*ido* должен быть переразмещен. Затем вновь вызывается FAST_DFT, и следующий возврат осуществляется с *ido* = 1. Переменные, необходимые для прямого и обратного преобразований, сохраняются в векторе *w*(:). Далее, когда подпрограмма FAST_DFT вызывается *ido* = 1 и с тем же значением *n*, содержимое *w*(:) будет использовано для рабочих переменных; дорогостоящий инициализирующий шаг повторно не выполняется. Необязательные параметры *ido*= и *work_array*= должны использоваться совместно / *входной/выходной* *Замечание*. Приведенное описание справедливо и для FAST_2DFT и FAST_3DFT	
work_array = *w*(:)	Комплексный вектор, употребляемый для хранения рабочих данных между вызовами FAST_DFT (FAST_2DFT, FAST_3DFT). Значение SIZE(*w*) должно быть не меньше величины -*ido*, когда *ido* < 0 / *входной/выходной*	COMPLEX(4) или COMPLEX(8)

2.2. ПОДПРОГРАММА FAST_DFT

Вычисляет дискретное преобразование Фурье комплексного массива *x* ранга 1. Имеет вызов

CALL FAST_DFT([*forward_in* = *x*], [*forward_out* = *y*], [*inverse_in* = *y*], &
[*inverse_out* = *x*], [*ndata* = *n*], [*ido* = *ido*], &
[*work_array* = *w*(:)], [*iopt* = *iopt*(:)])

Все параметры подпрограммы являются *необязательными*. Для выполнения прямых преобразований необходимо задать параметры *forward_in* = *x*, *forward_out* = *y*, а обратных - *inverse_in* = *y*, *inverse_out* = *x*.

Описание параметров подпрограммы FAST_DFT, кроме *ndata* = *n* и *iopt* = *iopt*(:), см. в табл. 2.1.

ndata = *n* (*входной*) - число используемых для преобразования данных; по умолчанию *n* = SIZE(*x*).

iopt = *iopt*(:) (*входной/выходной*) - массив производного типа с той же разновидностью, что и входной вектор. Используется для передачи в подпрограмму FAST_DFT необязательных опций. Принимает следующие значения:

Префиксы опций = ?	*Имя опции*	*Значение*
c_, z_	fast_dft_scan_for_NaN	1
" "	fast_dft_near_power_of_2	2
" "	fast_dft_scale_forward	3
" "	fast_dft_scale_inverse	4

iopt(*io*) = ?_options(?_*fast_dft_scan_for_NaN*, ?_*dummy*) - проверяет входной вектор на наличие в нем NaN (не числа). По умолчанию проверка не выполняется.

iopt(*io*) = ?_options(?_*fast_dft_near_power_of_2*, ?_*dummy*) - возвращает в *iopt*(*io* + 1)*%idummy* ближайшую большую *n* степень числа 2.

iopt(*io*) = ?_options(?_*fast_dft_scale_forward*, *real_part_of_scale*), *iopt*(*io* + 1) = ?_options(?_*dummy*, *imaginary_part_of_scale*) - комплексное число, определяемое коэффициентом CMPLX(*real_part_of_scale*, *imaginary_part_of_scale*), умножается на вектор, содержащий прямое преобразование. По умолчанию равно единице.

iopt(*io*) = ?_options(?_*fast_dft_scale_inverse*, *real_part_of_scale*), *iopt*(*io* + 1) = ?_options(?_*dummy*, *imaginary_part_of_scale*) - комплексное число, определяемое коэффициентом CMPLX(*real_part_of_scale*, *imaginary_part_of_scale*), умножается на вектор, содержащий обратное преобразование. По умолчанию равно единице.

Описание:

Максимальная вычислительная эффективность достигается, когда размер входного вектора может быть представлен в виде произведения $n = 2^{i_1}3^{i_2}4^{i_3}5^{i_4}$, где i_1, i_2, i_3, i_4 - неотрицательные целые числа. Других ограничений на $n \geq 1$ нет.

Замечание. Фатальные и завершающие ошибки, возникающие при работе с FAST_DFT, имеют номера 651-661 и 701-711.

Пример 1. Преобразовывается вектор случайных комплексных чисел. Первоначально выполняется генерация значений элементов вектора. Затем вычисляются прямое и обратное дискретные преобразования Фурье. Данные, полученные в результате обратного преобразования, сравниваются с исходным вектором.

```
program fast_dftTest1
 use fast_dft_int
 use rand_gen_int
 implicit none
 integer, parameter :: n = 1024
```

```
real(kind(1e0)), parameter :: one = 1e0
real(kind(1e0)) err, y(2 * n)
complex(kind(1e0)), dimension(n) :: a, b, c
! Создаем случайную комплексную последовательность и копируем данные
! в вектор c для последующей проверки результата преобразований
call rand_gen(y); a = cmplx(y(1:n), y(n + 1:2 * n), kind(one)); c = a
! Выполняем, используя одинарную точность,
! прямое и затем обратное дискретные преобразования Фурье
call c_fast_dft(forward_in = a, forward_out = b)
call c_fast_dft(inverse_in = b, inverse_out = a)
! Проверяем равенство начальной и результирующей последовательностей
err = maxval(abs(c - a)) / maxval(abs(c))
if(err <= sqrt(epsilon(one))) write(*, *) 'Example 1 for FAST_DFT is correct'
end program fast_dftTest1
```

Пример 2. На вход подаются циклические данные с линейным трендом. Набор данных порождается функцией $x(t) = at + b + y(t)$, где $y(t)$ - гармонические серии. Независимая переменная нормализуется так, что $-1 \le t \le 1$. На первом этапе линейные компоненты эффективно удаляются из данных решателем LIN_SOL_LSQ переопределенных линейных систем, использующим метод наименьших квадратов. Затем преобразовываются невязки и анализируются результирующие частоты.

```
program fast_dftTest2
 use fast_dft_int
 use lin_sol_lsq_int
 use rand_gen_int
 use sort_real_int
 implicit none
 integer, parameter :: n = 64, k = 4
 integer i, ip(n)
 real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
 real(kind(1e0)) delta_t, pi
 real(kind(1e0)) y(k), z(2), indx(k), t(n), temp(n)
 complex(kind(1e0)) a_trend(n, 2), a, b_trend(n, 1), b, c(k), f(n), r(n), x(n), x_trend(2, 1)
 ! Генерируем случайные данные для линейного тренда и гармонических серий
 call rand_gen(z); a = z(1); b = z(2)
 call rand_gen(y)
 ! Усиливаем гармоники 2, ..., k + 1
 c = y + one
 ! Определяем интервал выборки
 delta_t = two / n; t = (/ (-one + i * delta_t, i = 0, n - 1) /)
 ! Вычисляем число π и вектор indx
 pi = atan(one) * 4E0; indx = (/ (i * pi, i = 1, k) /)
```

```
! Формируем входной набор данных как линейный тренд плюс гармоники
x = a + b * t +  matmul(exp(cmplx(zero, spread(t, 2, k) *                &
      spread(indx, 1, n), kind(one))), c)
! Определяем матрицу наименьших квадратов для линейного тренда
a_trend(1:, 1) = one; a_trend(1:, 2) = t; b_trend(1:, 1) = x
! Решаем линейную систему с построенной матрицей
call Lin_sol_lsq(a_trend, b_trend, x_trend)
! Вычисляем гармонические невязки
r = x - reshape(matmul(a_trend, x_trend), (/ n /))
! Преобразовываем гармонические невязки
call c_fast_dft(forward_in = r, forward_out = f)
! Вектор перестановок для подпрограммы сортировки
ip = (/ (i, i = 1, n) /)
! Доминантные частоты должны иметь номера 2, …, k + 1
! Сортируем преобразованные данные
call s_sort_real(-(abs(f)), temp, iperm = ip)
! Доминантные частоты размещены в сечении ip(1:k)
! Сортируем эти значения для сравнения с величинами 2, …, k + 1
call s_sort_real(real(ip(1:k)), temp)
ip(1:k) = (/ (i, i = 2, k + 1) /)
! Проверяем результат
if(count(int(temp(1:k)) /= ip(1:k)) == 0) then
 write(*, *) 'Example 2 for FAST_DFT is correct'
end if
end program fast_dftTest2
```

Пример 3. Выполняются несколько преобразований с одной инициализацией. Необязательные параметры *ido* и *work_array* используются для хранения рабочих переменных в вызывающей программной единице. Это повышает производительность программы, поскольку рабочие данные вычисляются единожды для нескольких вызовов FAST_DFT.

```
program fast_dftTest3
 use fast_dft_int
 use rand_gen_int;
 implicit none
 integer, parameter :: n = 64
 ! Прежде в подпрограмме FAST_DFT будет определен размер вектора work(:)
 integer ido_value
 real(kind(1e0)) :: one=1e0
 real(kind(1e0)) err, y(2 * n)
 complex(kind(1e0)), dimension(n) :: a, b, save_a
 complex(kind(1e0)), allocatable :: work(:)
 ! Генерируем случайный комплексный массив
```

```
call rand_gen(y)
a = cmplx(y(1:n), y(n + 1:2 * n), kind(one))
save_a = a                              ! Храним массив для проверки результата
! Выполняем прямое и обратное преобразования,
! используя ранее найденные рабочие данные
ido_value = 0
do
 if(allocated(work)) deallocate(work)
 ! Выделяем память под вектор work(:)
 if(ido_value <= 0) allocate(work(-ido_value))
 call c_fast_dft(forward_in = a, forward_out = b, ido = ido_value, work_array = work)
 if(ido_value == 1) exit
end do
! Повторный вход в C_FAST_DFT с найденными ранее
! и записанными в work(:) рабочими данными
call c_fast_dft(inverse_in = b, inverse_out = a, ido = ido_value, work_array = work)
! Освобождаем занимаемую вектором work(:) память
if(allocated(work)) deallocate(work)
! Проверяем результат
err = maxval(abs(save_a - a)) / maxval(abs(save_a))
if(err <= sqrt(epsilon(one))) then
 write(*, *) 'Example 3 for FAST_DFT is correct'
end if
end program fast_dftTest3
```

Пример 4. Применение преобразований Фурье для вычисление сверток. Вычисляются суммы

$$c_k \sum_{j=0}^{n-1} a_j b_{k-j}\ ,\ k = 0,\ \ldots,\ n - 1.$$

Прямые вычисления посредством нахождения произведения матрицы и вектора требуют n^2 операций сложения и умножения. Эффективный метод состоит в вычислении произведений преобразованных векторов a и b с последующим инвертированием результата. Такой подход более предпочтителен (по сравнению с прямыми вычислениями) в задачах большой размерности.

```
program fast_dftTest4
 use fast_dft_int
 use rand_gen_int
 implicit none
 integer j; integer, parameter :: n=40
 real(kind(1e0)) :: one=1e0
 real(kind(1e0)) err
 real(kind(1e0)), dimension(n) :: x, y, yy(n, n)
```

```
complex(kind(1e0)), dimension(n) :: a, b, c, d, e, f
! Генерируем две случайные комплексные последовательности a и b
call rand_gen(x); call rand_gen(y)
a = x; b = y
! Вычисляем свертку c векторов a и b
! Используем для проверки результата произведение матрицы на вектор
yy(1:, 1) = y
do j = 2, n
 yy(2:, j) = yy(1:n - 1, j - 1); yy(1, j) = yy(n, j - 1)
end do
c = matmul(yy, x)
! Преобразовываем векторы a и b в последовательности d и e
call c_fast_dft(forward_in = a, forward_out = d)
call c_fast_dft(forward_in = b, forward_out = e)
! Вычисляем обратное преобразование произведения d * e
call c_fast_dft(inverse_in = d * e, inverse_out = f)
! Проверяем теорему о свертке:
! inverse(transform(a) * transform(b)) = convolution(a, b)
err = maxval(abs(c - f)) / maxval(abs(c))
if(err <= sqrt(epsilon(one))) then
 write(*, *) 'Example 4 for FAST_DFT is correct'
end if
end program fast_dftTest4
```

2.3. ПОДПРОГРАММА FAST_2DFT

Вычисляет дискретное преобразование Фурье комплексного массива *x* ранга 2. Имеет вызов

CALL FAST_2DFT([*forward_in* = *x*], [*forward_out* = *y*], [*inverse_in* = *y*], &
[*inverse_out* = *x*], [*mdata* = *m*], [*ndata* = *n*], &
[*ido* = *ido*], [*work_array* = *w*(:)], [*iopt* = *iopt*(:)])

Все параметры подпрограммы являются *необязательными*. Для выполнения прямых преобразований необходимо задать параметры *forward_in* = *x*, *forward_out* = *y*, а обратных - *inverse_in* = *y*, *inverse_out* = *x*.

Описание параметров подпрограммы FAST_2DFT, кроме *mdata* = *m*, *ndata* = *n* и *iopt* = *iopt*(:), см. в табл. 2.1.

mdata = *m* (*входной*) - используемый размер по первому измерению; по умолчанию *m* = SIZE(*x*, 1).

ndata = *n* (*входной*) - используемый размер по второму измерению; по умолчанию *n* = SIZE(*x*, 2).

iopt = *iopt*(:) (*входной/выходной*) - массив производного типа с той же разновидностью, что и входной массив. Используется для передачи в под-

программу FAST_2DFT необязательных установок. Принимает следующие значения:

Префиксы опций = ?	*Имя опции*	*Значение*
c_, z_	fast_2dft_scan_for_NaN	1
" "	fast_2dft_near_power_of_2	2
" "	fast_2dft_scale_forward	3
" "	fast_2dft_scale_inverse	4

iopt(*io*) = ?_options(?_*fast_2dft_scan_for_NaN*, ?_*dummy*) - проверяет входной массив на наличие в нем NaN. По умолчанию проверка не выполняется.

iopt(*io*) = ?_options(?_*fast_2dft_near_power_of_2*, ?_*dummy*) - возвращает в *iopt*(*io* + 1)*%idummy* и *iopt*(*io* + 2)*%idummy* ближайшие большие соответственно *m* и *n* степени числа 2.

iopt(*io*) = ?_options(?_*fast_2dft_scale_forward*, *real_part_of_scale*), *iopt*(*io* + 1) = ?_options(?_*dummy*, *imaginary_part_of_scale*) - комплексное число, определяемое коэффициентом CMPLX(*real_part_of_scale*, *imaginary_part_of_scale*), умножается на массив, содержащий прямое преобразование. По умолчанию равно единице.

iopt(*io*) = ?_options(?_*fast_2dft_scale_inverse*, *real_part_of_scale*), *iopt*(*io* + 1) = ?_options(?_*dummy*, *imaginary_part_of_scale*) - комплексное число, определяемое коэффициентом CMPLX(*real_part_of_scale*, *imaginary_part_of_scale*), умножается на массив, содержащий обратное преобразование. По умолчанию равно единице.

Описание:

Максимальная вычислительная эффективность достигается, когда *m* и *n* являются произведением небольших простых чисел, т. е. могут быть представлены в виде $2^{i_1}3^{i_2}4^{i_3}5^{i_4}$, где i_1, i_2, i_3, i_4 - неотрицательные целые числа.

Замечание. Фатальные и завершающие ошибки, возникающие при работе с FAST_2DFT, имеют номера 670-680 и 720-730.

Пример 1. Преобразовывается двумерный массив случайных комплексных чисел. Первоначально выполняется генерация значений элементов массива. Затем вычисляются прямое и обратное дискретные преобразования Фурье. Данные, полученные в результате обратного преобразования, сравниваются с исходным массивом.

```
program fast_2dftTest1
 use fast_2dft_int
 use rand_int
```

```
 implicit none
 integer, parameter :: n = 24, m = 40
 real(kind(1e0)) :: err, one = 1e0
 complex(kind(1e0)), dimension(n, m) :: a, b, c
 ! Создаем случайную комплексную последовательность и копируем данные
 ! в массив c для последующей проверки результата преобразований
 a = rand(a); c = a
 ! Выполняем, используя одинарную точность,
 ! прямое и затем обратное дискретные преобразования Фурье
 call c_fast_2dft(forward_in = a, forward_out = b)
 call c_fast_2dft(inverse_in = b, inverse_out = a)
 ! Проверяем равенство начальной и результирующей последовательностей
 err = maxval(abs(c - a)) / maxval(abs(c))
 if(err <= sqrt(epsilon(one))) then
  write(*, *) 'Example 1 for FAST_2DFT is correct.'
 end if
end program fast_2dftTest1
```

Пример 2. На вход подаются циклические двумерные данные с линейным трендом. Набор данных порождается функцией $x(s, t) = a + bs + ct + y(s, t)$, где $y(s, t)$ - гармонические серии. Независимые переменные нормализуются так, что $-1 \le s \le 1$ и $-1 \le t \le 1$. На первом этапе линейные компоненты эффективно удаляются из данных решателем LIN_SOL_LSQ переопределенных линейных систем, использующим метод наименьших квадратов. Затем преобразовываются невязки и анализируются результирующие частоты.

```
program fast_2dftTest2
 use fast_2dft_int
 use lin_sol_lsq_int
 use sort_real_int
 use rand_int
 implicit none
 integer, parameter :: n = 8, k = 15
 integer i, ip(n * n), order(k)
 real(kind(1e0)), parameter :: one = 1e0, two = 2e0, zero = 0e0
 real(kind(1e0)) delta_t
 real(kind(1e0)) rn(3), s(n), t(n), temp(n * n), new_order(k)
 complex(kind(1e0)) a, b, c, a_trend(n * n, 3), b_trend(n * n, 1),      &
       f(n, n), r(n, n), x(n, n), x_trend(3, 1)
 complex(kind(1e0)), dimension(n, n) :: g = zero, h = zero
 ! Генерируем случайные данные для линейного планарного тренда
 rn = rand(rn); a = rn(1); b = rn(2); c = rn(3)
 ! Генерируем частотные компоненты гармонических серий
 ! задаем ненулевые случайные амплитуды на двух границах квадратной области
```

```
g(1:, 1) = rand(g(1:, 1)); g(1, 1:) = rand(g(1, 1:))
! Преобразовываем g в гармонические серии h во временной области
call c_fast_2dft(inverse_in = g, inverse_out = h)
! Определяем интервал выборки
delta_t = two / n
s = (/ (-one + (i - 1) * delta_t, i = 1, n) /)
t = (/ (-one + (i - 1) * delta_t, i = 1, n) /)
! Формируем входной набор данных как линейный тренд плюс гармоники
x = a + b * spread(s, dim = 2, ncopies = n) + c * spread(t, dim = 1, ncopies = n) + h
! Определяем матрицу наименьших квадратов для линейного планарного тренда
a_trend(1:, 1) = one
a_trend(1:, 2) = reshape(spread(s, dim = 2, ncopies = n), (/ n * n /))
a_trend(1:, 3) = reshape(spread(t, dim = 1, ncopies = n), (/ n * n /))
b_trend(1:, 1) = reshape(x, (/ n * n /))
! Решаем линейную систему с построенной матрицей
call Lin_sol_lsq(a_trend, b_trend, x_trend)
! Вычисляем гармонические невязки
r = x - reshape(matmul(a_trend, x_trend), (/ n, n /))
! Преобразовываем гармонические невязки
call c_fast_2dft(forward_in = r, forward_out = f)
! Вектор перестановок для подпрограммы сортировки
ip = (/ (i, i = 1, n**2) /)
! Сортируем преобразованные данные
call s_sort_real(-(abs(reshape(f, (/ n * n /)))), temp, iperm = ip)
! Доминантные частоты размещены в сечении ip(1:k)
! Сортируем эти значения для сравнения с оригинальным порядком частот
call s_sort_real(real(ip(1:k)), new_order)
order(1:n) = (/ (i, i = 1, n) /)
order(n + 1:k) = (/ ((i - n) * n + 1, i = n + 1, k) /)
! Проверяем результат
if(count(order /= int(new_order)) == 0) then
 write(*, *) 'Example 2 for FAST_2DFT is correct'
end if
end program fast_2dftTest2
```

Пример 3. Выполняются несколько двумерных преобразований с одной инициализацией. Необязательные параметры *ido* и *work_array* используются для хранения рабочих переменных в вызывающей программной единице. Это повышает производительность программы, поскольку рабочие данные вычисляются единожды для нескольких вызовов FAST_2DFT.

```
program fast_3dftTest3
 use fast_2dft_int
 implicit none
 integer, parameter :: n = 256
```

```
real(kind(1e0)), parameter :: one = 1e0, zero = 0e0
real(kind(1e0)) r(n, n), err
complex(kind(1e0)) a(n, n), b(n, n), c(n, n)
! Прежде в подпрограмме FAST_2DFT будет определен размер вектора work(:)
integer i, j, ido_value
complex(kind(1e0)), allocatable :: work(:)
! Заносим значение one для точек, лежащих внутри круга с радиусом r = 64
a = zero
r = reshape((/ (((i - n / 2)**2 + (j - n / 2)**2, i = 1, n), j = 1, n) /), (/ n, n /))
where(r <= (n / 4)**2) a = one
c = a ! Храним массив для проверки результата
! Выполняем прямое и обратное преобразования,
! используя ранее найденные рабочие данные
ido_value = 0
do
 if(allocated(work)) deallocate(work)
 ! Выделяем память под вектор work(:)
 if(ido_value <= 0) allocate(work(-ido_value))
 call c_fast_2dft(forward_in = a, forward_out = b, ido = ido_value, work_array = work)
 if(ido_value == 1) exit
end do
! Повторный вход в C_FAST_2DFT с найденными ранее
! и записанными в work(:) рабочими данными
call c_fast_2dft(inverse_in = b, inverse_out = a, ido = ido_value, work_array = work)
! Освобождаем занимаемую вектором work(:) память
if(allocated(work)) deallocate(work)
! Проверяем, чтобы inverse(transform(image)) = image
err = maxval(abs(c - a)) / maxval(abs(c))
if(err <= sqrt(epsilon(one))) then
 write(*, *) 'Example 3 for FAST_2DFT is correct'
end if
end program fast_3dftTest3
```

2.4. ПОДПРОГРАММА FAST_3DFT

Вычисляет дискретное преобразование Фурье комплексного массива *x* ранга 3. Имеет вызов

```
CALL FAST_2DFT([forward_in = x], [forward_out = y], [inverse_in = y],  &
     [inverse_out = x], [mdata = m], [ndata = n], [kdata = k],         &
     [ido = ido], [work_array = w(:)], [iopt = iopt(:)])
```

Все параметры подпрограммы являются *необязательными*. Для выполнения прямых преобразований необходимо задать параметры *forward_in* = *x*, *forward_out* = *y*, а обратных - *inverse_in* = *y*, *inverse_out* = *x*.

Описание параметров подпрограммы FAST_3DFT, кроме *mdata* = *m*, *ndata* = *n*, *kdata* = *k* и *iopt* = *iopt*(:), см. в табл. 2.1.

mdata = *m* (*входной*) - используемый размер по первому измерению; по умолчанию *m* = SIZE(*x*, 1).

ndata = *n* (*входной*) - используемый размер по второму измерению; по умолчанию *n* = SIZE(*x*, 2).

kdata = *k* (*входной*) - используемый размер по третьему измерению; по умолчанию *k* = SIZE(*x*, 3).

iopt = *iopt*(:) (*входной/выходной*) - массив производного типа с той же разновидностью, что и входной массив. Используется для передачи в подпрограмму FAST_2DFT необязательных установок. Принимает следующие значения:

Префиксы опций = ?	*Имя опции*	*Значение*
c_, z_	fast_3dft_scan_for_NaN	1
" "	fast_3dft_near_power_of_2	2
" "	fast_3dft_scale_forward	3
" "	fast_3dft_scale_inverse	4

iopt(*io*) = ?_options(?*_fast_2dft_scan_for_NaN*, ?*_dummy*) - проверяет входной массив на наличие в нем NaN. По умолчанию проверка не выполняется.

iopt(*io*) = ?_options(?*_fast_2dft_near_power_of_*2, ?*_dummy*) - возвращает в *iopt*(*io* + 1)*%idummy*, *iopt*(*io* + 2)*%idummy* и *iopt*(*io* + 3)*%idummy* ближайшие большие соответственно *m*, *n* и *k* степени числа 2.

iopt(*io*) = ?_options(?*_fast_3dft_scale_forward*, *real_part_of_scale*), *iopt*(*io* + 1) = ?_options(?*_dummy*, *imaginary_part_of_scale*) - комплексное число, определяемое коэффициентом CMPLX(*real_part_of_scale*, *imaginary_part_of_scale*), умножается на массив, содержащий прямое преобразование. По умолчанию равно единице.

iopt(*io*) = ?_options(?*_fast_3dft_scale_inverse*, *real_part_of_scale*), *iopt*(*io* + 1) = ?_options(?*_dummy*, *imaginary_part_of_scale*) - комплексное число, определяемое коэффициентом CMPLX(*real_part_of_scale*, *imaginary_part_of_scale*), умножается на массив, содержащий обратное преобразование. По умолчанию равно единице.

Описание:

Максимальная вычислительная эффективность достигается, когда *m*, *n* и *k* являются произведением небольших простых чисел, т. е. могут быть представлены в виде $2^{i_1}3^{i_2}4^{i_3}5^{i_4}$, где i_1, i_2, i_3, i_4 - неотрицательные целые числа.

Замечание. Фатальные и завершающие ошибки, возникающие при работе с FAST_3DFT, имеют номера 685-695 и 740-750.

Пример. Преобразовывается двумерный массив случайных комплексных чисел. Первоначально выполняется генерация значений элементов массива. Затем вычисляются прямое и обратное дискретные преобразования Фурье. Данные, полученные в результате обратного преобразования, сравниваются с исходным массивом.

```
program fast_3dftTest
 use fast_3dft_int
 implicit none
 integer i, j, k
 integer, parameter :: n = 64
 real(kind(1e0)), parameter :: one = 1e0, zero = 0e0
 real(kind(1e0)) r(n, n, n), err
 complex(kind(1e0)) a(n, n, n), b(n, n, n), c(n, n, n)
 ! Заносим значение one для точек, лежащих внутри сферы с радиусом r = 64
 a = zero
 do i=1, n; do j=1, n; do k=1, n
  r(i, j, k) = (i - n / 2)**2 + (j - n / 2)**2 + (k - n / 2)**2
 end do; end do; end do
 where(r <= (n / 4)**2) a = one
 ! Храним массив для проверки результата
 c = a
 ! Выполняем прямое и обратное преобразования образа
 call c_fast_3dft(forward_in = a, forward_out = b)
 call c_fast_3dft(inverse_in = b, inverse_out = a)
 ! Проверяем равенство inverse(transform(image)) = image
 err = maxval(abs(c - a)) / maxval(abs(c))
 if(err <= sqrt(epsilon(one))) then
  write(*, *) 'Example for FAST_3DFT is correct'
 end if
end program fast_3dftTest
```

3. РЕШЕНИЕ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

3.1. МЕТОДЫ РЕШЕНИЯ НЕЛИНЕЙНЫХ УРАВНЕНИЙ С ОДНИМ НЕИЗВЕСТНЫМ

3.1.1. ПОСТАНОВКА ЗАДАЧИ

Поиск корней (в общем случае комплексных) нелинейного уравнения с одним неизвестным

$$f(x) = 0 \qquad (3.1)$$

выполняется в два этапа. На первом осуществляется *локализация корня*, т. е. ищется отрезок $[a, b]$, содержащий один корень уравнения (3.1). Такой отрезок называют *отрезком локализации* корня $\bar{x}$. На втором выполняется поиск корня на отрезке локализации.

Корни уравнения (3.1) разделяются на простые и кратные. Корень $\bar{x}$ уравнения (3.1) называется *простым*, если $f'(\bar{x}) \neq 0$. В противном случае корень называется *кратным*. Например, уравнение $(x - 1)(x - 2)^2 = 0$ имеет один простой ($\bar{x}_1 = 1$) и два кратных корня ($\bar{x}_2 = \bar{x}_3 = 2$). Целое число m называется *кратностью корня*, если $f^{(k)}(\bar{x}) = 0$ для $k = 1, 2, ..., m - 1$ и $f^{(m)}(\bar{x}) \neq 0$.

Заметим, что если функция $f(x)$ на отрезке $[a, b]$ непрерывна и $f(a) * f(b) \leq 0$, то этот отрезок содержит по крайней мере один корень уравнения (3.1).

Поскольку в общем случае поиск точных значений корней на компьютере невозможен, задача вычисления корней уравнения (3.1) формулируется так: "Найти с заданной точностью ε приближения всех корней уравнения (3.1)".

3.1.2. ПОИСК ВЕЩЕСТВЕННЫХ КОРНЕЙ

При поиске вещественных корней на заданном отрезке IMSL применяет гибридный алгоритм, включающий *метод бисекций, линейную интерполяцию* (*метод секущих*) и *обратную квадратичную интерполяцию*. На каждом шаге для вычисления очередного приближения выбирается наиболее подходящий метод.

Также IMSL использует для определения заданного числа вещественных корней метод Мюллера. Последний вдобавок применяется и при вычислении комплексных корней.

Изложим суть применяемых в библиотеке IMSL алгоритмов решения нелинейного уравнения $f(x) = 0$.

3.1.2.1. Метод бисекций

Будем считать, что задача локализации решена и известен отрезок $[a, b]$, на котором есть один корень. Причем он является простым. Рассмотрим на рис. 3.1 график функции $y = f(x)$ на отрезке локализации $[a, b]$, для которого, напомним, $f(a) * f(b) \le 0$. Функция на отрезке локализации непрерывна.

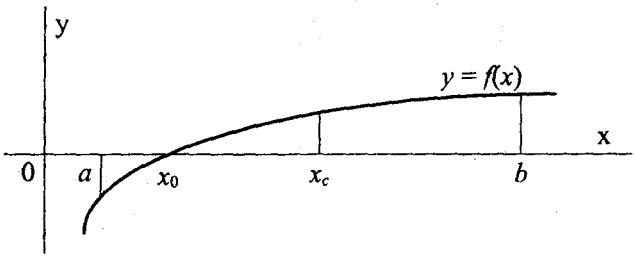


Рис. 3.1. Иллюстрация метода бисекций

Возьмем точку $x_c = (a + b) / 2$. Из анализа приведенного графика нетрудно понять, что дальнейший поиск корня следует выполнять на отрезке $[a, x_c]$, длина которого в 2 раза меньше исходного отрезка. Формально условие выбора отрезка для продолжения поиска корня запишем так:

Если $f(a) * f(x_c) \le 0$, то
  Продолжить поиск на отрезке $[a, x_c]$.
иначе
  Продолжить поиск на отрезке $[x_c, b]$.
конец если.

Далее подвергнем уже новый отрезок делению пополам и вновь сделаем выбор относительно половины, на которой находится корень.

Такое деление продолжается до тех пор, пока длина результирующего отрезка превышает заданную достаточно малую величину ε (например, $\varepsilon = 0.0001$), которая характеризует точность вычислений.

Замечание. Другое название метода бисекций - *метод деления отрезка пополам*.

Из приведенного анализа метода бисекций видно, что, если функция непрерывна и отрезок $[a, b]$ является отрезком локализации, алгоритм чисто теоретически всегда сходится. Про такие методы говорят, что они обладают *глобальной сходимостью*.

В то же время если для выбора отрезка использовать выражение $f(a) * f(x_c) \le 0$, то есть опасность, что при умножении произойдет переполнение или исчезновение порядка. Например, последнее случится при работе с

одинарной точностью, если $f(a) = 10^{-30}$, $f(x_c) = -10^{30}$ и выполняется умножение $f(a) * f(x_c)$; результат будет положен равным нулю. Можно, конечно, завершить вычисления при обнаружении этого факта, но тогда результат окажется ниже заданной точности. Поэтому для выбора подходящей половины используется выражение SIGN(1.0, $f(a)$) * SIGN(1.0, $f(x_c)$) < 0.0, в котором встроенная функция SIGN вернет 1.0, если второй ее аргумент больше нуля или равен ему, и -1.0 - в противном случае.

Теперь, когда ясна идея алгоритма, приведем и сам *алгоритм*:

1. Начало.
2. Задать значения a и b границ отрезка и точность вычислений ε.
3. Принять $xa = a$, $xb = b$ и вычислить $ya = f(xa)$, $yb = f(xb)$.
4. Если $ya = 0$ или $yb = 0$, то корень найден.
5. Если ya и yb одного знака, то
   Вывести сообщение: Неверно задан отрезок.
   Останов.
   конец если 5°.
6. Пока $xb - xa > \varepsilon$, выполнять:
   $xc = 0.5 * (xa + xb)$! Умножение быстрее деления
   ! Введем для снижения вычислительных затрат
   ! промежуточную переменную yc
   $yc = f(xc)$
   ! Вместо приводящей к переполнению проверки $ya * yc \le 0$ используем:

6.1. Если ya и yc имеют разные знаки, то
   $xb = xc$
   иначе
   $xa = xc$, $ya = yc$
   конец если 6.1.
   конец цикла 6.
7. $xc = 0.5 * (xa + xb)$! xc - искомый корень
8. Вывести xc и $f(xc)$.
9. Конец.

Замечания:

1. Центр текущего отрезка xc правильнее вычислять по формуле

   $xc = xa + 0.5 * (xb - xa)$

   вместо обычной формулы

   $xc = 0.5 * (xa + xb)$,

   поскольку при вычислениях с округлениями по последней формуле xc может оказаться за пределами отрезка $[xa, xb]$, см. [5, разд. 1.7.2].

2. В выражении $xc = xa + 0.5 * (xb - xa)$ второе слагаемое не должно быть меньше величины ε_n = NEAREST(xa, 1.0) - xa, в сумме с которой xa не изменит своего значения [5, разд. 1.7.1].

Пример. Методом бисекций на отрезке [0, 3] определяется корень функции $y = 1/(1{,}2\operatorname{arctg} x + \sqrt{|x+1|}) - x$. Ее график приведен на рис. 3. 2.

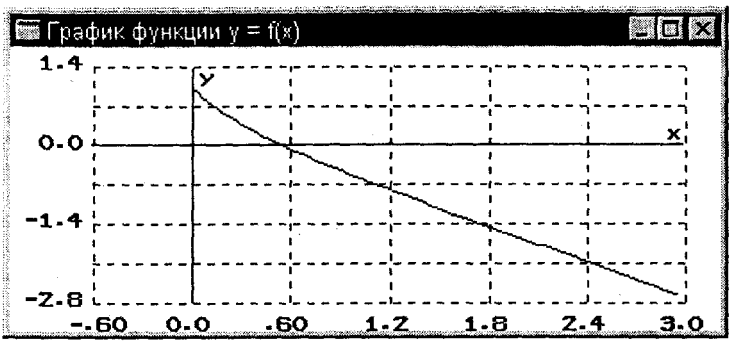


Рис. 3.2. График функции $y = 1/(1{,}2\operatorname{arctg} x + \sqrt{|x+1|}) - x$

```
! Вычисление корня функции y = f(x) методом бисекций
program rt1
 real(4) :: a, b, eps, xc                ! xc - искомый корень
 real(4), external :: f                  ! Теперь функцию f можно использовать
 logical(4) :: root                      ! в качестве фактического параметра
 a = 0.0; b = 3.0; eps = 1.0e-6
 if(root(f, a, b, eps, xc)) then         ! root - функция поиска корня
   print *, 'xc = ', xc, '; yc = ', f(xc)
 end if
end program rt1

! Функция root вернет .TRUE., если корень найден
! В противном случае она вернет .FALSE.
logical(4) function root(f, a, b, eps, xc)
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                       ! Для вывода русского текста
 real(4) :: f, xc, xa, xb, ya, dba
 xa = a; xb = b; ya = f(xa); yb = f(xb)
 root = sign(1.0, ya) * sign(1.0, yb) <= 0.0
 if(abs(yb) < tiny(0.0)) then            ! Корень - правая граница отрезка
   xc = xb
   return
 end if
 if(.not. root) then                     ! Ошибка при задании границ отрезка
```

```
  print *, trim(ru_doswin('Неверно заданы границы отрезка.', .false.))
  return
 end if
 do while(xb - xa > eps)                    ! Критерий окончания: xb - xa ≤ ε
  dba = 0.5 * (xb - xa)
  if(dba < nearest(xa, 1.0) - xa) then
   print *, trim(ru_doswin('Приращение меньше нормы.', .false.))
   exit
  end if
  xc = xa + dba
  yc = f(xc)
  ! Выбор нового отрезка
  if(sign(1.0, ya) * sign(1.0, yc) <= 0.0) then
   xb = xc
  else
   xa = xc; ya = yc
  end if
 end do
 xc = xa + 0.5 * (xb - xa)                  ! xc - приближение корня
end function root

function f(x)                               ! Оценка функции y = f(x)
 real(4) :: f, x
 f = 1.0 / (1.2 * atan(x) + sqrt(abs(x + 1.0))) - x
end function f
```

Результат:

```
xc =   5.435302E-01; yc =   4.723622E-08
```

Положительной стороной метода бисекций является его глобальная сходимость на отрезке локализации. К недостатком следует отнести его сравнительно невысокое быстродействие и отсутствие для него многомерных аналогов.

3.1.2.2. Метод Ньютона и метод секущих

Метод Ньютона - это метод, использующий специальную линеаризацию задачи, сводящую решение исходного нелинейного уравнения к решению последовательности линейных уравнений. То же справедливо и для метода секущих, который по существу является модификацией метода Ньютона.

Рассмотрим на рис. 3.3 график непрерывной функции $y = f(x)$, пересекающей в точке x_c ось x (т. е. x_c является корнем уравнения $f(x) = 0$).

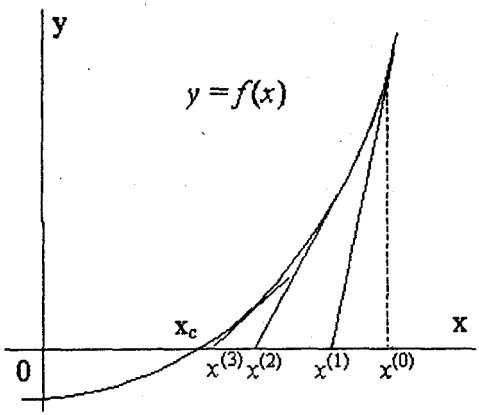


Рис. 3.3. Иллюстрация метода Ньютона

Выберем в качестве начального приближения точку $x^{(0)}$ и проведем касательную к графику функции $y = f(x)$ в точке с координатами $(x^{(0)}, f(x^{(0)}))$. Найдем точку $x^{(1)}$, в которой эта касательная пересекает ось x, и примем ее за новое приближение искомого корня.

Повторив этот процесс, получим точки $x^{(2)}$, $x^{(3)}$, ..., $x^{(n)}$, которые в нашем случае с каждым повторением все более и более приближаются к корню x_c.

Учитывая, что уравнение касательной к графику функции $y = f(x)$ в точке $x^{(k)}$ имеет вид

$$y = f(x^{(k)}) + f'(x^{(k)})(x - x^{(k)}),$$

найдем точку $x^{(k+1)}$ пересечения касательной с осью x. Точка $x^{(k+1)}$ в методе Ньютона принимается в качестве следующего приближения корня:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}. \qquad (3.2)$$

Замечание. Метод Ньютона имеет иное название - *метод касательных*.

Опираясь на приведенные соображения, запишем *алгоритм* метода касательных, применив в качестве критерия останова условие

$$| x^{(k+1)} - x^{(k)} | \le \varepsilon.$$

1. Начало.
2. Задать начальное приближение $x0$, точность вычислений ε и максимально допустимое число итераций *itmax*.
3. $k = 0$! k - число выполненных итераций
4. Найти $x1 = x0 - f(x0) / f'(x0)$.

5. Пока $|x1 - x0| > \varepsilon$, выполнять:
   $k = k + 1$
5.1. Если $k > itmax$, то ! Число итераций больше нормы
   Вывести сообщение: Корень не найден.
   Останов.
   конец если 5.1.
   $x0 = x1$
   $x1 = x0 - f(x0) / f'(x0)$! Следующее приближение корня
   конец цикла 5°.
6. Распечатать $x1$ и $f(x1)$. ! Принимаем $x1$ в качестве искомого корня
7. Конец.

Пример. Методом касательных вычисляется корень функции $y = x^3 - e^x$ (ее график приведен на рис. 3.4); в подпрограмму *root2*, вычисляющей корень, кроме функции, корень которой ищется, также передается и ее производная. (В нашем случае $y' = 3x^2 - e^x$.) В качестве начального приближения берется $x^{(0)} = 3.0$.

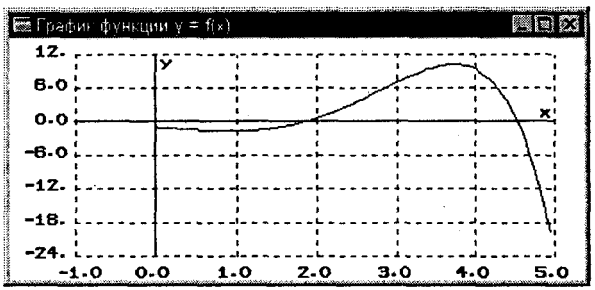


Рис. 3.4. График функции $y = x^3 - e^x$

```
! Поиск корня функции y = f(x) методом касательных
program rt2
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer              ! Для вывода русского текста
 real(4) :: eps, x0, xc         ! xc - искомый корень
 integer(4) :: itmax            ! Максимально допустимое число итераций
 real(4), external :: f, fd     ! f - функция, корень который ищется
 logical(4) :: root2            ! fd - функция, вычисляющая производную функции f
 x0 = 3.0                               ! Начальное приближение
 eps = 1.0e-6; itmax = 20               ! Точность и максимальное число итераций
 if(root2(f, fd, x0, eps, itmax, xc)) then  ! root2 - функция поиска корня
   print *, 'xc = ', xc, '; yc = ', f(xc)
 else
   print *, trim(ru_doswin('Корень не найден.', .false.))
 end if
```

```
end program rt2
! Функция root2 вернет .TRUE., если корень найден
! В противном случае она вернет .FALSE.
logical(4) function root2(f, fd, x0, eps, itmax, xc)
 real(4) :: f, fd, x0, eps, x1
 integer(4) :: itmax, k
 k = 0! Число выполненных итераций
 xc = x0 - f(x0) / fd(x0)                ! xc - следующее приближение
 do while(abs(xc - x0) > eps)            ! Критерий окончания - |xc- x0| ≤ ε или k > itmax
  k = k + 1
  if(k > itmax) exit                     ! Процесс не сошелся
  x0 = xc
  xc = x0 - f(x0) / fd(x0)               ! Следующее приближение корня
 end do
 root2 = k <= itmax                      ! Вернем .TRUE., если процесс сошелся
end function root2

function f(x)                            ! Оценка функции y = f(x)
 real(4) :: f, x
 f = x**3 - exp(x)
end function f

function fd(x)                           ! Оценка производной функции
 real(4) :: fd, x
 fd = 2.0 * x * x - exp(x)
end function fd
```

Результат:

```
x1 =      4:536404; y1 =  -1.510909E-05
```

Применение метода касательных не всегда приводит к успеху. Так, он не работает, если $f'(x) = 0$; если же $f'(x) \approx 0$, то метод может не сходиться. Про такие методы, обеспечивающие сходимость не изо всякой начальной точки, говорят, что они обладают *локальной сходимостью*. Кроме того, не всегда можно в явном виде задать функцию, вычисляющую производную $f'(x)$. В этом случае производные приближаются конечными разностями.

Другой способ избежать вычисления производных - это заменить в формуле Ньютона (3.2) $f'(x^{(k)})$ приближением $\dfrac{f(x^{(k-1)}) - f(x^{(k)})}{x^{(k-1)} - x^{(k)}}$, что даст расчетную формулу *метода секущих*:

$$x^{(k+1)} = x^{(k)} - \frac{x^{(k-1)} - x^{(k)}}{f(x^{(k-1)}) - f(x^{(k)})} f(x^{(k)})$$

Метод секущих является *двухшаговым*, поскольку для нахождения очередного приближения нужно знать два предшествующих. Методы бисекций и касательных - *одношаговые*. Геометрическая интерпретация метода секущих приведена на рис. 3.5.

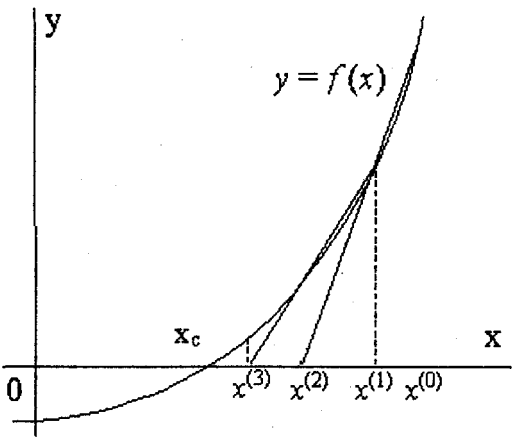


Рис. 3.5. Иллюстрация метода секущих

Так же как и метод Ньютона, метод секущих обладает локальной сходимостью.

3.1.2.3. Метод обратной квадратичной интерполяции

Идея метода состоит в том, чтобы по трем ранее найденным $x^{(k-2)}$, $x^{(k-1)}$ и $x^{(k)}$ приближениям корня $y = f(x)$ найти следующее приближение $x^{(k+1)}$. Для этого в методе обратной квадратичной интерполяции строится квадратный многочлен, такой, что

$$x^{(i)} = P_2(f(x^{(i)})),$$

где $i = k-2, k-1, k$. За очередное приближение корня принимается $x^{(k+1)} = P_2(0)$.

Метод является *трехшаговым*, поскольку для получения очередного приближения надо найти три предыдущих.

Необходимым условием построение многочлена $P_2(y)$ является различие значений $f(x^{(i)})$. Его выполнение будет обеспечено, если функция $y = f(x)$ на исследуемом отрезке строго монотонна.

Коэффициенты a, b, и c многочлена $P_2(y) = ay^2 + by + c$ получаются в результате решения системы линейных уравнений

$$\begin{aligned} a\left(y^{(k-2)}\right)^2 &+ by^{(k-2)} + c = x^{(k-2)}, \\ a\left(y^{(k-1)}\right)^2 &+ by^{(k-1)} + c = x^{(k-1)}, \\ a\left(y^{(k)}\right)^2 &+ by^{(k)} + c = x^{(k)}, \end{aligned}$$

в которой $y^{(i)} = f(x^{(i)})$, $i = k - 2, k - 1, k$.

Поскольку в качестве следующего приближения принимается $P_2(0)$, то нас интересует только свободный член многочлена $P_2(y)$, который найдем по формуле Крамера:

$$c = D_3 / D,$$

где

$$D = \begin{vmatrix} \left(y^{(k-2)}\right)^2 & y^{(k-2)} & 1 \\ \left(y^{(k-1)}\right)^2 & y^{(k-1)} & 1 \\ \left(y^{(k)}\right)^2 & y^{(k)} & 1 \end{vmatrix}$$

и

$$D_3 = \begin{vmatrix} \left(y^{(k-2)}\right)^2 & y^{(k-2)} & x^{(k-2)} \\ \left(y^{(k-1)}\right)^2 & y^{(k-1)} & x^{(k-1)} \\ \left(y^{(k)}\right)^2 & y^{(k)} & x^{(k)} \end{vmatrix}.$$

После преобразований получим:

$$x^{(k+1)} = P_2(0) = x^{(k-2)} \frac{y^{(k-1)} y^{(k)}}{(y^{(k-2)} - y^{(k-1)})(y^{(k-2)} - y^{(k)})} +$$

$$+ x^{(k-1)} \frac{y^{(k-2)} y^{(k)}}{(y^{(k-1)} - y^{(k-2)})(y^{(k-1)} - y^{(k)})} +$$

$$+ x^{(k)} \frac{y^{(k-2)} y^{(k-1)}}{(y^{(k)} - y^{(k-2)})(y^{(k)} - y^{(k-1)})}.$$

Метод обратной квадратичной интерполяции обладает локальной сходимостью и для начала его работы требуется задание трех хороших начальных приближений.

Пример. Методом обратной квадратичной интерполяции определяется корень функции $y = x^3 - e^x$. В качестве начальных приближений берутся $x^{(0)} = 5.0$, $x^{(1)} = 5.2$, $x^{(2)} = 5.4$.

```
program rt3
 real(4) :: eps, x0, x1, x2, xc              ! xc - искомый корень
 integer(4) :: itmax                         ! Максимально допустимое число итераций
 real(4), external :: f                      ! f - функция, корень который ищется
 logical(4) :: root3
 x0 = 5.0; x1 = 5.2; x2 = 5.4                ! Начальные приближения
 eps = 1.0e-6; itmax = 20                    ! Точность и максимальное число итераций
 if(root3(f, x0, x1, x2, eps, itmax, xc)) print *, 'xc = ', xc, '; yc = ', f(xc)
end program rt3

! root3 - функция поиска корня. Функция root3 вернет .TRUE.,
! если корень найден. В противном случае она вернет .FALSE.
logical(4) function root3(f, x0, x1, x2, eps, itmax, xc)
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                           ! Для вывода русского текста
 real(4) :: f, x0, x1, x2, eps, xc, y0, y1, y2, t01, t02, t12
 integer(4) :: itmax, k
 k = 0! Число выполненных итераций
 y0 = f(x0); y1 = f(x1); y2 = f(x2)
 t01 = y0 - y1; t02 = y0 - y2; t12 = y1 - y2
 if(check0( )) return                        ! Смотрим, может ли быть деление на нуль
 ! xc - следующее приближение
 xc = x0*(y1/t01)*(y2/t02) - x1*(y0/t01)*(y2/t12) + x2*(y0/t02)*(y1/t12)
 do while(abs(xc - x2) > eps)                ! Критерий окончания - |xc- x2| ≤ ε или k > itmax
  k = k + 1
  if(k > itmax) exit                         ! Процесс не сошелся
  x0 = x1; x1 = x2; x2 = xc
  y0 = y1; y1 = y2; y2 = f(x2)
  t01 = y0 - y1; t02 = y0 - y2; t12 = y1 - y2
  if(check0( )) return                       ! Смотрим, может ли быть деление на нуль
  ! xc - следующее приближение
  xc = x0*(y1/t01)*(y2/t02) - x1*(y0/t01)*(y2/t12) + x2*(y0/t02)*(y1/t12)
 end do
 root3 = k <= itmax                          ! Вернем .TRUE., если процесс сошелся
 if(.not. root3) then
  print *, trim(ru_doswin('Корень не найден.', .false.))
 end if

contains

 ! Функция вернет .TRUE., если t01, или t02, или t12 - машинный нуль
 logical(4) function check0( )
  check0 = abs(t01)< tiny(t01) .or. abs(t02)< tiny(t01) .or. abs(t12)< tiny(t01)
  if(check0) then
   print *, trim(ru_doswin('Деление на нуль.', .false.))
   root3 = .false.
  end if
```

```
 end function check0
end function root3

function f(x)                          ! Оценка функции y = f(x)
 real(4) :: f, x
 f = x**3 - exp(x)
end function f
```

Результат:

```
xc =  4.536404;   yc =  -3.263986E-08
```

Замечание. При вычислении *xc* операнды выражения группируются таким образом, что минимизируется опасность возникновения переполнения и исчезновения порядка.

3.1.3. ПОИСК КОМПЛЕКСНЫХ КОРНЕЙ

Комплексные корни в IMSL определяются методом Мюллера [29]. Метод является трехшаговым и основан на примерно тех же, что и метод обратной квадратичной интерполяции, идеях. В методе Мюллера для поиска очередного приближения корня по трем ранее найденным приближениям $z^{(k-2)}$, $z^{(k-1)}$ и $z^{(k)}$ строят интерполяционный многочлен $P_2(z)$ второй степени, для которого $P_2(z^{(i)}) = f(z^{(i)})$, $i = k-2, k-1, k$. За очередное приближение $z^{(k+1)}$ принимается тот из двух корней многочлена $P_2(z)$, который ближе расположен к $z^{(k)}$.

Коэффициенты c_1, c_2 и c_3 многочлена $P_2(z) = c_1z^2 + c_2z + c_3$ получаются в результате решения системы линейных уравнений

$$\begin{aligned} c_1\left(z^{(k-2)}\right)^2 &+ c_2 z^{(k-2)} + c_3 = y^{(k-2)}, \\ c_1\left(z^{(k-1)}\right)^2 &+ c_2 z^{(k-1)} + c_3 = y^{(k-1)}, \\ c_1\left(z^{(k)}\right)^2 &+ c_2 z^{(k)} + c_3 = y^{(k)}, \end{aligned} \qquad (3.3)$$

в которой $y^{(i)} = f(z^{(i)})$, $i = k-2, k-1, k$.

Пример. Методом Мюллера ищется комплексный корень уравнения $e^z - 0{,}2z + 1 = 0$. Для вычисления корней многочлена $P_2(z)$ вызывается подпрограмма LSLCG библиотеки IMSL. В качестве начального приближения берется $z^{(0)} = (1.0, 1.0)$.

```
program rt4
 integer(4) :: itmax                   ! Максимально допустимое число итераций
 integer(4) :: k                       ! Число выполненных итераций
 real(4) :: eps                        ! Точность вычислений
 complex(4) :: z0, zc                  ! xc - искомый корень
 complex(4), external :: f             ! f - функция, корень который ищется
```

```
 logical(4) :: root4
 z0 = (1.0, 1.0)                          ! Начальное приближение
 eps = 1.0e-6; itmax = 20                 ! Точность и максимальное число итераций
 z0 = (1.0, 1.0)                          ! Начальное приближение
 if(root4(f, z0, eps, itmax, zc, k)) print *, 'zc = ', zc, '; yc = ', f(zc), '; k = ', k
end program rt4

! root4 - функция поиска корня. Функция root4 вернет .TRUE.,
! если корень найден. В противном случае она вернет .FALSE.
logical(4) function root4(f, z0, eps, itmax, zc, k)
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                        ! Для вывода русского текста
 complex(4) :: f, z0, z1, z2, zc, y0, y1, y2
 real(4) :: eps                           ! b - вектор, содержащий y(i) системы (3)
 integer(4) :: itmax, k                   ! c - решение системы Ac = b
 z1 = z0-(0.1, 0.1); z2 = z0+(0.1, 0.1)   ! Еще два начальных приближения
 y0 = f(z0); y1 = f(z1); y2 = f(z2)
 ! Находим c1, c2 и c3 - коэффициенты многочлена P2(z)
 ! и вычисляем zc - следующее приближение
 call fzc( )
 k = 0! Число выполненных итераций
 ! Критерий окончания - |zc - z2| ≤ ε или k > itmax
 do while(abs(zc - z2) > eps)
  k = k + 1
  if(k > itmax) exit                      ! Процесс не сошелся
  z0 = z1; z1 = z2; z2 = zc               ! Приближаемся к корню
  y0 = y1; y1 = y2; y2 = f(z2)
  call fzc( )                             ! Формируем следующее приближение zc
 end do
 root4 = k <= itmax                       ! Вернем .TRUE., если процесс сошелся
 if(.not. root4) then
  print *, trim(ru_doswin('Корень не найден.', .false.))
 end if

contains

 subroutine fzc( )
  complex(4) :: d, zc1, zc2
  complex(4) :: a(3, 3), b(3), c(3)       ! Коэффициенты многочлена P2(z)
  real(4) :: dist1, dist2
  a(1, 1) = z0* z0;      a(1, 2) = z0;      a(1, 3) = 1
  a(2, 1) = z1* z1;      a(2, 2) = z1;      a(2, 3) = 1
  a(3, 1) = z2* z2;      a(3, 2) = z2;      a(3, 3) = 1
  b = (/ y0, y1, y2 /)                    ! Правая часть системы (3)
  call Lslcg(3, a, 3, b, 1, c)            ! Поиск коэффициентов многочлена P2(z)
  ! Решаем уравнение c1z^2 + c2z + c3 = 0
```

```
 d = sqrt(c(2) * c(2) - 4.0 * c(1) * c(3))
 zc1 = (-c(2) + d) / (2.0 * c(1)); zc2 = (-c(2) - d) / (2.0 * c(1))
 ! Вычисляем расстояния корней zc1 и zc2 до z2
 ! и находим zc - следующее приближение
 dist1 = abs(z2-zc1); dist2 = abs(z2-zc2)
 zc = zc1
 if(dist1 > dist2) zc = zc2
 end subroutine fzc
end function root4

function f(z)                          ! Оценка функции y = f(x)
 complex(4) :: f, z
 f = exp(z) - 0.2 * z + 1.0
end function f
```

Результат:

```
zc =  (1.066748E-01, 2.645904); yc =  (2.384186E-07,-5.960464E-08); k = 6
```

3.1.4. КРИТЕРИИ ОСТАНОВА

В приведенных примерах в качестве критерия завершения вычислений использовалось условие $|x^{(k)} - x^{(k-1)}| \le \varepsilon$, где k - номер последнего приближения корня. В процедурах IMSL отдается предпочтение критериям, задаваемым условиями

$$\left| \frac{x^{(k)} - x^{(k-1)}}{x^{(k)}} \right| \le \varepsilon_1$$

и

$$\left| f(x^{(k)}) \right| \le \varepsilon_2 .$$

Решение принимается, если выполняется хотя бы одно из них. Величины ε, ε_1 и ε_2 не следует брать меньшими машинной точности $\varepsilon_м$.

3.1.5. СКОРОСТЬ СХОДИМОСТИ АЛГОРИТМОВ

Введем применяемое для последовательности действительных чисел понятие сходимости с q-порядком, по меньшей мере равным p.

Говорят, что последовательность $\{x^{(k)}\} = \{x^{(0)}, x^{(1)}, x^{(2)}, ...\}$ сходится к $x^{(*)}$, если

$$\lim_{k \to \infty} \left| x^{(k)} - x^{(*)} \right| = 0 .$$

Если $\{x^{(k)}\}$ сходится к $x^{(*)}$ и существуют постоянные $p > 1$, $c \geq 0$ и $\widehat{k} \geq 0$, такие, что для всех $k \geq \widehat{k}$

$$\left|x^{(k)} - x^{(*)}\right| \leq c\left|x^{(k)} - x^{(*)}\right|^p,$$

то говорят, что $\{x^{(k)}\}$ сходится к $x^{(*)}$ с q-порядком, по меньшей мере равным p. Если $p = 2$, то говорят, что скорость сходимости является q-квадратичной; если $p = 3$, то она является q-кубичной. При $p = 1$ сходимость является q-линейной. Очевидно, что чем больше p, тем выше скорость сходимости.

Из рассмотренных алгоритмов наиболее медленным является метод бисекций, обладающий q-линейной сходимостью. Метод Ньютона обладает q-квадратичной локальной сходимостью. Для метода секущих известно, что он обладает локальной сходимостью с порядком $p = \left(\sqrt{5}+1\right)/2 \approx 1{,}618$, а метод обратной квадратичной интерполяции обладает локальной сходимостью с порядком $p \approx 1{,}839$.

3.2. РЕШЕНИЕ ТРАНСЦЕНДЕНТНЫХ УРАВНЕНИЙ С ОДНИМ НЕИЗВЕСТНЫМ ПРОЦЕДУРАМИ IMSL

3.2.1. СПИСОК И ОШИБКИ ПОДПРОГРАММ

Приводимые в табл. 3.1 подпрограммы возвращают корни трансцендентных уравнений с одним неизвестным. В табл. 3.2 приведены информационные ошибки, которые могут появляться при вызовах подпрограмм.

Таблица 3.1. Подпрограммы, возвращающие корни трансцендентных уравнений

Подпрограмма	*Назначение*
ZBREN (DZBREN)	Поиск корня вещественной функции, меняющей знак на заданном интервале
ZREAL (DZREAL)	Поиск вещественных корней методом Мюллера
ZANLY (DZANLY)	Поиск комплексных корней методом Мюллера

Таблица 3.2. Возможные ошибки подпрограмм из табл. 3.1

Тип	*Код*	*Причина ошибки*	*Где возникает*
3	1	Сходимость не достигнута за *itmax* итераций по крайней мере для одного из *nroot* (*nnew*) корней	ZREAL, ZANLY
4	1	Сходимость не достигнута за *maxfn* итераций	ZBREN

Замечания:

1. Подпрограммы табл. 3.1 могут быть использованы и для определения корней алгебраических уравнений (многочленов).
2. Вещественные параметры процедур имеют тип REAL(4) при работе с одинарной точностью и REAL(8) при работе с двойной. Аналогично комплексные параметры имеют тип COMPLEX(4) при работе с одинарной точностью и COMPLEX(8) при работе с двойной. Целые параметры процедур имеют тип INTEGER(4). Имена целых параметров начинаются с букв *i, j, k, l, m* или *n*.

3.2.2. ПОДПРОГРАММА ZBREN (DZBREN)

Выполняет поиск вещественного корня на заданном отрезке. Имеет вызов

CALL ZBREN(*f, errabs, errrel, a, b, maxfn*)

Параметры подпрограммы ZBREN:
Пользовательская функция: f.
Входные: errabs, errrel.
Входные/выходные: a, b, maxfn.

f - вещественная внешняя функция одного вещественного аргумента, корень которой нужно найти; имеет вид $f(x)$. Функция должна обладать атрибутом EXTERNAL. Ее аргумент нельзя изменять в теле функции *f*.

errabs - первый критерий завершения вычислений. Корень *b* принимается, если $|f(b)| \le errabs$. Причем можно задать $errabs = 0$.

errrel - относительная ошибка (второй критерий завершения вычислений). Корень принимается, если относительная разница между приближениями, найденными в двух последовательных аппроксимациях, меньше *errrel*.

a и *b* - начало и конец отрезка, на котором выполняется поиск вещественного корня, причем $f(a)$ и $f(b)$ должны иметь разный знак. На выходе и *a* и *b* отличны от начальных значений, причем *b* содержит приближенное значение корня функции *f*.

maxfn - на входе задает максимально допустимое число вычислений функции *f*; на выходе содержит число реальных вызовов функции *f*.

Если ZBREN завершается успешно, никаких сообщений не генерируется и *a* и *b* таковы, что:

1) $f(a) * f(b) \le 0.0$;
2) $|f(b)| \le |f(a)|$ и
3) либо $|f(b)| \le errabs$, либо $|a - b| \le \max(|b|, 0.1) * errrel$.

Подпрограмма ZBREN гарантирует сходимость в пределах *k* вычислений функции *f*, где

$$k = (\ln(b - a)/d) + 1.0)^2$$

и

$$d = \min_{x \in [a,b]} (\max(|x|, 0.1) * errrel) .$$

Однако на практике число обращений к функции f редко превышает $\sqrt{k}$. Значение d можно вычислить так:

```
p = max(0.1, min(abs(a), abs(b)))
if((a - 0.1) * (b - 0.1) < 0.0) p = 0.1
d = p * errrel
```

Описание:

Подпрограмма ZBREN применяет метод бисекций, линейную интерполяцию (метод секущих) и обратную квадратичную интерполяцию. На каждом шаге принимается решение, какой из трех методов будет использован для вычисления следующего приближения. Подробно алгоритм изложен в [9].

Пример. Определяется корень уравнения $x\lg x - 1 = 0$ на отрезке [1, 3]. График функции $y = x\lg x - 1$ дан на рис. 3.6.

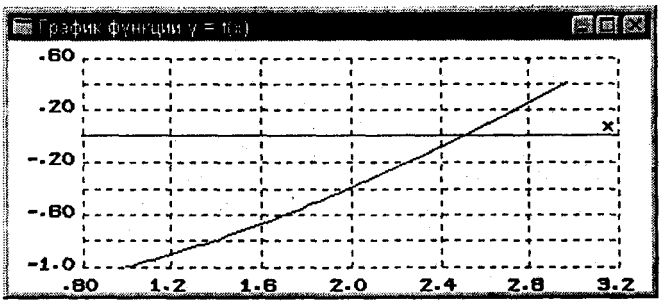


Рис. 3.6. График функции $y = x\lg x - 1$

```
program zer4
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer
 integer(4) :: maxfn
 real(4) :: a, b, errabs, errrel
 real(4), external :: f
 errabs = 0.0001; errrel = 0.0001          ! Параметры ZBREN
 a = 1.0; b = 3; maxfn = 100
 call zbren(f, errabs, errrel, a, b, maxfn)
 print *, trim(ru_doswin('Найден корень x0 =', .false.)), b
```

```
 print '(1x, a, i3)', trim(ru_doswin('Число обращений к функции: ', .false.)), maxfn
end program zer4

function f(x)
 real(4) :: f, x
 f = x * log10(x) - 1.0
end function f
```

Результат:

```
Найден корень x0 =      2.506193
Число обращений к функции:  5
```

3.2.3. ПОДПРОГРАММА ZREAL (DZREAL)

Выполняет поиск вещественных корней методом Мюллера. Имеет вызов

CALL ZREAL(*f*, *errabs*, *errrel*, *eps*, *eta*, *nroot*, *itmax*, *xguess*, *x*, *info*)

Параметры подпрограммы ZREAL:

Пользовательская функция: *f*.

Входные: *errabs*, *errrel*, *eps*, *eta*, *nroot*, *itmax*, *xguess*.

Входные/выходные: *x*, *info*.

f - вещественная внешняя функция одного вещественного аргумента, корни которой нужно найти; имеет вид *f*(*x*). Функция должна обладать атрибутом EXTERNAL. Ее аргумент нельзя изменять в теле функции *f*.

errabs - первый критерий завершения вычислений. Корень *b* принимается, если $|f(x(i)| \le errabs$.

errrel - относительная ошибка (второй критерий завершения вычислений). Корень *x*(*i*) принимается, если относительная разница между приближениями, найденными в аппроксимациях с номерами *k* - 1 и *k*, меньше *errrel*, т. е. $\left|\frac{x_i^{(k)} - x_i^{(k-1)}}{x_i^{(k-1)}}\right| < errrel$.

eps - см. *eta*.

eta - критерий локализации кратных корней. Если вычисляется корень *x*(*i*) и ABS(*x*(*i*) - *x*(*j*)) < *eps*, где *x*(*j*) - ранее найденный корень, то поиск *x*(*i*) начинается заново с начальным приближением, равным *x*(*i*) + *eta*.

nroot - число корней, которые должна найти ZREAL.

itmax - максимально допустимое число итераций на один корень.

xguess - вектор размера *nroot*, содержащий начальные приближения корней.

x - вектор размера *nroot*, содержащий вычисленные корни.

info - целочисленный вектор размера *nroot*. Элемент *info*(*j*) содержит число итераций, использованных при поиске *j*-го корня. Если, однако, про-

цесс поиска *j*-го корня не сошелся за *itmax* итераций, *info(j)* будет равен *itmax* + 1.

Замечания:

1. Подпрограмма ZREAL предполагает, что существует *nroot* различных вещественных корней функции *f* и к ним можно приблизиться от начальных (*xguess*) значений. Подпрограмма устроена так, что при одном вызове она никогда не сойдется к одному и тому же корню от разных начальных приближений.
2. Может потребоваться масштабирование вектора *x* в функции *f*, если известно, что какой-либо из корней меньше единицы.

Описание:

Подпрограмма ZREAL вычисляет, используя метод Мюллера, *n* вещественных корней функции *f*, начиная от заданных пользователем начальных приближений корней *x*(1), *x*(2), ..., *x*(*nroot*) [29].

Пример. Решается уравнение $4(1 - x^2) - e^x = 0$, имеющее два вещественных корня. В качестве начального приближения используется точка $x^{(0)} = 2.0$. График функции $y = 4(1 - x^2) - e^x = 0$ приведен на рис. 3.7.

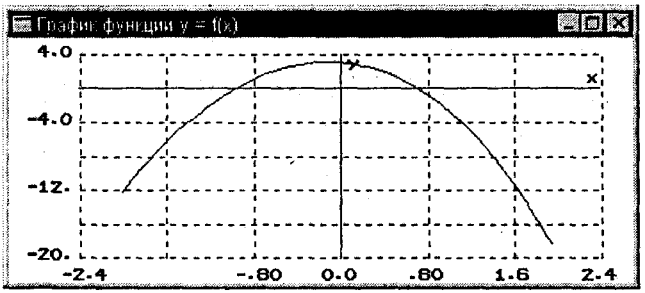


Рис. 3.7. График функции $y = 4(1 - x^2) - e^x$

```
program zer5
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                    ! Для вывода русского текста
 integer(4), parameter :: nroot = 2
 integer(4) :: info(nroot), itmax
 real(4) :: errabs, errrel, x(nroot), xguess(nroot)
 real(4), external :: f
 xguess = 2.0                         ! Начальные приближения
 errabs = 1.0e-5; errrel = 1.0e-5     ! Параметры ZREAL
 eps = 1.0e-5; eta = 1.0e-2; itmax = 100
 call zreal(f, errabs, errrel, eps, eta, nroot, itmax, xguess, x, info)
 call wrrrn(ru_doswin('Найдены корни', .false.), 1, nroot, x, 1, 0)
```

```
 call wrirn(ru_doswin('Число итераций', .false.), 1, nroot, info, 1, 0)
end program zer5

function f(x)
 real(4) :: f, x
 f = 4.0 * (1.0 - x * x) - exp(x)
end function f
```

Результат:

```
 Найдены корни
       1              2
 0.7034         -0.9505

Число итераций
       1          2
       7          8
```

Замечание. Если *nroot* - число определяемых корней - больше числа существующих вещественных корней, то может возникнуть ошибка переполнения, приводящая к останову программы. Так аварийное завершение произойдет, если в *zer5* задать *nroot* = 3. Сопровождающее сообщение таково:

```
run-time error M6104: MATH
- floating-point error: overflow
```

3.2.4. ПОДПРОГРАММА ZANLY (DZANLY)

Осуществляет вычисление комплексных корней методом Мюллера. Имеет вызов

CALL ZANLY(*f*, *errabs*, *errrel*, *nknown*, *nnew*, *nguess*, *zinit*, *itmax*, *z*, *info*)

Параметры подпрограммы ZANLY:
Пользовательская функция: *f*.
Входные: errabs, *errrel*, *nknown*, *nnew*, *nguess*, *zinit*, *itmax*.
Входные/выходные: z, *info*.

f - комплексная внешняя функция одного комплексного аргумента, корни которой нужно найти; имеет вид $f(x)$. Функция должна обладать атрибутом EXTERNAL. Ее аргумент нельзя изменять в теле функции *f*.

errabs - первый критерий завершения вычислений. Применяется так. Пусть $fp(z) = f(z)/p$, где $p = (z - z(1)) * (z - z(2)) *\ldots* (z - z(k - 1))$, здесь $z(1)$, ..., $z(k - 1)$ - ранее найденные корни. Если $|f(z)| \le errabs$ и $|fp(z)| \le errabs$, то z принимается в качестве корня.

errrel - относительная ошибка (второй критерий завершения вычислений). Корень принимается, если относительная разница между приближениями, найденными в двух последовательных аппроксимациях, меньше

errrel. Параметр *errrel* должен быть задан меньшим 0.01; в противном случае будет использовано значение 0.01.

nknown - число ранее найденных корней. Если *nknown* > 0, то найденные корни до вызова ZANLY должны быть занесены в *zinit*(1), ..., *zinit*(*nknown*).

nnew - число корней, которые должна найти ZANLY.

nguess - число начальных приближений. Сами приближения заносятся в *zinit*(*nknown* + 1), ..., *zinit*(*nknown* + *nguess*); *nguess* задается равным нулю при отсутствии приближений.

zinit - комплексный вектор размера *nknown* + *nnew*. Элементы *zinit*(1), ..., *zinit*(*nknown*) содержат ранее найденные корни; *zinit*(*nknown* + 1), ..., *zinit*(*nknown* + *nnew*) - начальные приближения для *nnew* корней, которые планируется найти, вызвав ZANLY.

itmax - максимально допустимое число итераций на один корень.

z - комплексный вектор размера *nknown* + *nnew*. Элементы *z*(1), ..., *z*(*nknown*) содержат ранее найденные корни; *z*(*nknown* + 1), ..., *z*(*nknown* + + *nnew*) - новые, найденные в текущем вызове ZANLY корни. Если *zinit* после вызова ZANLY не нужен, то на месте *zinit* можно использовать вектор *z*.

info - целочисленный вектор размера *nknown* + *nnew*. Элемент *info*(*j*) содержит число итераций, использованных при поиске *j*-го корня. Если, однако, процесс поиска *j*-го корня не сошелся за *itmax* итераций, *info*(*j*) будет больше *itmax*. Причем равенство *info*(*j*) = *itmax* + 1 означает, что сходимость не достигнута за *itmax* итераций. Возможно, следует для получения результата увеличить *itmax*. Из равенства *info*(*j*) = *itmax* + *k* (*k* > 1) следует, что сходимость достигнута на итерации *k* для функции $fp(z) = f(z) / ((z - z(1)) ... (z - z(j - 1)))$ с меньшим порядком, но не достигнута для $f(z)$. В таком случае можно попытаться задать более хорошие начальные приближения.

Подпрограмма ZANLY всегда возвращает в *z*(*j*) последнее приближение корня *j*, даже если сходимость не достигнута. Так же как и ZREAL, подпрограмма ZANLY использует метод Мюллера [29].

Пример. Решается уравнение $e^z - 0{,}2z + 1 = 0$, не имеющее вещественных корней.

```
program zer6
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer
 integer(4), parameter :: nknown = 0, nnew = 3
 integer(4) :: info(3), itmax, nguess
 real(4) :: errabs, errrel
 complex(4) :: z(nknown + nnew), zinit(nknown + nnew)
 complex(4), external :: f
 zinit = (/ (1.0,1.0), (1.0,1.0), (1.0,1.0) /) ! Начальные приближения
```

```
 errabs = 0.0001; errrel = 0.0001          ! Параметры ZANLY
 nguess = nnew; itmax = 100
 call zanly(f, errabs, errrel, nknown, nnew, nguess, zinit, itmax, z, info)
 call wrcrn(ru_doswin('Найдены корни', .false.), 1, nnew, z, 1, 0)
 call wrirn(ru_doswin('Число итераций', .false.), 1, nnew, info, 1, 0)
end program zer6

function f(z)
 complex(4) :: f, z
 f = exp(z) - 0.2 * z + 1.0
end function f
```

Результат:

```
                    Найдены корни
          1                    2                    3
( 0.107, 2.646)      ( 0.107,-2.646)      ( 0.632, 8.337)

 Число итераций
 1        2        3
 8        9        12
```

Замечание. Подпрограмму ZANLY можно, наряду с ZREAL, использовать и для определения вещественных корней. Так, если в *zer6* искать корни функции $4(1 - x^2) - e^x = 0$ из программы *zer5*, то будет получен *результат*:

```
                    Найдены корни
          1                    2                    3
( 0.70, 0.00)        ( -0.95, 0.00)       ( 6.56, 11.54)
```

3.3. ПОИСК КОРНЕЙ МНОГОЧЛЕНА

3.3.1. ВВЕДЕНИЕ

Общий вид алгебраического уравнения, левой частью которого является многочлен степени n:

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 = 0. \qquad (3.4)$$

В общем случае коэффициенты a_i могут быть комплексными числами.

Некоторые свойства уравнения (3.4) с вещественными коэффициентами:

1) уравнение степени n имеет n корней, среди которых могут быть как вещественные, так и комплексные;
2) комплексные корни образуют комплексно-сопряженные пары, т. е. каждому корню $x = c + id$ соответствует корень $x = c - id$;
3) коэффициенты многочлена, если $a_n = 1$, выражаются через его корни по формулам Вьета. В частности, произведение корней многочлена $(-1)^n x_1 x_2 \ldots x_n$ равно a_0.

В настоящее время задача поиска корней многочлена в известном смысле потеряла актуальность. Это объясняется тем, что после 60-х годов многочлены были вытеснены из той области вычислений, где они наиболее интенсивно применялись, - из задачи определения собственных значений матрицы. Ныне для ее решения применяются более совершенные и точные методы, основанные на *QR*-разложении матрицы (см. гл. 7 и 8). Тем не менее IMSL, как и другие библиотеки, содержит специальные методы вычисления корней многочлена, позволяющие решать эту задачу быстрее, чем описанные выше методы решения уравнения $f(x) = 0$. Отчасти ускорение достигается тем, что после вычисления очередного корня выполняется понижение степени многочлена. Причем если найденный корень комплексный и многочлен имеет вещественные коэффициенты, степень понижается сразу на 2.

3.3.2. ОПИСАНИЕ ПРОЦЕДУР, ВОЗВРАЩАЮЩИХ КОРНИ МНОГОЧЛЕНА

Приводимые в табл. 3.3 подпрограммы возвращают корни многочлена. Параметры подпрограмм описаны в табл. 3.4. В табл. 3.5 указаны ошибки, которые могут возникнуть при вызове подпрограмм из табл. 3.3.

Таблица 3.3. Подпрограммы, возвращающие корни многочлена

Подпрограмма	*Вызов*	*Назначение*
Многочлены с вещественными коэффициентами		
ZPLRC (DZPLRC)	CALL ZPLRC(*ndeg, coeff, root*)	Поиск корней многочлена методом Лагерра
ZPORC (DZPORC)	CALL ZPORC(*ndeg, coeff, root*)	Поиск корней многочлена методом Дженкинса -Трауба
Многочлены с комплексными коэффициентами		
ZPOCC (DZPOCC)	CALL ZPOCC(*ndeg, coeff, root*)	Поиск корней многочлена методом Дженкинса -Трауба

Таблица 3.4. Параметры подпрограмм, возвращающих корни многочлена

Имя	*Смысл/вид*	*Тип*
coeff	Вектор размера *ndeg* + 1, содержащий коэффициенты многочлена в порядке $a_0, a_1, \ldots, a_n$. Например, если решается уравнение $x^4 - x^3 - 4x^2 + 34x - 120 = 0$ и используется одинарная точность, то *coeff* определяется так: real(4) :: coeff(ndeg + 1) = (/ -120.0, 34.0, -4.0, -1.0, 1.0 /) / *входной*	REAL(4) или REAL(8); COMPLEX(4) или COMPLEX(8)

ndeg	Степень многочлена ($1 \le ndeg \le 100$) / *входной*	INTEGER(4)
root	Вектор размера *ndeg*, в который помещаются найденные корни / *выходной*	COMPLEX(4) или COMPLEX(8)

Таблица 3.5. Возможные ошибки подпрограмм из табл. 3.3

Тип	*Код*	*Причина ошибки*	*Где возникает*
3	1	Несколько старших коэффициентов многочлена равны нулю. По этой причине несколько последних корней будут возвращены равными машинной бесконечности	ZPLRC, ZPORC, ZPOCC
3	2	Число найденных корней меньше *ndeg*. Вектор будет содержать машинную бесконечность на месте ненайденных корней	То же

Примечания:

1. Подпрограмма ZPLRC является модификацией процедуры ZERPOL, [45], которая использует метод Лагерра. Метод обладает кубической сходимостью для некратных корней и линейной - для кратных. Максимальная длина шага между последовательными итерациями ограничена таким образом, что каждая новая итерация не выходит за пределы области, содержащей корень многочлена. При этом область локализации корня все время сужается. Корень считается найденным, когда значение многочлена становится меньше, чем вычисляемая для текущей итерации ошибка округления.
2. Подпрограмма ZPORC использует трехшаговый алгоритм Дженкинса - Трауба [21, 22].
3. Подпрограмма ZPOCC использует трехшаговый алгоритм Дженкинса - Трауба для многочленов с комплексными коэффициентами [22, 23]. Корни определяются в порядке увеличения их модулей.
4. Порядок многочлена понижается после нахождения вещественного корня или пары комплексно-сопряженных корней. Последующие корни вычисляются для многочлена меньшей степени.
5. Вещественные параметры процедур имеют тип REAL(4) при работе с одинарной точностью и REAL(8) при работе с двойной. Аналогично комплексные параметры имеют тип COMPLEX(4) при работе с одинарной точностью и COMPLEX(8) при работе с двойной. Целые параметры процедур имеют тип INTEGER(4). Как и ранее, имена целых параметров начинаются с букв *i*, *j*, *k*, *l*, *m* или *n*.

3.3.3. ПРИМЕРЫ ПРИМЕНЕНИЯ ПРОЦЕДУР IMSL, ВЫЧИСЛЯЮЩИХ КОРНИ МНОГОЧЛЕНОВ

Пример для ZPLRC. Найти корни уравнения $x^4 - x^3 - 4x^2 + 34x - 120 = 0$. График многочлена приведен на рис. 3.8.

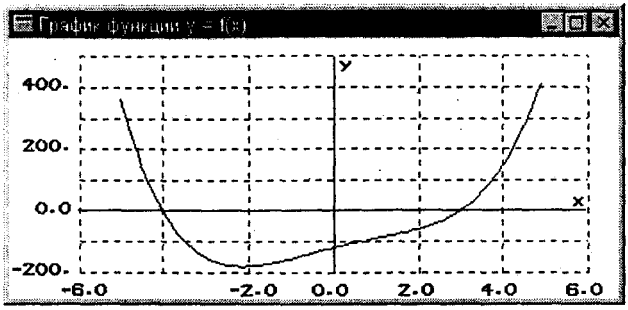


Рис. 3.8. График функции $y = x^4 - x^3 - 4x^2 + 34x - 120$

```
program zer1
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                    ! Для вывода русского текста
 integer(4), parameter :: ndeg = 4
 real(4) :: coeff(ndeg + 1) = (/ -120.0, 34.0, -4.0, -1.0, 1.0 /)
 complex(4) :: root(ndeg)
 call zplrc(ndeg, coeff, root)
 call wrcrn(ru_doswin('Найдены корни', .false.), 1, ndeg, root, 1, 0)
end program zer1
```

Результат:

```
                        Найдены корни
        1                2                3                4
 (-4.000,  .000)  ( 1.000, 3.000)  ( 1.000,-3.000)  ( 3.000,  .000)
```

Замечание. Подпрограмма ZPLRC, если использовать библиотеку, поставляемую с FPS 4.0, не найдет (попросту зависнет) корни уравнения $x^4 - 10x^3 + 35x^2 - 50x + 24 = (x - 1)(x - 2)(x - 3)(x - 4) = 0$. В случае CVF такой проблемы нет.

Пример для ZPORC. Решается уравнение $(x - 1)(x - 2)(x - 3)(x - 4) = 0$.

```
program zer2
 use dfimsl
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                    ! Для вывода русского текста в DOS-окно
 integer(4), parameter :: ndeg = 4
 real(4) :: coeff(ndeg + 1) = (/ 24.0, -50.0, 35.0, -10.0, 1.0 /)
```

```
 complex(4) :: root(ndeg)
 call zporc(ndeg, coeff, root)
 call wrcrn(ru_doswin('Найдены корни', .false.), 1, ndeg, root, 1, 0)
end program zer2
```

Результат:

```
                              Найдены корни
          1                   2                   3                   4
  (1.00, 0.00)        (2.00, 0.00)        (3.00, 0.00)        (4.00, 0.00)
```

Пример для ZPOCC. Решается уравнение $z^3 - (3 + 6i)z^2 - (8 - 12i)z + 10 = 0$; используется двойная точность.

```
program zer3                          ! Используем двойную точность
 use dfimsl
 use text_transfer
 integer(4), parameter :: ndeg = 3
 complex(8) :: coeff(ndeg+1) = (/ (10.0, 0.0), (-8.0, 12.0), (-3.0, -6.0), (1.0, 0.0) /)
 complex(8) :: root(ndeg)
 call dzpocc(ndeg, coeff, root)
 call dwrcrn(ru_doswin('Найдены корни', .false.), 1, ndeg, root, 1, 0)
end program zer3
```

Результат:

```
                    Найдены корни
          1                   2                   3
  (1.000, 1.000)      (1.000, 2.000)      (1.000, 3.000)
```

3.4. СИСТЕМЫ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

3.4.1. МЕТОД НЬЮТОНА ДЛЯ СИСТЕМ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

Пусть задана система из n нелинейных уравнений с n неизвестными

$$\begin{aligned} &f_1(x_1, x_2, \ldots, x_n) = 0, \\ &f_2(x_1, x_2, \ldots, x_n) = 0, \\ &\ldots \\ &f_n(x_1, x_2, \ldots, x_n) = 0. \end{aligned} \qquad (3.5)$$

Точное решение системы $\bar{x} = (\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)^T$, удовлетворяющее уравнениям (3.5), удается найти крайне редко. Поэтому на практике ищется приближенное решение $x^* = (x_1^*, x_2^*, \ldots, x_n^*)^T$, находящееся в некоторой δ-окрестности $\bar{x}$.

В основу многих современных алгоритмов (квазиньютоновские алгоритмы, методы секущих) положены идеи метода Ньютона для систем нелинейных уравнений, который, как и все иные алгоритмы для подобных систем, относится к классу итерационных алгоритмов. (В отличие от систем линейных уравнений прямые методы для нелинейных систем неприменимы.)

Пусть найдено приближение $x^{(k)}$ к решению $\bar{x}$ системы (3.5). Возникает вопрос: "Каким должно быть следующее приближение $x^{(k+1)}$, которое расположено к $\bar{x}$ ближе, чем $x^{(k)}$?"

Разложим левые части системы (3.5) в ряд Тейлора в точке $x^{(k)}$, ограничившись линейными членами разложения:

$$f_1(x_1, x_2, \ldots, x_n) = f_1(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}) + \frac{\partial f_1}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_1}{\partial x_2}\Delta x_2^{(k)} + \ldots + \frac{\partial f_1}{\partial x_n}\Delta x_n^{(k)},$$

$$f_2(x_1, x_2, \ldots, x_n) = f_2(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}) + \frac{\partial f_2}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_2}{\partial x_2}\Delta x_2^{(k)} + \ldots + \frac{\partial f_2}{\partial x_n}\Delta x_n^{(k)},$$

$\ldots$

$$f_n(x_1, x_2, \ldots, x_n) = f_n(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}) + \frac{\partial f_n}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_n}{\partial x_2}\Delta x_2^{(k)} + \ldots + \frac{\partial f_n}{\partial x_n}\Delta x_n^{(k)}.$$

Поскольку левые части получившегося разложения должны равняться нулю, приходим к системе линейных n уравнений с n неизвестными

$$\begin{aligned}
&\frac{\partial f_1}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_1}{\partial x_2}\Delta x_2^{(k)} + \ldots + \frac{\partial f_1}{\partial x_n}\Delta x_n^{(k)} = -f_1(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}),\\
&\frac{\partial f_2}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_2}{\partial x_2}\Delta x_2^{(k)} + \ldots + \frac{\partial f_2}{\partial x_n}\Delta x_n^{(k)} = -f_2(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}),\\
&\ldots\\
&\frac{\partial f_n}{\partial x_1}\Delta x_1^{(k)} + \frac{\partial f_n}{\partial x_2}\Delta x_2^{(k)} + \ldots + \frac{\partial f_n}{\partial x_n}\Delta x_n^{(k)} = -f_n(x_1^{(k)}, x_2^{(k)}, \ldots, x_n^{(k)}),
\end{aligned} \qquad (3.6)$$

решая которую относительно неизвестных $\Delta x_i^{(k)}$, $i = 1, \ldots, n$, найдем приращения, которые употребим для вычисления следующего приближения $x_i^{(k+1)} = x_i^{(k)} + \Delta x_i^{(k)}$, $i = 1, \ldots, n$.

Замечание. В процедурах IMSL для вектора приращений $\Delta x^{(k)}$ используется обозначение $s^{(k)}$.

Матрица

$$J(x^{(k)}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \dots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \dots & \frac{\partial f_2}{\partial x_n} \\ \dots & \dots & \dots & \dots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_n} & \dots & \frac{\partial f_n}{\partial x_n} \end{pmatrix} \tag{3.7}$$

коэффициентов системы (3.6) называется *матрицей Якоби*.

Таким образом, в методе Ньютона на каждом шаге нелинейная задача (3.5) заменяется линейной задачей (3.6), в результате решения которой определяется очередное приближение. Метод имеет q-квадратичную сходимость, что позволяет использовать простой критерий прекращения счета

$$\left\| x^{(k+1)} - x^{(k)} \right\| < \varepsilon .$$

Если использовать 1-норму, то критерий окончания запишется так:

$$\max_i \left(\left| \Delta x_i^{(k)} \right| \right) < \varepsilon . \tag{3.8}$$

Из описания метода следует, что он применим, если функции f_i системы (3.5) непрерывно дифференцируемы и на каждом его шаге определитель матрицы (3.7) отличен от нуля (т. е. система (3.6) не является вырожденной). Больше того, для получения надежного решения матрица (3.7) должна быть хорошо обусловленной.

Из приведенного материала видно, что использование метода Ньютона для систем нелинейных уравнений сопряжено с определенными трудностями:

1) на каждой итерации необходимо вычислять $J(x^{(k)})$;
2) на каждой итерации приходится решать систему линейных уравнений, которая может быть вырожденной или плохо обусловленной;
3) метод для многих задач не обладает глобальной сходимостью в том смысле, что решение системы (3.5) может быть получено лишь от удачно выбранных начальных приближений, найти которые в многомерном случае гораздо сложнее, чем в одномерном.

Существующие модификации метода во многом устраняют перечисленные недостатки, сохраняя присущее методу Ньютона для многомерного случая достоинство - быструю сходимость из хорошего начального приближения [6].

Пример. Найти методом Ньютона решение системы нелинейных уравнений

$$\begin{aligned} x_1^2 &+ x_2^2 + x_3^2 = 1, \\ 2x_1^2 &+ x_2^2 - 4x_3 = 0, \\ 3x_1^2 &- 4x_2 + x_3^2 = 0, \end{aligned} \qquad (3.9)$$

исходя из начального приближения $x_1^{(0)} = x_2^{(0)} = x_3^{(0)} = 0{,}5$.

```
program Newton
 use dfimsl
 integer(4), parameter :: n = 3          ! n - число уравнений
 integer(4) :: k, itmax = 100            ! itmax - максимально допустимое число итераций
 real(4) :: fjac(n, n), f(n), x(n), xguess(n), s(n), eps
 external fcn, jac                       ! Обязательный атрибут для подпрограмм fcn и jac
 xguess = 0.5                            ! Начальное приближение
 eps = 1.0e-5                            ! Точность
 k = 0! Число итераций
 do
  ! Вычисляем элементы матрицы Якоби в точке xguess
  call jac(n, xguess, fjac)
  ! Находим значения левых частей системы (3.8) в точке xguess
  call fcn(n, xguess, f)
  ! Решаем систему линейных уравнений J(x^(k))s^(k) = -f^(k)
  ! и находим вектор приращений s
  call Lslrg(n, fjac, n, -f, 1, s)
  x = xguess + s                         ! Вычисляем следующее приближение
  k = k + 1                              ! Число выполненных итераций
  if(all(abs(x - xguess) < eps) .or. k > itmax) exit
  xguess = x
 end do
 if(k > itmax) stop 'k > itmax'
 print '(1x, a, i3)', 'k = ', k          ! Число итераций
 call wrrrn('x', 1, n, x, 1, 0)          ! Вывод результата
 ! Находим для проверки значения левых частей системы (3.9) в точке x
 call fcn(n, x, f)
 ! Вывод значений функций в точке x. Они должны быть близки к нулю
 call wrrrn('f', 1, n, f, 1, 0)
end program Newton

subroutine fcn(n, x, f)                 ! Подпрограмма вычисления функций
 integer(4) :: n
 real(4) :: x(n), f(n)
 f(1) = x(1) * x(1) + x(2) * x(2) + x(3) * x(3) - 1.0
 f(2) = 2.0 * x(1) * x(1) + x(2) * x(2) - 4.0 * x(3)
 f(3) = 3.0 * x(1) * x(1)- 4.0 * x(2)+ x(3) * x(3)
end subroutine fcn
```

```
subroutine jac(n, x, fjac)                ! Подпрограмма, вычисляющая
 integer(4) :: n                          ! элементы матрицы Якоби
 real(4) :: x(n), fjac(n, n)
 fjac = reshape((/      2.0*x(1),    2.0*x(2),    2.0*x(3),          &
                        4.0*x(1),    2.0*x(2),      -4.0,            &
                        6.0*x(1),      -4.0,      2.0*x(3)  /),      &
                shape = (/ n, n /), order = (/ 2, 1 /))
end subroutine jac
```

Результат:

```
k =  5

                    x
          1                 2                 3
     0.7852            0.4966            0.3699

                    f
          1                 2                 3
  2.554E-08         1.341E-07         1.421E-07
```

3.4.2. КВАЗИНЬЮТОНОВСКАЯ СХЕМА ДЛЯ СИСТЕМ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

В квазиньютоновских алгоритмах есть возможность применения как рассмотренной выше ньютоновской стратегии, так и некоторой глобальной стратегии, позволяющей, во-первых, работать с плохо обусловленными матрицами Якоби и, во-вторых, решать проблему глобальной сходимости, т. е. обеспечивать сходимость к некоторому решению (если оно существует) из произвольной начальной точки.

Алгоритм квазиньютоновской схемы для k-й итерации решения системы (3.5):

1. Вычислить $f_i^{(k)}, i = 1, \ldots, n$
2. Сформировать матрицу $J(x^{(k)})$ или ее аппроксимацию.
3. Применить к $J(x^{(k)})$ некоторый способ разложения на множители, например QR-разложение, оценить ее число обусловленности и, если она плохо обусловлена, внести в нее соответствующее возмущение. (QR-разложение матрицы $A + \alpha xy^T$ по известному QR-разложению матрицы A возвращает подпрограмма LUPQR библиотеки IMSL.)
4. Решить систему $J(x^{(k)})s^{(k)} = -f^{(k)}$.
5. Вычислить $x^{(k+1)}$, сделав ньютоновский шаг - $x^{(k+1)} = x^{(k)} + s^{(k)}$, либо выбрать $x^{(k+1)}$ согласно используемой глобальной стратегии.
6. Принять решение - остановиться или продолжить вычисления.

С различными вариантами квазиньютоновских стратегий можно познакомиться, обратившись к [6].

3.5. ПРОЦЕДУРЫ IMSL ДЛЯ СИСТЕМ НЕЛИНЕЙНЫХ УРАВНЕНИЙ

3.5.1. СПИСОК И ОШИБКИ ПОДПРОГРАММ БИБЛИОТЕКИ IMSL

Перечень подпрограмм, решающих системы нелинейных уравнений, приводится в табл. 3.6; ошибки, которые могут возникнуть при вызове подпрограмм, - в табл. 3.7.

Таблица 3.6. Подпрограммы, решающие системы нелинейных уравнений

Подпрограммы	*Назначение*
NEQBF (DNEQBF) или N2QBF (DN2QBF)	Решают систему нелинейных уравнений методом секущих, применяя конечно-разностную аппроксимацию для вычисления матрицы Якоби
NEQBJ (DNEQBJ) или N2QBJ (DN2QBJ)	Решают систему нелинейных уравнений методом секущих, используя для вычисления матрицы Якоби пользовательскую подпрограмму
NEQNF (DNEQNF) или N2QNF (DN2QNF)	Решают систему нелинейных уравнений, применяя модифицированный гибридный алгоритм Пауэлла и конечно-разностную аппроксимацию для вычисления матрицы Якоби
NEQNJ (DNEQNJ) или N2QNJ (DN2QNJ)	Решают систему нелинейных уравнений, применяя модифицированный гибридный алгоритм Пауэлла; для вычисления матрицы Якоби применяется пользовательская подпрограмма

Таблица 3.7. Возможные ошибки подпрограмм из табл. 3.6

Тип	*Код*	*Причина ошибки*	*Где возникает*
3	1	Либо на последнем глобальном шаге не удалось достаточно уменьшить 2-норму $f(x)$, либо текущая точка находится рядом с корнем $f(x)$ и большей точности получить нельзя, либо аппроксимация матрицы Якоби методом секущих неточна, либо допуск для шага слишком велик	NEQBF, NEQBJ
3	3	Масштабированное расстояние между двумя последними шагами меньше, чем допуск для шага; вероятно, что текущая точка является приближенным корнем $f(x)$ (если только допуск *steptl* не слишком велик)	То же
3	4	Превышено максимально допустимое число итераций	"

3	5	Превышено максимально допустимое число вычислений функций	NEQBF, NEQBJ
3	7	Было выполнено 5 последовательных шагов размера *steptx*, однако либо 2-норма $f(x)$ приближается к конечной величине в некотором направлении, либо максимально допустимый размер шага *steptx* слишком мал	То же
4	1	Число вызовов FCN - подпрограммы, вычисляющей функции, превысило *itmax*(n + 1). Следует испробовать новое начальное приближение	NEQNF, NEQNJ
4	2	Значение *errrel* слишком мало. Дальнейшее улучшение решения невозможно	То же
4	3	Итерации не приводят к существенным улучшениям. Следует задать новое начальное приближение	"

Примечание. Вещественные параметры процедур имеют тип REAL(4) при работе с одинарной точностью и REAL(8) при работе с двойной. Целые параметры процедур имеют тип INTEGER(4). Как и всегда, имена целых параметров начинаются с букв *i*, *j*, *k*, *l*, *m* или *n*.

3.5.2. ПОДПРОГРАММЫ NEQBF (DNEQBF) И NEQBJ (DNEQBJ)

Подпрограмма NEQBF (DNEQBF) решает систему нелинейных уравнений методом секущих, используя конечно-разностную аппроксимацию для вычисления матрицы Якоби. Имеет вызов

CALL NEQBF(*fcn*, *n*, *xguess*, *xscale*, *fscale*, *iparam*, *rparam*, *x*, *fvec*)

Подпрограмма NEQBJ (DNEQBJ) решает систему нелинейных уравнений методом секущих, применяя для вычисления матрицы Якоби заданную пользователем подпрограмму *jac*. Имеет вызов

CALL NEQBJ(*fcn*, *jac*, *n*, *xguess*, *xscale*, *fscale*, *iparam*, *rparam*, *x*, *fvec*)

Параметры подпрограмм NEQBF и NEQBJ:
Пользовательские подпрограммы: fcn, jac.
Входные: *n*, *xguess*, *xscale*, *fscale*.
Входные/выходные: iparam, *rparam*.
Выходные: x, *fvec*.

fcn - имя подпрограммы, вычисляющей функции системы нелинейных уравнений. Подпрограмма должна обладать атрибутом EXTERNAL. Имеет заголовок

SUBROUTINE *fcn*(*n*, *x*, *f*)

Параметры подпрограммы fcn:
Входные: *n*, *x*.

Выходной: *f*.

n - размер векторов *x* и *f*; равен числу уравнений в системе (числу неизвестных).

x - вещественный вектор, содержащий координаты точки, в которой вычисляются значения функций; *x* не должен изменяться в подпрограмме *fcn*.

f - вещественный вектор, содержащий значения функций, вычисленные в точке *x*.

jac - пользовательская подпрограмма, вычисляющая элементы матрицы Якоби в заданной точке. Подпрограмма должна обладать атрибутом EXTERNAL. Имеет заголовок

SUBROUTINE jac(n, x, fjac, Ldfjac)

Параметры подпрограммы jac:

Входные: n, x, Ldfjac.

Выходной: fjac.

Тип *x* и *fjac* - REAL(4) или REAL(8); тип *n* и *Ldfjac* - INTEGER(4).

n - размер вектора *x*.

x - вещественный вектор, содержащий координаты точки, в которой вычисляются элементы матрицы Якоби. Не должен изменяться подпрограммой JAC.

fjac - вычисленная в точке *x* *n* × *n*-матрица Якоби.

Продолжение описания параметров подпрограмм NEQBF и NEQBJ:

Ldfjac - ведущий размер матрицы Якоби; обычно равен *n*.

xguess - вещественный вектор размера *n*, содержащий начальные приближения корней.

xscale - вещественный вектор размера *n*, содержащий диагональ матрицы масштабирования переменных. Вектор *xscale* преимущественно используется для масштабирования расстояний между точками. При отсутствии информации значения всех его элементов следует принять равными 1.0. Если задать *iparam*(6) = 1, то будет выполняться внутреннее масштабирование.

fscale - вещественный вектор размера *n*, содержащий диагональ матрицы масштабирования для функций. Вектор *fscale* главным образом используется для масштабирования разностей функций. При отсутствии информации значения всех его элементов следует принять равными 1.0.

iparam - целочисленный вектор параметров размера 6. Для задания значений *iparam* и *rparam* по умолчанию следует положить *iparam*(1) = 0.

rparam - вещественный вектор параметров размера 5. Описание *iparam* и *rparam* см. ниже.

x - вещественный вектор размера *n*, содержащий приближения корней.

fvec - вещественный вектор размера *n*, содержащий значения функций для найденных корней.

Автоматически для решения предоставляется память:

- $2n^2 + 11n$ байт в случае NEQBF и NEQBJ;
- $4n^2 + 22n$ байт в случае DNEQBF и DNEQBJ.

Память можно выделить явно, применив N2QBF (DN2QBF) или N2QBJ (DN2QBJ):

CALL N2QBF(*fcn*, *n*, *xguess*, *xscale*, *fscale*, *iparam*, *rparam*, *x*, *fvec*, *wk*, *Lwk*)

CALL N2QBJ(*fcn*, *jac*, *n*, *xguess*, *xscale*, *fscale*, &
param, *rparam*, *x*, *fvec*, *wk*, *Lwk*)

Дополнительные параметры подпрограмм N2QBF и N2QBJ:

wk - вещественный *рабочий вектор* размера *Lwk*. На выходе *wk* содержит:

- в элементах с $2n + 1$ по $3n$ последний выполненный шаг;
- в следующих *n* элементах последний ньютоновский шаг;
- в последних n^2 ячейках элементы матрицы Якоби в точке найденного приближения.

Lwk (*входной*) - размер вектора *wk*, $Lwk \geq 2n^2 + 11n$.

Останов NEQBF происходит, когда значения масштабированных норм функций становятся меньше масштабированных допусков, задаваемых *rparam*(1).

Если есть желание использовать заданные по умолчанию значения параметров, то установите *iparam*(1) = 0 и вызовите NEQBF. В противном случае, если задаются пользовательские значения *iparam* или *rparam*, нужно до вызова NEQBF вызвать N4QBJ:

CALL N4QBJ(*iparam*, *rparam*)

чтобы установить в *iparam* и *rparam* значения по умолчанию, а затем изменить подлежащие настройке элементы этих векторов, имея в виду что:

- *iparam*(1) - флаг инициализации; устанавливается равным нулю, если *iparam* и *rparam* должны содержать заданные по умолчанию величины;
- *iparam*(2) - число значащих десятичных цифр функций; значение по умолчанию зависит от типа ЭВМ;
- *iparam*(3) - максимальное число итераций; значение по умолчанию - 100;
- *iparam*(4) - максимальное число вычислений функций (вызовов FCN); значение по умолчанию - 400;
- *iparam*(5) - максимальное число вычислений матрицы Якоби; по умолчанию NEQBF эту настройку не использует;

- *iparam*(6) - флаг внутреннего масштабирования. Если *iparam*(6) = 1, то значения *xscale* устанавливаются NEQBF; значение по умолчанию - 0;
- *rparam*(1) - масштабированный допуск для функций. Масштабированная норма функций равна

$$\max_j(|f_i| * fs_i),$$

где f_i - i-й компонент вектора f, а fs_i - i-й компонент вектора *fscale*; значение по умолчанию равно $\sqrt{\varepsilon_м}$; здесь и далее $\varepsilon_м$ - машинная точность;

- *rparam*(2) - масштабированный допуск *steptl* для шага. Масштабированная норма шага между точками x и y равна

$$\max_j\left(\frac{|x_i - y_i|}{\max(x_i, 1/s_i)}\right),$$

где s_i - i-й компонент *xscale*; значение по умолчанию - $\varepsilon_м^{2/3}$;

- *rparam*(3) - допуск для ложной сходимости; по умолчанию NEQBF эту настройку не использует;
- *rparam*(4) - максимально допустимый размер *steptmx* шага; значение по умолчанию - 1000 * max(ε_1, ε_2), где $\varepsilon_1 = \sqrt{\sum_{i=1}^{n}(s_i t_i)^2}$, $\varepsilon_2 = \|s\|_2$, s = *xscale*, t = *xguess*;
- *rparam*(5) - размер начальной доверительной области; значение по умолчанию основано на начальном масштабированном шаге Коши.

Замечания:

1. Если используется двойная точность, то вызывается DN4QBJ и *rparam* имеет тип REAL(8).
2. Чтобы изменить заданные по умолчанию настройки вывода и останова подпрограммы, вызывайте подпрограмму ERSET [5, разд. 3.4.3].

Описание:

Подпрограмма NEQBF (NEQBJ) основана на методе секущих для решения системы нелинейных уравнений $f(x) = 0$, где $f: R^n \to R^n$ - и $x \in R^n$.

Начиная с текущей точки алгоритм использует метод, применяющий шаг с двойным изломом, для приближенного решения проблемы $\min_{x^{(k)} \in R^n} \left\| f(x^{(k)}) + J(x^{(k)}) s^{(k)} \right\|_2$ при условии, что $\|s^{(k)}\|_2 \le \delta^{(k)}$, и получения в результате направления $s^{(k)}$, где $f(x^{(k)})$ и $J(x^{(k)})$ - соответственно значение вектор-функций и приближение матрицы Якоби, вычисленные в текущей точке $x^{(k)}$. Затем вычисляются значения функций в точке $x^{(k+1)} = x^{(k)} + s^{(k)}$ и прини-

мается решение, следует ли принять новую точку $x^{(k+1)}$ в качестве очередного приближения или нет.

Когда точка $x^{(k+1)}$ отвергается, подпрограмма уменьшает размер доверительной области $\delta^{(k)}$ и возвращается назад, чтобы повторить вычисления. Процедура повторяется, пока не находится более хорошая точка.

Если новая точка удовлетворяет критерию останова, то алгоритм завершится. В противном случае приближение матрицы Якоби обновляется по формуле Бройдена:

$$J(x^{(k+1)}) = J(x^{(k)}) + \frac{(y - J(x^{(k)})s^{(k)})s^{(k)^T}}{s^{(k)^T}s^{(k)}},$$

где $y = f(x^{(k+1)}) - f(x^{(k)})$.

Алгоритм продолжается, используя в качестве новой точки $x^{(k+1)} \leftarrow x^{(k)}$. С деталями алгоритма можно познакомиться в [6, гл. 8].

Поскольку в случае NEQBF для вычисления матрицы Якоби применяются конечные разности, то при одинарной точности приближение матрицы Якоби может оказаться неудовлетворительным, вследствие чего алгоритм завершится далеко от корня. В таких случаях следует употреблять двойную точность. Когда же пользователь может обеспечить точное вычисление матрицы Якоби, лучше заменить NEQBF на NEQBJ.

Пример для NEQBF. Решается система нелинейных уравнений с четырьмя неизвестными

$$x_1 + 2x_2 + x_3 + 4x_4 - 20.7 = 0,$$
$$x_1^2 + 2x_1x_2 + x_4^3 - 15.88 = 0,$$
$$x_1^3 + x_3^2 + x_4 - 21.218 = 0,$$
$$3x_2 + x_3x_4 - 21.1 = 0$$

с начальным приближением (1.0, 1.0, 1.0, 1.0).

```
program ne1
 use dfimsl
 integer(4), parameter :: n = 4              ! n - число уравнений
 integer(4) :: iparam(6)
 real(4) :: fscale(n), fvec(n), rparam(5), x(n), xguess(n), xscale(n)
 external fcn                                ! Обязательный атрибут для fcn
 xguess = 1.0                                ! Начальное приближение
 xscale = 1.0; fscale = 1.0
 iparam(1) = 0                               ! Используем установки по умолчанию
 ! Поиск решения
 call neqbf(fcn, n, xguess, xscale, fscale, iparam, rparam, x, fvec)
```

```
 call wrrrn('x', 1, n, x, 1, 0)              ! Вывод результата
 call wrrrn('fvec', 1, n, fvec, 1, 0)        ! Вывод невязки
end program ne1

subroutine fcn(n, x, f)                      ! Подпрограмма вычисления функций
 integer(4) :: n
 real(4) :: x(n), f(n)
 f(1) = x(1) + 2.0*x(2) + x(3) + 4.0 * x(4) - 20.7
 f(2) = x(1) * x(1) + 2.0 * x(1) * x(2) + x(4)**3 - 15.88
 f(3) = x(1)**3 + x(3) * x(3) + x(4) - 21.218
 f(4) = 3.0 * x(2) + x(3) * x(4) - 21.1
end subroutine fcn
```

Результат:

```
                          x
          1             2             3             4
      1.200         5.600         4.300         1.000

                       fvec
          1             2             3             4
 4.768E-07     -2.505E-04     3.190E-04     -1.758E-05
```

Пример для NEQBJ. Решается система нелинейных уравнений с тремя неизвестными

$$f_1(x) = x_1 + e^{x_1 - 1} + (x_2 + x_3)^2 - 27 = 0,$$

$$f_2(x) = e^{x_2 - 2} / x_1 + x_3^2 - 10 = 0,$$

$$f_3(x) = x_3 + \sin(x_2 - 2) + x_2^2 - 7 = 0.$$

с начальным приближением (4.0, 4.0, 4.0).

```
program ne2
 use dfimsl
 integer(4), parameter :: n = 3              ! n - число уравнений
 integer(4) :: iparam(6)
 real(4) :: fscale(n), fvec(n), rparam(5), x(n), xguess(n), xscale(n)
 external fcn, jac                           ! Обязательный атрибут для fcn и jac
 xguess = (/ 4.0, 4.0, 4.0 /)                ! Начальное приближение
 xscale = (/1.0, 1.0, 1.0 /); fscale = (/1.0, 1.0, 1.0 /)
 iparam(1) = 0                               ! Используем установки по умолчанию
 ! Поиск решения
 call neqbj(fcn, jac, n, xguess, xscale, fscale, iparam, rparam, x, fvec)
 call wrrrn('x', 1, n, x, 1, 0)              ! Вывод результата
 call wrrrn('fvec', 1, n, fvec, 1, 0)        ! Вывод невязки
end program ne2
```

```
subroutine fcn(n, x, f)                    ! Подпрограмма вычисления функций
 integer(4) :: n
 real(4) :: x(n), f(n)
 f(1) = x(1) + exp(x(1) - 1.0) + (x(2) + x(3)) * (x(2)+x(3)) - 27.0
 f(2) = exp(x(2) - 2.0) / x(I) + x(3) * x(3) - 10.0
 f(3) = x(3) + sin(x(2) - 2.0) + x(2) * x(2) - 7.0
end subroutine fcn
```

```
subroutine jac(n, x, fjac, Ldfjac)         ! Пользовательская подпрограмма,
 integer(4) :: n, Ldfjac                   ! вычисляющая элементы матрицы Якоби
 real(4) :: x(n), fjac(Ldfjac, n)
 fjac = reshape((/                                                        &
            1.0+exp(x(1)-1.0),          2.0*(x(2)+x(3)),  2.0*(x(2)+x(3)),  &
  -exp(x(2)-2.0)*(1.0/x(1)**2),  exp(x(2)-2.0)*(1.0/x(1)),        2.0*x(3),  &
                           0.0,  cos(x(2)-2.0) + 2.0*x(2),             1.0  &
      /), shape = (/ Ldfjac, n /), order = (/ 2, 1 /))
end subroutine jac
```

Результат:

```
                x
      1         2         3
  1.000     2.000     3.000

                fvec
          1             2             3
 1.550E-06     1.848E-06     2.384E-07
```

3.5.3. ПОДПРОГРАММЫ NEQNF (DNEQNF) И NEQNJ (DNEQNJ)

Подпрограмма NEQNF (DNEQNF) решает систему нелинейных уравнений модифицированным гибридным алгоритмом Пауэлла, используя конечно-разностную аппроксимацию для вычисления матрицы Якоби. Имеет вызов

CALL NEQNF(*fcn*, *errrel*, *n*, *itmax*, *xguess*, *x*, *fnorm*)

Подпрограмма NEQNJ (DNEQNJ) решает систему нелинейных уравнений модифицированным гибридным алгоритмом Пауэлла, применяя для вычисления матрицы Якоби заданную пользователем подпрограмму *Lsjac*. Имеет вызов

CALL NEQNJ(*fcn*, *Lsjac*, *errrel*, *n*, *itmax*, *xguess*, *x*, *fnorm*)

Параметры подпрограмм NEQNF и NEQNJ:
Пользовательские подпрограммы: *fcn*, *Lsjac*.
Входные: *errrel*, *n*, *itmax*, *xguess*, *x*.
Выходные: *x*, *fnorm*.
Описание параметров *n*, *xguess* и *x* выполнено в предшествующем разделе.

fcn - имя подпрограммы, вычисляющей функции системы нелинейных уравнений. Подпрограмма должна обладать атрибутом EXTERNAL. Имеет заголовок, несколько отличающийся порядком следования элементов по сравнению с заголовком одноименной подпрограммы предшествующего раздела:

SUBROUTINE *fcn*(*x*, *f*, *n*)

Входные: *n*, *x*.
Выходной: *f*.
Описание параметров *fcn* дано в предшествующем разделе.

Lsjac - пользовательская подпрограмма, вычисляющая элементы матрицы Якоби в заданной точке. Подпрограмма должна обладать атрибутом EXTERNAL. Имеет заголовок

SUBROUTINE *Lsjac*(*n*, *x*, *fjac*)

Параметры подпрограммы Lsjac:
Входные: n, x.
Выходной: fjac.
Смысл параметров подпрограммы *Lsjac* такой же, как и у одноименных параметров приведенной в предшествующем разделе подпрограммы *jac.*

Продолжение описания параметров подпрограмм NEQNF и NEQNJ:

errrel - критерий останова. Точка принимается в качестве корня, если относительная ошибка между двумя последовательными приближениями меньше *errrel.*

itmax - максимально допустимое число итераций. Максимальное число вызовов FCN равно *itmax*(*n* + 1). Рекомендуемое значение *itmax* = 200.

fnorm - вещественный скаляр, имеющий в точке *x* значение $f(1)^2 + ... + f(n)^2$.

Автоматически для решения предоставляется память:

- $1.5n^2 + 7.5n$ байт в случае NEQNF и NEQNJ;
- $3n^2 + 15n$ байт в случае DNEQNF и DNEQNJ.

Память можно выделить явно, применив N2QNF (DN2QNF) или N2QNJ (DN2QNJ):

CALL N2QNF(*fcn*, *errrel*, *n*, *itmax*, *xguess*, *x*, *fnorm*, *fvec*, *fjac*, *r*, *qtf*, *wk*)
CALL N2QNJ(*fcn*, *lsjac*, *errrel*, *n*, *itmax*, *xguess*, *x*, *fnorm*, *fvec*, *fjac*, *r*, *qtf*, *wk*)

Дополнительные параметры подпрограмм N2QNF и N2QNJ:

fvec (*выходной*) - вещественный вектор размера *n*, содержащий значения функций в точке *x*.

fjac (*выходной*) - вещественная *n*×*n*-матрица, содержащая ортогональную матрицу *Q* из *QR*-разложения итоговой матрицы Якоби.

r (*выходной*) - вещественный вектор размера $n(n + 1)/2$, содержащий верхнюю треугольную матрицу *R* из *QR*-разложения итоговой матрицы Якоби. Данные о матрице хранятся в *r* по строкам.

qtf (*выходной*) - вещественный вектор размера *n*, содержащий вектор $Q^T fvec$.

wk - вещественный *рабочий вектор* размера 5*n*.

Описание:

Подпрограмма NEQNF (NEQNJ) основана на процедуре HYBRD1 пакета MINPACK, использующей модификацию гибридного алгоритма Пауэлла. Эта модификация является вариантом метода Ньютона, в котором не допускаются большие величины для шага и увеличения невязки. Детальное описание алгоритма дано в [28].

В случае приближения матрицы Якоби конечными разностями для достижения большей точности рекомендуется работать с типом REAL(8), т. е. вызвать DNEQNF. Когда же пользователь может обеспечить точное вычисление матрицы Якоби, лучше вместо NEQNF употребить NEQNJ.

Пример для NEQNF. Решается та же, что в примере для NEQBF, система нелинейных уравнений.

```
program ne3
 use dfimsl
 integer(4), parameter :: n = 4              ! n - число уравнений
 integer(4) :: itmax = 200                   ! Максимально допустимое число итераций
 real(4) :: x(n), xguess(n), errrel, fnorm
 external fcn                                ! Обязательный атрибут для fcn
 xguess = 1.0                                ! Начальное приближение
 errrel = 0.001                              ! Относительная ошибка
 ! Поиск решения
 call neqnf(fcn, errrel, n, itmax, xguess, x, fnorm)
 call wrrrn('x', 1, n, x, 1, 0)              ! Вывод результата
 print *, 'fnorm =', fnorm
end program ne3

subroutine fcn(x, f, n)                      ! Следует обратить внимание на порядок
 integer(4) :: n                             ! следования параметров подпрограммы fcn,
 real(4) :: x(n), f(n)                       ! иной, чем в случае NEQBF
 f(1) = x(1) + exp(x(1) - 1.0) + (x(2) + x(3)) * (x(2)+x(3)) - 27.0
 f(2) = exp(x(2) - 2.0) / x(1) + x(3) * x(3) - 10.0
 f(3) = x(3) + sin(x(2) - 2.0) + x(2) * x(2) - 7.0
end subroutine fcn
```

Результат:

```
                  x
      1           2           3           4
  1.200       5.600       4.300       1.000

fnorm =  1.431741E-09
```

Пример для NEQNJ. Решается та же, что в примере для NEQBJ, система нелинейных уравнений.

```
program ne4
 use dfimsl
 integer(4), parameter :: n = 3
 integer(4) :: itmax = 200                ! Максимально допустимое число итераций
 real(4) :: x(n), xguess(n), errrel, fnorm
 external fcn, jac                        ! Обязательный атрибут для fcn и jac
 xguess = 4.0                             ! Начальное приближение
 errrel = 0.001                           ! Относительная ошибка
 ! Поиск решения
 call neqnj(fcn, jac, errrel, n, itmax, xguess, x, fnorm)
 call wrrrn('x', 1, n, x, 1, 0)           ! Вывод результата
 print *, 'fnorm =', fnorm
end program ne4

subroutine fcn(x, f, n)                   ! Следует обратить внимание на порядок
 integer(4) :: n                          ! следования параметров подпрограммы fcn,
 real(4) :: x(n), f(n)                    ! иной, чем в случае NEQBF
 f(1) = x(1) + exp(x(1) - 1.0) + (x(2) + x(3)) * (x(2)+x(3)) - 27.0
 f(2) = exp(x(2) - 2.0) / x(1) + x(3) * x(3) - 10.0
 f(3) = x(3) + sin(x(2) - 2.0) + x(2) * x(2) - 7.0
end subroutine fcn

subroutine jac(n, x, fjac)                ! Пользовательская подпрограмма,
 integer(4) :: n                          ! вычисляющая элементы матрицы Якоби
 real(4) :: x(n), fjac(n, n)
 fjac = reshape((/                                                    &
          1.0+exp(x(1)-1.0),          2.0*(x(2)+x(3)),  2.0*(x(2)+x(3)),  &
  -exp(x(2)-2.0)*(1.0/x(1)**2), exp(x(2)-2.0)*(1.0/x(1)),       2.0*x(3),  &
                        0.0, cos(x(2)-2.0) + 2.0*x(2),             1.0  &
       /), shape = (/ n, n /), order = (/ 2, 1 /))
end subroutine jac
```

Результат:

```
                x
      1           2           3
  1.000       2.000       3.000

fnorm =  9.909558E-08
```

4. ОПТИМИЗАЦИЯ

4.1. ВВЕДЕНИЕ

4.1.1. БЕЗУСЛОВНАЯ МИНИМИЗАЦИЯ

Задача безусловной минимизации (минимизации без ограничений) состоит в поиске минимума $\min_{x \in \mathbf{R}^n} f(x)$, где функция f: $\mathbf{R}^n \to \mathbf{R}$ - является по крайней мере непрерывной. Процедуры безусловной минимизации подразделяются на 3 категории:

- оперирующие функцией одной переменной (с префиксом UV);
- работающие с функцией нескольких переменных (с префиксом UM);
- использующие нелинейный метод наименьших квадратов (с префиксом UNLS).

В случае функции одной переменной предполагается, что на исследуемом отрезке она имеет один экстремум. В противном случае осуществляется поиск локального минимума.

Поиск минимума функции нескольких переменных можно выполнить квазиньютоновским методом (процедуры UMINF и UMING), модифицированным алгоритмом Ньютона (процедуры UMIDH и UMIAH), методом сопряженных градиентов (процедуры UMCGF и UMCGG) и методом деформируемого многогранника (процедура UMPOL). Процедуры UNLSF и UNLSJ, применяющие нелинейный метод наименьших квадратов, основаны на модифицированном алгоритме Левенберга - Маркварда.

Перечисленные процедуры обеспечивают поиск локального минимума. Если же функция имеет несколько локальных минимумов и необходимо найти наилучший, то следует испытать разные начальные точки и интервалы поиска. С процедурами, использующими только значения функции, следует употреблять двойную точность. Также полезно использовать процедуры контроля производной (имеют префикс CH), обеспечивающие проверку работы пользовательских процедур, оценивающих производные.

Как и ранее, с целью экономии места опускаются процедуры 2-го уровня. Исключение составляют случаи, когда такие процедуры предоставляют дополнительные по сравнению с главными процедурами возможности.

4.1.2. МИНИМИЗАЦИЯ ФУНКЦИИ НЕСКОЛЬКИХ ПЕРЕМЕННЫХ С ПРОСТЫМИ ОГРАНИЧЕНИЯМИ

Состоит в поиске минимума $\min_{x \in \mathbf{R}^n} f(x)$ с ограничениями $l_i \le x_i \le u_i$, для $i = 1, 2, \ldots, n$, где f: $\mathbf{R}^n \to \mathbf{R}$. В общем случае ограничения на значения части

переменных могут не накладываться. Процедуры, выполняющие минимизацию функции нескольких переменных, имеют префиксы BCO, BCLS или BCP.

Процедуры с префиксом BCO используют те же алгоритмы, что и UMI-подпрограммы; BCLS-процедуры подобны UNLS-подпрограммам. Единственное отличие в том, что стратегия поиска минимума сохраняет значение переменных в заданных пределах. Процедура BCPOL, основанная на методе деформируемого многогранника, применяет метод сравнения функции, схожий с методом, употребляемым UMPOL. Сходимость методов деформируемого многогранника не гарантируется, однако BCPOL и UMPOL могут быть использованы как последняя альтернатива.

4.1.3. МИНИМИЗАЦИЯ ФУНКЦИИ НЕСКОЛЬКИХ ПЕРЕМЕННЫХ С ЛИНЕЙНЫМИ ОГРАНИЧЕНИЯМИ

Заключается в нахождении минимума $\min_{x \in \mathbf{R}^n} f(x)$ при условии, что $Ax = b$, где f: $\mathbf{R}^n \to \mathbf{R}$, A - $m \times n$-матрица коэффициентов, b - вектор размера m.

Если $f(x)$ - линейная функция, то решается задача линейного программирования; если $f(x)$ - квадратичная функция, то мы имеем дело с задачей квадратичного программирования.

Подпрограмма DLPRS использует модифицированный симплекс-метод для решения небольших и среднего размера задач линейного программирования. Матрица коэффициентов хранится как полная матрица.

Подпрограмма QPROG решает выпуклую задачи квадратичного программирования. Если гессиан задачи не является положительно определенным, то QPROG приводит его к положительно определенному. В этом случае, однако, результат должен быть тщательно проверен.

Подпрограммы LCONF и LCONG используют итерационный метод решения задачи с линейными ограничениями и целевой функцией общего вида. Детали см. в [38] и [39].

4.1.4. МИНИМИЗАЦИЯ ФУНКЦИИ НЕСКОЛЬКИХ ПЕРЕМЕННЫХ С НЕЛИНЕЙНЫМИ ОГРАНИЧЕНИЯМИ

Заключается в нахождении минимума $\min_{x \in \mathbf{R}^n} f(x)$ с ограничениями

$$g_i(x) = 0, \; i = 1, 2, \ldots, m_1,$$

$$g_i(x) \geq 0, \; i = m_1 + 1, \ldots, m,$$

где f: $\mathbf{R}^n \to \mathbf{R}$ - и g_i: $\mathbf{R}^n \to \mathbf{R}$, для $i = 1, 2, \ldots, m$.

Минимизация выполняется подпрограммами NCONF и NCONG, использующими последовательный алгоритм квадратичного программирования.

4.1.5. ВЫБОР ПРОЦЕДУРЫ

4.1.5.1. Безусловная минимизация

Придерживайтесь следующих рекомендаций:

1) в случае функции одной переменной используйте UVMID, если можно вычислить градиент, и UVMIF - в противном случае. Если функция имеет разрывы, применяйте UVMGS;
2) при работе с функцией нескольких переменных используйте UMCG*, когда есть проблемы с памятью, для негладких функций применяйте UMPOL, в противном случае употребляйте UMI** в зависимости от наличия и вида градиента или гессиана;
3) в задаче наименьших квадратов используйте либо UNLSJ, если есть якобиан, или UNLSF, если его нет.

Графически приведенные рекомендации представлены на рис. 4.1.

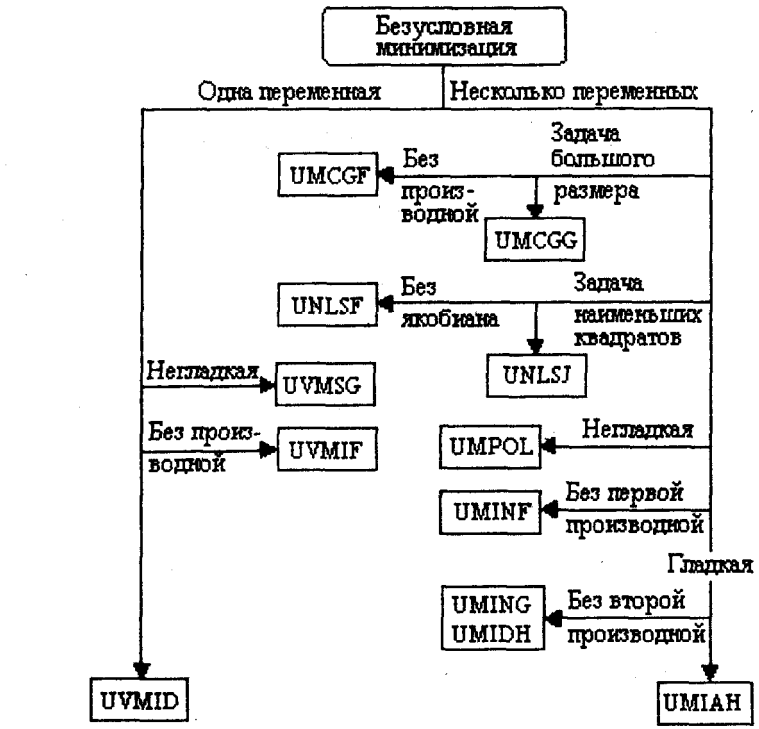


Рис. 4.1. Выбор процедуры безусловной минимизации

4.1.5.2. Минимизация с простыми ограничениями

Выбирая процедуру для решения задачи минимизации с простыми ограничениями, придерживайтесь следующих рекомендаций:

1) используйте BCONF, если можно оценить только функцию. Когда можно оценить первую производную, применяйте либо BCONG, либо BCODH. Если доступны и первая и вторая производные, употребляйте BCOAH;
2) в задаче наименьших квадратов применяйте BCLSF или, если доступен якобиан, BCLSJ;
3) в случае негладких функций, когда неэффективны иные процедуры, используйте BCPOL.

Выбор процедуры облегчит и приводимый ниже рис. 4.2.

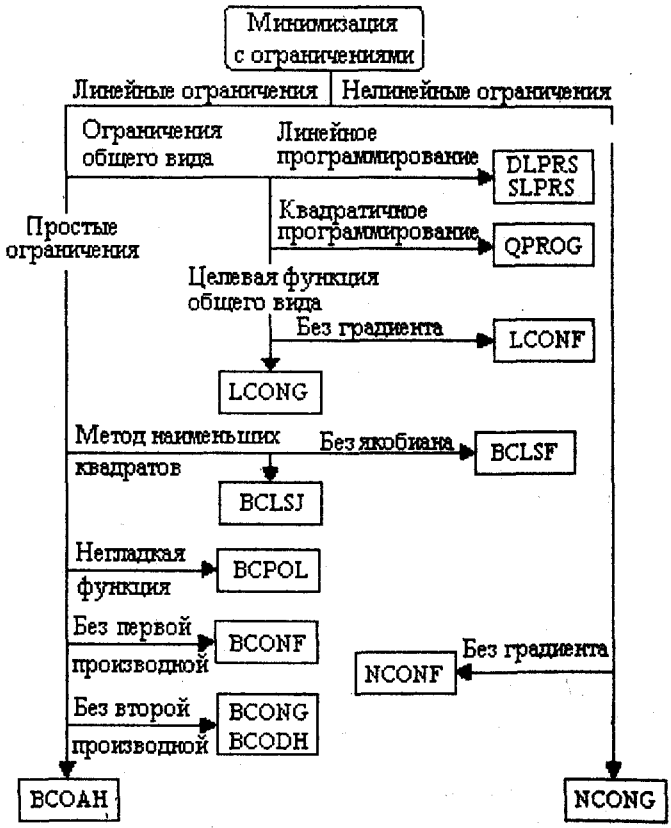


Рис. 4.2. Выбор процедуры при работе с ограничениями

4.2. БЕЗУСЛОВНАЯ МИНИМИЗАЦИЯ ФУНКЦИИ ОДНОЙ ПЕРЕМЕННОЙ

4.2.1. ПЕРЕЧЕНЬ ПОДПРОГРАММ

Минимум функции одной переменной находят приведенные в табл. 4.1 подпрограммы.

Таблица 4.1. Подпрограммы, определяющие минимум функции одной переменной

Подпрограмма	*Описание*
UVMIF	Находит минимум гладкой функции одной переменной, используя только оценки функции
UVMID	Находит минимум гладкой функции одной переменной, используя оценки функции и ее первой производной
UVMGS	Находит минимум негладкой функции одной переменной

4.2.2. ПОДПРОГРАММА UVMIF (DUVMIF)

Находит минимум гладкой функции одной переменной, используя только оценки функции. Имеет вызов

CALL UVMIF(*f, xguess, step, bound, xacc, maxfn, x*)

Параметры подпрограммы UVMIF:

Входные: f, xguess, step, bound, xacc, maxfn.

Выходной: x.

f - пользовательская функция, возвращающая значение минимизируемой функции в точке *x*. Имеет вид *f*(*x*). Переменная *x* не должна изменяться *f*. Функция *f* должна в вызывающей программе получить атрибут EXTERNAL.

xguess - начальное предположение о точке минимума функции *f*.

step - начальное значение шага для *x*; может быть положительным или отрицательным.

bound - положительное число, ограничивающее предельное отклонение *x* от начального значения.

xacc - абсолютная точность результирующего значения *x*. При нормальном завершении значение функции в точках *x* - *xacc* и *x* + *xacc* не меньше *f*(*x*).

maxfn - максимально допустимое число оценок функции.

x - точка, в которой найден минимум функции *f*.

Комментарий. При работе с UVMIF могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
3	1	Ошибка округления не позволяет изменять x
3	2	Окончательное значение x находится на границе. Возможно, минимум лежит за пределами границы
4	3	Число оценок функции превысило *maxfn*

Описание:

Подпрограмма UVMIF использует метод квадратичной интерполяции для поиска минимума функции одной переменной. Код и алгоритм основаны на процедуре ZXLSF, приведенной в [39].

Пусть x_0 = *xguess* и b = *bound*, тогда x не должен выходить за пределы отрезка $[x_0 - b, x_0 + b]$. Обычно поиск начинается с перемещения от x_0 к $x = x_0 + s$, где s = *step*. Первые две оценки функции позволяют определить направление поиска, используя которое алгоритм либо находит точку минимума, либо достигает границы $x_0 \pm b$. Во время поиска шаг увеличивается в 2-9 раз при каждой оценке функции. Коэффициент увеличения шага зависит от прогноза расположения точки минимума, осуществляемого квадратичной интерполяцией по трем последним оценкам функции.

После обнаружения интервала, содержащего решение, мы имеем 3 значения аргумента: x_1, x_2 и x_3. Причем $x_1 < x_2 < x_3$ и $f(x_2) \le f(x_1)$ и $f(x_2) \le f(x_3)$. Для выбора следующего значения x оценивается минимальная точка квадратичной интерполяционной функции, и новый x берется как можно ближе к найденному минимуму с учетом допуска ε, зависящего от близости f к квадратичной интерполяционной функции и положения x_2 относительно центра отрезка $[x_1, x_3]$ и его концов.

Алгоритм обеспечивает быструю сходимость, когда f имеет положительную и непрерывную вторую производную в минимуме. Также он эффективен и в таких трудных случаях, как, например, $f(x) = x + 1.001|x|$.

Рекомендуется использовать двойную точность.

Пример. Вычисляется минимум функции $e^x - 5x$.

```
program uvmifTest
 use dfimsl
 integer(4) :: maxfn
 real(4) :: bound, f, step, x, xacc, xguess
 external f
 ! Инициализация переменных
 xguess = 0.0; xacc = 0.001; bound = 100.0; step = 0.1; maxfn = 50
 ! Находим минимум функции f = EXP(x) - 5x
 call uvmif (f, xguess, step, bound, xacc, maxfn, x)
 ! Печатаем результат
 write(*,"(' The minimum is at ', 7x, f7.3, //, ' The function ', 'value is ', f7.3)") x, f(x)
```

```
end program uvmifTest

real function f(x)                    ! f = EXP(x) - 5.0x
 real(4) :: x
 f = exp(x) - 5.0e0*x
end function f
```

Результат:

```
The minimum is at 1.609
The function value is -3.047
```

4.2.3. ПОДПРОГРАММА UVMID (DUVMID)

Находит минимум гладкой функции одной переменной, используя оценки функции и ее первой производной. Имеет вызов

CALL UVMID(*f, g, xguess, errrel, gtol, maxfn, a, b, x, fx, gx*)

Параметры подпрограммы UVMID:
Пользовательские функции: f, g.
Входные: xguess, errrel, gtol, maxfn, a, b.
Выходные: x, fx, gx.

Описание параметров *f, xguess, maxfn, x* см. в предшествующем разделе.

g - пользовательская функция, возвращающая значение производной функции *f* в точке *x*. Имеет вид *g*(*x*). Переменная *x* не должна изменяться *g*. Функция *g* должна в вызывающей программе получить атрибут EXTERNAL.

errrel - требуемая относительная точность конечного значения *x*. Это первый критерий завершения. При нормальном возврате решение *x* находится на отрезке, содержащем локальный минимум, и меньше или равно max(1.0, ABS(*x*)) * *errrel*. Когда *errrel* меньше машинной точности $\varepsilon_м$, возвращаемой AMACH(4) или DMACH(4), вместо *errrel* употребляется $\sqrt{\varepsilon_м}$.

gtol - допуск для производной, используемый для определения того, является ли текущая точка локальным минимумом. Это второй критерий завершения. Переменная *x* возвращается как решение, когда $gx \le gtol$. Значение *gtol* должно быть неотрицательным; в противном случае для *gtol* используется нуль.

a, b - соответственно левая и правая границы отрезка поиска локального минимума функции *f*.

fx, gx - соответственно значения функции и ее производной в точке *x*.

Комментарий. При работе с UVMID могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
3	1	Окончательное значение *x* находится на левой границе. Возможно, минимум лежит за пределами границы
3	2	Окончательное значение *x* находится на правой границе. Возможно, минимум лежит за пределами границы
4	3	Число оценок функции превысило *maxfn*

Описание:

Подпрограмма UVMID ищет минимум функции одной переменной, используя либо метод секущих, либо кубическую интерполяцию. Подпрограмма стартует с начального предположения и двух конечных точек. Если какая-либо из этих трех точек является локальным минимумом, подпрограмма завершается и возвращает решение. В противном случае точка с наименьшим значением функции будет использована как начальная.

В начальной точке x_c вычисляется значение функции $f_c = f(x_c)$ и производной $g_c = g(x_c)$; затем определяется новая точка $x_n = x_c - g_c$. Далее оцениваются функция $f_n = f(x_n)$ и ее производная $g_n = g(x_n)$. Если $f_n \geq f_c$ или знак g_n противоположен знаку g_c, то минимальная точка находится между x_c и x_n, т. е. отрезок поиска локализован. В противном случае выполняется обмен x_n и x_c. Новая точка x_s находится по методу секущих:

$$x_s = x_c - g_c \frac{g_n - g_c}{x_n - x_c}.$$

Далее выполняется присваивание $x_n \leftarrow x_s$, и процесс повторяется до тех пор, пока не найден отрезок, содержащий минимум, или не удовлетворен один из критериев сходимости: $|x_c - x_n| \leq \varepsilon_c$ (критерий 1) или $|g_c| \leq \varepsilon_g$ (критерий 2), где $\varepsilon_c = \max(1.0, |x_c|)\varepsilon$, $\varepsilon = errrel$, а $\varepsilon_g = gtol$.

Если процесс не сошелся, то для получения новой точки на найденном отрезке выполняется кубическая интерполяция функции. В новой точке оцениваются функция и ее производная и выбирается новый (меньший) отрезок, содержащий точку минимума. На новом интервале опять вычисляется кубическая интерполяция и повторяются описанные выше действия до тех пор, пока не удовлетворен один из критериев завершения.

Пример. Ищется минимум функции $e^x - 5x$.

```
program uvmidTest
 use dfimsl
 integer(4) :: maxfn
 real(4) :: a, b, errrel, f, fx, g, gtol, gx, x, xguess
 external f, g
 ! Инициализация переменных
```

```
xguess = 0.0; errrel = 0.0; gtol = 0.0; a = -10.0; b = 10.0; maxfn = 50
! Поскольку errrel = 0, то для относительной ошибки
! будет использовано значение √εм
! Находим минимум функции f = e^x - 5x и выводим результаты
call uvmid(f, g, xguess, errrel, gtol, maxfn, a, b, x, fx, gx)
write(*, 1) x, fx, gx
1 format(' The minimum is at ', 7x, f7.3 / ' The function value is ',      &
      f7.3 / ' The derivative is ', f7.3)
end program uvmidTest

real function f(x)                          ! f = EXP(x) - 5.0x
 real(4) :: x
 f = exp(x) - 5.0e0*x
end function f

real function g(x)                          ! Производная g(x) = f'(x)
 real(4) :: x
 g = exp(x) - 5.0e0
end function g
```

Результат:

```
The minimum is at 1.609
The function value is -3.047
The derivative is -0.001
```

4.2.4. ПОДПРОГРАММА UVMGS (DUVMGS)

Находит минимум негладкой функции одной переменной. Имеет вызов

CALL UVMGS(*f, a, b, tol, xmin*)

Параметры подпрограммы UVMGS:

Пользовательская функция: f.

Входной: tol.

Входные/выходные: a, b.

Выходной: xmin.

Описание параметра *f* то же, что и для подпрограммы UVMIF.

a - на входе это левая граница отрезка поиска интервала, содержащего минимум функции *f*; на выходе - левая граница искомого интервала.

b - на входе это правая граница отрезка поиска интервала, содержащего минимум функции *f*; на выходе - правая граница искомого интервала.

tol - допустимая длина искомого интервала.

xmin - приблизительное значение точки, в которой расположен минимум функции *f*; $xmin \in [a, b]$.

Комментарии:

1. При работе с UVMGS могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
3	1	Значение *tol* чрезмерно мало и не может быть удовлетворено
4	2	Из-за ошибок округления функция *f* не является унимодальной

2. При выходе из UVMGS без ошибок имеем: $(b - a) \le tol$, $a \le xmin \le b$, $f(xmin) \le f(a)$ и $f(xmin) \le f(b)$.
3. При выходе из UVMGS с ошибкой, имеющей код 2, имеем: $a \le xmin \le b$, $f(xmin) \ge f(a)$ и $f(xmin) \ge f(b)$ (нестрогим может быть только одно неравенство). Необходимо продолжить анализ функции f, чтобы выяснить, является ли функция унимодальной, и принять возвращенные значения a, b и $xmin$ в качестве решения или отвергнуть их.

Описание:

Подпрограмма UVMGS использует для поиска интервала, содержащего минимум, метод золотого сечения. Если для *tol* использовать обозначение τ, то для получения искомого интервала потребуется не менее

$$\left]\frac{\ln(\tau/(b-a))}{\ln(1-c)}+1\right[$$

итераций, где $c = (3 - \sqrt{5})/2$ и $]x[$ - наименьшее целое, большее или равное x.

Первые две тестовые точки соответственно равны $v_1 = a + c(b - a)$ и $v_2 = b - c(b - a)$, и если $f(v_1) < f(v_2)$, то минимум принадлежит интервалу (a, v_2). В этом случает параметры поиска меняются следующим образом: $b \leftarrow v_2$, $v_2 \leftarrow v_1$ и $v_1 \leftarrow a + c(b - a)$. Если $f(v_1) \ge f(v_2)$, то минимум принадлежит интервалу (v_1, b) и новые параметры поиска таковы: $a \leftarrow v_1$, $v_1 \leftarrow v_2$ и $v_2 \leftarrow b - c(b - a)$. Поиск продолжается в указанном ключе; на каждом шаге вычисляется только одна новая точка. Процесс прекращается при достижении заданной точности τ; *xmin* принимается равным тестовой точке с минимальным значением функции.

Математически алгоритм всегда достигает заданной точности. Могут возникнуть, однако, численные проблемы. Так, если f - чрезмерно пологая на отрезке поиска функция, то с вычислительной точки зрения она может восприниматься как константа, о чем сообщит ошибка с кодом 2. Требования к результату можно смягчить, увеличив τ, или преодолеть проблему, масштабируя функцию или повышая точность вычислений.

Пример. Ищется минимум функции $y = 3x^2 - 2x + 4$.

```
program uvmgsTest
 use dfimsl
 real(4) :: a, b, fcn, fmin, tol, xmin
 external fcn
 a = 0.0e0; b = 5.0e0; tol = 1.0e-3        ! Инициализация переменных
 call uvmgs(fcn, a, b, tol, xmin)          ! Находим минимум функции fcn
 fmin = fcn(xmin)                          ! Вывод результата
 write(*, 1) xmin, fmin, a, b
 1 format (' The minimum is at ', f5.3, //, ' The function value is ', f5.3, //,
      ' The final interval is (', f6.4, ',', f6.4, ')', /)
end program uvmgsTest

real function fcn(x)                       ! Функция y = 3x^2 - 2x + 4
 real(4) :: x
 fcn = 3.0e0 *x * x - 2.0e0 * x + 4.0e0
end function fcn
```

Результат:

```
The minimum is at 0.333
The function value is 3.667
The final interval is (0.3331, 0.3340)
```

4.3. БЕЗУСЛОВНАЯ МИНИМИЗАЦИЯ ФУНКЦИИ НЕСКОЛЬКИХ ПЕРЕМЕННЫХ

4.3.1. ПАРАМЕТРЫ И ИНФОРМАЦИОННЫЕ ОШИБКИ ПОДПРОГРАММ

Многие подпрограммы, вычисляющие минимум функции нескольких переменных, как с ограничениями, так и без них, имеют одинаковые параметры и возможные информационные ошибки. Поэтому для экономии места и параметры и ошибки ряда подпрограмм сведены в отдельные таблицы. Так, параметры подпрограмм UMINF, UMING, UMIDH, UMIAH, UMCGF, UMCGG, UMPOL, BCONF, BCONG, BCODH, BCOAH и BCPOL приведены в табл. 4.2, а параметры подпрограмм UNLSF, UNLSJ, BCLSF, BCLSJ и BCNLS, решающих задачу наименьших квадратов, - в табл. 4.3. Возможные информационные ошибки подпрограмм занесены в табл. 4.4.

Таблица 4.2. Параметры подпрограмм UMINF, UMING, UMIDH, UMIAH, UMCGF, UMCGG, UMPOL, BCONF, BCONG, BCODH, BCOAH и BCPOL

Имя	*Смысл/вид*	*Тип*
dfpred	Грубая оценка ожидаемого уменьшения функции; используется для определения размера начального изменения *x* / *входной*	REAL(4) или REAL(8)

ibtype	Скаляр, задающий вид ограничений на переменные; оказывает следующие действия: *ibtype* = 0 - все ограничения задаются пользователем; *ibtype* = 1 - все переменные неотрицательны; *ibtype* = 2 - все переменные неположительны; *ibtype* = 3 - пользователь задает ограничения на первую переменную; все другие переменные будут иметь те же ограничения / *входной*	INTEGER(4)
iparam	Вектор целочисленных параметров размера 7. Содержит следующие параметры: *iparam*(1) - флаг инициализации. Если задать *iparam*(1) = = 0, то будут использоваться установленные по умолчанию параметры *iparam* и *rparam*; *iparam*(2) - число значимых десятичных знаков для функции; на Pentium-совместимых ЭВМ по умолчанию равно 7; *iparam*(3) - максимально допустимое число итераций; по умолчанию равно 100; *iparam*(4) - максимально допустимое число оценок функции; по умолчанию равно 400; *iparam*(5) - максимально допустимое число оценок градиента; по умолчанию равно 400; *iparam*(6) - параметр инициализации гессиана (подпрограммами UMIDH и UMIAH не используется). Если *iparam*(6) = 0, гессиан инициализируется как единичная матрица, в противном случае изначально гессиан - это диагональная матрица, содержащая на диагонали $\max(\lvert f(t), f_s \rvert) * s_i^2$, где $t = xguess$, $f_s = fscale$ и $s = xscale$; по умолчанию *iparam*(6) = 0; *iparam*(7) - максимально допустимое число оценок гессиана (подпрограммами UMINF и UMING не используется). Значение по умолчанию - 100 / *входной/выходной*	"
fcn	Пользовательская подпрограмма, оценивающая минимизируемую функцию. Имеет вызов CALL *fcn*(*n*, *x*, *f*) Параметры *n*, *x* подпрограммы *fcn* являются *входными*, параметр *f* - *выходным*. Параметры имеют следующий смысл: *n* - размер вектора *x*, содержащего координаты текущей точки; *x* - точка (вектор размера *n*), в которой выполняется оценка функции; параметр *x* не должен изменяться подпрограммой *fcn*;	-

	f - оценка функции в точке *x*. Подпрограмма *fcn* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	
fscale	Скаляр, применяемый главным образом для масштабирования градиента. При отсутствии информации о величине *fscale* задайте *fscale* = 1.0 / *входной*	REAL(4) или REAL(8)
ftol	Используется для выработки критерия сходимости. Первый критерий сходимости: алгоритм останавливается, когда относительная ошибка значений функции меньше, чем *ftol*, т. е. когда *f*(*worst*) - *f*(*best*) < *ftol* * (1 + ABS(*f*(*best*))), где *f*(*worst*) и *f*(*best*) - соответственно значения функции в текущих лучшей и худшей точках. Второй критерий сходимости: алгоритм останавливается, когда стандартное отклонение значений функции в *n* + 1 текущих точках меньше, чем *ftol*. Если подпрограмма завершается преждевременно, выполните новую попытку с меньшим значением *ftol* / *входной*	То же
fvalue	Скаляр, содержащий значение функции в точке *x* / *выходной*	"
g	Вектор размера *n*, содержащий компоненты градиента после последней его оценки / *выходной*	"
grad	Пользовательская подпрограмма, оценивающая градиент в точке *x*. Имеет вызов CALL *grad*(*n, x, g*) Параметры *n, x* подпрограммы *grad* являются *входными*, параметр *g* - *выходным*. Параметры имеют следующий смысл: *n* - размер вектора *x*, содержащего координаты текущей точки; *x* - точка, в которой выполняется оценка градиента; параметр не должен изменяться подпрограммой *grad*; *g* - оценка градиента в точке *x*. Подпрограмма *grad* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	"
gradtl	Критерий сходимости. Вычисления прекращаются, когда сумма квадратов компонентов вектора *g* меньше, чем *gradtl* / *входной*	"
hess	Пользовательская подпрограмма, оценивающая гессиан в точке *x*. Имеет вызов CALL *hess*(*n, x, h, Ldh*) Параметры *n, x, Ldh* подпрограммы *hess* являются *вход-*	"

	ными, параметр *h* - *выходным*. Параметры имеют следующий смысл: *n* - размер вектора *x*, содержащего координаты текущей точки; *x* - точка, в которой выполняется оценка гессиана; параметр не должен изменяться подпрограммой *hess*; *h* - массив формы (*Ldh*, *n*), содержащий оценку гессиана в точке *x*; *Ldh* - ведущий размер массива *h*, представляющего гессиан *H*. В этой подпрограмме равен *n*. Подпрограмма *hess* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	
maxfn	Максимально допустимое число оценок функции. Если задать *maxfn* = 0, то ограничений на число оценок существовать не будет / *входной*	INTEGER(4)
maxfcn	На входе - максимально допустимое число оценок функции, на выходе - реальное число оценок функции / *входной/выходной*	"
n	Размер задачи / *входной*	"
rparam	Вектор вещественных параметров размера 7. Содержит следующие параметры: *rparam*(1) - масштабируемый допуск для градиента. Компонент *i* масштабированного градиента вычисляется по формуле $\frac{\lvert g_i \rvert * \max(\lvert x_i \rvert, 1/s_i)}{\max(\lvert f(x) \rvert, f_s)}$, где $g = \nabla f(x)$, s = *xscale* и f_s = *fscale*; по умолчанию равен $\sqrt{\varepsilon_м}$ при одинарной точности вычислений и $\sqrt[3]{\varepsilon_м}$ при двойной, где $\varepsilon_м$ - машинная точность; *rparam*(2) - масштабированный допуск для шага *steptl*. Компонент *i* масштабированного допуска для шага между двумя точками *x* и *y* вычисляется по формуле $\frac{\lvert x_i - y_i \rvert}{\max(\lvert x_i \rvert, 1/s_i)}$, где s = *xscale*; по умолчанию равен $\varepsilon_м^{2/3}$; *rparam*(3) - относительный допуск для функции; по умолчанию равен $\max(10^{-10}, \varepsilon_м^{2/3})$ при одинарной точ-	REAL(4) или REAL(8)

	ности вычислений и $\max(10^{-20}, \varepsilon_м^{2/3})$ при двойной; *rparam*(4) - абсолютный допуск для функции (подпрограммами UMINF, UMING, UMIDH и UMIAH не используется); *rparam*(5) - допуск для ложной сходимости; по умолчанию равен $100\varepsilon_м$; *rparam*(6) - минимально допустимый шаг; по умолчанию равен $1000\max(\varepsilon_1, \varepsilon_2)$, где $\varepsilon_1 = \sqrt{\sum_{i=1}^{n}(s_i t_i)^2}$, $\varepsilon_2 = \|s\|_2$, $s = xscale$ и $t = xguess$; *rparam*(7) - начальный радиус доверительной области (подпрограммами UMINF и UMING не используется). Основывается на масштабированном шаге Коши / *входной/выходной*	
s	На входе скаляр, содержащий длину стороны начального симплекса. (Напомним, что *симплексом* называется набор из $n + 1$ точек в n-мерном пространстве. Процесс минимизации состоит в замене точки с наибольшим значением функции новой, более хорошей точкой.) Если нет существенной информации о величине *s*, то *s* можно установить меньшим или равным нулю. В этом случае UMPOL самостоятельно создаст начальный симплекс. На выходе *s* - это среднее расстояние от вершин симплекса до его центра, который и принимается в качестве решения / *входной/выходной*	REAL(4) или REAL(8)
x	Вектор размера *n*, содержащий вычисленное решение / *выходной*	То же
xguess	Вектор размера *n*, содержащий начальное предположение о точке минимума / *входной*	"
xlb	Вектор размера *n*, содержащий нижние границы переменных / *входной*, если *ibtype* = 0; *выходной*, если *ibtype* = 1 или 2; *входной/выходной*, если *ibtype* = 3	"
xscale	Вектор размера *n*, содержащий диагональную матрицу масштабирования переменных. Вектор *xscale* применяется главным образом для масштабирования градиента и расстояния между двумя точками. При отсутствии информации о компонентах вектора задайте все его элементы, равными 1.0 / *входной*	"
xub	Вектор размера *n*, содержащий верхние границы переменных / *входной*, если *ibtype* = 0; *выходной*, если *ibtype* = 1 или 2; *входной/выходной*, если *ibtype* = 3	"

Таблица 4.3. Параметры подпрограмм UNLSF, UNLSJ, BCLSF, BCLSJ и BCNLS

Имя	*Смысл*	*Тип*
ibtype	Скаляр, задающий вид ограничений на переменные; оказывает следующие действия: *ibtype* = 0 - все ограничения задаются пользователем; *ibtype* = 1 - все переменные неотрицательны; *ibtype* = 2 - все переменные неположительны; *ibtype* = 3 - пользователь задает ограничения на первую переменную; все другие переменные будут иметь те же ограничения / *входной*	INTEGER(4)
iparam	Вектор целочисленных параметров размера 6. Содержит следующие параметры: *iparam*(1) - флаг инициализации. Если задать *iparam*(1) = = 0, то будут использоваться установленные по умолчанию параметры *iparam* и *rparam*; *iparam*(2) - число значимых десятичных знаков для функции; на Pentium-совместимых ЭВМ по умолчанию равно 7; *iparam*(3) - максимально допустимое число итераций; по умолчанию равно 100; *iparam*(4) - максимально допустимое число оценок функции; по умолчанию равно 400; *iparam*(5) - максимально допустимое число оценок якобиана (подпрограммой UNLSF не используется); по умолчанию равно 400; *iparam*(6) - внутренний флаг масштабирования. Если *iparam*(6) = 1, то вектор *xscale* задается внутри подпрограммы; по умолчанию *iparam*(6) = 1 / *входной/выходной*	"
fcn	Пользовательская подпрограмма, задающая задачу наименьших квадратов. Имеет вызов CALL *fcn*(*m, n, x, f*) Параметры *m, n, x* подпрограммы *fcn* являются *входными*, параметр *f* - *выходным*. Параметры имеют следующий смысл: *m* - размер вектора *f*; *n* - размер вектора *x*; *x* - точка (вектор размера *n*), в которой выполняется оценка вектор-функции; параметр *x* не должен изменяться подпрограммой *fcn*; *f* - вектор размера *n*, содержащий значения вектор-функции *f*(*x*) в точке *x*. Подпрограмма *fcn* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	-

fjac	Массив формы (*Ldfjac, n*), содержащий конечно-разностную аппроксимацию *m*×*n*-матрицы Якоби в точке приблизительного решения *x* (для подпрограмм UNLSF и BCLSF) / *выходной*	-
fjac	Массив формы (*Ldfjac, n*), содержащий вычисленный в точке *x* *m*×*n*-якобиан (для подпрограмм UNLSJ и BCLSJ) / *выходной*	
fscale	Вектор размера *m*, содержащий диагональную матрицу масштабирования функций. Вектор *fscale* применяется главным образом для масштабирования градиента. При отсутствии информации о величине *fscale* задайте *fscale* = 1.0 / *входной*	REAL(4) или REAL(8)
fvec	Вектор размера *m*, содержащий невязки в точке приблизительного решения *x* / *выходной*	То же
grad	Пользовательская подпрограмма, оценивающая градиент в точке *x*. Имеет вызов CALL *grad*(*n, x, g*) Параметры *n, x* подпрограммы *grad* являются *входными*, параметр *g* - *выходным*. Параметры имеют следующий смысл: *n* - размер вектора *x*, содержащего координаты текущей точки; *x* - точка, в которой выполняется оценка градиента; параметр не должен изменяться подпрограммой *grad*; *g* - оценка градиента в точке *x*. Подпрограмма *grad* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	"
jac	Пользовательская подпрограмма, оценивающая якобиан в точке *x*. Имеет вызов: CALL *jac*(*m, n, x, fjac, Ldfjac*) Параметры *m, n, x, Ldfjac* подпрограммы *jac* являются *входными*, параметр *fjac* - *выходным*. *m* - размер вектора *f*; *n* - размер вектора *x*; *x* - точка (вектор размера *n*), в которой выполняется оценка якобиана; параметр *x* не должен изменяться подпрограммой *jac*; *fjac* - массив формы (*Ldfjac, n*), содержащий вычисленный в точке *x* *m*×*n*-якобиан; *Ldfjac* - ведущий размер массива *fjac*. Подпрограмма *jac* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	"

Ldfjac	Ведущий размер массива *fjac* / *входной*	INTEGER(4)
m	Число функций / *входной*	"
n	Размер задачи (число переменных; $n \le m$) / *входной*	"
rparam	Вектор вещественных параметров размера 7. Содержит следующие параметры: *rparam*(1) - масштабируемый допуск для градиента. Компонент *i* масштабированного градиента вычисляется по формуле $\frac{\lvert g_i \rvert * \max(\lvert x_i \rvert, 1/s_i)}{\lVert f(x) \rVert_2^2}$, где $g_i = \left(J(x)^T f(x)\right)_i * (f_s)_i^2$, $J(x)$ - якобиан, s = *xscale* и f_s = *fscale*; по умолчанию равен $\sqrt{\varepsilon_м}$ при одинарной точности вычислений и $\sqrt[3]{\varepsilon_м}$ при двойной, где $\varepsilon_м$ - машинная точность; *rparam*(2) - масштабированный допуск для шага (*steptl*). Компонент *i* масштабированного допуска для шага между двумя точками *x* и *y* вычисляется по формуле $\frac{\lvert x_i - y_i \rvert}{\max(\lvert x_i \rvert, 1/s_i)}$, где s = *xscale*; по умолчанию равен $\varepsilon_м^{2/3}$; *rparam*(3) - относительный допуск для функции; по умолчанию равен $\max(10^{-10}, \varepsilon_м^{2/3})$ при одинарной точности вычислений и $\max(10^{-20}, \varepsilon_м^{2/3})$ при двойной; *rparam*(4) - абсолютный допуск для функции; по умолчанию равен $\max(10^{-20}, \varepsilon_м^{2})$ при одинарной точности вычислений и $\max(10^{-40}, \varepsilon_м^{2})$ при двойной; *rparam*(5) - допуск для ложной сходимости; по умолчанию равен $100\varepsilon_м$; *rparam*(6) - минимально допустимый шаг; по умолчанию равен $1000\max(\varepsilon_1, \varepsilon_2)$, где $\varepsilon_1 = \sqrt{\sum_{i=1}^{n} (s_i t_i)^2}$, $\varepsilon_2 = \lVert s \rVert_2$, s = *xscale* и t = *xguess*; *rparam*(7) - начальный радиус доверительной области. Основывается на масштабированном шаге Коши / *входной/выходной*	REAL(4) или REAL(8)

x	Вектор размера *n*, содержащий вычисленное решение / *выходной*	REAL(4) или REAL(8)
xguess	Вектор размера *n*, содержащий начальное предположение о точке минимума / *входной*	То же
xlb	Вектор размера *n*, содержащий нижние границы переменных / *входной*, если *ibtype* = 0; *выходной*, если *ibtype* = 1 или 2; *входной/выходной*, если *ibtype* = 3	"
xscale	Вектор размера *n*, содержащий диагональную матрицу масштабирования переменных. Вектор *xscale* применяется главным образом для масштабирования градиента и расстояния между двумя точками; по умолчанию вектор *xscale* определяется подпрограммой UNLSF самостоятельно (см. также параметр *iparam*(6)) / *входной*	"
xub	Вектор размера *n*, содержащий верхние границы переменных / *входной*, если *ibtype* = 0; *выходной*, если *ibtype* = 1 или 2; *входной/выходной*, если *ibtype* = 3	"

Таблица 4.4. Информационные ошибки подпрограмм, решающих задачу минимизации функции нескольких переменных

Тип	*Код*	*Описание*	*Где возникает*
4	1	Превыщено максимально допустимое число оценок функции	**POL
3	1	Реальное и предсказанное относительное уменьшение функции меньше или равно относительного допуска для функции	***NF, ***NG, ***DH, ***AH, **LSF, **LSJ
4	2	Итерации сходятся к неминимальной точке	То же
4	3	Превышено максимально допустимое число итераций	"
4	4	Превышено максимально допустимое число оценок функции	"
4	5	Превышено максимально допустимое число оценок градиента (якобиана в случае UNLSF и UNLSJ)	"
4	6	Пять последовательных шагов выполнены с максимальной длиной шага	"
4	7	Превышено максимально допустимое число оценок гессиана	***DH, ***AH

2	7	Масштабированный допуск для шага удовлетворен; текущая точка может быть приблизительным локальным решением, или алгоритм прогрессирует чрезмерно медленно и не находится около решения, или величина *steptl* слишком велика	***NF, ***NG, ***DH, ***AH, **LSF, **LSJ
3	8	На последнем глобальном шаге не удалось переместиться в более низкую, чем текущее значение *x*, точку	***NF, ***NG, ***DH, ***AH

Примечание. В приведенной таблице 3 звездочки обозначают UMI или BCO, две - UM или BC.

4.3.2. ПЕРЕЧЕНЬ ПОДПРОГРАММ

Минимум функции нескольких переменных (без ограничений) вычисляют приведенные в табл. 4.5 подпрограммы. Их параметры в алфавитном порядке перечислены в табл. 4.2 и 4.3, а информационные ошибки - в табл. 4.4.

Таблица 4.5. Подпрограммы, определяющие минимум функции нескольких переменных

Подпрограмма	*Описание*
UMINF	Находит минимум функции *n* переменных, используя квазиньютоновский метод и конечно-разностный градиент
UMING	Находит минимум функции *n* переменных, используя квазиньютоновский метод и предоставленный пользователем градиент
UMIDH	Находит минимум функции *n* переменных, используя модифицированный метод Ньютона и конечно-разностный гессиан
UMIAH	Находит минимум функции *n* переменных, используя модифицированный метод Ньютона и предоставленный пользователем гессиан
UMCGF	Находит минимум функции *n* переменных, используя метод сопряженных градиентов и конечно-разностный градиент
UMCGG	Находит минимум функции *n* переменных, используя метод сопряженных градиентов и предоставленный пользователем градиент
UMPOL	Находит минимум функции *n* переменных, используя метод деформируемого многогранника
Нелинейный метод наименьших квадратов без ограничений	
UNLSF	Решает нелинейную задачу наименьших квадратов, используя модифицированный алгоритм Левенберга - Маркварда и конечно-разностный якобиан

UNLSJ	Решает нелинейную задачу наименьших квадратов, используя модифицированный алгоритм Левенберга - Марквардта и предоставленный пользователем якобиан

4.3.3. ПОДПРОГРАММА UMINF (DUMINF)

Находит минимум функции n переменных, используя квазиньютоновский метод и конечно-разностный градиент. Имеет вызов

CALL UMINF(*fcn, n, xguess, xscale, fscale, iparam, rparam, x, fvalue*)

Описание параметров см. в табл. 4.2, информационных ошибок - в табл. 4.4.

Комментарии:

1. При работе с UMINF могут возникать приведенные в табл. 4.4 информационные ошибки.
2. Первый критерий останова UMINF удовлетворяется, когда норма масштабированного градиента меньше, чем заданный допуск для градиента (*rparam*(1)). Второй критерий завершения работы UMINF выполняется, когда масштабированное расстояние между двумя последними шагами меньше, чем допуск для шага (*rparam*(2)).
3. Если необходимо использовать заданные по умолчанию параметры UMINF, содержащиеся в *iparam* и *rparam*, установите *iparam*(1) = 0 и вызовите UMINF. В противном случае перед вызовом UMINF выполните вызов

   CALL U4INF(*iparam, rparam*)

   и задайте требуемые значения элементов *iparam* и *rparam*. Заметьте, что вызов U4INF установит в *iparam* и *rparam* величины, заданные по умолчанию. Поэтому после этого вызова достаточно определить изменяемые элементы *iparam* и *rparam*. При работе с двойной точностью вызывается подпрограмма DU4INF и вектор *rparam* объявляется как REAL(8).
4. Характер реагирования на информационные ошибки изменяется подпрограммой ERSET, приведенной в [5, разд. 3.3.3].

Описание:

Подпрограмма UMINF использует квазиньютоновский метод поиска минимума функции $f(x)$ от n переменных ($\min_{x \in \mathbf{R}^n} f(x)$). При работе нужны только значения функции. Начиная со стартовой точки x_c направление поиска вычисляется по формуле

$$d = -B^{-1} g_c,$$

где B - положительно определенная аппроксимация гессиана, а g_c - градиент, вычисленный в точке x_c. Для определения следующей точки x_n применяется линейный поиск:

$$x_n = x_c + \lambda d, \lambda > 0,$$

такой, что

$$f(x_n) \le f(x_c) + \alpha g^T d, \alpha \in (0, 0.5).$$

Затем проверяется условие оптимальности $\|g(x)\| = \varepsilon$, где ε - допуск для градиента. Если условие не выполняется, то матрица B обновляется по положительно определенной формуле секущих

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s},$$

где $s = x_n - x_c$ и $y = g_n - g_c$. Вычисляется другое направление поиска и осуществляется следующая итерация. Детали см. в [6, прил. А].

Если при работе с одинарной точностью погрешность оценки градиента приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если градиент можно оценить пользовательской процедурой, то вместо подпрограммы UMINF лучше употребить подпрограмму UMING.

Пример. Ищется минимум функции Розенброка (рис. 4.3) $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

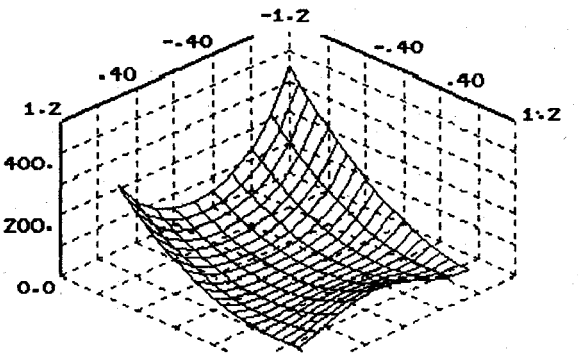


Рис. 4.3. Функция Розенброка

```
program uminfTest
 use dfimsl
 integer(4), parameter :: n = 2
 integer(4) :: iparam(7), k
```

```
real(4) :: f, fscale, rparam(7), x(n), xguess(n), xscale(n)
external rosbrk
data xguess / -1.2e0, 1.0e0 /, xscale / 1.0e0, 1.0e0 /, fscale / 1.0e0 /
call u4inf(iparam, rparam)                ! Ослабляем допуск для градиента
rparam(1) = 10.0e0 * rparam(1)
! Минимизируем функцию Розенброка, используя начальное предположение
! о минимуме (-1.2, 1.0)
call uminf(rosbrk, n, xguess, xscale, fscale, iparam, rparam, x, f)
write(*, 1) x, f, (iparam(k), k = 3, 5)   ! Вывод результата
1 format (' The solution is ', 6x, 2f8.3, /, ' The function value is ', f8.3, /,
      ' The number of iterations is ', 10x, i3, /, ' The number of function evaluations is ',
      i3, /, ' The number of gradient evaluations is ', i3)
end program uminfTest

subroutine rosbrk(n, x, f)                ! Оценивает функцию в точке x
 integer(4) :: n; real(4) :: x(n), f
 f = 1.0e2 * (x(2) - x(1) * x(1))**2 + (1.0e0 - x(1))**2
end subroutine rosbrk
```

Результат:

```
The solution is 1.000 1.000
The function value is 0.000
The number of iterations is 15
The number of function evaluations is 52
The number of gradient evaluations is 26
```

4.3.4. ПОДПРОГРАММА UMING (DUMING)

Находит минимум функции n переменных, используя квазиньютоновский метод и предоставленный пользователем градиент (первую производную). Имеет вызов

CALL UMING(*fcn, grad, n, xguess, xscale, fscale, iparam, rparam, x, fvalue*)

Описание параметров подпрограммы UMING см. в табл. 4.2, информационных ошибок - в табл. 4.4.

Комментарии и *описание* см. в разд. 4.3.3.

Пример. Ищется минимум функции Розенброка (рис. 4.3) $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

```
program umingTest
 . . .                                    ! Объявления см. в разд. 4.3.3
external rosbrk, rosgrd
data xguess / -1.2e0, 1.0e0 /, xscale / 1.0e0, 1.0e0 /, fscale / 1.0e0 /
iparam(1) = 0
call uming(rosbrk, rosgrd, n, xguess, xscale, fscale, iparam, rparam, x, f)
! Вывод результата. Операторы, используемые при выводе, см. в разд. 4.3.3
```

```
...
end program umingTest

subroutine rosbrk(n, x, f)                    ! Оценивает функцию в точке x
... ! Текст подпрограммы см. в предыдущем разделе
end subroutine rosbrk

! Оценивает градиент (первые частные производные) в точке x
subroutine rosgrd(n, x, g)
 integer(4) :: n; real(4) :: x(n), g(n)
 g(1) = -4.0e2 * (x(2) - x(1) * x(1)) * x(1) - 2.0e0 * (1.0e0 - x(1))
 g(2) = 2.0e2 * (x(2) - x(1) * x(1))
end subroutine rosgrd
```

Результат:

```
The solution is 1.000 1.000
The function value is 0.000
The number of iterations is 10
The number of function evaluations is 31
The number of gradient evaluations is 22
```

4.3.5. ПОДПРОГРАММА UMIDH (DUMIDH)

Находит минимум функции *n* переменных, используя модифицированный метод Ньютона и конечно-разностный гессиан. Имеет вызов:

CALL UMIDH(*fcn, grad, n, xguess, xscale, fscale, iparam, rparam, x, fvalue*)

Описание параметров подпрограммы UMIDH см. в табл. 4.2, информационных ошибок - в табл. 4.

Комментарий см. в разделе, содержащем описание подпрограммы UMINF.

Описание:

Подпрограмма UMIDH использует модифицированный метод Ньютона для вычисления минимума функции $f(x)$ от *n* переменных. Алгоритм вычисляет оптимальный локальный шаг [13], ограниченный доверительной областью, что позволяет работать с неопределенным гессианом и с отрицательной кривизной.

В целом процедура поиска минимума аналогична процедуре, используемой в подпрограмме UMINF, за тем исключением, что направление поиска вычисляется по формуле

$$d = -H^{-1}g_c,$$

где H - гессиан, а g_c - градиент, вычисленный в текущей точке x_c. Детали см. в [6, прил. А] и [13].

Если при работе с одинарной точностью погрешность оценки гессиана приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если гессиан можно оценить пользовательской процедурой, то вместо подпрограммы UMIDH лучше употребить подпрограмму UMIAH.

Пример. Ищется минимум функции Розенброка (рис. 4.3) $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

```
program umidhTest
 . . .  ! Объявления и инициализацию см. выше
 call umidh(rosbrk, rosgrd, n, xguess, xscale, fscale, iparam, rparam, x, f)
 ! Вывод результата
 . . .  ! Операторы, используемые при выводе результата, см. в разд. 4.3.3
end program umidhTest

subroutine rosbrk(n, x, f)                ! Оценивает функцию в точке x
 . . .  ! Текст подпрограммы см. в разд. 4.3.3
end subroutine rosbrk

! Оценивает градиент (первые частные производные) в точке x
subroutine rosgrd(n, x, g)
 . . .  ! Текст подпрограммы см. разд. 4.3.4
end subroutine rosgrd
```

Результат:

```
The solution is 1.000 1.000
The function value is 0.000
The number of iterations is 21
The number of function evaluations is 30
The number of gradient evaluations is 22
```

4.3.6. ПОДПРОГРАММА UMIAH (DUMIAH)

Находит минимум функции *n* переменных, используя модифицированный метод Ньютона и предоставленный пользователем гессиан (вторую производную). Имеет вызов

```
CALL UMIAH(fcn, grad, hess, n, xguess, xscale, fscale,                &
     iparam, rparam, x, fvalue)
```

Описание параметров подпрограммы UMIAH см. в табл. 4.2, информационных ошибок - в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UMIDF.

Описание см. в предыдущем разделе.

Пример. Ищется минимум функции Розенброка (рис. 4.3) $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

```
program umidhTest
 . . .   ! Объявления см. выше
 external rosbrk, rosgrd, roshes
 data xguess / -1.2e0, 1.0e0 /, xscale / 1.0e0, 1.0e0 /, fscale / 1.0e0 /
 iparam(1) = 0
 call umiah(rosbrk, rosgrd, roshes, n, xguess, xscale, fscale, iparam, rparam, x, f)
 ! Вывод результата
 . . .   ! Операторы, используемые при выводе результата, см. в разд. 4.3.3
end program umidhTest

subroutine rosbrk(n, x, f)                    ! Оценивает функцию в точке x
 . . .   ! Текст подпрограммы см. в разд. 4.3.3
end subroutine rosbrk

! Оценивает градиент (первые частные производные) в точке x
subroutine rosgrd(n, x, g)
 . . .   ! Текст подпрограммы см. разд. 4.3.4
end subroutine rosgrd

! Оценивает гессиан (вторые частные производные) в точке x
subroutine roshes(n, x, h, Ldh)
 integer(4) :: n, Ldh; real(4) :: x(n), h(Ldh, n)
 h(1, 1) = -4.0e2 * x(2) + 1.2e3 * x(1) * x(1) + 2.0e0; h(2, 1) = -4.0e2 * x(1)
 h(1, 2) = h(2, 1); h(2, 2) = 2.0e2
end subroutine roshes
```

Результат:

```
The solution is 1.000 1.000
The function value is 0.000
The number of iterations is 21
The number of function evaluations is 31
The number of gradient evaluations is 22
```

4.3.7. ПОДПРОГРАММА UMCGF (DUMCGF)

Находит минимум функции *n* переменных, используя метод сопряженных градиентов и конечно-разностный градиент. Имеет вызов

CALL UMCGF(*fcn, n, xguess, xscale, gradtl, maxfn, dfpred, x, g, fvalue*)

Описание параметров подпрограммы UMCGF см. в табл. 4.2.

Комментарии:

1. При работе с UMCGF могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
4	1	Линейный поиск прекращен. Эта ошибка может быть вызвана ошибкой вычисления градиента
4	2	Вычисления не могут продолжаться, поскольку вместо спуска наблюдается подъем
4	3	Итерации прекращены, т. к. превышено максимально допустимое число оценок функции *maxfn*
3	4	Вычисления прекращены, т. к. две последовательные итерации не смогли уменьшить функцию

2. Из-за схожести методов сопряженных градиентов и наискорейшего спуска полезно задавать вектор масштабирования *xscale* таким, чтобы он балансировал компоненты вектора градиента. Метод может быть неэффективным, если несколько компонентов градиента существенно больше остальных.
3. Если *gradtl* = 0.0, то подпрограмма продолжает вычисления до тех пор, пока снижается значение целевой функции. В этом случае часто оказывается, что уменьшение целевой функции вызвано ошибками округления. Поэтому в окончательном решении может быть точной только половина определяемых компьютерной арифметикой значащих цифр. В то же время наименьшее значение *fvalue* обычно оказывается весьма точным.

Описание:

Подпрограмма UMCGF основана на версии метода сопряженных градиентов, приведенной в [34]. Основные преимущества метода состоят в его относительно быстрой сходимости и отсутствии промежуточных матриц. Поэтому он хорош для задач безусловной минимизации большой размерности. В то же время подпрограммы, использующие матрицы, например UMINF, как правило, более эффективны, поскольку на каждой итерации используется дополнительная информация о предшествующих итерациях.

Если при работе с одинарной точностью погрешность оценки градиента приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если градиент можно оценить пользовательской процедурой, то вместо подпрограммы UMCGF лучше употребить подпрограмму UMCGG.

Пример. Ищется минимум функции Розенброка (рис. 4.3) $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

```
program umcgfTest
 use dfimsl
 integer(4), parameter :: n = 2
 integer(4) :: i, maxfn
```

```
real(4) :: dfpred, fvalue, g(n), gradtl, x(n), xguess(n), xs(n)
external rosbrk
data xguess / -1.2e0, 1.0e0 /, xs / 2*1.0e0 /
dfpred = 0.2; gradtl = 1.0e-6; maxfn = 100
! Ищем минимум функции Розенброка
call umcgf(rosbrk, n, xguess, xs, gradtl, maxfn, dfpred, x, g, fvalue)
! Вывод результата
write(*, 1) (x(i), i = 1, n), fvalue, (g(i), i = 1, n)
1 format(' The solution is ', 2f8.3, /, ' The function evaluated at the solution is ', f8.3, /,
      ' the gradient is ', 2f8.3, /)
end program umcgfTest

subroutine rosbrk(n, x, f)                    ! Оценивает функцию в точке x
... ! Текст подпрограммы см. в разд. 4.3.3
end subroutine rosbrk
```

Результат:

```
The solution is  1.000  1.000
The function evaluated at the solution is 0.000
The gradient is  0.000  0.000
```

4.3.8. ПОДПРОГРАММА UMCGG (DUMCGG)

Находит минимум функции *n* переменных, используя метод сопряженных градиентов и предоставленный пользователем градиент. Имеет вызов

CALL UMCGG(*fcn, grad, n, xguess, gradtl, maxfn, dfpred, x, g, fvalue*)

Описание параметров подпрограммы UMCGG см. в табл. 4.2.

Комментарии и *описание* см. в предшествующем разделе.

В задаче поиска минимума функции Розенброка получается тот же результат, что и для подпрограммы UMCGF.

4.3.9. ПОДПРОГРАММА UMPOL (DUMPOL)

Находит минимум функции *n* переменных, используя метод деформируемого многогранника (симплекса). Имеет вызов

CALL UMPOL(*fcn, n, xguess, s, ftol, maxfcn, x, fvalue*)

Описание параметров подпрограммы UMPOL см. в табл. 4.2, информационной ошибки - в табл. 4.4.

Комментарии:

1. Поскольку подпрограмма UMPOL использует для определения следующей точки только значения функции, подпрограмма может оказаться неэффективной в гладких задачах по сравнению с методом, реализованным в UMINF, учитывающим дополнительно информацию о теку-

щей производной. Поэтому UMPOL следует использовать как последний ресурс.

2. Возвращаемая величина *s* полезна для оценки пологости функции вычисленного минимума. Чем больше *s*, тем более полога функция.

Описание:

Метод деформируемого многогранника основывается на сравнении значений функции, в общем случае негладкой. Для старта метода задаются $n + 1$ точка: $x_1, x_2, \ldots, x_{n+1}$. На каждой итерации генерируется новая точка, заменяющая наихудшую (из $n + 1$ точки) точку x_j, т. е. точку, в которой функция имеет наибольшее значение. Новая точка находится по формуле

$$x_k = c + \alpha(c - x_j),$$

где

$$c = \frac{1}{n}\sum_{i \neq j} x_i$$

и α ($\alpha > 0$) - коэффициент отражения.

Когда x_k является лучшей точкой, т. е. $f(x_k) \le f(x_i)$ для $i = 1, \ldots, n + 1$, вычисляется точка расширения $x_c = c + \beta(x_k - c)$, где β ($\beta > 1$) - коэффициент расширения. Если новая точка окажется наихудшей, то многогранник (симплекс) сжимается, чтобы получить более хорошую новую точку. Если сжатие окажется нерезультативным, то симплекс деформируется путем приближения вершин к текущей наилучшей точке. В результате приближения расстояние между точками x_i и x_k ($i = 1, \ldots, n + 1$ и $i \neq k$) сокращается в 2 раза. Описанные действия повторяются, пока не выполняется один из следующих критериев останова:

$$f_{наилучшая} - f_{наихудшая} \le \varepsilon_f(1.0 + |f_{наилучшая}|)$$

или

$$\sum_{i=1}^{n+1}\left(f_i - \frac{\sum_{j=1}^{n+1} f_j}{n+1}\right)^2 \le \varepsilon_f,$$

где $f_i = f(x_i)$, $f_j = f(x_j)$, а ε_f - заданный допуск. Детали см. в [16] или [32].

Пример. Ищется минимум функции Розенброка (рис. 4.3) $f(x) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$.

```
program umpolTest
 use dfimsl
 integer(4), parameter :: n = 2
 integer(4) :: maxfn
```

```
real(4) :: ftol, fvalue, s, x(n), xguess(n)
external fcn
data xguess / -1.2, 1.0 /                    ! Инициализация
ftol = 1.0e-10; maxfcn = 200; s = 1.0  ! Поиск минимума
call umpol(fcn, n, xguess, s, ftol, maxfcn, x, fvalue)
write(*, 1) x(1:n), fvalue
1 format(' The best estimate for the minimum value of the function is x = (',
      2(2x, f4.2),')', /, ' with function value fvalue = ', e12.6)
end program umpolTest

subroutine fcn(n, x, f)                      ! Оценивает функцию в точке x
integer(4) :: n; real(4) :: x(n), f
f = 100.0 * (x(1) * x(1) - x(2))**2 + (1.0 - x(1))**2
end subroutine fcn
```

Результат:

```
The best estimate for the minimum value of the function is x = ( 1.00 1.00)
with function value fvalue = 0.247835E-10
```

4.3.10. НЕЛИНЕЙНЫЙ МЕТОД НАИМЕНЬШИХ КВАДРАТОВ С ПРОСТЫМИ ОГРАНИЧЕНИЯМИ

4.3.10.1. Подпрограмма UNLSF (DUNLSF)

Решает нелинейную задачу наименьших квадратов, используя модифицированный алгоритм Левенберга - Маркварда и конечно-разностный якобиан. Имеет вызов

CALL UNLSF(*fcn, m, n, xguess, xscale, fscale, iparam,* &
rparam, x, fvec, fjac, Ldfjac)

Описание параметров подпрограммы UNLSF см. в табл. 4.3.

Комментарии:

1. При работе с UNLSF могут возникать приведенные в табл. 4.4 информационные ошибки.
2. Останов UNLSF по первому критерию происходит, когда норма функции меньше, чем абсолютный допуск для функции (*rparam*(4)). Второй критерий останова UNLSF удовлетворяется, когда норма масштабированного градиента меньше, чем заданный допуск для градиента (*rparam*(1)). Третий критерий завершения работы UNLSF выполняется, когда масштабированное расстояние между двумя последними шагами меньше, чем допуск для шага (*rparam*(2)).
3. См. комментарии 3 и 4 для подпрограммы UMINF.

Описание:

Подпрограмма UNLSF основана на процедуре LMDIF пакета MINPACK [28]. Решаемая подпрограммой UNLSF задача наименьших квадратов формулируется следующим образом:

$$\min_{x\in\mathbf{R}^n}\frac{1}{2}f(x)^T f(x)=\frac{1}{2}\sum_{i=1}^{m}f_i(x)^2,$$

где $m \geq n$, f: $\mathbf{R}^n \to \mathbf{R}^m$ - и $f_i(x)$ - i-й компонент вектор-функции $f(x)$.

Начиная со стартовой точки x_c подпрограмма вычисляет направление поиска по формуле

$$d = -(J^T J + \mu I)^{-1} J^T f,$$

где μ - параметр Левенберга - Маркварда, $f = f(x)$, а J - якобиан. Далее алгоритм использует доверительную область

$$\min_{x\in\mathbf{R}^n}\left\|f(x_c)+J(x_c)(x_n-x_c)\right\|_2$$

с ограничениями для длины шага $||x_n - x_c||_2 \leq \delta_c$, чтобы получить новую точку x_n, вычисляя ее по формуле

$$x_n = x_c - \left(J(x_c)^T J(x_c)+\mu_c I\right)^{-1} J(x_c)^T f(x_c),$$

где $\mu_c = 0$, если $\delta_c \geq \|(J(x_c)^T J(x_c))^{-1} J(x_c)^T f(x_c)\|_2$, и $\mu_c > 0$ - в противном случае; $f(x_c)$ и $J(x_c)$ - соответственно значения функций и якобиана, вычисленные в текущей точке x_c. Приведенная процедура повторяется, пока не удовлетворяется хотя бы один из критериев останова. Детали см. в [6, гл. 10)], [25] и [27].

Если при работе с одинарной точностью погрешность оценки якобиана приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если якобиан можно оценить пользовательской процедурой, то вместо подпрограммы UNLSF лучше употребить подпрограмму UNLSJ.

Пример. Поиск минимума функции Розенброка сводится к решению нелинейной задачи наименьших квадратов

$$\min_{x\in\mathbf{R}^2}\frac{1}{2}\sum_{i=1}^{2}f_i(x)^2,$$

где $f_1(x) = 10(x_2 - x_1^2)$ и $f_2 = (1 - x_1)$. Задается отличное от установленного по умолчанию значение элемента *rparam*(4).

```
program unlsfTest
 integer(4), parameter :: Ldfjac = 2, m = 2, n = 2
 integer(4) :: iparam(6)
```

```
real(4) :: fjac(Ldfjac, n), fscale(m), fvec(m), rparam(7), x(n), xguess(n), xscale(n)
external rosbck
data xguess / -1.2e0, 1.0e0 /, xscale/ 2*1.0e0 /, fscale/ 2*1.0e0 /
! Ослабляем первый критерий останова, вызывая U4LSF
! и увеличивая абсолютный допуск для функции в 10 раз
call u4lsf(iparam, rparam); rparam(4) = 10.0e0 * rparam(4)
! Решаем задачу наименьших квадратов
call unlsf(rosbck, m, n, xguess, xscale, fscale, iparam, rparam, x, fvec, fjac, Ldfjac)
write(*, 1) x, fvec, iparam(3), iparam(4)
1 format (' The solution is ', 2f9.4, //, ' The function evaluated at the solution is ',
      2f9.4, //, ' The number of iterations is ', 10x, i3, /,                    &
      ' The number of function evaluations is ', i3, /)
end program unlsfTest

subroutine rosbck(m, n, x, f)              ! Оценка задачи наименьших квадратов
 integer(4) :: m, n; real(4) :: x(n), f(m)
 f(1) = 10.0e0 * (x(2) - x(1) * x(1)); f(2) = 1.0e0 - x(1)
end subroutine rosbck
```

Результат:

```
The solution is 1.0000 1.0000
The function evaluated at the solution is 0.0000 0.0000
The number of iterations is 22
The number of function evaluations is 30
```

4.3.10.2. Подпрограмма UNLSJ (DUNLSJ)

Решает нелинейную задачу наименьших квадратов, используя модифицированный алгоритм Левенберга - Марквардта и предоставленный пользователем якобиан. Имеет вызов

CALL UNLSJ(*fcn, jac, m, n, xguess, xscale, fscale,* &
iparam, rparam, x, fvec, fjac, Ldfjac)

Описание параметров подпрограммы UNLSJ см. в табл. 4.3.

Комментарии и *описание* см. в предшествующем разделе.

Пример. Решается та же, что и для приведенной в предшествующем разделе функции UNLSF, задача. Заданные по умолчанию значения параметров сохраняются.

```
program unlsjTest
 . . .   ! Объявления см. в предшествующем примере
external rosbck, rosjac
data xguess / -1.2e0, 1.0e0 /, xscale/ 2*1.0e0 /, fscale/ 2*1.0e0 /
! Сохраняем заданные по умолчанию значения параметров
iparam(1) = 0
! Решаем задачу наименьших квадратов
```

```
call unlsj(rosbck, rosjac, m, n, xguess, xscale, fscale, iparam, rparam, x, fvec, fjac, Ldfjac)
write(*, 1) x, fvec, iparam(3), iparam(4), iparam(5)
1 format (' The solution is ', 2f9.4, //, ' The function evaluated at the solution is ',
      2f9.4, //, ' The number of iterations is ', 10x, i3; /,                    &
      ' The number of function evaluations is ', i3, /                          &
      ' The number of jacobian evaluations is ', i3, /)
end program unlsjTest

subroutine rosbck(m, n, x, f)          ! Оценка задачи наименьших квадратов
...   ! Код см. в разд. 4.3.10.1
end subroutine rosbck

subroutine rosjac(m, n, x, fjac, Ldfjac) ! Оценивает якобиан
 integer(4) :: m, n, Ldfjac; real(4) :: x(n), fjac(Ldfjac, n)
 fjac(1, 1) = -20.0e0 * x(1); fjac(2, 1) = -1.0e0; fjac(1, 2) = 10.0e0; fjac(2, 2) = 0.0e0
end subroutine rosjac
```

Результат:

```
The solution is 1.0000 1.0000
The function evaluated at the solution is 0.0000 0.0000
The number of iterations is 22
The number of function evaluations is 31
The number of Jacobian evaluations is 23
```

4.4. МИНИМИЗАЦИЯ С ПРОСТЫМИ ОГРАНИЧЕНИЯМИ

4.4.1. ПЕРЕЧЕНЬ, ПАРАМЕТРЫ И ИНФОРМАЦИОННЫЕ ОШИБКИ ПОДПРОГРАММ

Минимум функции n переменных с ограничениями вида $l_i \le x_i \le u_i$ для $i = 1, 2, \ldots, n$ вычисляют приведенные в табл. 4.6 подпрограммы. Их параметры в алфавитном порядке перечислены в табл. 4.2 и 4.3, а информационные ошибки в - табл. 4.4.

Таблица 4.6. Подпрограммы, определяющие минимум функции нескольких переменных с простыми ограничениями

Подпрограмма	*Описание*
BCONF	Находит минимум функции n переменных, используя квазиньютоновский метод и конечно-разностный градиент
BCONG	Находит минимум функции n переменных, используя квазиньютоновский метод и предоставленный пользователем градиент
BCODH	Находит минимум функции n переменных, используя модифицированный метод Ньютона и конечно-разностный гессиан

BCOAH	Находит минимум функции *n* переменных, используя модифицированный метод Ньютона и конечно-разностный гессиан
BCPOL	Находит минимум функции *n* переменных, используя метод деформируемого многогранника
BCLSF	Решает нелинейную задачу наименьших квадратов, используя модифицированный алгоритм Левенберга - Марквардa и конечно-разностный якобиан
BCLSJ	Решает нелинейную задачу наименьших квадратов, используя модифицированный алгоритм Левенберга - Марквардa и предоставленный пользователем якобиан
BCNLS	Решает нелинейную задачу наименьших квадратов с ограничениями на значения переменных и линейными ограничениями общего вида

4.4.2. ПОДПРОГРАММА BCONF (DBCONF)

Находит минимум функции *n* переменных с простыми ограничениями, используя квазиньютоновский метод и конечно-разностный градиент. Имеет вызов

CALL BCONF(*fcn, n, xguess, ibtype, xlb, xub, xscale, fscale,* &
iparam, rparam, x, fvalue)

Смысл параметров подпрограммы BCONF см. в табл. 4.2; информационные ошибки приведены в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UMINF.

Описание:

Подпрограмма BCONF решает следующую задачу:

$$\min_{x \in \mathbf{R}^n} f(x)$$

при

$$l < x < u.$$

Начиная со стартовой точки x_c создается активный набор переменных IA, содержащий индексы переменных и их границы. Переменные, не находящиеся в активном наборе, называются *свободными переменными*. Затем подпрограмма вычисляет направление поиска для свободных переменных по формуле

$$d = -B^{-1} g_c,$$

где B - положительно определенная аппроксимация гессиана, а g_c - градиент, оцененный в точке x_c; оба вычисляются для свободных переменных.

Направление поиска в наборе IA устанавливается равным нулю. Для поиска следующей точки x используется линейный поиск

$$x_n = x_c + \lambda d, \lambda \in (0, 1],$$

такой, что

$$f(x_n) \le f(x_c) + \alpha g^T d, \alpha \in (0, 0.5).$$

Далее проверяются условия оптимальности

$$\|g(x_i)\| \le \varepsilon, l_i < x_i < u_i,$$

$$g(x_i) < 0, x_i = u_i,$$

$$g(x_i) > 0, x_i = l_i,$$

где ε - допуск для градиента. Когда оптимальность не достигается, матрица B пересчитывается по положительно определенной формуле секущих

$$B \leftarrow B - \frac{Bss^T B}{s^T Bs} + \frac{yy^T}{y^T s},$$

где $s = x_n - x_c$ и $y = g_n - g_c$; вычисляется другое направление поиска, и начинается следующая итерация.

Процесс продолжается, пока не выполнены условия оптимальности.

Активный набор IA изменяется, лишь когда свободная переменная достигает своей границы или когда условия оптимальности удовлетворяются для всех свободных переменных, но не для всех переменных в наборе IA. В последнем случае переменная, нарушающая условия оптимальности, будет извлечена из набора IA. Детали, связанные с квазиньютоновским линейным поиском, см. в [6]. Стратегия работы с активным набором рассмотрена в [14].

Если при работе с одинарной точностью погрешность оценки градиента приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если градиент можно оценить пользовательской процедурой, то вместо подпрограммы BCONF лучше употребить подпрограмму BCONG.

Пример. Ищется минимум функции Розенброка

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

с ограничениями

$$-2 \le x_1 \le 0.5,$$

$$-1 \le x_2 \le 2.$$

Начальная точка имеет координаты (-1.2, 1.0). Используются заданные по умолчанию параметры.

```
program bconfTest
 use dfimsl
 integer(4), parameter :: n = 2
 integer(4) :: iparam(7), itp, k
 real(4) :: f, fscale, rparam(7), x(n), xguess(n), xlb(n), xscale(n), xub(n)
 external rosbrk
 data xguess / -1.2e0, 1.0e0 /, xscale / 1.0e0, 1.0e0 /, fscale / 1.0e0 /
 data xlb / -2.0e0, -1.0e0 /, xub / 0.5e0, 2.0e0 /
 itp = 0                                  ! Накладываем все ограничения
 call u4inf(iparam, rparam)               ! Ослабляем допуск для градиента
 rparam(1) = 10.0e0 * rparam(1)
 ! Ищем минимум функции Розенброка с простыми ограничениями
 call bconf(rosbrk, n, xguess, itp, xlb, xub, xscale, fscale, iparam, rparam, x, f)
 write(*, 1) x, f, (iparam(k), k = 3, 5)
 1 format (' The solution is ', 6x, 2f8.3, /, ' The function value is ', f8.3, /,
       ' The number of iterations is ', 10x, i3, /, ' The number of function evaluations is ',
       i3, /, ' The number of gradient evaluations is ', i3)
end program bconfTest

subroutine rosbrk(n, x, f)               ! Оценивает функцию в точке x
 integer(4) :: n; real(4) :: x(n), f
 f = 1.0e2 * (x(2) - x(1) * x(1))**2 + (1.0e0 - x(1))**2
end subroutine rosbrk
```

Результат:

```
The solution is 0.500 0.250
The function value is 0.250
The number of iterations is 23
The number of function evaluations is 33
The number of gradient evaluations is 25
```

4.4.3. ПОДПРОГРАММА BCONG (DBCONG)

Находит минимум функции *n* переменных с простыми ограничениями, используя квазиньютоновский метод и предоставленный пользователем градиент. Имеет вызов

CALL BCONG(*fcn, grad, n, xguess, ibtype, xlb, xub,* &
xscale, fscale, iparam, rparam, x, fvalue)

Смысл параметров подпрограммы BCONG см. в табл. 4.2; информационные ошибки приведены в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UMINF.

Описание см. в предшествующем разделе.

Для задачи, решенной в качестве *примера* в предыдущем разделе, употребление BCONG даст следующий *результат*:

```
The solution is 0.500 0.250
The function value is 0.250
The number of iterations is 23
The number of function evaluations is 32
The number of gradient evaluations is 24
```

Подпрограмма, оценивающая градиент решаемой задачи, приведена в разделе с описанием подпрограммы UMING.

4.4.4. ПОДПРОГРАММА BCODH (DBCODH)

Находит минимум функции n переменных с простыми ограничениями, используя модифицированный метод Ньютона и конечно-разностный гессиан. Имеет вызов

```
CALL BCODH(fcn, grad, n, xguess, ibtype, xlb, xub,                &
     xscale, fscale, iparam, rparam, x, fvalue)
```

Смысл параметров подпрограммы BCODH см. в табл. 4.2; информационные ошибки приведены в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UMINF.

Описание:

В целом процедура поиска минимума аналогична процедуре, используемой в подпрограмме BCONF, за тем исключением, что направление поиска вычисляется по формуле

$$d = -H^{-1}g_c,$$

где H - гессиан, а g_c - градиент, вычисленный в текущей точке x_c.

Если при работе с одинарной точностью погрешность оценки гессиана приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если гессиан можно оценить пользовательской процедурой, то вместо подпрограммы BCODH лучше употребить подпрограмму BCOAH.

Для задачи, решенной в качестве *примера* в разделе, содержащем описание BCONF, употребление BCODH даст следующий *результат*:

```
The solution is 0.500 0.250
The function value is 0.250
The number of iterations is 17
The number of function evaluations is 26
The number of gradient evaluations is 18
```

4.4.5. ПОДПРОГРАММА BCOAH (DBCOAH)

Находит минимум функции *n* переменных с простыми ограничениями, используя модифицированный метод Ньютона и предоставленный пользователем гессиан. Имеет вызов

CALL BCOAH(*fcn, grad, hess, n, xguess, ibtype, xlb, xub,* &
xscale, fscale, iparam, rparam, x, fvalue)

Смысл параметров подпрограммы BCOAH см. в табл. 4.2; информационные ошибки приведены в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UMINF.

Описание см. в предшествующем разделе.

Для задачи, решенной в качестве *примера* в разделе, содержащем описание BCONF, употребление BCOAH даст следующий *результат*:

```
The solution is 0.500 0.250
The function value is 0.250
The number of iterations is 18
The number of function evaluations is 29
The number of gradient evaluations is 19
The number of Hessian evaluations is 18
```

Подпрограмма, оценивающая гессиан решаемой задачи, приведена в разделе с описанием подпрограммы UMIAH.

4.4.6. ПОДПРОГРАММА BCPOL (DBCPOL)

Находит минимум функции *n* переменных, используя метод деформируемого многогранника (комплекса). Имеет вызов

CALL BCPOL(*fcn, n, xguess, ibtype, xlb, xub, ftol, maxfcn, x, fvalue*)

Описание параметров подпрограммы BCPOL см. в табл. 4.2, информационной ошибки - в табл. 4.4.

Комментарий аналогичен комментарию, приведенному для подпрограммы UMPOL.

Описание:

Подпрограмма BCPOL использует для поиска минимума функции (в общем случае негладкой) комплекс-метод. (Напомним, что *комплексом* называется набор из $2n$ точек в n-мерном пространстве.) Процесс минимизации состоит в замене точки с наибольшим значением функции новой, более хорошей точкой. Итерации продолжаются до тех пор, пока все точки комплекса не подойдут близко к минимуму.

Для старта метода задаются $2n$ точек: $x_1, x_2, \ldots, x_{2n}$. На каждой итерации генерируется новая точка, заменяющая наихудшую (из $2n$ точек) точку x_j, т. е. точку, в которой функция имеет наибольшее значение. Новая точка находится по формуле

$$x_k = c + \alpha(c - x_j),$$

где

$$c = \frac{1}{2n-1}\sum_{i \neq j} x_i$$

и α ($\alpha > 0$) - коэффициент отражения.

Когда x_k является лучшей точкой, т. е. $f(x_k) \leq f(x_i)$ для $i = 1, \ldots, 2n$, вычисляется точка расширения $x_c = c + \beta(x_k - c)$, где β ($\beta > 1$) - коэффициент расширения. Если новая точка окажется наихудшей, то комплекс сжимается, чтобы получить более хорошую новую точку. Если сжатие окажется нерезультативным, то комплекс деформируется путем приближения вершин к текущей наилучшей точке. Если новая точка оказывается за пределами границ, она устанавливается на границе. Описанные действия повторяются, пока не выполняется один из следующих критериев останова:

$$f_{наилучшая} - f_{наихудшая} \leq \varepsilon_f(1.0 + |f_{наилучшая}|)$$

или

$$\sum_{i=1}^{2n}\left(f_i - \frac{\sum_{j=1}^{2n} f_j}{2n}\right)^2 \leq \varepsilon_f,$$

где $f_i = f(x_i)$, $f_j = f(x_j)$, а ε_f - заданный допуск. Детали см. в [16] или [32].

Для задачи, решенной в качестве *примера* в разделе, содержащем описание BCONF, и с формой вывода, примененной в примере для UMPOL, употребление BCPOL даст следующий *результат*:

```
The best estimate for the minimum value of the function is x = (0.50 0.25)
with function value fvalue = 0.250002E+00
```

4.4.7. НЕЛИНЕЙНЫЙ МЕТОД НАИМЕНЬШИХ КВАДРАТОВ С ПРОСТЫМИ ОГРАНИЧЕНИЯМИ

4.4.7.1. Подпрограмма BCLSF (DBCLSF)

Решает нелинейную задачу наименьших квадратов с ограничениями на значения переменных, используя модифицированный алгоритм Левенберга - Марквардта и конечно-разностный якобиан. Имеет вызов

CALL BCLSF(*fcn, m, n, xguess, ibtype, xlb, xub, xscale, fscale,* &
iparam, rparam, x, fvec, fjac, Ldfjac)

Описание параметров подпрограммы BCLSF см. в табл. 4.3, информационных ошибок - в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UNLSF.

Описание:

Подпрограмма BCLSF использует модифицированный алгоритм Левенберга - Марквардa и стратегию активного набора. Решаемая ей задача наименьших квадратов с простыми ограничениями формулируется следующим образом:

$$\min_{x \in \mathbf{R}^n} \frac{1}{2} f(x)^T f(x) = \frac{1}{2} \sum_{i=1}^{m} f_i(x)^2$$

при

$$l < x < u,$$

где $m \geq n$, $f: \mathbf{R}^n \to \mathbf{R}^m$ - и $f_i(x)$ - i-й компонент вектор-функции $f(x)$.

Начиная со стартовой точки x_c создается активный набор переменных IA, содержащий индексы переменных и их границы. Затем подпрограмма вычисляет направление поиска для свободных переменных по формуле

$$d = -(J^T J + \mu I)^{-1} J^T f,$$

где μ - параметр Левенберга - Марквардa, $f = f(x)$, а J - якобиан для свободных переменных. Направление поиска в наборе IA устанавливается равным нулю. Следующая точка определяется исходя из модели "доверительная область" [6], после чего проверяются условия оптимальности

$$\|g(x_i)\| \leq \varepsilon, \; l_i < x_i < u_i,$$

$$g(x_i) < 0, \; x_i = u_i,$$

$$g(x_i) > 0, \; x_i = l_i,$$

где ε - допуск для градиента.

Процесс продолжается, пока не выполнены условия оптимальности.

Активный набор IA изменяется, лишь когда свободная переменная достигает своей границы или когда условия оптимальности удовлетворяются для всех свободных переменных, но не для всех переменных в наборе IA. В последнем случае переменная, нарушающая условия оптимальности, будет извлечена из набора IA. Детали, связанные с методом Левенберга - Марквардa, см. в [25] или [27]. Стратегия работы с активным набором рассмотрена в [14].

Если при работе с одинарной точностью погрешность оценки якобиана приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если якобиан можно оценить пользовательской процедурой, то вместо подпрограммы BCLSF лучше употребить подпрограмму BCLSJ.

Пример. Поиск минимума функции Розенброка с ограничениями на значения переменных сводится к решению нелинейной задачи наименьших квадратов

$$\min_{x \in \mathbf{R}^2} \frac{1}{2} \sum_{i=1}^{2} f_i(x)^2$$

при

$$-2 \le x_1 \le 0.5,$$

$$-1 \le x_2 \le 2,$$

где $f_1(x) = 10(x_2 - x_1^2)$ и $f_2 = (1 - x_1)$. Начальная точка имеет координаты (-1.2, 1.0). Используются заданные по умолчанию параметры.

```
program bclsfTest
 use dfimsl
 integer(4), parameter :: Ldfjac = 2, n = 2, m = 2
 integer(4) :: iparam(7), itp, k
 real(4) :: fjac(Ldfjac, n), fscale(m), fvec(m), rparam(7), x(n),                &
    xguess(n), xlb(n), xs(n), xub(n)
 external rosbck
 data xguess / -1.2e0, 1.0e0 /, xs / 2*1.0e0 /, fscale / 2*1.0e0 /
 data xlb / -2.0e0, -1.0e0 /, xub / 0.5e0, 2.0e0 /
 itp = 0                                  ! Накладываем все ограничения
 ! Используются заданные по умолчанию параметры
 iparam(1) = 0
 ! Ищем минимум функции Розенброка с простыми ограничениями
 call bclsf(rosbck, m, n, xguess, itp, xlb, xub, xs, fscale,                    &
      iparam, rparam, x, fvec, fjac, Ldfjac)
 write(*, 1) x, fvec, (iparam(k), k = 3, 4)
 1 format(' The solution is ', 2f9.4, //, ' The function evaluated at the solution is ',  &
      2f9.4, //, ' The number of iterations is ', 10x, i3, /,                   &
      ' The number of function evaluations is ', i3, /)
end program bclsfTest

subroutine rosbck(m, n, x, f)            ! Оценивает функцию в точке x
 integer(4) :: m, n; real(4) :: x(n), f(m)
 f(1) = 1.0e1 * (x(2) - x(1) * x(1)); f(2) = 1.0e0 - x(1)
end subroutine rosbck
```

Результат:

The solution is 0.5000 0.2500
The function evaluated at the solution is 0.0000 0.5000
The number of iterations is 15
The number of function evaluations is 20

4.4.7.2. Подпрограмма BCLSJ (DBCLSJ)

Решает нелинейную задачу наименьших квадратов с ограничениями на значения переменных, используя модифицированный алгоритм Левенберга - Марквардa и предоставленный пользователем якобиан. Имеет вызов

CALL BCLSJ(*fcn, jac, m, n, xguess, ibtype, xlb, xub, xscale, fscale,* &
iparam, rparam, x, fvec, fjac, Ldfjac)

Описание параметров подпрограммы BCLSJ см. в табл. 4.3, информационных ошибок - в табл. 4.4.

Комментарии см. в разделе, содержащем описание подпрограммы UNLSF.

Описание см. в предшествующем разделе.

Для задачи, решенной в качестве *примера* в предшествующем разделе, употребление BCLSJ даст следующий *результат*:

The solution is 0.5000 0.2500
The function evaluated at the solution is 0.0000 0.5000
The number of iterations is 13
The number of function evaluations is 21

Подпрограмма, оценивающая якобиан решаемой задачи, приведена в разделе с описанием подпрограммы UNLSJ.

4.4.7.3. Подпрограмма BCNLS (DBCNLS)

Решает нелинейную задачу наименьших квадратов с ограничениями на значения переменных и линейными ограничениями общего вида. Имеет вызов

CALL BCNLS(*fcn, m, n, mcon, c, Ldc, bl, bu, irtype,* &
xlb, xub, xguess, x, rnorm, istat)

Параметры подпрограммы BCNLS:
Пользовательская подпрограмма: fcn.
Входные: m, n, mcon, c, Ldc, bl, bu, irtype, xlb, xub, xguess.
Выходные: x, rnorm, istat.
Описание параметров *fcn, m, n, xguess, x* см. в табл. 4.3.
mcon - число ограничений общего вида, не считая простые ограничения.
c - массив формы (*Ldc, n*), представляющий *mcon*×*n*-матрицу *C*, содержащую коэффициенты *mcon* ограничений общего вида.

Ldc - ведущий размер массива *c*; $Ldc \geq mcon$.

bl, bu - векторы размера *mcon*, содержащие соответственно нижние и верхние границы ограничений общего вида.

irtype - вектор размера *mcon*, указывающий на тип ограничений общего вида, задаваемых матрицей *C*. Пусть $r(i) = c(i, 1) * x(1) + \ldots + c(i, n) * x(n)$. Тогда если

$irtype(i) = 0$, то $bl(i) = r(i) = bu(i)$;

$irtype(i) = 1$, то $r(i) \leq bu(i)$;

$irtype(i) = 2$, то $r(i) \geq bl(i)$;

$irtype(i) = 3$, то $bl(i) \leq r(i) \leq bu(i)$.

xlb - вектор размера *n*, содержащий нижние границы переменных; если нижние границы для какой-либо переменной не задаются, то соответствующий элемент вектора должен быть равен 1.0E30.

xub - вектор размера *n*, содержащий верхние границы переменных; если верхние границы для какой-либо переменной не задаются, то соответствующий элемент вектора должен быть равен -1.0E30.

rnorm - евклидова норма компонентов вектор-функции $f(x)$ (далее - длина функции) после вычисления приблизительного решения.

istat - скаляр, характеризующий приблизительное решение *x*. Принимает следующие значения (смысл параметров *iparam* и *rparam* см. ниже):

- 1 - длина функции $f(x)$ меньше, чем $tolf = rparam(1)$. Это величина возвращается, когда ожидается нулевое значение $f(x)$;
- 2 - функция $f(x)$ достигла локального минимума. Это величина возвращается, когда ожидается ненулевое значение $f(x)$;
- 3 - незначительное (абсолютное) изменение вектора *x*. Выполнен полный шаг. Также может быть удовлетворено условие для $istat = 2$ и найден локальный минимум. Однако этот тест выполняется до теста с $istat = 2$;
- 4 - незначительное (относительное) изменение вектора *x*. Выполнен полный шаг. Также может быть удовлетворено условие для $istat = 2$ и найден локальный минимум. Однако этот тест выполняется до теста с $istat = 2$;
- 5 - число точек квадратичной модели ограничено объемом отведенной для вычислений памяти. Можно, но необязательно, добавить память для точек квадратичной модели. Это достигается посредством рассматриваемого ниже вектора *iparam*;
- 6 - возврат для выполнения оценки функции и якобиана (пояснения см. ниже).

Автоматически для решения предоставляется память:

- $51n + m + 15mcon + (m + mcon)(n + 1) + 4mx + na(na + 8) + 5(m + mx + 14) + 70$ байт в случае BCNLS;

- $92n + m + 26mcon + 2(m + mcon)(n + 1) + 7mx + 2na(na + 8) + 10(m + mx + 14) + 99$ байт в случае DBCNLS, где $mx = \max(m, n)$, $na = mcon + 2n + 6$.

Память можно выделить явно, применив B2NLS (DB2NLS):

```
CALL B2NLS(fcn, m, n, mcon, c, Ldc, bl, bu, irtype, xlb, xub,        &
     xguess, x,rnorm, istat, iparam, rparam, jac, f, fj,             &
     Ldfj, iwork, Liwork, work, Lwork)
```

Дополнительные параметры подпрограммы B2NLS: *iparam, rparam, jac, f, fj, Ldfj, iwork, Liwork, work, Lwork.* Все дополнительные параметры являются *входными.*

iparam - целочисленный вектор размера 6, используемый для изменения заданных по умолчанию параметров (настроек) подпрограммы BCNLS. Если необходимы заданные по умолчанию значения параметров, то задайте *iparam*(1) = 0. Для изменения настроек перед обращением к B2NLS выполните вызов

```
CALL B7NLS(iparam, rparam)
```

и установите вслед требуемые значения *iparam* и *rparam*. В случае двойной точности вместо B7NLS вызывается подпрограмма DB7NLS.

Элементы вектора *iparam* имеют следующий смысл:

- *iparam*(1) - флаг инициализации. Его назначение рассмотрено выше;
- *iparam*(2) = *itmax* - максимально допустимое число итераций; по умолчанию равно 75;
- *iparam*(3) - флаг, подавляющий использование квадратичной модели во внутреннем цикле. Если задать *iparam*(3) = 1, то квадратичная модель никогда не будет использоваться. В противном случае квадратичная модель употребляется по необходимости. Отказ от использования квадратичной модели снижает требования к памяти и вычислительные затраты; по умолчанию *iparam*(3) = 0;
- *iparam*(4) = *nterms* - число точек в квадратичной модели; по умолчанию равно 5;
- *iparam*(5) = *rcstat* - флаг, определяющий, будет ли использоваться дифференцирование вперед или дифференцирование назад. Если равен нулю, то функции *fcn* и *jac* применяются для дифференцирования вперед. Если равен 1, то употребляется дифференцирование назад и формальные процедуры B10LS (DB10LS) и B11LS (DB11LS) могут быть соответственно использованы вместо *fcn* и *jac*. Когда BCNLS завершатся с *istat* = 6, массивы *f* и *fj* содержат соответственно $f(x)$ и якобиан функции $f(x)$. Подпрограмма BCNLS вызывается вновь. Значение по умолчанию - 0;

- *iparam*(6) - флаг, определяющий, будет ли использоваться (если *iparam*(6) = 1) предоставленный как *jac* пользовательский якобиан или будет вычисляться (если *iparam*(6) = 0) его конечно-разностная аппроксимация. Значение по умолчанию - 0.

rparam - вещественный вектор размера 7, используемый для изменения заданных по умолчанию вещественных параметров (настроек) подпрограммы BCNLS.

Для описания вектора *rparam* введем следующие обозначения:

fc - текущая длина $f(x)$;

fb - наилучшая длина $f(x)$;

fl - длина $f(x)$ на предыдущем шаге;

pv - предполагаемая длина $f(x)$ после выполнения шага, использующего аппроксимирующую модель;

$\varepsilon_м$ - машинная точность, возвращаемая функцией AMACH(4).

Завершение BCNLS с *istat* = 2 произойдет, если удовлетворяются условия $|fb - pv| \le tolsnr * fb$, $|fc - pv| \le tolp * fb$ и $|fc - fl| \le tolsnr * fb$ и выполнен полный шаг модели. (Уменьшение любой из величин *tolf*, *told*, *tolx*, *tolsnr* или *tolp*, скорее всего, приведет к росту числа необходимых для сходимости итераций.)

Элементы вектора *rparam* имеют следующий смысл:

- *rparam*(1) = *tolf* - допуск, применяемый для останова в ситуации, когда $fc \le tolf$. Значение по умолчанию - $\min(10^{-5}, \sqrt{\varepsilon_м})$;
- *rparam*(2) = *tolx* - допуск, применяемый для останова, когда изменение длины вектора *x* меньше, чем *tolx* * (длина вектора *x*). Значение по умолчанию - $\min(10^{-5}, \sqrt{\varepsilon_м})$;
- *rparam*(3) = *told* - допуск, применяемый для останова, когда изменение длины вектора *x* меньше или равно *told*. Значение по умолчанию - $\min(10^{-5}, \sqrt{\varepsilon_м})$;
- *rparam*(4) = *tolsnr* - допуск, применяемый для останова в ситуации, когда *istat* = 2. Значение по умолчанию - 10^{-5};
- *rparam*(5) = *tolp* - допуск, применяемый для останова в ситуации, когда *istat* = 2. Значение по умолчанию - 10^{-5};
- *rparam*(6) = *toluse* - допуск, используемый для снижения значений *x* в квадратичной модели интерполяции предыдущих точек. Уменьшение *toluse* может привести к росту числа включаемых в квадратичную модель точек. Значение по умолчанию - $\sqrt{\varepsilon_м}$;

- *rparam*(7) = *cond* - наибольшее допустимое число обусловленности в задаче нахождения коэффициентов квадратичной модели. Увеличение *cond* может привести к росту числа включаемых в квадратичную модель точек. Значение по умолчанию - 30.

jac - пользовательская подпрограмма, оценивающая якобиан. Описание подпрограммы см. в табл. 4.3.

f - вещественный вектор размера *n*, употребляемый для пересылки в подпрограмму вектор-функции $f(x)$, если используется дифференцирование назад.

fj - вещественный массив формы (*Ldfj, n*), используемый для хранения $m \times n$-матрицы (якобиана) вектор-функции $f(x)$. Необходим, если используется дифференцирование назад. Элемент массива

$$fj(i, j) = \frac{\partial f_i}{\partial x_j}.$$

Ldfj - ведущий размер массива *fj*.

iwork - целочисленный рабочий вектор размера *Liwork*.

Liwork - размер вектора *iwork*; $Liwork \geq 5mcon + 12n + 47 + \max(m, n)$.

work - вещественный рабочий вектор размера *Lwork*.

Lwork - размер вектора *work*. $Lwork \geq 41n + 6m + 11mcon + (m + mcon) \times (n + 1) + na(na + 7) + 8\max(m, n) + 99$, где $na = mcon + 2n + 6$.

Комментарий. При работе с подпрограммой BCNLS могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
3	1	Функция $f(x)$ достигла значения, которое может быть локальным минимумом. Однако границы доверительной области, определяющей шаг, нарушаются на каждом шаге. Таким образом, решение не вызывает доверия (такая ситуация может произойти, когда в решении один или несколько компонентов вектора x бесконечны)
3	2	Длина вектора невязок в линейной или квадратичной модели решателя больше или равна текущей длине функции $f(x)$. Такая ситуация означает, что оценка $f(x)$ имеет слишком большую неопределенность, чтобы рассчитывать на допуск при определении минимума. В то же время минимум, возможно, найден
3	3	Для получения решения выполнено более чем *itmax* итераций. Хотя решение и не вызывает полного доверия, оно является наилучшим среди найденных во время вычислений значений x. Величина *itmax* может быть увеличена в результате изменения элемента *iparam*(2)

Описание:

Подпрограмма BCNLS решает нелинейную задачу наименьших квадратов

$$\min \sum_{i=1}^{m} f_i(x)^2$$

с ограничениями

$$b_l \le Cx \le b_u,$$

$$x_l \le x \le x_u.$$

Подпрограмма BCNLS основывается на процедуре DQED [18].

Пример 1. Вычисляются значения переменных x_1, x_2, x_3, x_4 модельной функции

$$h(t) = x_1 e^{x_2 t} + x_3 e^{x_4 t}.$$

Известны величины $h(t)$ при пяти значениях t: $h(0.05) = 2.206$, $h(0.1) = 1.994$, $h(0.4) = 1.35$, $h(0.5) = 1.216$, $h(1.0) = 0.7358$.

Также заданы следующие ограничения: $x_2 \le 0$, $x_4 \le 0$, $x_1 \ge 0$, $x_3 \ge 0$, и расстояние между x_2 и x_4 должно быть не менее 0.05. Поскольку о переменных больше ничего не известно, задаем в качестве стартовой точки 0.

```
program bcnlsTest1
 use dfimsl
 integer(4), parameter :: mcon = 1, n = 4, m = 5, Ldc = mcon
 integer(4) :: irtype(mcon), istat
 real(4) :: bl(mcon) = 0.0, c(mcon, n), rnorm, x(n), xguess(n), xlb(n), xub(n)
 external fcn
 ! Устанавливаем расстояние между x2 и x4
 c(1, 1) = 0.0; c(1, 2) = 1.0; c(1, 3) = 0.0; c(1, 4) = -1.0; bl(1) = 0.05
 irtype(1) = 2
 ! Нижние границы на значения переменных
 xlb(1) = 0.0; xlb(2) = 1.0e30; xlb(3) = 0.0; xlb(4) = 1.0e30
 ! Верхние границы на значения переменных
 xub(1) = -1.0e30; xub(2) = 0.0; xub(3) = -1.0e30; xub(4) = 0.0
 xguess = 0.0                              ! Начальная точка
 ! Поиск решения
 call bcnls(fcn, m, n, mcon, c, Ldc, bl, bl, irtype, xlb, xub, xguess, x, rnorm, istat)
 call wrrrn('x', 1, n, x, 1, 0); write(*, "(/, 'rnorm = ', e10.5)") rnorm
end program bcnlsTest1

subroutine fcn(m, n, x, f)
 integer(4) :: m, n, i
 real(4) :: x(*), f(*)
 real(4), save :: h(5), t(5)               ! Сохраняемые переменные
```

```
data t / 0.05, 0.1, 0.4, 0.5, 1.0 /, h / 2.206, 1.994, 1.35, 1.216, 0.7358 /
do i = 1, m; f(i) = x(1) * exp(x(2) * t(i)) + x(3) * exp(x(4) * t(i)) - h(i); end do
end subroutine fcn
```

Результат:

```
                    x
      1           2           3           4
  1.999      -1.000       0.500      -9.954
rnorm = .42424E-03
```

Пример 2. Решается та же задача, что и в предыдущем примере. Для оценки $f(x)$ и якобиана $f(x)$ используется дифференцирование назад. Применение квадратичной модели исключено.

```
program bcnlsTest2
 use dfimsl
 integer(4), parameter :: m = 5, mcon = 1, n = 4, Ldc = mcon, Ldfj = m
 integer(4) :: i, iparam(6), irtype(mcon), istat, iwork(1000), Liwork, Lwork
 real(4) :: bl(mcon) = 0.0, c(mcon, n), f(m), fj(m, n), rnorm, rparam(7),          &
      work(1000), x(n), xguess(n), xlb(n), xub(n)
 real(4), save :: h(5), t(5)                    ! Сохраняемые переменные
 data t / 0.05, 0.1, 0.4, 0.5, 1.0 /, h / 2.206, 1.994, 1.35, 1.216, 0.7358 /
 ! Устанавливаем расстояние между x2 и x4
 c(1, 1) = 0.0; c(1, 2) = 1.0; c(1, 3) = 0.0; c(1, 4) = -1.0; bl(1) = 0.05
 irtype(1) = 2
 ! Нижние границы переменных
 xlb(1) = 0.0; xlb(2) = 1.0e30; xlb(3) = 0.0; xlb(4) = 1.0e30
 ! Верхние границы переменных
 xub(1) = -1.0e30; xub(2) = 0.0; xub(3) = -1.0e30; xub(4) = 0.0
 xguess = 0.0                                   ! Стартовая точка
 ! Устанавливаем заданные по умолчанию значения параметров
 call b7nls(iparam, rparam)
 ! Подавляем использование квадратичной модели; оцениваем функцию и якобиан,
 ! применяя дифференцирование назад
 iparam(3) = 1; iparam(5) = 1; iparam(6) = 1; Lwork = 1000; Liwork = 1000
 ! Задаем формальные процедуры b10ls и b11ls для fcn и jac,
 ! что необходимо при дифференцировании назад
 do
  call b2nls(b10ls, m, n, mcon, c, Ldc, bl, bl, irtype, xlb, xub, xguess, x, rnorm,  &
       istat, iparam, rparam, b11ls, f, fj, Ldfj, iwork, Liwork, work, Lwork)
  ! Оцениваем функции, если подпрограмма завершается с istat = 6
  if(istat /= 6) exit
  do i = 1, m
    fj(i, 1) = exp(x(2) * t(i)); fj(i, 2) = t(i) * x(1) * fj(i, 1)
    fj(i, 3) = exp(x(4) * t(i)); fj(i, 4) = t(i) * x(3) * fj(i, 3)
    f(i) = x(1) * fj(i, 1) + x(3) * fj(i, 3) - h(i)
```

```
 end do
 end do
 call wrrrn('x', 1, n, x, 1, 0); write(*, "(/, 'rnorm = ', e10.5)") rnorm
end program bcnlsTest2
```

Результат:

```
                    x
      1           2           3           4
  1.999      -1.000       0.500      -9.954
rnorm = .42450E-03
```

4.5. МИНИМИЗАЦИЯ С ЛИНЕЙНЫМИ ОГРАНИЧЕНИЯМИ

4.5.1. ПЕРЕЧЕНЬ ПОДПРОГРАММ

Минимизация с линейными ограничениями выполняется подпрограммами, приведенными в табл. 4.7.

Таблица 4.7. Подпрограммы, выполняющие минимизацию с линейными ограничениями

Подпрограмма	*Описание*
DLPRS	Решает задачу линейного программирования, применяя модифицированный симплекс-алгоритм
SLPRS	Решает разреженную задачу линейного программирования, применяя модифицированный симплекс-алгоритм
QPROG	Решает задачу квадратичного программирования с линейными ограничениями общего вида равенства и неравенства
LCONF	Минимизирует целевую функцию общего вида с линейными ограничениями равенства и неравенства, используя конечно-разностный градиент
LCONG	Минимизирует целевую функцию общего вида с линейными ограничениями равенства и неравенства, используя предоставленный пользователем градиент

4.5.2. ПОДПРОГРАММА DLPRS (DDLPRS)

Решает задачу линейного программирования, применяя модифицированный симплекс-алгоритм. Имеет вызов

CALL DLPRS(*m, nvar, a, Lda, bl, bu, c, irtype, xlb, xub, obj, xsol, dsol*)

Параметры подпрограммы DLPRS:
Входные: m, nvar, a, Lda, bl, bu, c, irtype, xlb, xub.
Входной/выходной: maxfcn.

Выходные: obj, xsol, dsol.

m - число ограничений.

nvar - число переменных.

a - массив формы (*Lda, nvar*), представляющий *m*×*nvar*-матрицу *A*, содержащую коэффициенты *m* ограничений.

Lda - ведущий размер массива *a*; *Lda* ≥ *m*.

bl - вектор размера *m*, содержащий нижние границы ограничений общего вида. Если *i*-е ограничение не задано, то элемент вектора *bl*(*i*) не адресуется.

bu - вектор размера *m*, содержащий верхние границы ограничений общего вида. Если *i*-е ограничение не задано, то элемент вектора *bu*(*i*) не адресуется.

c - вектор размера *nvar*, содержащий коэффициенты целевой функции.

irtype - вектор размера *m*, указывающий на тип ограничений общего вида, задаваемых матрицей *A*. Пусть $r(i) = a(i, 1) * xsol(1) + \ldots + a(i, nvar) \times \times xsol(nvar)$. Тогда если

- *irtype*(*i*) = 0, то *bl*(*i*) = *r*(*i*) = *bu*(*i*);
- *irtype*(*i*) = 1, то *r*(*i*) ≤ *bu*(*i*);
- *irtype*(*i*) = 2, то *r*(*i*) ≥ *bl*(*i*);
- *irtype*(*i*) = 3, то *bl*(*i*) ≤ *r*(*i*) ≤ *bu*(*i*).

xlb - вектор размера *nvar*, содержащий нижние границы переменных; если нижние границы для какой-либо переменной не задаются, то соответствующий элемент вектора должен быть равен 1.0E30.

xub - вектор размера *nvar*, содержащий верхние границы переменных; если верхние границы для какой-либо переменной не задаются, то соответствующий элемент вектора должен быть равен -1.0E30.

obj - значение целевой функции.

xsol - вектор размера *nvar*, содержащий решение прямой задачи линейного программирования.

dsol - вектор размера *m*, содержащий решение двойственной задачи линейного программирования.

Комментарий. При работе с подпрограммой DLPRS могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
3	1	Задача не имеет ограничений
4	2	Превышено максимально допустимое число итераций
3	3	Задача неразрешима
4	4	Возникли трудности, связанные с машинной арифметикой. Возможно, поможет двойная точность
4	5	Ограничения несовместимы

Описание:

Подпрограмма DLPRS решает задачу линейного программирования

$$\min_{x \in \mathbf{R}^n} c^T x$$

с ограничениями

$$b_l \le Ax \le b_u,$$

$$x_l \le x \le x_u,$$

где c - вектор коэффициентов целевой функции, A - матрица коэффициентов, векторы b_l, b_u, x_l и x_u - нижняя и верхняя границы ограничений общего вида и на значения переменных. Детали см. в [30] или [31].

Пример. Решается задача линейного программирования.

```
program dlprsTest
 use dfimsl
 integer(4), parameter :: m = 2, nvar = 2, Lda = m
 ! m - число ограничений; nvar - число переменных
 integer(4) :: i, irtype(m)
 real(4) :: a(Lda, nvar), b(m), c(nvar), dsol(m), obj, xlb(nvar), xsol(nvar), xub(nvar)
 ! Решаем следующую задачу:
 ! max(1.0 * xsol(1) + 3.0 * xsol(2))
 ! xsol(1) + xsol(2) ≤ 1.5
 ! xsol(1) + xsol(2) ≥ 0.5
 ! 0 ≤ xsol(1) ≤ 1
 ! 0 ≤ xsol(2) ≤ 1
 data xlb / 2*0.0 /, xub / 2*1.0 /, a /4*1.0 /, b / 1.5, 0.5 /, c / 1.0, 3.0 /, irtype / 1, 2 /
 ! Необходимое для поиска максимума изменения знака
 c = -c
 ! Решаем задачу линейного программирования
 ! Поскольку нет ограничений диапазона, нужен только вектор b
 call dlprs(m, nvar, a, Lda, b, b, c, irtype, xlb, xub, obj, xsol, dsol)
 ! Необходимые для поиска максимума изменения знака
 obj = -obj; dsol = -dsol
 write(*, 1) obj, (xsol(i), i = 1, nvar), (dsol(i), i = 1, m)
 1 format(/, ' Objective = ', f9.4, /, ' Primal solution =', 2f9.4, /, ' Dual solution =', 2f9.4)
end program dlprsTest
```

Результат:

```
Objective = 3.5000
Primal solution = 0.5000 1.0000
Dual solution = 1.0000 0.0000
```

4.5.3. ПОДПРОГРАММА SLPRS (DSLPRS)

Решает разреженную задачу линейного программирования, применяя модифицированный симплекс-алгоритм. Имеет вызов

CALL SLPRS(*m, nvar, nz, a, irow, jcol, bl, bu, c, irtype,* &
xlb, xub, obj, xsol, dsol)

Параметры подпрограммы SLPRS, кроме приведенных ниже, см. в предшествующем разделе.

Параметры, нерассмотренные выше (все они являются *входными*):

nz - число ненулевых коэффициентов матрицы *A*.

a - вектор размера *nz*, содержащий коэффициенты *m* ограничений.

irow - вектор размера *nz*, содержащий номера строк расположения элементов вектора *a* в матрице *A*.

jcol - вектор размера *nz*, содержащий номера столбцов расположения элементов вектора *a* в матрице *A*.

Автоматически для решения предоставляется память:

- $5nvar + 62m + 2\max(nz + nvar + 8, 4nvar + 7)$ байт в случае SLPRS;
- $9nvar + 85m + 3\max(nz + nvar + 8, 4nvar + 7)$ байт в случае DSLPRS.

Память можно выделить явно, применив S2PRS (DS2PRS):

CALL S2PRS(*m, nvar, nz, a, irow, jcol, bl, bu, c, irtype, xlb, xub,* &
obj, xsol, dsol, iparam, rparam, colscl, rowscl, &
work, Lw, iwork, Liw)

Дополнительные параметры подпрограммы S2PRS: *iparam, rparam, colscl, rowscl, work, Lw, iwork, Liw*. Все дополнительные параметры являются *входными*.

iparam - целочисленный вектор параметров размера 12. Если используются заданные по умолчанию значения вектора *iparam*, то задается *iparam*(1) = 1 и вызывается S2PRS. Если необходимо изменить заданные по умолчанию установки, хранящиеся в *iparam* и *rparam*, то перед обращением к S2PRS делается вызов

CALL S5PRS(*iparam, rparam*)

затем изменяются нужные элементы *iparam* и *rparam*. Заметьте, что вызов S5PRS устанавливает в *iparam* и *rparam* заданные по умолчанию значения, поэтому после вызова S5PRS изменяются лишь те элементы *iparam* и *rparam*, значения которых должны отличаться от заданных по умолчанию.

Элементы вектора *iparam* имеют следующий смысл:

- *iparam*(1) = 0 - означает, что решается задача минимизации, если *iparam*(1) = 1, то будет решаться задача максимизации. Значение по умолчанию - 0;
- *iparam*(2) - максимальное число итераций, выполняемых подпрограммой. Если задать *iparam*(2) = 0, то может быть выполнено не более $3(nvars + m)$ итераций. Значение по умолчанию - 0;
- *iparam*(3) - способ выбора столбцов в алгоритме замены базисных переменных. Если равен нулю, то выполняется наилучший локальный шаг; если - единице, то выполняется более быстрый локальный шаг. При первой стратегии общее число итераций, как правило, меньше, но стоимость каждой итерации выше, чем при второй стратегии. Значение по умолчанию - 0;
- *iparam*(4) = *mxitbr* - число итераций между пересчетом ошибки решения прямой задачи. Не следует задавать чрезмерно малым, поскольку расчет ошибки - дорогостоящая процедура. Значение по умолчанию - 10;
- *iparam*(5) = *npp* - число просматриваемых отрицательных элементов на каждой итерации при выборе перемещаемой в базис переменной. Если равен нулю, то *npp* = *nvars*, что означает просмотр всех кандидатов на каждом шаге. Такой выбор может увеличить число итераций, но снижает объем вычислений на каждой итерации. Значение по умолчанию - 0;
- *iparam*(6) = *iredfq* - число шагов между повторными разложениями базисной матрицы. Новое разложение также выполняется, когда линейная система для прямого и двойственного решения теряет половину своей рабочей точности. Значение по умолчанию - 50;
- *iparam*(7) = *Lamat* - размер части вектора *work*, отводимой для хранения разреженной матрицы и ее разложения. *Lamat* > *nz* + *nvars* + 4; по умолчанию *Lamat* = *nz* + *nvars* + 5;
- *iparam*(8) = *Lbm* - размер части вектора *work*, отводимой для хранения ограничений. *Lbm* > 0. Значение по умолчанию *Lbm* = 8*m*;
- *iparam*(9) - номер устройства, используемого для записи данных после максимального числа итераций или оптимального решения. Данные записываются в файл, подсоединенный к устройству *iparam*(9). Данные содержат информацию о разреженной матрице и текущем базисе. Если *iparam*(9) = 0, то автоматически данные не сохраняются. Значение по умолчанию - 0;
- *iparam*(10) - номер устройства, используемого для сохранения промежуточных результатов. Если *iparam*(10) = 0, то данные не сохраняются и решается следующая задача. Значение по умолчанию - 0;
- *iparam*(11) - способ вычисления коэффициента масштабирования столбцов матрицы *A*. Если *iparam*(11) = 0, то подпрограмма S2PRS бе-

рет коэффициент масштабирования равным обратной величине ∞-нормы каждого столбца. Если *iparam*(11) = 1, то *i*-й элемент вектора *colscl* употребляется в качестве коэффициента масштабирования *i*-го столбца матрицы *A*. Масштабирование выполняется неявно, поэтому в действительности входные данные не меняются. Значение по умолчанию - 0;

- *iparam*(12) - способ вычисления коэффициента масштабирования строк матрицы *A*. Если *iparam*(12) = 0, то коэффициент масштабирования равен единице. Если *iparam*(12) = 1, то *i*-й элемент вектора *rowscl* употребляется в качестве коэффициента масштабирования *i*-й строки матрицы *A*. Масштабирование выполняется неявно, поэтому в действительности входные данные не меняются. Значение по умолчанию - 0.

rparam - вещественный вектор размера 7. Элементы вектора *rparam* имеют следующий смысл:

- *rparam*(1) = *costsc* - коэффициент масштабирования вектора цен. Если *rparam*(1) = 0.0, то подпрограмма SLPRS вычисляет *costsc* равным обратной величине ∞-нормы вектора. Значение по умолчанию - 0.0;
- *rparam*(2) = *asmall* - наименьшая величина ненулевого элемента матрицы *A*. Проверка, в процессе которой выявляется, все ли ненулевые элементы матрицы *A* больше или равны *asmall*, выполняется, если *rparam*(2) > 0, и не выполняется - в противном случае. Значение по умолчанию - 0.0;
- *rparam*(3) = *abig* - наибольшая величина ненулевого элемента матрицы *A*. Проверка, в процессе которой выявляется, все ли ненулевые элементы матрицы *A* меньше или равны *abig*, выполняется, если *rparam*(3) > 0, и не выполняется - в противном случае. Значение по умолчанию - 0.0;
- *rparam*(4) = *tolls* - относительный допуск, используемый для оценки невязок. Если *rparam*(4) = 0.0, то для *tolls* используется заданная по умолчанию величина, равная 1000.0*AMACH(4);
- *rparam*(5) = *phi* - коэффициент масштабирования, употребляемый для масштабирования оценки ошибки уменьшения целевой функции. Если *rparam*(5) = 0.0, то для *phi* используется заданная по умолчанию величина, равная 1.0;
- *rparam*(6) = *tolabs* - допуск для абсолютной ошибки. Первым проводится тест с применением *tolls* (см. *rparam*(4)). Если тест не проходит, то выполняется тест, использующий *tolabs*. Значение по умолчанию - 0.0;
- *rparam*(7) - допуск, используемый в схеме частичного выбора процедуры разложения разреженной матрицы *A*. Если *rparam*(7) = 0.0, используется заданное по умолчанию значение, равное 0.1.

colscl - вектор размера *nvars*, содержащий коэффициенты масштабирования столбцов матрицы *A*. Вектор *colscl* не используется, если *iparam*(11) = 0.

rowscl - вектор размера *m*, содержащий коэффициенты масштабирования строк матрицы *A*. Вектор *rowscl* не используется, если *iparam*(12) = 0.

work - вещественный рабочий вектор размера *Lw*.

Lw - размер вектора *work*. $Lw \geq 4nvar + 23m + \max(nz + nvar + 8, 4nvar + 7)$.

iwork - целочисленный рабочий вектор размера *Liw*.

Liw - размер вектора *iwork*. $Liw \geq nvar + 39m + \max(nz + nvar + 8, 4nvar + 7)$.

Описание:

Подпрограмма SLPRS решает ту же задачу, что и рассмотренная выше подпрограмма DLPRS, но с разреженной матрицей. Основана на процедуре DPLO [19].

Пример. Решается задача линейного программирования с матрицей

$$A = \begin{pmatrix} 0 & 0.5 & & & & \\ & 1 & 0.5 & & & \\ & & 1 & \ddots & & \\ & & & \ddots & 0.5 & \\ & & & & & 1 \end{pmatrix}.$$

```
program slprsTest
 use dfimsl
 integer(4), parameter :: m = 200, nvar = 200
 integer(4) :: index, irow(3 * m), j, jcol(3 * m), nz, irtype(m)
 real(4) :: a(3 * m), dsol(m), obj, xsol(nvar), b(m), c(nvar), xl(nvar), xu(nvar)
 data b / 199*1.7, 1.0 /
 data c / -1.0, -2.0, -3.0, -4.0, -5.0, -6.0, -7.0, -8.0, -9.0, -10.0, 190*-1.0 /
 data xl / 200*0.1 /, xu / 200*2.0 /, irtype / 200*1 /
 index = 1                              ! Задаем вектор a
 do j = 2, m
  irow(index) = j - 1                   ! Верхняя кодиагональ
  jcol(index) = j; a(index) = 0.5
  irow(index + 1) = j                   ! Главная диагональ
  jcol(index+1) = j; a(index + 1) = 1.0
  index = index + 2
 end do
 nz = index - 1; xl(4) = 0.2
 call slprs(m, nvar, nz, a, irow, jcol, b, b, c, irtype, xl, xu, obj, xsol, dsol)
 write(*, "(/, 'The value of the objective function is ', e12.6)") obj
end program slprsTest
```

Результат:
The value of the objective function is -.280971E+03

4.5.4. ПОДПРОГРАММА QPROG (DQPROG)

Решает задачу квадратичного программирования (КП) с линейными ограничениями общего вида равенства и неравенства. Имеет вызов

CALL QPROG(*nvar, ncon, neq, a, Lda, b, g, h, Ldh, diag,* &
sol, nact, iact, alamda)

Параметры подпрограммы QPROG:
Входные: nvar, ncon, neq, a, Lda, b, g, h, Ldh.
Выходные: diag, sol, nact, iact, alamda.
nvar - число переменных.
ncon - число линейных ограничений.
neq - число линейных ограничений равенства.
a - массив формы (*Lda, ncon*), представляющий *ncon*×*nvar*-матрицу *A*, содержащую в первых *neq* рядах ограничения равенства и ограничения неравенства в последующих.
Lda - ведущий размер массива *A*.
b - вектор размера *ncon*, содержащий правые части линейных ограничений.
g - вектор размера *nvar*, содержащий коэффициенты при элементах целевой функции.
h - массив формы (*Ldh, nvar*), представляющий *nvar*×*nvar*-матрицу *H*, содержащую гессиан целевой функции. Матрица *H* должна быть симметрической положительно определенной. Если *H* не является положительно определенной, то процедура QPROG пытается решить КП-задачу матрицей *H* + *diagI*, где *diag* - скаляр, а *I* - единичная матрица. Причем матрица *H* + *diagI* является положительно определенной (см. комментарий 2).
Ldh - ведущий размер массива *h*.
diag - скаляр, употребленный для модификации матрицы *H* с целью получения положительно определенной матрицы *H* + *diagI*.
sol - вектор размера *nvar*, содержащий решение.
nact - окончательное число активных ограничений.
iact - вектор размера *nvar*, содержащий в первых *nact* позициях индексы окончательных активных ограничений.
alamda - вектор размера *nvar*, содержащий в первых *nact* позициях оценки множителей Лагранжа окончательных активных ограничений.

Комментарии:
1. При работе с QPROG могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
3	1	Из-за ошибок округления изменения переменных не привели к улучшению целевой функции; обычно решение близко к оптимальному
4	2	Решения нет. Система несовместна

2. Если в результате модификации матрицы H используется матрица $H + diagI$, то та же матрица ($H + diagI$) также будет использована и для определения множителей Лагранжа.

Описание:

Подпрограмма QPROG основана на реализации Пауэлла алгоритма квадратичного программирования, приведенного в [17], для выпуклой КП-задачи с линейными ограничениями общего вида равенства и неравенства, т. е. решает задачу

$$\min_{x \in \mathbf{R}^n} (g^T x + \frac{1}{2} x^T H x)$$

с ограничениями

$$A_1 x = b_1,$$

$$A_2 x \geq b_2$$

и данными векторами b_1, b_2 и g и матрицами H, A_1 и A_2. Матрица H должна быть положительно определенной. В этом случае находится единственное решение x или выясняется, что ограничения несовместны. Если H не является положительно определенной, то вместо H используется положительно определенная модификация исходной матрицы H. Детали см. в [36] и [37].

Пример. Решается задача квадратичного программирования.

```
program qprogTest
 use dfimsl
 integer(4), parameter :: ncon = 2, neq = 2, nvar = 5, Lda = ncon, Ldh = nvar
 integer(4) :: iact(nvar), k, nact
 real(4) :: a(Lda, nvar), alamda(nvar), b(ncon), diag, g(nvar), h(Ldh, Ldh), sol(nvar)
 ! Задаем матрицы A и H, векторы b и g
 a = reshape((/      1.0,  1.0,  1.0,  1.0,  1.0,  &
                     0.0,  0.0,  1.0, -2.0,  -2.0/), shape= (/ Lda, nvar /), order = (/ 2, 1 /))
 h = reshape((/      2.0,  0.0,  0.0,  0.0,  0.0,  &
                     0.0,  2.0, -2.0,  0.0,  0.0,  &
                     0.0, -2.0,  2.0,  0.0,  0.0,  &
                     0.0,  0.0,  0.0,  2.0, -2.0,  &
                     0.0,  0.0,  0.0, -2.0,  2.0/), shape= (/ Ldh, Ldh /), order = (/ 2, 1 /))
 b = (/ 5.0, -3.0 /); g = (/ -2.0, 0.0, 0.0, 0.0, 0.0 /)
 call qprog(nvar, ncon, neq, a, Lda, b, g, h, Ldh, diag, sol, nact, iact, alamda)
```

```
 write(*, "(' The solution vector is sol = (', 5f6.1, ' )')") (sol(k), k = 1, nvar)
end program qprogTest
```

Результат:

```
The solution vector is sol = ( 1.0 1.0 1.0 1.0 1.0 )
```

4.5.5. ПОДПРОГРАММА LCONF (DLCONF)

Находит минимум целевой функции общего вида с линейными ограничениями равенства и неравенства, используя конечно-разностный градиент. Имеет вызов

```
CALL LCONF(fcn, nvar, ncon, neq, a, Lda, b, xlb, xub, xguess, acc,      &
     maxfcn, sol, obj, nact, iact, alamda)
```

Параметры подпрограммы LCONF:

Пользовательская подпрограмма: fcn.

Входные: nvar, ncon, neq, a, Lda, b, xlb, xub, xguess, acc.

Входной/выходной: maxfcn.

Выходные: sol, obj, nact, iact, alamda.

fcn - пользовательская подпрограмма, оценивающая минимизируемую функцию в точке x. Описание *fcn* см. в табл. 4.2.

nvar - число переменных.

ncon - число линейных ограничений, не включая простые ограничения.

neq - число линейных ограничений равенства.

a - массив формы (*Lda, nvar*), представляющий *ncon*×*nvar*-матрицу A, содержащую в первых *neq* рядах ограничения равенства градиентов и ограничения неравенства градиентов в последующих.

Lda - ведущий размер массива *a*.

b - вектор размера *ncon*, содержащий правые части линейных ограничений. Ограничения на переменные $x(i)$, $i = 1, \ldots, nvar$ таковы: $a(k, 1) * x(1) + \ldots + a(k, nvar) * x(nvar) = b(k)$, $k = 1, \ldots, neq$; $a(k, 1) * x(1) + \ldots + a(k, nvar) \times x(nvar) \le b(k)$, $k = neq + 1, \ldots, ncon$. То есть ограничения равенства предшествуют ограничениям неравенства.

xlb - вектор размера *nvar*, содержащий нижние границы переменных. Если переменная не имеет ограничений снизу, то в соответствующий элемент вектора *xlb* следует поместить очень большое (по абсолютной величине) отрицательное число. Если задать $xlb(i) = xub(i)$, то переменная i будет заморожена. Простые ограничения задаются неравенством $xlb(i) \le x(i)$, $i = 1, \ldots, nvar$.

xub - вектор размера *nvar*, содержащий верхние границы переменных. Если переменная не имеет ограничений сверху, то в соответствующий элемент вектора *xub* следует поместить очень большое положительное число. Простые ограничения задаются неравенством $x(i) \le xub(i)$, $i = 1, \ldots, nvar$.

xguess - вектор размера *nvar*, содержащий начальные предположения о точке минимума.

acc - неотрицательный допуск на условия первого порядка для вычисленного решения.

maxfcn - на входе - максимально допустимое число оценок функции; на выходе - реально произведенное число оценок функции.

sol - вектор размера *nvar*, содержащий решение.

obj - значение целевой функции.

nact - окончательное число активных ограничений.

iact - вектор, содержащий в первых *nact* позициях индексы активных ограничений. Его размер должен быть не меньше *ncon* + 2*nvar*.

alamda - вектор размера *nvar*, содержащий в первых *nact* позициях оценки множителей Лагранжа окончательных активных ограничений.

Автоматически для решения предоставляется память:

- $nvar^2 + 11nvar + ncon$ байт в случае LCONF;
- $2(nvar^2 + 11nvar + ncon)$ байт в случае DLCONF.

Память можно выделить явно, применив L2ONF (DL2ONF):

```
CALL L2ONF(fcn, nvar, ncon, neq, a, Lda, b, xlb, xub, xguess, acc,      &
     maxfcn, sol, obj, nact, iact, alamda, iprint, info, wk)
```

Дополнительные параметры подпрограммы L2ONF: *iprint, info, wk*. Параметр *iprint* является *входным*, *info* - *выходным*, *wk* - *рабочий вектор*.

iprint - целочисленный параметр печати, задающий интенсивность печати во время выполнения подпрограммы L2ONF. Если *iprint* = 0, то печать отсутствует. В противном случае при обнаружении подходящей точки печать выполняется через каждые IABS(*iprint*) итераций. На печать, если *iprint* > 0, выводятся значения x, f и $g = grad(f)$. Если *iprint* < 0, к этой информации добавляются текущие значения векторов *iact*(1:*nact*), *par*(1:*nact*) и *reskt*(1:*n*). Также, если *iprint* не равен нулю, выводится значение параметра *info*.

info - информационный флаг; после завершения L2ONF параметр *info* будет иметь одно из следующих целочисленных значений, информирующих о результате, полученном в L2ONF:

- *info* = 1 - вектор *sol* является приемлемым и условия, зависящие от *acc*, удовлетворены;
- *info* = 2 - вектор *sol* является приемлемым; ошибки округления не позволяют продолжить вычисления;
- *info* = 3 - вектор *sol* является приемлемым, но не удалось уменьшить целевую функцию, хотя уменьшение прогнозировалось по текущему значению вектора градиента;

- *info* = 4 - вычисления не могут быть продолжены, поскольку либо *Lda* < *ncon*, либо нижнее ограничение на переменную больше верхнего;
- *info* = 5 - ограничения равенства несовместны; такие ограничения замораживаются путем установки *xl*(*i*) = *xu*(*i*);
- *info* = 6 - возврат с ошибкой, поскольку ограничения равенства и простые ограничения несовместны;
- *info* = 7 - не существует вектора переменных, удовлетворяющего всем ограничениям. Когда L2ONF возвращает *info* = 6 или *info* = 7, текущие активные ограничения, индексы которых находятся в векторе *iact*(*k*), *k* = 1, ..., *nact*, не позволяют изменять вектор *x* так, чтобы уменьшить суммарные нарушения ограничений. Простые ограничения добавляются в эту сумму, если *info* = 6;
- *info* = 8 - превышено максимально допустимое число вызовов функции;
- *info* = 9 - значения переменных определяются ограничениями равенства.

wk - вещественный рабочий вектор размера $nvar^2 + 11nvar + ncon$.

Комментарий. При работе с LCONF могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
4	4	Ограничения равенства несовместны
4	5	Ограничения равенства и ограничения на переменные несовместны
4	6	Нет вектора *x*, удовлетворяющего всем ограничениям. В частности, текущие активные ограничения не позволяют изменить *x* так, чтобы уменьшить суммарные нарушения ограничений
4	7	Превышено максимально допустимое число оценок функции
4	9	Значения переменных определяются ограничениями равенства

Описание:

Подпрограмма LCONF основана на процедуре TOLMIN, приведенной в [39], и решает задачу

$$\min_{x \in \mathbf{R}^n} f(x)$$

с линейными ограничениями

$$A_1 x = b_1,$$

$$A_2 x \le b_2,$$

$$x_l \le x \le x_u$$

и с данными векторами b_1, b_2, x_l, x_u и матрицами A_1, A_2.

Процедура начинается с проверки на совместимость и избыточность ограничений равенства. Если ограничения равенства совместны, процедура

проверит, удовлетворяет ли начальное предположение о точке минимума $x_0 = xguess$ ограничениям $A_1x = b_1$. Затем вектор x_0 изменяется так, что он в наибольшей мере удовлетворяет простым ограничениям и ограничениям неравенства. Для этого решается последовательность подзадач квадратичного программирования по минимизации суммарных нарушений обобщенных и простых ограничений.

Пусть J_k - набор индексов ограничений неравенства, имеющих маленькие невязки для вектора x_k. (Простые ограничения рассматриваются как ограничения неравенства.) Пусть I_k - набор индексов активных ограничений. Далее решается задача квадратичного программирования

$$\min(f(x_k) + d^T\nabla f(x_k) + 1/2\, d^T B_k d$$

с ограничениями

$$a_j d = 0, j \in I_k,$$

$$a_j d \le 0, j \in J_k$$

и находятся (d_k, λ_k), где a_j - вектор-строка, представляющий либо ограничения A_1, либо A_2, либо простые ограничения на x. В последнем случае $a_j = e^{(i)}$ для простых ограничений вида $x_i \le (x_u)_i$ и $a_j = -e^{(i)}$ для простых ограничений вида $-x_i \le (-x_l)_i$. Здесь $e^{(i)}$ - i-й канонический вектор, i-й компонент которого равен единице, а остальные - нулю; d_k - направление поиска; λ_k - множители Лагранжа; B_k - положительно определенная аппроксимация второй производной $\nabla^2 f(x_k)$.

После получения направления поиска d_k выполняется линейный поиск более хорошей точки. Новая точка $x_{k+1} = x_k + \alpha_k d_k$, где α_k - размер шага, должна удовлетворять условиям

$$f(x_k + \alpha_k d_k) \le f(x_k) + 0.1\, \alpha_k d_k^T \nabla f(x_k)$$

и

$$d_k^T \nabla f(x_k + \alpha_k d_k) \ge 0.7\, d_k^T \nabla f(x_k).$$

Набор J_k не содержит индексы ограничений неравенства, влияющих на размер шага α_k. Следовательно, маленькие шаги будут достаточно редким явлением.

И наконец, аппроксимация второй производной - матрица B_k - пересчитывается по положительно определенной формуле секущих, если, впрочем, выполняется условие

$$d_k^T \nabla f(x_k + \alpha_k d_k) - \nabla f(x_k) > 0.$$

Перед началом следующей итерации имеем: $x_k \leftarrow x_{k+1}$.

Итерации повторяются, пока не выполняется критерий останова

$$\|\nabla f(x_k) - A_k\lambda_k\|_2 \le \tau,$$

где τ - заданный пользователем допуск. Детали см. в [38] и [39].

Если при работе с одинарной точностью погрешность оценки градиента приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если градиент можно оценить пользовательской процедурой, то целесообразно вместо подпрограммы LCONF употребить подпрограмму LCONG.

Пример. Решается задача, приведенная в [42]:

$$\min f(x) = -x_1x_2x_3$$

с ограничениями

$$-x_1 - 2x_2 - 2x3 \le 0,$$

$$x_1 + 2x_2 + 2x_3 \le 72,$$

$$0 \le x_1 \le 20,$$

$$0 \le x_2 \le 11,$$

$$0 \le x_3 \le 42.$$

В качестве начальной выбирается точка $x_1 = 10$, $x_2 = 10$ и $x_3 = 10$.

```
program LconfTest
 use dfimsl
 integer(4), parameter :: ncon = 2, neq = 0, nvar = 3, Lda = ncon
 integer(4) :: iact(8), maxfcn, nact
 real(4) :: a(ncon, nvar), acc, alamda(nvar), b(ncon), obj, sol(nvar),                &
       xguess(nvar), xlb(nvar), xub(nvar)
 external fcn
 ! Решается задача: min(-x(1) * x(2) * x(3)) с ограничениями
 !       -x(1)    - 2 * x(2)      - 2 * x(3) ≤ 0
 !        x(1)    + 2 * x(2)      + 2 * x(3) ≤ 72
 ! 0 ≤ x(1) ≤ 20, 0 ≤ x(2) ≤ 11, 0 ≤ x(3) ≤ 42
 data a / -1.0, 1.0, -2.0, 2.0, -2.0, 2.0 /, b / 0.0, 72.0 /
 data xlb / 3*0.0 /, xub / 20.0, 11.0, 42.0 /, xguess / 3*10.0 /
 data acc / 0.0 /, maxfcn / 400 /
 call Lconf(fcn, nvar, ncon, neq, a, Lda, b, xlb, xub,                               &
       xguess, acc, maxfcn, sol, obj, nact, iact, alamda)
 write(*, "(/, ' ', a, 5f16.6)") 'Solution: ', sol
 write(*,"(/, ' ', a, f16.6)") 'Function value at solution: ', obj
 write(*, "(/, ' ', a, i4)") 'Number of function evaluations:', maxfcn
end program LconfTest

subroutine fcn(n, x, f)                    ! Оценивает функцию в точке x
```

```
 integer(4) :: n; real(4) :: x(*), f
 f = -x(1) * x(2) * x(3)
end subroutine fcn
```

Результат:

```
Solution: 20.000000 11.000000 15.000000
Function value at solution: -3300.000000
Number of function evaluations: 5
```

4.5.6. ПОДПРОГРАММА LCONG (DLCONG)

Находит минимум целевой функции общего вида с линейными ограничениями равенства и неравенства, используя предоставленный пользователем градиент. Имеет вызов

CALL LCONG(*fcn, grad, nvar, ncon, neq, a, Lda, b, xlb, xub,* &
xguess, acc, maxfcn, sol, obj, nact, iact, alamda)

Описание параметров подпрограммы LCONG, кроме параметра *grad*, см. в предшествующем разделе.

grad - пользовательская подпрограмма, оценивающая градиент минимизируемой функции в точке *x*. Описание *grad* см. в табл. 4.2.

Комментарий и *описание* см. в предшествующем разделе.

Пример. Решается та же, что и в предшествующем разделе, задача.

```
program LcongTest
! Объявления см. в программе LconfTest, приведенной в предшествующем разделе
...
external fcn, grad
...  ! Задание данных см. в программе LconfTest
! Поиск решения
call Lcong(fcn, grad, nvar, ncon, neq, a, Lda, b, xlb, xub,                &
     xguess, acc, maxfcn, sol, obj, nact, iact, alamda)
...  ! Вывод см. в программе LconfTest
end program LcongTest

subroutine fcn(n, x, f)                    ! Оценивает функцию в точке x
 integer(4) :: n; real(4) :: x(*), f
 f = -x(1) * x(2) * x(3)
end subroutine fcn

subroutine grad(n, x, g)                   ! Оценивает градиент в точке x
 integer(4) :: n; real(4) :: x(*), g(*)
 g(1) = -x(2) * x(3); g(2) = -x(1)*x(3); g(3) = -x(1)*x(2)
end subroutine grad
```

Результат тот же, что и для программы LconfTest.

4.6. МИНИМИЗАЦИЯ С НЕЛИНЕЙНЫМИ ОГРАНИЧЕНИЯМИ

4.6.1. ПЕРЕЧЕНЬ ПОДПРОГРАММ

Минимизацию функции с нелинейными ограничениями выполняют приведенные в табл. 4.8 подпрограммы.

Таблица 4.8. Подпрограммы, выполняющие минимизацию с нелинейными ограничениями

Подпрограмма	*Описание*
NCONF	Решает общую задачу нелинейного программирования, используя последовательный алгоритм квадратичного программирования и конечно-разностный градиент
NCONG	Решает общую задачу нелинейного программирования, используя последовательный алгоритм квадратичного программирования и предоставленный пользователем градиент

4.6.2. ПОДПРОГРАММА NCONF (DNCONF)

Решает общую задачу нелинейного программирования, используя последовательный алгоритм квадратичного программирования и конечно-разностный градиент. Имеет вызов

CALL NCONF(*fcn, m, me, n, xguess, ibtype, xlb, xub,* &
xscale, iprint, maxitn, x, fvalue)

Параметры подпрограммы NCONF:
Пользовательская подпрограмма: fcn.
Входные: m, me, n, xguess, ibtype, xscale, iprint, maxitn.
Входной/выходной: xlb, xub.
Выходные: x, fvalue.

fcn - пользовательская подпрограмма, оценивающая функцию в заданной точке. Имеет вызов

CALL *fcn*(*m, me, n, x, active, f, g*)

Параметры *f* и *g* подпрограммы *fcn* являются *выходными*, остальные параметры - *входные.*

m - общее число ограничений.

me - число ограничений равенства.

n - число переменных.

x - точка (вектор размера *n*), в которой оценивается функция; параметр *x* не должен изменяться подпрограммой *fcn*.

active - логический вектор размера *mmax*, указывающий на активные ограничения; *mmax* = max(1, *m*).

f - значение функции в точке *x*.

g - вектор размера *mmax*, содержащий значения ограничений в точке *x*.

Подпрограмма *fcn* должна получить атрибут EXTERNAL в вызывающей программе.

Продолжение описания параметров подпрограммы NCONF:

m, me, n - имеют тот же смысл, что и для подпрограммы *fcn*.

xguess - вектор размера *n*, содержащий начальное предположение о точке минимума.

ibtype - скаляр, задающий вид ограничений на переменные; оказывает следующие действия:

- *ibtype* = 0 - все ограничения задаются пользователем;
- *ibtype* = 1 - все переменные неотрицательны;
- *ibtype* = 2 - все переменные неположительны;
- *ibtype* = 3 - пользователь задает ограничения на первую переменную; все другие переменные будут иметь те же ограничения.

xlb - вектор размера *n*, содержащий нижние границы переменных; *входной*, если *ibtype* = 0; *выходной*, если *ibtype* = 1 или 2; *входной/выходной*, если *ibtype* = 3. Если переменная не имеет нижней границы, то соответствующий элемент *xlb* должен быть задан равным -1.0E6.

xub - вектор размера *n*, содержащий верхние границы переменных; *входной*, если *ibtype* = 0; *выходной*, если *ibtype* = 1 или 2; *входной/выходной*, если *ibtype* = 3. Если переменная не имеет верхней границы, то соответствующий элемент *xub* должен быть задан равным 1.0E6.

xscale - вектор размера *n*, содержащий диагональную матрицу масштабирования переменных. Все элементы вектора *xscale* должны быть больше нуля. При отсутствии информации о компонентах вектора задайте все его элементы равными 1.0.

iprint - параметр, задающий желаемый уровень печати; принимает следующие значения:

- 0 - нет печати;
- 1 - выводится итоговый анализ о работе подпрограммы;
- 2 - дополнительно на каждой итерации выводится одна строка с промежуточными результатами;
- 3 - о каждой итерации выводится детальная информация.

maxitn - максимально допустимое число итераций.

x - вектор размера *n*, содержащий вычисленное решение.

fvalue - скаляр, содержащий значение целевой функции в полученном решении.

Комментарий. При работе с UVMGS могут возникать следующие информационные ошибки:

Тип	*Код*	*Описание*
4	1	Поиск осуществляется в гору
4	2	При поиске без снижения выполнено 5 вызовов функции
4	3	Превышено максимально допустимое число итераций
4	4	Направление поиска близко к нулю
4	5	Ограничения подзадачи квадратичного программирования несовместны

При желании можно решать задачу, выполняя дифференцирование назад. Для этих целей употребляется подпрограмма N0ONF (DN0ONF), имеющая вызов

CALL N0ONF(*ido, m, me, n, ibtype, xlb, xub, iprint, maxitn,* &
x, fvalue, g, df, dg, Lddg, u, c, Ldc, d, acc, scbou, &
maxfun, active, mode, wk, iwk, conwk)

Дополнительные параметры подпрограммы N0ONF:
Входные: fvalue, g, df, dg, Lddg, Ldc, acc, scbou, maxfun, mode.
Входные/выходные: ido, x, active.
Выходные: u, c, d.
Рабочие массивы: wk, iwk, conwk.

ido - параметр, задающий решаемую задачу. При первом вызове подпрограммы N0ONF параметр *ido* должен равняться нулю и в N0ONF надо передать начальные значения *x, fvalue, g, df* и *dg*. Если подпрограмма вернет *ido* = 1, то пользователь должен вычислить *fvalue* и *g* для заданного *x*. Если подпрограмма вернет *ido* = 2, то для заданного *x* вычисляются *df* и *dg*. Далее пользователь должен повторять вызов подпрограммы до тех пор, пока она не вернет значение параметра *ido*, равное 1 или 2.

x - вектор размера *n*, содержащий на входе начальное предположение о точке минимума и решение на выходе.

fvalue - скаляр, содержащий значение целевой функции, оцененной в текущей точке *x*.

g - вектор размера *mmax*, содержащий значения ограничений в текущей точке *x*; *mmax* = max(1, *m*).

df - вектор размера *n*, содержащий градиент целевой функции, оцененный в текущей точке *x*.

dg - массив формы (*Lddg, n*), представляющий *mmax*×*n*-матрицу, содержащую градиент ограничений, оцененный в текущей точке *x*.

Lddg - ведущий размер массива *dg*.

u - вектор размера $m + n + n + 2$, содержащий множители нелинейных и простых ограничений. Первые *m* позиций вектора содержат множители нелинейных ограничений; последующие *n* позиций - множители нижних границ простых ограничений; дальнейшие *n* позиций - множители верхних границ простых ограничений.

c - массив формы (*Ldc*, $n + 1$), представляющий $n + 1 \times n + 1$-матрицу, содержащую итоговую аппроксимацию гессиана.

Ldc - ведущий размер массива *c*.

d - вектор размера $n + 1$, содержащий диагональные элементы гессиана.

acc - точность для окончательного результата.

scbou - скаляр, содержащий переменную масштабирования минимизируемой функции. При отсутствии информации о потребной величине *scbou* он может быть задан равным 1.0E3.

maxfun - скаляр, содержащий максимально допустимое число вызовов функции в процессе поиска решения.

active - логический вектор размера $2mmax + 13$. Первые *mmax* его позиций используются для задания активных ограничений градиента и должны на входе быть инициализированы значением .TRUE.. Если *active*(*i*) = = .TRUE., то выполняется оценка элемента $dg(i, k)$, $k = 1, \ldots, n$. Последние $mmax + 13$ позиций вектора используются как рабочее пространство.

mode - версия алгоритма. Если *mode* = 2, то используется дифференцирование назад. Если *mode* = 3, то используется дифференцирование назад и начальные предположения о множителях и гессиане функции Лагранжа предоставляются пользователем на входе.

wk - вещественный рабочий вектор размера $2n(n + 16) + 4mmax + 5m + 68$.

iwk - целочисленный рабочий вектор размера $19 + \max(m, n)$.

conwk - вещественный рабочий вектор размера *m*.

Описание:

Подпрограмма NCONF основана на процедуре NLPQL, приведенной в [41]. Она использует метод последовательного квадратичного программирования для решения общей задачи нелинейного программирования, состоящей в поиске

$$\min_{x \in \mathbf{R}^n} f(x)$$

с ограничениями

$$g_j(x) = 0, j = 1, \ldots, m_c,$$

$$g_j(x) \geq 0, j = m_c + 1, \ldots, m_c,$$

$$x_l \leq x \leq x_u,$$

где все функции задачи являются непрерывно дифференцируемыми. Метод основан на последовательном выделении и решении подзадач квадратичного программирования, которые получаются в результате применения квадратичной аппроксимации лагранжиана и линеаризации ограничений. Таким образом, на каждой итерации решается подзадача

$$\min_{d \in \mathbf{R}^n} \frac{1}{2} d^T B_k d + \nabla f(x_k)^T d$$

с ограничениями

$$\nabla g_j(x_k)^T d + g_j(x_k) = 0, \; j = 1, \ldots, m_c,$$

$$\nabla g_j(x_k)^T d + g_j(x_k) \geq 0, \; j = m_c + 1, \ldots, m_c,$$

$$x_l - x_k \leq d \leq x_u - x_k,$$

где B_k - положительно определенная аппроксимация гессиана и x_k - текущая точка.

Пусть d_k - решение подзадачи. Тогда новая точка x_{k+1} определяется в результате линейного поиска:

$$x_{k+1} = x_k + \lambda d_k, \; \lambda \in (0, 1].$$

Новая точка такова, что в ней функция качества имеет наименьшее значение. В качестве функции качества употребляется функция Лагранжа [41]. Если оптимум не достигнут, то матрица B_k пересчитывается по положительно определенной формуле секущих [35].

Заметьте, что приведенный алгоритм может в процессе решения порождать неподходящие точки (т. е. не приводящие к снижению). Следовательно, если все промежуточные точки должны вызывать снижение, то настоящий алгоритм может оказаться неприменимым. Детали см. в [16], [31], [41] и [46].

Если при работе с одинарной точностью погрешность оценки градиента приводит к останову в неминимальной точке, то следует перейти к вычислениям с двойной точностью. Если градиент можно оценить пользовательской процедурой, то целесообразно вместо подпрограммы NCONF употребить подпрограмму NCONG.

Подпрограмма NCONF содержит подпрограмму N0ONF, которая может быть вызвана пользователем и которая позволяет применить дифференцирование назад. В этом случае функция и ее градиент оцениваются в основной программе, что полезно, если затруднительно выполнить оценку функции, применяя фиксированный интерфейс пользовательской подпрограммы *fcn*.

Пример 1. Решается задача

$$\min f(x) = (x_1 - 2)^2 + (x_2 - 1)^2$$

с ограничениями

$$g_1(x) = x_1 - 2x_2 + 1 = 0,$$

$$g_2(x) = -(x_1^2)/4 - x_2^2 + 1 \geq 0,$$

и начальной точкой (2.0, 2.0).

```
program nconfTest1
 use dfimsl
 integer(4), parameter :: ibtype = 0, iprint = 0, m = 2, maxitn = 100, me = 1, n = 2
 real(4) :: fvalue, x(n), xguess(n), xlb(n), xscale(n), xub(n)
 external fcn
 data xguess / 2.0e0, 2.0e0 /, xscale / 2*1.0e0 /,                              &
      xlb / -1.0e6, -1.0e6 /, xub / 1.0e6, 1.0e6 /
 call nconf(fcn, m, me, n, xguess, ibtype, xlb, xub, xscale, iprint, maxitn, x, fvalue)
 call wrrrn(' The solution is', 1, n, x, 1, 0)
end program nconfTest1

! Оценивает функцию и ограничения в точке x
subroutine fcn(m, me, n, x, active, f, g)
 integer(4) :: m, me, n; real(4) :: x(*), f, g(*); logical(4) :: active(*)
 ! Химмельблау задача 1
 f = (x(1)-2.0e0)**2 + (x(2)-1.0e0)**2
 if(active(1)) g(1) = x(1) - 2.0e0 * x(2) + 1.0e0
 if(active(2)) g(2) = -(x(1)**2) / 4.0e0 - x(2)**2 + 1.0e0
end subroutine fcn
```

Результат:

```
 The solution is
 0.8229          0.9114
```

Пример 2. Та же, что и в примере 1, задача решается с применением дифференцирования назад.

```
program nconfTest2
 use dfimsl
 integer(4), parameter :: m = 2, me = 1, n = 2, Ldc = n + 1, Lddg = m, &
      Lwk = 2 * n * (n + 16) + 9 * m + 68
integer(4) :: ibtype, ido, iprint, iwk(19 + m), maxfun, maxitn, mode
real(4) :: acc, c(Ldc, n + 1), conwk(m), d(n + 1), df(n), dg(Lddg, n), fvalue, g(m),
      scbou, u(m + n + n + 2), wk(Lwk), x(n), xlb(n), xub(n)
 logical(4) :: active(2 * m + 13)
 data ibtype / 3 /, maxitn / 100 /, mode / 2 /, maxfun / 10 /, iprint / 0 /
 data x / 2.0e0, 2.0e0 /, xlb(1) / -1.0e6 /, xub(1) / 1.0e6 /, scbou / 1.0e3 /
 acc = sqrt(amach(4))                    ! Итоговая точность
 active(1) = .true.; active(2) = .true.
 ido = 0
```

```
do
 if(ido == 0 .or. ido == 1) then
  ! Оцениваем функцию и ограничения в точке x
  fvalue = (x(1) - 2.0e0)**2 + (x(2) - 1.0e0)**2
  g(1) = x(1) - 2.0e0 * x(2) + 1.0e0; g(2) = -(x(1)**2) / 4.0e0 - x(2)**2 + 1.0e0
 end if
 if(ido == 0 .or. ido == 2) then
  ! Оцениваем градиент функции в точке x
  df(1) = 2.0e0 * (x(1) - 2.0e0)
  df(2) = 2.0e0 * (x(2) - 1.0e0)
  ! Оцениваем ограничения градиента в точке x
  if(active(1)) then
   dg(1, 1) = 1.0e0; dg(1, 2) = -2.0e0
  end if
  if(active(2)) then
   dg(2, 1) = -0.5e0 * x(1); dg(2, 2) = -2.0e0 * x(2)
  end if
 end if
 ! Вызываем N0ONF для следующей модификации
 call n0onf(ido, m, me, n, ibtype, xlb, xub, iprint, maxitn, x, fvalue, g, df, dg,&
      Lddg, u, c, Ldc, d, acc, scbou, maxfun, active, mode, wk, iwk, conwk)
 if(ido /= 1 .and. ido /= 2) exit          ! Выход, если ido ≠ 1 и ido ≠ 2
end do                                     ! Вывод результата
call wrrrn(' The solution is', 1, n, x, 1, 0)
end program nconfTest2
```

Результат тот же, что и в примере 1.

4.6.3. ПОДПРОГРАММА NCONG (DNCONG)

Решает общую задачу нелинейного программирования, используя последовательный алгоритм квадратичного программирования и предоставленный пользователем градиент. Имеет вызов

CALL NCONG(*fcn, grad, m, me, n, xguess, ibtype, xlb, xub,* &
iprint, maxitn, x, fvalue)

Описание параметров подпрограммы NCONG, кроме параметра *grad*, см. в предшествующем разделе.

grad - пользовательская подпрограмма, оценивающая градиент минимизируемой функции в текущей точке. Имеет вызов

CALL *grad*(*m, me, mmax, n, x, active, f, g, df, dg*)

Параметры *df* и *dg* подпрограммы *grad* являются *выходными*, остальные параметры - *входные*.

m - общее число ограничений.

me - число ограничений равенства.

mmax = max(1, *m*).

n - число переменных.

x - точка (вектор размера *n*), в которой оценивается градиент; параметр *x* не должен изменяться подпрограммой *grad*.

active - логический вектор размера *mmax*, указывающий на активные ограничения.

f - значение функции в точке *x*.

g - вектор размера *mmax*, содержащий значения ограничений в точке *x*.

df - вектор размера *n*, содержащий вычисленные значения градиента целевой функции.

dg - массив формы (*mmax, n*), содержащий значения градиентов активных ограничений.

Подпрограмма *grad* должна получить атрибут EXTERNAL в вызывающей программе.

Комментарий и *описание* см. в предшествующем разделе.

Замечание. Подпрограмму NCONG нельзя заменить вызовом подпрограммы N0ONF.

Пример. Решается та же, что и в примере 1 из предшествующего раздела, задача.

```
program ncongTest
 ! Объявления см. в программе nconfTest1, приведенной в предшествующем разделе
 . . .
 external fcn, grad
 ! Задание данных см. в программе nconfTest1
 . . .
 ! Поиск решения
 call ncong(fcn, grad, m, me, n, xguess, ibtype, xlb, xub, iprint, maxitn, x, fvalue)
. . .   ! Вывод см. в программе nconfTest1
end program ncongTest

! Оценивает функцию и ограничения в точке x
subroutine fcn(m, me, n, x, active, f, g)
 ! Текст подпрограммы см. в предшествующем разделе
 . . .
end subroutine fcn

! Оценивает градиент функции и ограничений в точке x
subroutine grad(m, me, mmax, n, x, active, f, g, df, dg)
 integer(4) :: m, me, mmax, n; real(4) :: x(*), f, g(*),                    &
       df(*), dg(mmax,*); logical(4) :: active(*)
 df(1) = 2.0e0 * (x(1) - 2.0e0); df(2) = 2.0e0 * (x(2)-1.0e0)
```

```
if(active(1)) then
 dg(1, 1) = 1.0e0; dg(1, 2) = -2.0e0
end if
if(active(2)) then
 dg(2, 1) = -0.5e0 * x(1); dg(2, 2) = -2.0e0 * x(2)
end if
end subroutine grad
```

Результат тот же, что и для программы *nconfTest*1.

4.7. ВСПОМОГАТЕЛЬНЫЕ ПОДПРОГРАММЫ

4.7.1. ПЕРЕЧЕНЬ И ПАРАМЕТРЫ ПОДПРОГРАММ

Вспомогательные подпрограммы, применяемые в задаче минимизации, приведены в табл. 4.9 их параметры - в табл. 4.10.

Таблица 4.9. Вспомогательные подпрограммы

Подпрограмма	*Описание*
CDGRD	Аппроксимирует градиент, используя центральные разности
FDGRD	Аппроксимирует градиент, используя правые разности
FDHES	Аппроксимирует гессиан, используя правые разности и значения функции
GDHES	Аппроксимирует гессиан, используя правые разности и предоставленный пользователем градиент
FDJAC	Аппроксимирует якобиан *m* функций от *n* переменных, используя правые разности
CHGRD	Проверяет предоставленный пользователем градиент функции
CHHES	Проверяет предоставленный пользователем гессиан аналитической функции
CHJAC	Проверяет предоставленный пользователем якобиан системы уравнений *m* функций от *n* переменных
GGUES	Генерирует точку *n*-мерного пространства

Таблица 4.10. Параметры вспомогательных подпрограмм

Имя	*Смысл / вид*	*Тип*
epsfcn	Оценка относительного шума функции; *epsfcn* ≤ 0.1. При отсутствии информации об относительном шуме положите *epsfcn* = 0.0 / *входной*	REAL(4) или REAL(8)
fc	Скаляр, содержащий значение функции в заданной точке *xc* / *входной* (в случае подпрограмм FDGRD и FDHES)	То же

fc	Вектор размера *m*, содержащий значения функций в точке *xc* / *входной* (в случае подпрограммы FDJAC)	REAL(4) или REAL(8
fcn	Пользовательская подпрограмма, оценивающая минимизируемую функцию. Имеет вызов CALL *fcn*(*n, x, f*) Параметры *n, x* подпрограммы *fcn* являются *входными*, параметр *f* - *выходным*. Параметры имеют следующий смысл: *n* - размер вектора *x*, содержащего координаты текущей точки; *x* - точка (вектор размера *n*), в которой выполняется оценка функции; параметр *x* не должен изменяться подпрограммой *fcn*; *f* - оценка функции в точке *x*. Подпрограмма *fcn* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	-
fjac	Массив формы (*Ldfjac, n*), представляющий *m*×*n*-матрицу, содержащую оцениваемый якобиан / *выходной*	REAL(4) или REAL(8)
grad	Пользовательская подпрограмма, оценивающая градиент в точке *x*. Имеет вызов CALL *grad*(*n, x, g*) Параметры *n, x* подпрограммы *grad* являются *входными*, параметр *g* - *выходным*. Параметры имеют следующий смысл: *n* - размер вектора *x*, содержащего координаты текущей точки; *x* - точка, в которой выполняется оценка градиента; параметр не должен изменяться подпрограммой *grad*; *g* - оценка градиента в точке *x*. Подпрограмма *grad* должна получить в вызывающей программе атрибут EXTERNAL / *пользовательская подпрограмма*	"
h	Массив формы (*Ldh, n*), представляющий *n*×*n*-матрицу *H*, содержащую в нижней треугольной части конечно-разностную аппроксимацию гессиана / *выходной*	"
Ldh	Ведущий размер массива *h* / *входной*	INTEGER(4)
Ldfjac	Ведущий размер массива *fjac* / *входной*	То же
m	Число функций / *входной*	"
n	Число переменных / *входной*	"

xc	Вектор размера *n*, содержащий точку, в которой оценивается градиент (в случае CDGRD, FDGRD и FDJAC) или гессиан (в случае FDHES и GDHES) / *входной*	REAL(4) или REAL(8)
xscale	Вектор размера *n*, содержащий диагональную матрицу масштабирования переменных. При отсутствии информации о компонентах вектора задайте все его элементы равными 1.0 / *входной*	То же

4.7.2. ПОДПРОГРАММА CDGRD (DCDGRD)

Аппроксимирует градиент, используя центральные разности. Имеет вызов

CALL CDGRD(*fcn, n, xc, xscale, epsfcn, gc*)

Смысл параметров подпрограммы CDGRD см. в табл. 4.10.

Описание:

Подпрограмма CDGRD использует для оценки градиента функции *n* переменных в точке *x* следующую конечно-разностную формулу:

$$\frac{f(x+h_i e_i)-f(x-h_i e_i)}{2h_i}, \; i = 1, \ldots, n,$$

где $h_i = \sqrt{\varepsilon_м}\, \max(|x_i|, 1/s_i)\mathrm{sign}(x_i)$, $\varepsilon_м$ - машинная точность, s_i - коэффициент масштабирования *i*-й переменной, $e^{(i)}$ - *i*-й канонический вектор. Детали см. в [6].

Замечание. Конечно-разностному методу присущи ошибки усечения, потеря значащих разрядов (при вычитании) и ошибки округления, приводящие иногда к неточным результатам; по возможности следует использовать вычисления с двойной точностью.

Пример. Оценивается градиент функции $f(x) = x_1 - x_1x_2 - 2$ в точке (1.0, 1.0).

```
program cdgrdTest
 use dfimsl
 integer(4), parameter :: n = 2
 real(4) :: epsfcn, gc(n), xc(n), xscale(n)
 external fcn
 data xscale / 2*1.0e0 /, xc / 2*1.0e0 /
 epsfcn = 0.01                          ! Шум функции
 call cdgrd(fcn, n, xc, xscale, epsfcn, gc)
 write(*, "(' The gradient is ', 2f8.2, /)") gc(1:n)
end program cdgrdTest

subroutine fcn(n, x, f)                 ! Выполняет оценку функции в точке x
 integer(4) :: n; real(4) :: x(n), f
 f = x(1) - x(1) * x(2) - 2.0e0
end subroutine fcn
```

Результат:
The gradient is 0.00 -1.00

4.7.3. ПОДПРОГРАММА FDGRD (DFDGRD)

Аппроксимирует градиент, используя правые разности. Имеет вызов

CALL FDGRD(*fcn, n, xc, xscale, fc, epsfcn, gc*)

Смысл параметров подпрограммы FDGRD раскрыт в табл. 4.10.

Описание:

Подпрограмма FDGRD использует для оценки градиента функции n переменных в точке x следующую конечно-разностную формулу:

$$\frac{f(x+h_i e_i) - f(x)}{h_i}, \; i = 1, \ldots, n,$$

где $h_i = \sqrt{\varepsilon_м}\, \max(|x_i|, 1/s_i)\mathrm{sign}(x_i)$, $\varepsilon_м$ - машинная точность, si - коэффициент масштабирования i-й переменной, $e^{(i)}$ - i-й канонический вектор. Детали см. в [6].

Когда важна точность градиента, рекомендуется употреблять приведенную выше подпрограмму CDGRD.

Пример. Решается та же, что и в примере для CDGRD, задача.

```
program fdgrdTest
 use dfimsl
 integer(4), parameter :: n = 2
 real(4) :: epsfcn, cf, gc(n), xc(n), xscale(n)
 external fcn
 data xscale / 2*1.0e0 /, xc / 2*1.0e0 /
 epsfcn = 0.01                          ! Шум функции
 call fcn(n, xc, fc)                    ! Получаем значение функции в точке xc
 call fdgrd(fcn, n, xc, xscale, fc, epsfcn, gc)
 write(*, "(' The gradient is ', 2f8.2, /)") gc(1:n)
end program fdgrdTest

subroutine fcn(n, x, f)                 ! Выполняет оценку функции в точке x
 integer(4) :: n; real(4) :: x(n), f
 f = x(1) - x(1) * x(2) - 2.0e0
end subroutine fcn
```

Результат тот же, что и для подпрограммы CDGRD.

4.7.4. ПОДПРОГРАММА FDHES (DFDHES)

Аппроксимирует гессиан, используя правые разности и значения функции. Имеет вызов

CALL FDHES(*fcn, n, xc, xscale, fc, epsfcn, h, Ldh*)

Смысл параметров см. в табл. 4.10.

Описание:

Подпрограмма FDHES использует для оценки гессиана функции $f(x)$ следующую конечно-разностную формулу:

$$\frac{f(x+h_i e_i + h_j e_j) - f(x+h_i e_i) - f(x+h_j e_j) + f(x_i)}{h_i h_j},$$

где $h_i = \varepsilon^{1/3}\max(|x_i|, 1/s_i)\text{sign}(x_i)$, $h_j = \varepsilon^{1/3}\max(|x_j|, 1/s_j)\text{sign}(x_j)$, ε - машинная точность или пользовательская оценка относительного шума, s_i и s_j - коэффициент масштабирования соответственно i-й и j-й переменных, e_i и e_j - соответственно i-й и j-й канонические векторы. Детали см. в [6].

Пример. Оценивается гессиан функции $f(x) = x_1^2 - x_1x_2 - 2$ в точке (1, -1).

```
program fdhesTest
 use dfimsl
 integer(4), parameter :: n = 2, Ldhes = 2
 integer(4) :: i, j
 real(4) :: xc(n), xscale(n), fvalue, hes(Ldhes, n), epsfcn
 external fcn
 data xscale/2*1.0e0/, xc/1.0e0,-1.0e0/      ! Инициализация
 epsfcn = 0.001                              ! Шум функции
 call fcn(n, xc, fvalue)                     ! Оцениваем функцию в текущей точке
 ! Получаем оценку гессиана
 call fdhes(fcn, n, xc, xscale, fvalue, epsfcn, hes, Ldhes)
 write(*, 1) ((hes(i, j), j =1, i), i =1, n)
 1 format(' The lower triangle of the hessian is', /, 5x, f10.2, /, 5x, 2f10.2, /)
end program fdhesTest

subroutine fcn(n, x, f)                      ! Выполняет оценку функции в точке x
 integer(4) :: n; real(4) :: x(n), f
 f = x(1) * (x(1) - x(2)) - 2.0e0
end subroutine fcn
```

Результат:

```
The lower triangle of the Hessian is
    2.00
   -1.00        0.00
```

4.7.5. ПОДПРОГРАММА GDHES (DGDHES)

Аппроксимирует гессиан, используя правые разности и предоставленный пользователем градиент. Имеет вызов

CALL GDHES(*grad, n, xc, xscale, gc, epsfcn, h, Ldh*)

Смысл параметров подпрограммы GDHES см. в табл. 4.10.

Описание:

Подпрограмма GDHES использует для оценки гессиана функции $f(x)$ следующую конечно-разностную формулу:

$$\frac{g(x+h_j e_j) - g(x)}{h_j},$$

где $h_j = \sqrt{\varepsilon_м}\ \max(|x_j|, 1/s_j)\text{sign}(x_j)$, $\varepsilon_м$ - машинная точность, s_j - коэффициент масштабирования j-й переменной, g - аналитический градиент функции $f(x)$, $e^{(j)}$ - j-й канонический вектор. Детали см. в [6].

Пример. Оценивается гессиан в точке (1.0, 1.0) по следующим функциям градиента:

$$g_1 = 2x_1x_2 - 2,$$

$$g_2 = x_1x_1 + 1.$$

```
program gdhesTest
 use dfimsl
 integer(4), parameter :: n = 2, Ldhes = 2
 integer(4) :: i, j
 real(4) :: xc(n), xscale(n), gc(n), hes(Ldhes, n), epsfcn
 external grad
 data xscale/ 2*1.0e0 /, xc/ 2*1.0e0 /
 epsfcn = 0.0                          ! Шум функции
 call grad(n, xc, gc)                  ! Оцениваем градиент в точке xc
 ! Получаем конечно-разностную аппроксимацию гессиана
 call gdhes(grad, n, xc, xscale, gc, epsfcn, hes, Ldhes)
 write(*, "(' The hessian is', /, 2(5x, 2f10.2, /), /)") ((hes(i, j), j = 1, n), i = 1, n)
end program gdhesTest

subroutine grad(n, x, g)
 integer(4) :: n; real(4) :: x(n), g(n)
 g(1) = 2.0e0 * x(1) * x(2) - 2.0e0; g(2) = x(1) * x(1) + 1.0e0
end subroutine grad
```

Результат:

```
The hessian is
    2.00      2.00
    2.00      0.00
```

4.7.6. ПОДПРОГРАММА FDJAC (DFDJAC)

Аппроксимирует якобиан *m* функций от *n* переменных, используя правые разности. Имеет вызов

CALL FDJAC(*fcn, m, n, xc, xscale, fc, epsfcn, fjac, Ldfjac*)

Смысл параметров подпрограммы FDJAC, кроме *пользовательской подпрограммы fcn*, см. в табл. 4.10.

fcn - пользовательская подпрограмма, оценивающая минимизируемую функцию. Имеет вызов

CALL *fcn*(*m, n, x, f*)

Параметр *f* подпрограммы *fcn* является *выходным*, остальные параметры - *входные*.

m - размер вектора *f*.

n - размер вектора *x*.

x - точка, в которой оценивается функция; параметр *x* не должен изменяться подпрограммой *fcn*.

f - оценка функции в точке *x*.

Подпрограмма *fcn* должна получить в вызывающей программе атрибут EXTERNAL.

Описание:

Подпрограмма FDJAC использует для оценки якобиана функции $f(x)$ следующую конечно-разностную формулу:

$$\frac{f(x+h_j e_j)-f(x)}{h_j},$$

где $h_j = \sqrt{\varepsilon_м}\ \max(|x_j|, 1/s_j)\text{sign}(x_j)$, $\varepsilon_м$ - машинная точность, s_j - коэффициент масштабирования j-й переменной, $e^{(j)}$ - j-й канонический вектор. Детали см. в [6].

Пример. В точке (1.0, 1.0) оценивается якобиан функций

$$f_1(x) = x_1x_2 - 2,$$

$$f_2(x) = x_1 - x_1x_2 + 1.$$

```
program fdjacTest
 use dfimsl
 integer(4), parameter :: n = 2, m = 2, Ldfjac = 2
 real(4) :: fjac(Ldfjac, n), xscale(n), xc(n), fc(m), epsfcn
 external fcn
 data xscale / 2*1.0e0 /, xc / 2*1.0e0 /
 epsfcn = 0.01                          ! Шум функции
```

```
 call fcn(m, n, xc, fc)                    ! Оцениваем функции в точке xc
 ! Получаем якобиан
 call fdjac(fcn, m, n, xc, xscale, fc, epsfcn, fjac, Ldfjac)
 write(*, "(' The jacobian is', /, 2(5x, 2f10.2, /), /)") ((fjac(i, j), j = 1, n), i = 1, m)
end program fdjacTest

subroutine fcn(m, n, x, f)
 integer(4) :: m, n; real(4) :: x(n), f(m)
 f(1) = x(1) * x(2) - 2.0e0; f(2) = x(1) - x(1) * x(2) + 1.0e0
end subroutine fcn
```

Результат:

```
The Jacobian is
    1.00      1.00
    0.00     -1.00
```

4.7.7. ПОДПРОГРАММА CHGRD (DCHGRD)

Проверяет предоставленный пользователем градиент функции. Имеет вызов

CALL CHGRD(*fcn, grad, n, x, info*)

Параметр подпрограммы CHGRD:
Пользовательская подпрограмма: fcn.
Входные: grad, n.
Выходной: info.

fcn - пользовательская подпрограмма, оценивающая функцию, градиент которой проверяется. Имеет вызов

CALL *fcn*(*n, x, f*)

Описание параметров подпрограммы *fcn* см. в табл. 4.10.

grad - вектор размера *n*, содержащий оценки градиента в точке *x*.

n - число переменных (размерность задачи).

x - вектор размера *n*, содержащий точку, в которой проверяется градиент.

info - целочисленный вектор размера *n*; принимает следующие значения:

- *info*(*i*) = 0 - значение предоставленного пользователем градиента расходится в точке *x*(*i*) с численной оценкой подпрограммы CHGRD;
- *info*(*i*) = 1 - значение предоставленного пользователем градиента хорошо согласуется в точке *x*(*i*) с численной оценкой подпрограммы CHGRD;
- *info*(*i*) = 2 - значение предоставленного пользователем градиента не согласуется в точке *x*(*i*) с оценкой подпрограммы CHGRD, но, возможно, подпрограмма CHGRD не может выполнить численную оценку градиента;
- *info*(*i*) = 3 - значение предоставленного пользователем градиента и численная оценка подпрограммы CHGRD равны нулю в точке *x*(*i*); градиент нужно проверить в другой точке.

Автоматически для решения предоставляется память:

- $2n + 2$ байт в случае CHGRD;
- $4n + 4$ байт в случае DCHGRD.

Память можно выделить явно, применив C2GRD (DC2GRD):

CALL C2GRD(*fcn, grad, n, x, info, fx, xscale, epsfcn, xnew*)

Дополнительные параметры подпрограммы C2GRD: *fx, xscale, epsfcn, xnew*. Все дополнительные параметры являются *входными*.

fx - значение функции в точке *x*.

xscale - вещественный вектор размера *n*, содержащий диагональную матрицу масштабирования переменных. Все элементы вектора *xscale* должны быть больше нуля.

epsfcn - относительный шум функции *fcn*.

xnew - вещественный рабочий вектор размера *n*.

Комментарий. При работе с CHGRD может возникнуть информационная ошибка типа 4 с кодом 2, означающая, что значение предоставленного пользователем градиента расходится с численной оценкой подпрограммы CHGRD.

Описание:

Подпрограмма CHGRD использует для оценки градиента функции $f(x)$ от n переменных в точке x следующую конечно-разностную формулу:

$$g_i(x) = \frac{f(x + h_i e_i) - f(x)}{h_i}, \; i = 1, \ldots, n,$$

где $h_i = \sqrt{\varepsilon_м} \max(|x_i|, 1/s_i)\text{sign}(x_i)$, $\varepsilon_м$ - машинная точность, s_i - коэффициент масштабирования i-й переменной, $e^{(i)}$ - i-й канонический вектор.

Подпрограмма CHGRD проверяет предоставленный пользователем градиент $\nabla f(x)$, сравнивая его с конечно-разностной аппроксимацией градиента $g(x)$. Если

$$|g_i(x) - (\nabla f(x))_i| < \tau|(\nabla f(x))_i|,$$

где $\tau = \varepsilon_м^{1/4}$, то $(\nabla f(x))_i$ - i-й элемент вектора $\nabla f(x)$ - является верным, в противном случае CHGRD определяет ошибки вычисления и аппроксимации. Когда обе ошибки достаточно малы, элемент $(\nabla f(x))_i$ классифицируется как неверный. В случае больших ошибок, подпрограмма CHGRD, используя почти оптимальный размер шага, заново вычисляет $g_i(x)$ и сообщает, что значение элемента $(\nabla f(x))_i$ верно, если

$$|g_i(x) - (\nabla f(x))_i| < 2\tau|(\nabla f(x))_i|.$$

В противном случае элемент $(\nabla f(x))_i$ классифицируется как неверный, если только ошибка оптимального шага не больше чем $\tau|(\nabla f(x))_i|$. Если же она больше, то, скорее всего, численное значение градиента нельзя оценить правильно. Детали см. в [29].

Пример. Проверяется предоставленное пользователем значение градиента функции

$$f(x) = x_1 + x_2 \exp(-(t - x_3)^{2/x_4})$$

в точке (625, 1, 3.125, 0.25) при $t = 2.125$.

```
program chgrdTest
 use dfimsl
 integer(4), parameter :: n = 4
 integer(4) :: info(n)
 real(4) :: grad(n), x(n)
 external fcn
 data x / 625.0e0, 1.0e0, 3.125e0, 0.25e0 /
 call driv(n, x, grad)                    ! Формируем пользовательский градиент
 call chgrd(fcn, grad, n, x, info)        ! Проверяем пользовательский градиент
 call wrirn('The information vector', 1, n, info, 1, 0)
end program chgrdTest

subroutine fcn(n, x, fx)                  ! Выполняет оценку функции в точке x
 integer(4) :: n; real(4) :: x(n), fx
 fx = x(1) + x(2)*exp(-1.0e0*(2.125e0-x(3))**2/x(4))
end subroutine fcn

subroutine driv(n, x, grad)               ! Вычисляет предоставленный пользователем
 integer(4) :: n; real(4) :: x(n), grad(n) ! градиент
 real(4) :: t = 2.125e0
 grad(1) = 1.0e0; grad(2) = exp(-1.0e0 * (t - x(3))**2 / x(4))
 grad(3) = x(2) * exp(-1.0e0 * (t - x(3))**2 / x(4)) * 2.0e0 / x(4) * (t - x(3))
 grad(4) = x(2) * exp(-1.0e0 * (t - x(3))**2 / x(4)) * (t - x(3))**2 / (x(4) * x(4))
end subroutine driv
```

Результат:

```
The information vector
1 2 3 4
1 1 1 1
```

4.7.8. ПОДПРОГРАММА CHHES (DCHHES)

Проверяет предоставленный пользователем гессиан аналитической функции. Имеет вызов

CALL CHHES(*grad, hess, n, x, info, Ldinfo*)

Параметр подпрограммы CHHES:
Пользовательские подпрограммы: grad, hess.
Входные: n, x, Ldinfo.
Выходной: info.

grad - пользовательская подпрограмма, оценивающая градиент в точке *x*. Имеет вызов

CALL *grad*(*n, x, g*)

Описание параметров подпрограммы *grad* см. в табл. 4.10.

hess - пользовательская подпрограмма, вычисляющая гессиан в точке *x*. Имеет вызов

CALL *hess*(*n, x, h, Ldh*)

Параметры *n, x, Ldh* подпрограммы *hess* являются *входными*, параметр *h* - *выходным*.

n - размер вектора *x*, содержащего координаты текущей точки.

x - точка, в которой выполняется оценка гессиана; параметр не должен изменяться подпрограммой *hess*.

h - массив формы (*Ldh, n*), содержащий оценку гессиана в точке *x*.

Ldh - ведущий размер массива *h*, представляющего гессиан *H*. В этой подпрограмме равен *n*.

Подпрограмма *hess* должна получить в вызывающей программе атрибут EXTERNAL

Продолжение описания параметров подпрограммы CHHES:

n, x - имеют тот же смысл, что и для подпрограммы *hess*.

info - целочисленный массив формы (*Ldinfo, n*), информирующий о результатах проверки *n*×*n*-гессиана; имеет следующий смысл:

- *info*(*i, j*) = 0 - значение предоставленного пользователем гессиана функции *i* в точке *x*(*j*) расходится с численной оценкой подпрограммы CHHES;
- *info*(*i, j*) = 1 - значение предоставленного пользователем гессиана функции *i* в точке *x*(*j*) хорошо согласуется с численной оценкой подпрограммы CHHES;
- *info*(*i, j*) = 2 - значение предоставленного пользователем гессиана функции *i* в точке *x*(*j*) не согласуется с численной оценкой подпрограммы CHHES, но, возможно, подпрограмма CHHES не может выполнить численную оценку гессиана;
- *info*(*i, j*) = 3 - значение предоставленного пользователем гессиана функции *i* в точке *x*(*j*) и соответствующая численная оценка подпрограммы CHHES равны нулю; гессиан нужно проверить в другой точке.

Ldinfo - ведущий размер массива *info*.

Описание:

Подпрограмма CHHES использует для оценки гессиана функции $f(x)$ от n переменных в точке x следующую конечно-разностную формулу:

$$B_{ij}(x) = \frac{g_i(x + h_j e_j) - g_i(x)}{h_j}, \quad j = 1, \ldots, n,$$

где $h_j = \sqrt{\varepsilon_м}\ \max(|x_j|, 1/s_j)\text{sign}(x_j)$, $\varepsilon_м$ - машинная точность, s_j - коэффициент масштабирования j-й переменной, g_i - градиент функции $f(x)$ по i-й переменной, $e^{(j)}$ - j-й канонический вектор.

Затем CHHES проверяет предоставленный пользователем гессиан $H(x)$, сравнивая его с конечно-разностной аппроксимацией $B(x)$. Если

$$|B_{ij}(x) - H_{ij}(x)| < \tau|H_{ij}(x)|,$$

где $\tau = \varepsilon_м^{1/4}$, то элемент $H_{ij}(x)$ классифицируется как верный, в противном случае CHHES определяет ошибки вычисления и аппроксимации. Когда обе ошибки достаточно малы, элемент $H_{ij}(x)$ классифицируется как неверный. В случае больших ошибок, подпрограмма CHHES, используя почти оптимальный размер шага, заново вычисляет $B_{ij}(x)$ и сообщает, что значение элемента $H_{ij}(x)$ верно, если

$$|B_{ij}(x) - H_{ij}(x)| < 2\tau|H_{ij}(x)|.$$

В противном случае элемент $H_{ij}(x)$ классифицируется как неверный, если только ошибка оптимального шага не больше чем $\tau|H_{ij}(x)|$. Если же она больше, то, скорее всего, численную аппроксимацию гессиана нельзя выполнить правильно. Детали см. в [29].

Пример. В точке (-1.2, 1.0) проверяется предоставленный пользователем гессиан функции

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

В пользовательскую подпрограмму *hes*, оценивающую гессиан, вносится ошибка.

```
program chhesTest
 use dfimsl
 integer(4), parameter :: n = 2, Ldinfo = n
 integer(4) :: info(Ldinfo, n)
 real(4) :: x(n)
 external grd, hes
 data x / -1.2, 1.0 /
 call chhes(grd, hes, n, x, info, Ldinfo) ! Выполняем проверку гессиана
end program chhesTest
```

```
subroutine grd(n, x, ug)                    ! Возвращает оценку градиента
 integer(4) :: n; real(4) :: x(n), ug(n)
 ug(1) = -400.0 * x(1) * (x(2) - x(1) * x(1)) + 2.0 * x(1) - 2.0
 ug(2) = 200.0 * x(2) - 200.0 * x(1) * x(1)
end subroutine grd

subroutine hes(n, x, hx, Ldhs)              ! Возвращает оценку гессиана
 integer(4) :: n, Ldhs; real(4) :: x(n), hx(Ldhs, n)
 hx(1, 1) = -400.0 * x(2) + 1200.0 * x(1) * x(1) + 2.0
 hx(1, 2) = -400.0 * x(1); hx(2, 1) = -400.0 * x(1)
 hx(2, 2) = -200.0                          ! Вносим ошибку, изменяя знак для hx(2, 2)
end subroutine hes
```

Результат:

```
*** FATAL ERROR 1 from CHHES. The Hessian evaluation with respect to
*** X(2) and X(2) is a poor estimate.
```

4.7.9. ПОДПРОГРАММА CHJAC (DCHJAC)

Проверяет предоставленный пользователем якобиан системы уравнений из *m* функций от *n* неизвестных. Имеет вызов

CALL CHJAC(*fcn, jac, m, n, x, info, Ldinfo*)

Параметр подпрограммы CHJAC:
Пользовательские подпрограммы: fcn, jac.
Входные: m, n, x, Ldinfo.
Выходной: info.

fcn - пользовательская подпрограмма, оценивающая функцию, для которой вычисляется якобиан. Имеет вызов

CALL *fcn*(*m, n, x, f*)

Параметр *f* подпрограммы *fcn* является *выходным*, остальные параметры - *входные.*

m - размер вектора *f.*

n - размер вектора *x.*

x - точка (вектор размера *n*), в которой оценивается функция; параметр *x* не должен изменяться подпрограммой *fcn.*

f - оценка функции в точке *x.*

jac - пользовательская подпрограмма, оценивающая якобиан в точке *x.* Имеет вызов

CALL *jac*(*m, n, x, fjac, Ldfjac*)

Параметры *m, n, x, Ldfjac* подпрограммы *jac* являются *входными*; параметр *fjac* - *выходным.*

m, n - имеют тот же смысл, что и одноименные параметры подпрограммы *fcn*.

x - точка (вектор размера *n*), в которой выполняется оценка якобиана; параметр *x* не должен изменяться подпрограммой *jac*.

fjac - массив формы (*Ldfjac, n*), содержащий вычисленный в точке *x* *m*×*n*-якобиан.

Ldfjac - ведущий размер массива *fjac*.

Подпрограмма *jac* должна получить в вызывающей программе атрибут EXTERNAL.

Продолжение описания параметров подпрограммы CHJAC:

m - число функций в системе уравнений.

n - число неизвестных в системе уравнений.

x - точка (вектор размера *n*), в которой выполняется проверка якобиана.

info - целочисленный массив формы (*Ldinfo, n*), информирующий о результатах проверки *m*×*n*-якобиана; имеет следующий смысл:

- *info*(*i, j*) = 0 - значение предоставленного пользователем якобиана функции *i* в точке *x*(*j*) расходится с численной оценкой подпрограммы CHJAC;
- *info*(*i, j*) = 1 - значение предоставленного пользователем якобиана функции *i* в точке *x*(*j*) хорошо согласуется с численной оценкой подпрограммы CHJAC;
- *info*(*i, j*) = 2 - значение предоставленного пользователем якобиана функции *i* в точке *x*(*j*) не согласуется с численной оценкой подпрограммы CHJAC, но, возможно, подпрограмма CHJAC не может выполнить численную оценку якобиана;
- *info*(*i, j*) = 3 - значение предоставленного пользователем якобиана функции *i* в точке *x*(*j*) и соответствующая численная оценка подпрограммы CHJAC равны нулю; якобиан нужно проверить в другой точке.

Ldinfo - ведущий размер массива *info*.

Комментарий. При работе с CHJAC может возникнуть информационная ошибка типа 4 с кодом 1, означающая, что значение предоставленного пользователем якобиана расходится с численной оценкой подпрограммы CHJAC.

Описание:

Подпрограмма CHJAC использует для оценки градиента *i*-й функции $f(x)$ от *n* переменных в точке *x* следующую конечно-разностную формулу:

$$g_{ij}(x) = \frac{f_i(x + h_j e_j) - f_i(x)}{h_j}, \; j = 1, \ldots, n,$$

где $h_j = \sqrt{\varepsilon_м}\ \max(|x_j|, 1/s_j)\text{sign}(x_j)$, $\varepsilon_м$ - машинная точность, s_j - коэффициент масштабирования j-й переменной, $e^{(j)}$ - j-й канонический вектор.

Затем CHJAC проверяет предоставленный пользователем якобиан $J(x)$, сравнивая его с конечно-разностной аппроксимацией градиента $g_{ij}(x)$. Если

$$|g_{ij}(x) - J_{ij}(x)| < \tau|J_{ij}(x)|,$$

где $\tau = \varepsilon_м^{1/4}$, то элемент $J_{ij}(x)$ классифицируется как верный, в противном случае CHJAC определяет ошибки вычисления и аппроксимации. Когда обе ошибки достаточно малы, то элемент $J_{ij}(x)$ классифицируется как неверный. В случае больших ошибок, подпрограмма CHJAC, используя почти оптимальный размер шага, заново вычисляет $g_{ij}(x)$ и сообщает, что значение элемента $J_{ij}(x)$ верно, если

$$|g_{ij}(x) - J_{ij}(x)| < 2\tau|J_{ij}(x)|.$$

В противном случае элемент $J_{ij}(x)$ классифицируется как неверный, если только ошибка оптимального шага не больше чем $\tau|J_{ij}(x)|$. Если же она больше, то, скорее всего, численную аппроксимация градиента $g_{ij}(x)$ нельзя выполнить правильно. Детали см. в [43].

Пример. В точке (-1.2, 1.0) проверяется предоставленный пользователем якобиан системы

$$f_1(x) = 1 - x_1,$$

$$f_2(x) = 10(x_2 - x_1^2).$$

Поскольку *fjac*(1, 2) и соответствующая этому элементу числовая оценка равны нулю, то подпрограммой второго уровня C2JAC выводится надлежащее предупреждение.

```
program chjacTest
 use dfimsl
 integer(4), parameter :: m = 2, n = 2, Ldinfo = m
 integer(4) :: info(Ldinfo, n)
 real(4) :: x(n)
 external fcn, jac
 data x / -1.2, 1.0 /
 ! Проверяем пользовательский якобиан
 call chjac(fcn, jac, m, n, x, info, Ldinfo)
 call wrirn(' The information matrix', m, n, info, Ldinfo, 0)
end program chjacTest

subroutine fcn(m, n, x, f)
 integer(4) :: m, n; real(4) :: x(n), f(m)
 f(1) = 1.0 - x(1); f(2) = 10.0 * (x(2) - x(1) * x(1))
end subroutine fcn
```

```
subroutine jac(m, n, x, fjac, Ldfjac)
 integer(4) :: m, n, Ldfjac; real(4) :: x(n), fjac(Ldfjac, n)
 fjac(1, 1) = -1.0; fjac(1, 2) = 0.0; fjac(2, 1) = -20.0 * x(1); fjac(2, 2) = 10.0
end subroutine jac
```

Результат:

```
*** WARNING ERROR 2 from C2JAC. The numerical value of the Jacobian
*** evaluation for function 1 at the point X(2) = 1.000000E+00 and
*** the user-supplied value are both zero. The Jacobian for this
*** function should probably be re-checked at another value for
*** this point.
The information matrix
1 3
1 1
```

4.7.10. ПОДПРОГРАММА GGUES (DGGUES)

Генерирует точку n-мерного пространства. Имеет вызов

CALL GGUES(*n, a, b, k, ido, s*)

Параметры подпрограммы GGUES:

Входные: n, a, b, k.

Входной/выходной: ido.

Выходной: s.

n - размерность пространства.

a - вещественный вектор размера *n* (см. параметр *b*).

b - вещественный вектор размера *n*. Векторы *a* и *b* определяют прямоугольную область, в которой генерируются точки, т. е. $a(i) < s(i) < b(i)$ для $i = 1, \ldots, n$. Причем если $b(i) < a(i)$, то $b(i) < s(i) < a(i)$.

k - число генерируемых точек.

ido - параметр инициализации; при первом вызове *ido* должен равняться нулю. Далее подпрограмма GGUES устанавливает в *ido* единицу и возвращает в *s* первую сгенерированную точку. Последующие вызовы должны выполняться с $ido = 1$.

s - вещественный вектор размера *n*, содержащий сгенерированную точку. Каждый вызов GGUES порождает новую точку, которая записывается в *s*.

Комментарии:

1. При работе с GGUES может возникнуть информационная ошибка типа 4 с кодом 1, означающая, что выполнена попытка сгенерировать больше чем *k* точек.
2. Подпрограмму GGUES можно использовать с любой процедурой нелинейной оптимизации, требующей одну стартовую точку. Для генерации задаются прямоугольник, в пределах которого генерируются точки (оп-

ределяется параметрами *a*, *b* и *n*), и число точек *k*. Далее вызывается GGUES, генерируется точка и осуществляется обращение к подпрограмме оптимизации. Приведенная последовательность вызовов повторяется *k* раз. Число итераций оптимизационной подпрограммы задается небольшим (5-10). Лучшая точка, найденная таким образом, может быть употреблена затем в качестве начального предположения о точке минимума. Такой порядок поиска начальной точки позволяет более точно находить минимум функции.

3. Детали, связанные с генерацией точек, см. в [8].

Пример. Генерируются 10 точек в прямоугольнике, задаваемом вершинами (1, 1), (3, 1), (3, 2) и (1, 2).

```
program gguesTest
 use dfimsl
 integer(4), parameter :: n = 2
 integer(4) :: ido, j, k
 real(4) :: a(n) = (/1.0, 1.0 /), b(n) = (/ 3.0, 2.0 /), s(n)
 write(*, "(' Point number', 7x, 'Generated point')")
 k = 10; ido = 0
 do j = 1, k
  call ggues(n, a, b, k, ido, s)
  write(*, "(1x, i7, 14x, '(', f4.1, ',', f6.3, ')')") j, s(1), s(2)
 end do
end program gguesTest
```

Результат:

```
 Point Number      Generated Point
            1          ( 1.5, 1.125)
            2          ( 2.0, 1.500)
            3          ( 2.5, 1.750)
            4          ( 1.5, 1.375)
            5          ( 2.0, 1.750)
            6          ( 1.5, 1.625)
            7          ( 2.5, 1.250)
            8          ( 1.5, 1.875)
            9          ( 2.0, 1.250)
           10          ( 2.5, 1.500)
```

5. СОРТИРОВКА И ПОИСК ДАННЫХ

5.1. МЕТОДЫ СОРТИРОВКИ ДАННЫХ

Основное назначение сортировки - обеспечить быстрый поиск данных. Помимо этого, в отсортированном файле или массиве гораздо быстрее выполнять многие вычисления. Например, существенно быстрее подсчитывается число элементов, равных заданному значению. Также во многих случаях отсортированный файл более удобен для просмотра и визуального анализа данных.

5.1.1. ВНЕШНЯЯ И ВНУТРЕННЯЯ СОРТИРОВКА

Подпрограммы IMSL сортируют только числовые векторы. После их применения данные в отсортированном векторе располагаются в порядке возрастания, а точнее, неубывания их значений или модулей значений. Сортировка символьных векторов может быть выполнена подпрограммой SORTQQ. Для сортировки данных производного типа по произвольному ключу потребуется создать пользовательскую процедуру.

Применяя подпрограммы сортировки, можно отсортировать данные в последовательном файле. Порядок действий очевиден: а) ввести данные в массив; б) выполнить его сортировку; в) переместиться в начало файла; г) перенести данные из отсортированного массива в файл. Такая сортировка файла называется *внутренней* - все данные файла одновременно находятся в выделенной под процесс памяти.

Сортировка, при которой часть данных находится в принадлежащей программе памяти, а часть во внешней памяти, называется *внешней*. Крайним проявлением внешней сортировки является сортировка файла прямого доступа, в котором, как известно, можно редактировать отдельные записи.

5.1.2. ПОНЯТИЕ КЛЮЧА

Пусть файл данных содержит некоторую последовательность из n элементов: $r_1, r_2, ..., r_n$. Каждый элемент файла будем называть *записью*. Как правило, записи файла состоят из одинакового числа компонентов, называемых *полями записи*. В общем случае компоненты записи могут быть разного типа.

Сформируем для примера структуры, отображающие данные о студенте и его экзаменационных оценках, и объявим переменные (записи) созданных типов данных:

```
type person                          ! Тип для хранения данных о студенте
 integer(4) stud_id                  ! Идентификационный номер студента,
 character(30) name, phone           ! Ф. И. О. студента, его телефон,
```

```
 character(70) address                ! адрес
 character(15) group                  ! и учебная группа
end type person

type exam                             ! Описание производного типа exam
 integer(4) stud_id                   ! Идентификационный номер студента
 integer(4) m1, m2, m3, m4            ! Оценки студента
end type exam
type(person) ps, group(30)            ! Объявляем объекты производных типов
type(exam) stud
```

Используем для хранения определенных в структурах данных, например, файлы прямого доступа pers.dir и exam.dir. Поля записей файлов совпадают с соответствующими полями введенных структур.

Свяжем с каждой записью файлов r_i *ключ* k_i, понимая под ключом одно из полей записи. Правда, такое понятие ключа является узким: в общем случае ключом записи r_i является некоторое выражение, среди операндов которого присутствует одно или несколько полей записи.

Выбор ключа диктуется практическими целями поиска и выбора данных. В рассматриваемых файлах целесообразно в качестве ключа использовать поле *stud_id*. Кроме того, в файле pers.dir полезным будет и ключ *name*, например для отображения списка студентов в алфавитном порядке.

Файл отсортирован по ключу, если для любых $k_i < k_j$ запись r_i всегда предшествует r_j, где i и j - номера записей в файле до выполнения сортировки. (Напомним, что первая запись файла имеет номер 1.)

Вполне возможно, что две записи имеют в некотором файле одинаковый ключ. Метод сортировки называется *устойчивым*, если для всех записей r_i и r_j, таких, что $k_i = k_j$, выполняется условие: в отсортированном файле r_i предшествует r_j, если r_i предшествует r_j в первоначальном файле.

Подпрограммы сортировки IMSL выполняют, по существу, сортировку числовых ключей, хранящихся в одномерных массивах.

5.1.3. СОРТИРОВКА ТАБЛИЦЫ УКАЗАТЕЛЕЙ

Сортировать можно или сами записи файла, или указатели некоторой вспомогательной таблицы. Пример первого случая приведен на рис. 5.1.

Если объем данных в каждой записи файла велик, то сортировка самих записей является нецелесообразной, в частности по причине больших временных затрат на перемещение записей в процессе сортировки. Кроме того, часто файл имеет не один, а несколько ключей. В этом случае можно ввести вспомогательную *таблицу указателей* или несколько таких таблиц и перемещать при сортировке не записи, а указатели на записи (рис. 5.2). Такая

сортировка называется *сортировкой таблицы адресов* (*указателей*). В базах данных таблица указателей может размещаться в отдельном индексном файле.

Номер записи	*Ключ*	*Другие поля*
1	8	...
2	20	
3	15	
4	4	
5	10	

а

Ключ	*Другие поля*
4	...
8	
10	
15	
20	

б

Рис. 5.1. Сортировка записей: а - исходный файл; б - отсортированный файл

Таблица указателей может содержать два поля: ключ и номер соответствующей записи исходного файла.

Номер записи	*Файл*			*Отсортированная таблица указателей*	
	Ключ	*Другие поля*		*Ключ*	*Номер записи*
1	8	...			
2	20			4	4
3	15			8	1
4	4			10	5
5	10			15	3
				20	2

Рис. 5.2. Отсортированная таблица указателей

При работе с таблицами указателей задача поиска записи решается в два этапа: а) в таблице указателей ищется ключ; б) в файле данных ищется запись (по ее номеру), с которой связан найденный в таблице указателей ключ.

Рассмотрим теперь алгоритмы внутренней пузырьковой и быстрой сортировки ключей. Расширение этих методов для сортировки записей файла или таблицы указателей представляется очевидным.

5.1.4. СОРТИРОВКА МЕТОДОМ ПУЗЫРЬКА

5.1.4.1. СОРТИРОВКА МАССИВА

Сортировка методом пузырька наиболее проста для реализации, но имеет по сравнению с другими методами наименьшую вычислительную эффективность.

Не теряя общности, будем для простоты изложения в дальнейшем рассматривать задачу сортировки массива x целых чисел, в котором первые n чисел должны быть отсортированы так, чтобы $x_i \le x_j$ для $1 \le i \le j \le n$.

Идея сортировки методом пузырька в том, чтобы просмотреть массив последовательно несколько раз. Один просмотр состоит из сравнения каждого элемента массива со следующим за ним элементом (x_i сравнивается с x_{i+1}) и обмена этих двух элементов, если они располагаются не в нужном порядке (если $x_i > x_{i+1}$).

Рассмотрим массив:	25	57	48	37	12	92	86	33.
Результат сортировки:	12	25	33	37	48	57	86	92.

Проанализируем процесс сортировки.

На первом просмотре делаются сравнения:

x_1 с x_2	(25 с 57)	Нет обмена
x_2 с x_3	(57 с 48)	Обмен
x_3 с x_4	(57 с 37)	Обмен
x_4 с x_5	(57 с 12)	Обмен
x_5 с x_6	(57 с 92)	Нет обмена
x_6 с x_7	(92 с 86)	Обмен
x_7 с x_8	(92 с 33)	Обмен

После первого просмотра в результате обменов элементы массива расположатся в таком порядке:

25 48 37 12 57 86 33 92

Отметим, что наибольший элемент (в данном случае 92) находится после первого просмотра в нужной позиции. В общем случае элемент $x_{n-pass+1}$ результирующего массива будет находиться в нужной позиции после итерации *pass*. Отсюда и идет название метода: каждое число медленно "всплывает", как пузырек, вверх на свою конечную позицию.

После второго просмотра в результате обменов элементы массива расположатся так:

25 37 12 48 57 33 86 92

Как и ожидалось, в нужной позиции оказалось второе по величине число, 86. Поскольку каждая итерация помещает в нужную позицию очередной элемент, для сортировки массива из n чисел потребуется не более n - 1 итераций.

Полный набор итераций при сортировке методом пузырька таков:

Массив до начала сортировки	25 57 48 37 12 92 86 33
Итерация 1	25 48 37 12 57 86 33 92
Итерация 2	25 37 12 48 57 33 86 92
Итерация 3	25 12 37 48 33 57 86 92
Итерация 4	12 25 37 33 48 57 86 92
Итерация 5	12 25 33 37 48 57 86 92
Итерация 6	12 25 33 37 48 57 86 92
Итерация 7	12 25 33 37 48 57 86 92

В каждой строчке приведенного списка итераций подчеркнуты элементы массива, находящиеся в нужной позиции.

Анализ полного набора итераций подсказывает два очевидных улучшения метода.

Во-первых, поскольку все элементы в позициях $k \geq n - pass + 1$ уже находятся после итерации *pass* на нужных местах, их рассмотрение на последующих итерациях избыточно. Следовательно, на каждой итерации число необходимых сравнений уменьшается на единицу. Так, при первом просмотре нужно сделать $n - 1$ сравнений, на втором - $n - 2$, на просмотре с номером *pass* - только $n - pass$. То есть данный процесс ускоряется с каждым просмотром.

Во-вторых, не всегда следует выполнять все $n - 1$ просмотров. В частности, приведенный массив был отсортирован после пяти итераций (вместо семи). Для исключения холостых проходов нужно иметь возможность обнаружения факта завершения сортировки массива и прекращать итерации при его обнаружении. Признаком того, что массив отсортирован, является отсутствие перестановок на очередном проходе. Применительно к нашему примеру это означает, что факт завершения сортировки будет установлен только на шестом проходе.

Используя эти два улучшения, напишем подпрограмму *bubble* сортировки методом пузырька, принимающую в качестве параметра неотсортированный массив и *n* - число сортируемых элементов - и возвращающую массив, в котором отсортированы первые *n* элементов.

```
program sort0
 integer(4), parameter :: n = 8
 integer(4) :: x(n) = (/ 25, 57, 48, 37, 12, 92, 86, 33 /)
 call bubble(x, n)
 print '(1x, 10i4)', x(1:n)                    ! 12 25 33 37 48 57 86 92
end program sort0
subroutine bubble(x, n)
```

```
integer(4) :: n, x(n), i, pass, hold
logical(4) :: fl
pass = 1                              ! Номер просмотра массива
fl = .true.
do while(fl .and. pass < n)           ! Число просмотров меньше n
 ! fl равен .FALSE., если нет обменов, т. е. когда массив отсортирован
 fl = .false.
 do i = 1, n - pass                   ! Реализация каждого отдельного
  if(x(i) > x(i+1)) then              ! просмотра
   fl = .true.                        ! Нужен обмен
   hold = x(i); x(i) = x(i+1); x(i+1) = hold
  end if
 end do
 pass = pass + 1
end do
end subroutine bubble
```

Замечание. Можно, используя родовой интерфейс, создать 3 подпрограммы сортировки массивов целого, вещественного и символьного типов и обращаться к ним посредством родового или специфических имен.

Проанализируем вычислительную эффективность метода. Всего алгоритм (без усовершенствований) предусматривает выполнение n - 1 просмотров и n - 1 сравнений на каждом просмотре. Таким образом, общее число сравнений на всех возможных проходах равно $(n - 1)*(n - 1) = n^2 - 2n + 1$, что составляет $O(n^2)$.

Введенные усовершенствования метода хотя и сокращают общее число сравнений, но не изменяют порядка вычислительной сложности. В самом деле, число сравнений на итерации *pass* равно n - *pass*. При наличии k итераций общее число сравнений равно $(n - 1) + (n - 2) + ... + (n - k) = (2k*n - k^2 - k)/2$. Можно показать, что среднее число итераций k представляет собой $O(n)$, так что общая формула имеет по-прежнему порядок $O(n^2)$, хотя постоянный сомножитель теперь меньше, чем ранее.

Если массив полностью отсортирован, то вычислительная сложность метода составляет $O(n)$ - необходимо n - 1 сравнений, чтобы в этом убедиться.

Для улучшения метода имеются и иные способы. Один из них таков: сортировка методом пузырька может быть ускорена при помощи выполнения следующих один за другим просмотров в *противоположных* направлениях, так что небольшие по величине элементы быстро перемещаются в начало массива таким же способом, как большие элементы перемещаются в его конец. Это приводит к уменьшению необходимого числа просмотров.

5.1.4.2. СОРТИРОВКА ФАЙЛА ПРЯМОГО ДОСТУПА

Рассмотрим прямой файл exam.dir, каждая запись которого содержит определенные в типе *exam* поля: код студента и 4 оценки. (Тип *exam* введен в разд. 5.1.2.) Длина первого поля - 6 символов, а остальных - по 2 символа. То есть длина записи равна 14 байтам.

Выполним, используя метод пузырька, сортировку файла по первому полю. Приводимую ниже программу проверим на следующем тестовом наборе:

Stud_id	*m1*	*m2*	*m3*	*m4*
25	4	3	5	4
57	3	4	5	4
48	5	3	4	4
37	4	5	4	3
12	5	4	3	4
92	4	4	5	5
86	3	3	5	4
33	4	4	3	3

```
program sortDir
 type exam                          ! Описание производного типа exam
  integer(4) stud_id                ! Идентификационный номер студента
  integer(4) m1, m2, m3, m4         ! Оценки студента
 end type exam
 type(exam) stud1, stud2
 integer(4) :: n, i, pass
 logical(4) :: fl
 character(10) :: fmt = '(i6, 4i2)'  ! Строка формата
 open(1, file = 'exam.dir', access = 'direct', form = 'formatted', recl = 14, status = 'old')
 n = 0
 do while(.not. eof(1))             ! Подсчитаем число записей в файле
  read(1, fmt) stud1
  n = n + 1
 end do
 pass = 1                           ! Номер просмотра файла
 fl = .true.
 do while(fl .and. pass < n)        ! Число просмотров меньше n
  ! fl равен .FALSE., если нет обменов, т. е. когда файл отсортирован
  fl = .false.
  rewind(1)                         ! "Перемотка" файла в его начало
  read(1, fmt) stud1                ! Аналог x(i) предыдущей программы
  do i = 1, n - pass                ! Реализация одного просмотра
   read(1, fmt) stud2               ! Аналог x(i + 1) предыдущей программы
```

```
    if(stud1%stud_id > stud2%stud_id) then
      fl = .true.                    ! Нужен обмен
      backspace(1), backspace(1)     ! Подготовка к обмену
      write(1, fmt) stud2            ! Выполняем обмен
      write(1, fmt) stud1
    else
      stud1 = stud2
    end if
  end do
  pass = pass + 1
end do
rewind(1)                            ! Контрольный вывод на экран
do while(.not. eof(1))
  read(1, fmt) stud1
  write(*, fmt) stud1
end do
end program sortDir
```

5.1.5. БЫСТРАЯ СОРТИРОВКА

Подпрограммы сортировки библиотеки IMSL используют одну из модификаций алгоритма быстрой сортировки, к изложению которого мы переходим.

Рассмотрим массив x:

25 37 12 33 **48** 57 92 86

В нем число 48 характеризуется тем, что, во-первых, все расположенные левее него числа меньше 48 и, во-вторых, числа, расположенные правее него, больше 48. Назовем такое число *разделителем массива*. Нетрудно понять, что теперь мы можем отдельно решать задачу сортировки для чисел до разделителя и для чисел после него. Кроме того, сам разделитель находится в нужной позиции, т. е. в дальнейшей сортировке он уже не рассматривается.

Рассмотрим теперь массив, в котором нет разделителя:

37 25 57 48 12 92 86 33

Чтобы воспользоваться только что приведенной идеей уменьшения размерности задачи сортировки, нам надо научиться выполнять такие перестановки в массиве, после которых один из его элементов станет разделителем. Выберем для будущего разделителя первый элемент массива, т. е. число 37. Разделитель окажется в нужной позиции, если разместить числа 25, 12 и 33 слева от 37. Выполним это так: будем просматривать последовательно элементы массива, начиная с его первой позиции, до тех пор, пока не встретим число, большее разделителя. Присвоим этой позиции имя L1. Далее просмотрим элементы массива начиная с последней позиции, останавливаясь

при обнаружении числа, меньшего 37. Дадим такой позиции имя R1. В нашем случае после выполнения этих двух просмотров возникнет следующая ситуация:

37	25	**57**	48	12	92	86	**33**
		L1					R1

Нетрудно предугадать следующий шаг алгоритма - числа 57 и 33 должны быть переставлены местами, тогда получится массив

37	25	33	48	12	92	86	57
		L1					R1

Продолжим теперь поиск элемента, большего 37, перемещаясь вправо, начиная с позиции L1, и элемента, меньшего 37, перемещаясь влево, начиная с позиции R1. Такой поиск даст следующий результат:

37	25	33	48	12	92	86	57
			L1	R1			

Выполним и перестановку:

37	25	33	12	48	92	86	57
			L1	R1			

Еще одна пара перемещений не изменит картины, но даст нам основание для прекращения левых и правых перемещений вдоль массива (критерий прекращения перемещений - истинность выражения L1 >= R1) и перестановки разделителя (числа 37) в его окончательную позицию. После выполнения перемещений, предшествующих перестановке разделителя, имена R1 и L1 расположатся так:

37	25	33	12	48	92	86	57
			R1	L1			

Сам же разделитель должен быть размещен в позиции R1, что выполняется в результате обмена элементов $x(1)$ и $x(R1)$, т. е. чисел 12 и 37. Приведем и результат:

12	25	33	**37**	48	92	86	57

Итак, массив разбит на две части (до и после разделителя), которые можно сортировать отдельно, применяя к каждой из частей только что приведенную схему. Завершим же сортировку, когда длина каждой из полученных в результате разбиения частей будет равна единице. Выполняющая данный процесс программа содержит рекурсивную реализацию алгоритма быстрой сортировки и подпрограмму перестановки двух элементов массива.

```
program qs
 integer(4), parameter :: n = 100
```

```
integer(4) :: x(n), i
real(4) :: r
do i = 1, n                              ! Генерация целочисленного вектора x
  call random_number(r)
  x(i) = int(n * r + 1)
end do
call qsort(x, 1, n)                      ! Сортировка первых n элементов массива
print '(20i4)', x                        ! Вывод отсортированного массива

contains

recursive subroutine qsort(x, L, R)
 integer x(:), L, R, L1, R1
 if(L < R) then                  ! Выход из подпрограммы будет выполнен, когда
  L1 = L; R1 = R                 ! длина каждой из полученных в результате
  do  ! разбиения частей массива будет равна единице
   do while(L1 < R .and. x(L1) <= x(L))        ! Перемещаем L1 вправо
    L1 = L1 + 1
   end do
   do while(R1 > L .and. x(R1) >= x(L))        ! Перемещаем R1 вправо
    R1 = R1 - 1
   end do
   if(L1 < R1) call swap(x(L1), x(R1))         ! Промежуточная перестановка
   if(L1 >= R1) exit                           ! Положение разделителя найдено
  end do
  if(R1 > L) then
   call swap(x(L), x(R1))                      ! Размещение разделителя в R1
   call qsort(x, L, R1 - 1)                    ! Сортировка левого фрагмента
  end if
  call qsort(x, R1 + 1, R)                     ! Сортировка правого фрагмента
 end if
end subroutine qsort

subroutine swap(a, b)
 integer a, b, hold
 hold = a; a = b; b = hold
end subroutine swap
end program qs
```

Замечание. Выбор в качестве будущего разделителя первого элемента массива (или последующего фрагмента) необязателен и в ряде случаев неудовлетворителен. Так, в случае почти отсортированного массива будущий разделитель лучше выбирать ближе к середине рассматриваемого фрагмента.

Вычислительная сложность быстрой сортировки оценивается как $O(n\log_2 n)$. Так, если сортируется массив из 1024 элементов, то пузырьковая сортировка будет в среднем выполняться в

$$\frac{n^2}{2}\Big/(n\log_2 n)=\frac{1024}{20}=51,2 \quad \text{раза медленнее.}$$

5.2. СОРТИРОВКА ПОДПРОГРАММАМИ IMSL

Рассмотрим процедуры сортировки библиотек IMSL 77 и IMSL 90.

5.2.1. ПОДПРОГРАММЫ СОРТИРОВКИ IMSL 77

В табл. 5.1 приводится синтаксис вызова и назначение подпрограмм из библиотеки IMSL, выполняющих сортировку данных; табл. 5.2 содержит описание их параметров.

Таблица 5.1. Подпрограммы сортировки данных

Вызовы	*Назначение*
Сортировка целочисленных данных	
CALL SVIBN(*n*, *ia*, *ib*)	Сортирует элементы вектора *ia* в неубывающем порядке их абсолютных значений, размещая результат в *ib*
CALL SVIBP(*n*, *ia*, *ib*, *iperm*)	Делает то же, что и SVIBN, дополнительно возвращая в *iperm* данные о перестановках элементов *ia*
CALL SVIGN(*n*, *ia*, *ib*)	Сортирует элементы вектора *ia* в неубывающем порядке их алгебраических значений, размещая результат в *ib*
CALL SVIGP(*n*, *ia*, *ib*, *iperm*)	Делает то же, что и SVIGN, дополнительно возвращая в *iperm* данные о перестановках элементов *ia*
Сортировка вещественных данных	
CALL SVRBN(*n*, *ra*, *rb*) CALL DSVRBN(*n*, *ra*, *rb*)	Сортируют элементы вектора *ra* в неубывающем порядке их абсолютных значений, размещая результат в *rb*
CALL SVRBP(*n*, *ra*, *rb*, *iperm*) CALL DSVRBP(*n*, *ra*, *rb*, *iperm*)	Делают то же, что и SVRBN, дополнительно возвращая в *iperm* данные о перестановках элементов *ra*
CALL SVRGN(*n*, *ra*, *rb*) CALL DSVRGN(*n*, *ra*, *rb*)	Сортируют элементы вектора *ra* в неубывающем порядке их алгебраических значений, размещая результат в *rb*

CALL SVRGP(*n*, *ra*, *rb*, *iperm*) CALL DSVRGP(*n*, *ra*, *rb*, *iperm*)	Делает то же, что и SVRGN, дополнительно возвращая в *iperm* данные о перестановках элементов *ra*

Таблица 5.2. Параметры подпрограмм сортировки данных

Имя	*Смысл / вид*	*Тип*
	Сортировка целочисленных данных	
ia	Вектор размера *n*, подлежащий сортировке / *входной*	INTEGER(4)
ib	Отсортированный вектор размера *n*; если после завершения сортировки вектор *ia* не нужен, то на месте *ib* может стоять вектор *ia*, например: CALL SVIBN(*n*, *ia*, *ia*) / *выходной*	"
	Сортировка вещественных данных	
ra	Вектор размера *n*, подлежащий сортировке / *входной*	REAL(4) или REAL(8)
rb	Отсортированный вектор размера *n*; если после завершения сортировки вектор *ra* не нужен, то на месте *rb* может стоять вектор *ra*, например: CALL SVRBN(*n*, *ra*, *ra*) / *выходной*	То же
	Параметры, применяемые со всеми подпрограммами сортировки данных	
iperm	Вектор, содержащий данные о перестановках элементов *ia* или *ra*. На входе *iperm* необходимо проинициализировать величинами 1, 2, ..., *n*. Причем эти величины могут быть занесены в *iperm* в любом порядке без повторений. Тогда если элемент, например *ia*(*i*), имел в *iperm* номер *k* и если после сортировки этот элемент размещен в позиции *j* вектора *ib*, то *iperm*(*j*) = *k* / *входной/выходной*	INTEGER(4)
n	Число элементов подлежащих сортировке / *входной*	"

Замечания:

1. После завершения сортировки подпрограммой, сортирующей абсолютные значения, $|a_j| \le |a_i|$ для $j < i$; в случае сортировки алгебраических значений $a_j \le a_i$ для $j < i$.
2. Подпрограммы, начинающиеся с буквы D, сортируют данные типа REAL(8).
3. Все подпрограммы сортировки используют один и тот же модифицированный алгоритм быстрой сортировки [18, 33, 44];
4. Вектор перестановок *iperm* можно применять, например, при сортировке данных производного типа по числовому ключу.

Пример для SVIBP и SVIGP. Первоначально выполняется сортировка 10 элементов *ia* в неубывающем порядке их абсолютных значений, а затем - алгебраических значений.

```
program sort1
 use dfimsl
 integer(4), parameter :: n = 10
 integer(4) :: ia(n), ib(n), iperm(n), i
 ia = (/ 10, 9, -4, 1, 6, 5, 8, 3, -2, 7 /)
 iperm = (/ (i, i = 1, n) /)              ! Инициализация вектора iperm
 call svibp(n, ia, ib, iperm)             ! Сортировка абсолютных значений ia
 print *, 'SVIBP'
 print '(1x, a, 20i4)', 'ib:   ', ib(1:n)
 print '(1x, a, 20i4)', 'iperm:', iperm(1:n)
 iperm = (/ (i, i = 1, n) /)
 call svigp(n, ia, ib, iperm)             ! Сортировка элементов ia
 print *, 'SVIGP'
 print '(1x, a, 20i4)', 'ib:   ', ib(1:n)
 print '(1x, a, 20i4)', 'iperm:', iperm(1:n)
end program sort1
```

Результат:

SVIBP

ib:	1	-2	3	-4	5	6	7	8	9	10
iperm:	4	9	8	3	6	5	10	7	2	1

SVIGP

ib:	-4	-2	1	3	5	6	7	8	9	10
iperm:	3	9	4	8	6	5	10	7	2	1

5.2.2. ПОДПРОГРАММА SORT_REAL БИБЛИОТЕКИ IMSL 90

Сортирует вектор (массив ранга 1) вещественных чисел x, размещая результат в вектор y, в котором $y_1 \le y_2 \le \ldots \le y_n$. Имеет вызов

CALL SORT_REAL(*x, y,* [*nsize* = *n*], [*iperm* = *iperm*], &
[*icycle* = *icycle*], [*iopt* = *iopt*(:)])

Обязательные параметры подпрограммы SORT_REAL:
Входной: x.
Выходной: y.
x - вектор, содержащий подлежащие сортировке данные.
y - вектор, содержащий отсортированные данные.

Необязательные параметры подпрограммы SORT_REAL:
Входной: nsize.

Входной/выходной: iperm.

Выходные: icycle, iopt.

nsize = *n* - число сортируемых элементов вектора *x*; по умолчанию *n* = SIZE(*x*).

iperm = *iperm* - если до вызова SORT_REAL задать

```
iperm(i) = (/ (i, i = 1, n) /)
```

то после вызова SORT_REAL вектор *iperm* получит значения, используя которые отсортированный вектор можно получить, выполнив присваивание

```
y = x(iperm(1:n))
```

icycle = *icycle* - вектор, содержащий циклические перестановки, позволяющие получить исходный порядок элементов из отсортированного вектора *y* в результате выполнения следующего цикла:

```
call sort_real(x, y, icycle = icycle)     ! Сортируем вектор x, размещая результат в y
do i = 1, n                               ! После выполнения цикла имеем: x(1:n) = y(1:n)
 j = icycle(i)
 hold = y(j)
 y(j) = y(i)
 y(i) = hold
end do
```

iopt = *iopt*(:) - массив производного типа с той же разновидностью, что и входной вектор. Используется для передачи в подпрограмму SORT_REAL необязательных опций. Принимает следующее значение:

Префикс опции = ?	*Имя опции*	*Значение*
s_, d_	sort_real_scan_for_NaN	1

iopt(*IO*) = ?_options(?_*sort_real_scan_for_NaN*, ?_*dummy*) - проверяет входной вектор на наличие в нем NaN (не числа). По умолчанию проверка не выполняется.

Описание:

Подпрограмма SORT_REAL является реализованной в стиле Фортран 90 версией подпрограммы SVRGN библиотеки IMSL 77.

Замечание. Фатальные и завершающие ошибки, возникающие при работе с SORT_REAL, имеют номера 561-567 и 581-587.

Пример 1. Первоначально случайным образом формируется вектор. Затем он сортируется в неубывающем порядке.

```
program sort_realTest1
 use sort_real_int
 use rand_gen_int
 implicit none
 integer, parameter :: n = 100
 real(kind(1e0)), dimension(n) :: x, y
 call rand_gen(x)                    ! Формируем вектор x из случайных чисел
 call sort_real(x, y)                ! Сортируем вектор x, размещая результат в y
 ! Элементы вектора y должны быть размещены в неубывающем порядке
 if(count(y(1:n - 1) > y(2:n)) == 0) then
  write(*, *) 'Example 1 for SORT_REAL is correct'
 end if
end program sort_realTest1
```

Пример 2. Первоначально n случайных чисел сортируются в невозрастающем порядке. Чтобы получить такой порядок вместо вектора x используется вектор $-x$, а затем в результирующем векторе y меняется знак его элементов. Также результат можно получить, употребив вектор перестановок: $y = x(ip(1{:}n))$. Далее столбцы случайной $n \times n$-матрицы перемещаются в порядке, задаваемом вектором перестановок ip.

```
program sort_realTest2
 use sort_real_int; use rand_gen_int
 implicit none
 integer, parameter :: n = 100
 integer i, ip(n)
 real(kind(1e0)) a(n, n), x(n), y(n), temp(n * n)
 call rand_gen(x)                    ! Генерируем случайные вектор и матрицу
 call rand_gen(temp)
 a = reshape(temp, (/ n, n /))
 ip = (/ (i, i=1, n) /)              ! Инициализация вектора перестановок
 ! Сортируем, используя отрицательные значения вектора x, чтобы получить
 ! невозрастающий вектор y
 call sort_real(-x, y, iperm = ip)
 y = -y                                ! То же даст присваивание y = x(ip(1:n))
 a = a(:, ip(1:n))                     ! Перестановки столбцов матрицы a
 if(count(y(1:n - 1) < y(2:n)) == 0) then   ! Проверка результата
  write(*, *) 'Example 2 for SORT_REAL is correct'
 end if
end program sort_realTest2
```

5.2.3. СРАВНЕНИЕ ПРОЦЕДУР СОРТИРОВКИ IMSL И DFLIB

Сортировка вектора любого встроенного типа, в том числе и символьного, может быть выполнена подпрограммой библиотеки DFLIB (MSFLIB):

CALL SORTQQ(*adrarray*, *count*, *size*)

Параметры подпрограммы SORTQQ:
Входные: adrarray, size.
Входной/выходной: count.
Тип всех параметров - INTEGER(4).

adrarray - адрес сортируемого вектора. Для вычисления адреса применяется функция LOC. Сортируемый массив не должен быть производного типа.

count - при вызове равен числу элементов массива, подлежащих сортировке, а на выходе - числу реально отсортированных элементов.

size - положительная константа (*size* < 32'767), задающая тип и разновидность типа сортируемого массива. Если сортируется целочисленный или вещественный массив, то *size* должен принимать одно из приведенных в табл. 5.3 значений.

Таблица 5.3. Значения параметра size

Константа	*Значение*	*Тип массива*
SRT$REAL4	#00010000	REAL(4)
SRT$REAL8	#00020000	REAL(8)
SRT$INTEGER1	#00030000	INTEGER(1)
SRT$INTEGER2	#00040000	INTEGER(2)
SRT$INTEGER4	#00050000	INTEGER(4)

Если *size* не является именованной константой приведенной таблицы и *size* < 32'767, то предполагается, что задан символьный массив типа CHARACTER(*size*).

Чтобы убедиться в том, что сортировка выполнена успешно, следует сравнить значения параметра *count* до и после сортировки. При положительном результате они совпадают.

Предупреждение. Адрес сортируемого массива должен быть вычислен функцией LOC. Значения параметров *count* и *size* должны точно описывать характеристики массива. Если же подпрограмма SORTQQ получила неверные параметры, то будет выполнена попытка сортировки некоторой области памяти. Если память принадлежит текущему процессу, то сортировка будет выполнена, иначе операционная система реализует функции защиты памяти и остановит программу.

Сравним подпрограммы SORTQQ и SVRGN по быстродействию. Сгенерируем для этого вещественный вектор из 3'000'000 элементов, содержащий величины, возвращаемые датчиком случайных чисел. Процессорное

время, затрачиваемое на сортировку, замерим встроенной подпрограммой DATE_AND_TIME. Для датчика случайных чисел используем встроенную подпрограмму RANDOM_NUMBER.

```
program sortCompare
 use dfimsl
 use dflib
 use sort_real_int
 ! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer
 integer(4), parameter :: n = 3000000
 integer(4) :: i
 real(4) :: ra(n), rb(n), rnd
 real(8) :: start_time, finish_time        ! Время начала и завершения вычислений
 ! Функция timer возвращает процессорное время в миллисекундах
 real(8) :: timer, t1, t2, t3, t4
 start_time = timer( )                     ! Начало формирования ra
 do i = 1, n
  call random_number(rnd)                  ! Случайное число 0 ≤ rnd < 1
  ra(i) = rnd
 end do
 finish_time = timer( )                    ! Завершение формирования ra
 t1 = finish_time - start_time             ! Время формирования ra
 start_time = timer( )                     ! Начало сортировки подпрограммой SVRGN
 call svrgn(n, ra, rb)                     ! Сортируем ra и размещаем результат в rb
 finish_time = timer( )
 t2 = finish_time - start_time             ! Время сортировки подпрограммой SVRGN
 start_time = timer( )                     ! Начало сортировки подпрограммой SORTQQ
 call sortqq(loc(ra), n, srt$real4)        ! Сортируем ra и там же размещаем результат
 finish_time = timer( )
 t3 = finish_time - start_time             ! Время сортировки подпрограммой SORTQQ
 print *, trim(ru_doswin('Число сортируемых элементов n =', .false.)), n
 print *, trim(ru_doswin('Время формирования вектора ra =', .false.)), t1
 print *, trim(ru_doswin('Время сортировки подпрограммой SVRGN =', .false.)), t2
 print *, trim(ru_doswin('Время сортировки подпрограммой SORTQQ =', .false.)), t3
end program sortCompare

function timer( )                          ! Определение процессорного времени timer
 real(8) :: timer                          ! в миллисекундах
 integer(4) :: ival(8)
 call date_and_time(values = ival)
 timer = dble(ival(8)) * 0.001_8 + dble(ival(7)) +                              &
      dble(ival(6)) * 60.0_8 + dble(ival(5)) * 3600.0_8
end function timer
```

Результат:

```
Число сортируемых элементов n =    3000000
Время формирования вектора ra =      1.000000000000000
Время сортировки подпрограммой SVRGN =      4.000000000000000
Время сортировки подпрограммой SORTQQ =      9.000000000000000
```

Из результатов следует, что SVRGN несколько быстрее SORTQQ, однако последняя позволяет сортировать не только числовые, но и символьные векторы.

5.3. ПОИСК ДАННЫХ

Поиск данных в векторе и файле выполняется по ключу. Как и при сортировке, выделяют два метода поиска: *внутренний* и *внешний*. В первом случае весь файл находится в отведенной под программу памяти ЭВМ. В случае внешнего поиска большая часть данных находится во внешней памяти, например на жестком диске.

Рассмотрим внутренний поиск применительно к одномерному массиву.

Предположим, что x - это вектор, содержащий n ключей. Пусть *key* является аргументом поиска, т. е. искомым значением ключа. Сформулируем задачу поиска: установить в переменную *search* наименьшее целое число i, такое, что $x(i) = key$, если такое i существует, и нуль - в противном случае. (При такой постановке задачи нижняя граница массива x должна быть больше нуля.)

5.3.1. ПОСЛЕДОВАТЕЛЬНЫЙ ПОИСК

Последовательный поиск является простейшей формой поиска. В поставленной формулировке запись фрагмента поиска тривиальна:

```
integer(4), parameter :: n = 20
integer(4) :: x(n), search, key, i
...
search = 0
i = 1
do while(i <= n .and. search == 0)
  if(x(i) == key) search = i
  i = i + 1
end do
...
```

Эффективность последовательного поиска оценим из предположения, что аргумент поиска может равновероятно располагаться в любой позиции массива. Подсчитаем среднее число сравнений вида *if*($x(i)$ == *key*), необходимых для завершения поиска. Если аргумент является первым элементом

массива, то необходимо одно сравнение; если последним, то необходимо n сравнений. То есть среднее число сравнений для успешного поиска составит $(n + 1)/2$. А в случае неуспешного - n. В любом случае вычислительная сложность алгоритма последовательного поиска равна $O(n)$.

Область применения последовательного поиска - поиск в неупорядоченных массивах (файлах). Если же данные упорядочены, то следует применять бинарный поиск, имеющий и иное название - дихотомия.

5.3.2. БИНАРНЫЙ ПОИСК

Подпрограммы поиска библиотеки IMSL основаны на алгоритме бинарного поиска, к рассмотрению которого мы переходим.

Пусть упорядоченный массив $x(1{:}n)$ содержит элементы

5 7 11 18 26 32 44 57 81 90 94 97 107 116 129 147 179

и пусть задан аргумент поиска *key*, равный, например, 129.

Идея алгоритма бинарного поиска такова:

- сравнить аргумент поиска *key* со значением среднего элемента $x(mid)$ массива x, где $mid = [n/2]$, а $[c]$ - целая часть числа c;
- если они равны, то поиск завершен, иначе, если $key < x(mid)$, выполнить аналогичным образом поиск в позициях массива x, предшествующих позиции *mid*, в противном случае, если $key < k(mid)$, выполнить аналогичным образом поиск в позициях массива k, следующих за позицией *mid*.

Исключить из дальнейшего рассмотрения часть массива позволяет тот факт, что массив упорядочен.

Проиллюстрируем процесс бинарного поиска. Число элементов массива $n = 17$, тогда $[n/2] = 8$. Поэтому первоначально выполняется сравнение *key* с $x_8 = 57$. Так как $key > x_8$, то зона поиска на следующем шаге ограничивается участком от 9-го элемента до 17-го. Этот участок состоит из девяти элементов и его серединой является элемент $x_{13} = 107$ ($[(9 + 17)/2] = 13$). Поскольку $key > x_{13}$, то зона поиска ограничивается участком от 14-го до 17-го элемента. Его серединой является элемент x_{15}. На этом процесс поиска завершен, так как $x_{15} = key$. Отобразим на рис. 5.3 процесс поиска элемента $key = 129$, выделяя посредством подчеркивания на каждом шаге зоны поиска.

Итерация 0	5 7 11 18 26 32 44 57 81 90 94 97 107 116 129 147 179
Итерация 0	5 7 11 18 26 32 44 57 81 90 94 97 107 116 129 147 179
Итерация 1	5 7 11 18 26 32 44 57 81 90 94 97 107 116 129 147 179
Итерация 3	5 7 11 18 26 32 44 57 81 90 94 97 107 116 129 147 179

Рис. 5.3. Пример дихотомии

Приведем нерекурсивную реализацию алгоритма бинарного поиска.

```
program bs
! Текст модуля text_transfer см. в [5, прил. 1]
 use text_transfer                    ! Для вывода русского текста
 integer(4), parameter :: n = 17
 integer(4) :: x(n)
 integer(4) :: key, search
 key = 129                            ! Ищем элемент key
 x = (/ 5, 7, 11, 18, 26, 32, 44, 57, 81, 90, 94, 97, 107, 116, 129, 147, 179 /)
 search = bsearch(x, key)
 if(search > 0) then
  print *, trim(ru_doswin('Искомый элемент находится в позиции ', .false.)), search
 else
  print *, trim(ru_doswin('Элемент ', .false.)), key, trim(ru_doswin(' не найден', .false.))
 end if

 contains

 function bsearch(x, key)             ! Функция выполняет бинарный поиск
  integer(4) :: bsearch               ! ключа key в векторе x
  integer(4) :: x(:), key, low, hi, mid
  low = 1                             ! Нижняя граница поиска
  hi = size(x)                        ! Верхняя граница поиска
  bsearch = 0
  do while(low <= hi .and. bsearch == 0)
    mid = (low + hi)/2
    if(key == x(mid)) then
      bsearch = mid
    else if(key < x(mid)) then
      hi = mid - 1
    else
      low = mid + 1
    end if
  end do                              ! После этого цикла должен следовать
 end function bsearch                 ! приведенный в конце раздела код
end program bs
```

Каждое сравнение уменьшает число возможных кандидатов в 2 раза. Максимальное число шагов поиска будет в том случае, когда аргумент поиска находится в начале или в конце массива. В этом случае при завершении поиска *low* = *hi*. Для достижения такого равенства потребуется $\log_2 n + 1$ итераций. Действительно, если число элементов в массиве равно $n = 2^m$, равенство *low* = *hi* будет достигнуто, когда нерассмотренным останется только один элемент, т. е. через m шагов. В свою очередь, при заданном n имеем $m = \log_2 n$. После анализа последнего элемента получаем общее число итера-

ций $\log_2 n + 1$. Поэтому вычислительная сложность бинарного поиска составляет $O(\log_2 n)$.

Однако приведенный алгоритм и его реализация не позволяют в общем случае точно решить задачу поиска, когда файл или массив содержат повторяющиеся значения ключей. Рассмотрим, например, массив

5 7 11 18 26 32 44 57 81 90 94 97 107 129 129 147 179

в котором элемент (ключ) 129 содержится 2 раза. Тогда, если аргумент поиска будет равен 129, функция *bsearch*, как и прежде, вернет значение 15, т. е. будет найдено не первое, а второе значение ключа 129 (первое значение ключа расположено в позиции 14). В ряде случаев эта ошибка принципиальна и должна быть устранена. Это достигается после включения в функцию *bsearch* следующего кода:

```
! Этот код следует после END DO функции bsearch
if(bsearch > 0) then
 do while(bsearch > 1 .and. x(bsearch - 1) == key)
   bsearch = bsearch - 1
 end do
end if
```

5.3.3. СРАВНЕНИЕ ПОСЛЕДОВАТЕЛЬНОГО И БИНАРНОГО ПОИСКА

Пусть файл, в котором выполняется поиск, отсортирован и содержит 1024 (2^{10}) элемента. В случае последовательного поиска наибольшее число итераций будет равно 1024, а бинарного - 11. То есть разница в два порядка.

Сравним теперь временные затраты на поиск в случае неотсортированного файла. При последовательном поиске максимальное число итераций, разумеется, сохраняется. Бинарный поиск неприменим. Выполним, однако, быструю сортировку файла. В среднем для этого потребуется итераций $1024*\log_2 1024 = 10240$. Далее выполним бинарный поиск, на который будет затрачено не более 11 итераций.

Приведенные цифры позволяют сделать вывод: если файл неотсортирован и в процессе вычислений задача поиска в файле возникает сравнительно редко (в нашем примере не более 10 раз), то можно применять последовательный поиск, в противном случае более целесообразно прежде отсортировать файл или таблицу указателей и при вычислениях применять бинарный поиск. Как правило, именно такой подход используется на практике.

5.4. ПОИСК ПОДПРОГРАММАМИ IMSL

5.4.1. ВЫЗОВЫ И ПАРАМЕТРЫ ПОДПРОГРАММ

В табл. 5.4 приводится синтаксис вызова и назначение подпрограмм из библиотеки IMSL, выполняющих поиск данных; табл. 5.5 содержит описание их параметров.

Таблица 5.4. Подпрограммы поиска данных

Вызовяы подпрограмм	*Назначение*
CALL ISRCH(*n*, *ivalue*, *ix*, *incx*, *index*)	Находит в отсортированном целочисленном векторе заданный элемент и возвращает его индекс
CALL SRCH(*n*, *value*, *x*, *incx*, *index*) CALL DSRCH(*n*, *value*, *x*, *incx*, *index*)	Делают то же, но в вещественном векторе типа REAL(4) или REAL(8)
CALL SSRCH(*n*, *string*, *chx*, *incx*, *index*)	Делает то же, но в символьном векторе

Таблица 5.5. Параметры подпрограмм поиска данных

Имя	*Назначение / вид*	*Тип*
n	Длина вектора *iy* (*y* или *chy*), в котором выполняется поиск / *входной*	INTEGER(4)
ivalue	Скаляр, который ищется в векторе *iy* / *входной*	"
value	" " " " " *y* / *входной*	REAL(4) или REAL(8)
string	" " " " " *chy* / *входной*	CHARACTER(*)
ix	Вектор размера *n* * *incx*. Вектор *iy* получается из *ix* так: *iy*(*i*) = *ix*(1 + (*i* - 1) * *incx*) для *i* = 1, 2, ..., *n*. Элементы *iy*(*1*), *iy*(2), ..., *iy*(*n*) должны быть отсортированы по возрастанию их значений / *входной*	INTEGER(4)
x	Вектор размера *n* * *incx*. Вектор *y* получается из *x* так же, как и вектор *iy* из *ix* / *входной*	REAL(4) или REAL(8)
chx	Вектор размера *n* * *incx*. Вектор *chy* получается из *chx* так же, как и вектор *iy* из *ix* / *входной*	CHARACTER(*)
incx	Расстояние между элементами вектора *ix* (*x* или *chx*); *incx* должен быть больше нуля / *входной*	INTEGER(4)
index	Индекс *iy*, указывающий на *ivalue* (*value* или *string*). Если скаляр *ivalue* (*value* или *string*) найден в *iy* (*y* или *chy*), то *index* > 0, иначе *index* < 0 / *выходной*	"

Замечания:

1. Если параметр *index* принимает значение (на примере отсортированного целочисленного вектора): от 1 до *n*, то *ivalue* = *iy*(*index*); 1, то *ivalue* <

< *iy*(1) или *n* = 0; от -*n* до -2, то *iy*(-*index* - 1) < *ivalue* < *iy*(-*index*); (*n* + 1), то *ivalue* > *iy*(*n*).

2. Параметр *incx*, в частности, полезен, если выполняется поиск в отдельной строке матрицы, например в строке *i* матрицы *ix*. (Элементы строки *i* должны быть отсортированы в неубывающем порядке.) Поиск в строке *i* будет выполняться, если задать *incx* равным ведущему измерению матрицы *ix*, например:

```
integer(4) :: ix(Ldix, n)          ! Ldix - ведущее измерение матрицы ix
. . .
! Поиск в строке i
call isrch(n, ivalue, ix(i, 1), Ldix, index)
```

3. Поиск в отсортированном столбце *j* матрицы *ix* выполняется при вызове

```
integer ix(Ldix, n)                ! Ldix - ведущее измерение матрицы ix
. . .
! Поиск в столбце j
call isrch(Ldix, ivalue, ix(1, j), Ldix, index)
```

4. Подпрограмма ISRCH выполняет бинарный поиск. Алгоритм обсуждается в [24, с. 407-411].
5. Подпрограмма SSRCH использует при сравнении символьных данных встроенные функции LLT и LGT. Причем если сравниваются две символьные строки разной длины, более короткая строка расширяется справа за счет добавления пробелов до длины более длинной строки.

Пример 1. Выполняется поиск в отсортированном символьном векторе строки 'cc'.

```
program search1
 use dfimsl
 integer(4), parameter :: n = 9
 integer(4) :: incx, index
 character(2) :: chx(n), string
 chx = (/ 'aa', 'bb', 'cc', 'dd', 'ee', 'ff', 'gg', 'hh', 'ii' /)
 incx = 1; string = 'cc'
 call ssrch(n, string, chx, incx, index)   ! Поиск строки string
 print '(1x, a, i3)', 'index = ', index
end program search1
```

Результат:

```
index =  3
```

Пример 2. Выполняется поиск во втором столбце, а затем в третьей строке 4×6-матрицы *A*. И столбец и строка отсортированы в неубывающем порядке.

```
program search2
 use dfimsl
 integer(4), parameter :: m = 4, n = 6
 integer(4) :: index1, index2
 real(4) :: a(m, n)                          ! m - ведущее измерение матрицы A
 real(4) :: val1 = 5.0, val2 = 16.0          ! Числа, которые надо найти
 a = reshape((/      5.0,  -4.0,      -  1.0,   1.0,  -2.0,          &
                                  10.0,
                     1.0,   1.0,  -4.0,  6.0,  -4.0,  -3.0,         &
                     1.0,   5.0,  10.0, 14.0,  16.0,  25.0,         &
                    -2.0,   8.0,   1.0, -3.0,  -4.0,   5.0 /),      &
                   shape = (/ m, n /), order = (/ 2, 1 /))
 call srch(m, val1, a(1, 2), 1, index1)    ! Поиск во втором столбце матрицы A
 call srch(n, val2, a(3, 1), m, index2)    ! Поиск в третьей строке матрицы A
 print '(1x, 2(a, i3))', 'index1 = ', index1, ';  index2 = ', index2
end program search2
```

Результат:

```
index1 =   3;  index2 =   5
```

Замечание. Поиск в строке и столбце матрицы можно выполнить, передавая в *srch* соответствующие ее сечения. В обоих случаях *incx* = 1. Например:

```
call srch(m, val1, a(:, 2), 1, index1)    ! Поиск во втором столбце матрицы A
call srch(n, val2, a(3, :), 1, index2)    ! Поиск в третьей строке матрицы A
```

5.4.2. ОСОБЕННОСТИ ПРИМЕНЕНИЯ ПРОЦЕДУР ПОИСКА БИБЛИОТЕК IMSL И DFLIB

Поиск в отсортированном в неубывающем порядке векторе (числовом или символьном) можно также выполнить функцией BSEARCHQQ библиотеки DFLIB (MSFLIB):

index = BSEARCHQQ(*adrkey*, *adrarray*, *Length*, *size*)

Параметры функции BSEARCHQQ:

Все параметры являются *входными* и имеют тип INTEGER(4).

adrkey - адрес переменной, поиск которой выполняется.

adrarray - адрес вектора, в котором выполняется поиск.

Length - число элементов вектора, в котором выполняется поиск.

Смысл параметра *size* пояснен при рассмотрении подпрограммы SORTQQ (разд. 5.2.3).

Функция BSEARCHQQ вернет индекс искомого элемента или нуль, если элемент не найден. Элементы вектора не могут быть производного типа.

Предупреждение. Адреса сортируемого массива и элемента должны быть вычислены функцией LOC. Значения параметров *count* и *size* должны точно описывать характеристики массива. К тому же искомый элемент должен иметь те же тип и разновидность типа, что и массив, в котором выполняется поиск. Эти характеристики задаются параметром *size*. Если же функция BSEARCHQQ получила неверные параметры, то, если память принадлежит текущему процессу, будет выполнена попытка поиска в некоторой области памяти, иначе операционная система выполнит функции защиты памяти и остановит программу.

Пример. Выполняется поиск элемента *ival* = 3 в векторе *ix*.

```
program search3
 use dflib                              ! В этой программе бинарный поиск возвращает
 use dfimsl                             ! число 5 вместо правильного значения 2
 integer(4) :: ix(10), ival = 3, index, index2
 ix(1) = 1; ix(2:) = 3                  ! Вектор ix:  1  3  3  3  3  3  3  3  3  3
 ! Поиск функцией BSEARCHQQ из библиотеки DFLIB
 index = bsearchqq(loc(ival), loc(ix), 10, srt$integer4)
 ! Поиск подпрограммой ISRCH из библиотеки IMSL
 call isrch(10, ival, ix, 1, index2)
 ! Далее должен следовать приводимый ниже код, уточняющий значение index
 print '(1x, 2(a, i3))', 'index = ', index, ';  index2 = ', index2
end program search3
```

Результат:

```
index =   5;  index2 =   5
```

Из результата видно, что и функция BSEARCHQQ библиотеки DFLIB (MSFLIB), и подпрограмма ISRCH библиотеки IMSL находят не первый, равный трем, элемент с *index* = 2, а элемент с тем же значением, но с *index* = 5 (то же справедливо и для иных подпрограмм поиска данных библиотеки IMSL). Во многих приложениях такой результат неудовлетворителен. Например, в задаче "Найти в векторе все равные заданному значению элементы".

Чтобы устранить этот недостаток, дополним программу *search3* кодом, который позволяет найти первый равный заданному значению элемент:

```
if(index > 1) then
  it = index
  do index = it, 1, -1
    if(ix(index) /= ival) exit
```

```
 end do
 index = index + 1                    ! Вернет index = 2
end if
```

5.5. ПЕРЕСТАНОВКИ В МАССИВАХ

5.5.1. ПЕРЕСТАНОВКИ СТРОК И СТОЛБЦОВ МАТРИЦЫ

Выполняются подпрограммой PERMA (DPERMA):

CALL PERMA(*nra*, *nca*, *a*, *Lda*, *ipermu*, *ipath*, *aper*, *Ldaper*)

Параметры подпрограммы PERMA:
Входные: nra, *nca*, *a*, *Lda*, *ipermu*, *ipath*, *Ldaper*.
Выходной: aper.

Параметры *a* и *aper* имеют тип REAL(4). Тип остальных параметров - INTEGER(4).

nra и *nca* - соответственно число строк и число столбцов в матрицах *A* и *APER*.

a - массив, представляющий *nra*×*nca*-матрицу *A*, в которой выполняются перестановки.

Lda - ведущий размер массива *a*, как правило *Lda* = *nra*.

ipermu - вектор размера *k*, содержащий указания о перестановках. Значениями *ipermu*(1), ..., *ipermu*(*k*) являются целые числа 1, 2, ..., *k*, где *k* = *nra*, если переставляются строки, и *k* = *nca*, если - столбцы.

ipath - флаг, указывающий на характер перестановок; *ipath* = 1, если переставляются строки, и *ipath* = 2, если - столбцы.

aper - массив, представляющий *nra*×*nca*-матрицу *APER*, содержащий результат перестановок. Если *a* после перестановок не нужен, то на месте *aper* можно разместить *a*.

Ldaper - ведущий размер массива *aper*, как правило *Ldaper* = *nra*.

Описание:
Пусть *p* = *ipermu*(*i*), тогда после перестановок *aper*(*i*, *j*) = *a*(*p*, *j*) для всех *i*, *j*.

Пример. Переставляются столбцы матрицы *A*.

```
program perm1
 use dfimsl
 integer(4), parameter :: nra = 2, nca = 6, Lda = nra, ipath = 2
 integer(4) :: ipermu(nca)
 real(4) :: a(Lda, nca)
 a = reshape((/    1.0,  2.0,  3.0,  4.0,  5.0,  6.0,                &
                   1.0,  2.0,  3.0,  4.0,  5.0,   6.0                &
                      /), shape = (/ Lda, nca /), order = (/ 2, 1 /))
```

```
ipermu = (/ 3, 4, 6, 5, 2, 1 /)          ! Вектор перестановок
! Перестановка столбцов матрицы A; оставляем результат в массиве a
call perma(nra, nca, a, Lda, ipermu, ipath, a, Lda)
call wrrrn('a', nra, nca, a, Lda, 0)
end program perm1
```

Результат

	a					
	1	2	3	4	5	6
1	3.000	4.000	6.000	5.000	2.000	1.000
3	3.000	4.000	6.000	5.000	2.000	1.000

5.5.2. ПЕРЕСТАНОВКИ В ВЕКТОРЕ

Выполняются подпрограммой PERMU (DPERMU):

CALL PERMU(*n*, *x*, *ipermu*, *ipath*, *xpermu*)

Параметры подпрограммы PERMU:

Входные: n, x, ipermu, ipath.

Входной: xpermu.

Тип *x* и *xpermu* - REAL(4). Тип остальных параметров - INTEGER(4).

n - размер векторов *x*, *xpermu* и *ipermu*.

x - вектор размера *n*, элементы которого подлежат перестановке.

ipermu - вектор размера *n*, содержащий указания о предстоящих перестановках. Значениями *ipermu*(1), ..., *ipermu*(*n*) являются числа 1, 2, ..., *n*.

ipath - флаг, указывающий на характер перестановок; *ipath* = 1, если выполняются прямые перестановки, т. е. *x*(*ipermu*(*i*)) перемещается в *xpermu*(*i*); *ipath* = 2, если выполняются обратные перестановки, т. е. *x*(*i*) перемещается в *xpermu*(*ipermu*(*i*)).

xpermu - вектор размера *n*, содержащий переставленные элементы вектора *x*. Если *x* после перестановок не нужен, то на месте *xpermu* можно разместить *x*.

Замечание. Вектор перестановок *ipermu* может быть получен, например, из подпрограммы сортировки данных SVRBP или SVRGP.

Пример. Выполняются прямые перестановки элементов вектора *x*.

```
program perm2
 use dfimsl
 integer(4), parameter :: n = 6, ipath = 1
 integer(4) :: ipermu(n)
 real(4) :: x(n), xpermu(n)
 x = (/ 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 /)
 ipermu = (/3, 4, 6, 5, 2, 1 /)          ! Вектор перестановок
```

```
! Перестановка элементов вектора x; результат размещаем в xpermu
 call permu(n, x, ipermu, ipath, xpermu)
 call wrrrn('xpermu', 1, n, xpermu, 1, 0)
end program perm2
```

```
                    xpermu
     1           2           3           4           5           6
 3.000       4.000       6.000       5.000       2.000       1.000
```

Приложение 1. ОТОБРАЖАТЕЛЬ МАССИВОВ

П.-1.1. НАЗНАЧЕНИЕ ОТОБРАЖАТЕЛЯ МАССИВОВ

Отображатель массивов (ОМ) фирмы Compaq® позволяет наблюдать как данные числовых массивов, так и их графическое представление. Отображатель содержит в качестве ядра графическую библиотеку OpenGL®, процедуры которой обеспечивают графический вывод. (Об употреблении OpenGL® в Фортране см., например, [3].) Дополнительно ОМ помогает манипулировать графическими данными, предоставляя возможности для перемещения, поворота и масштабирования изображения, а также для изменения способа его представления на экране. ОМ содержит:

- автономно запускаемое приложение, выполняющее отображение массивов;
- библиотеку Aview процедур, вызываемых из приложений Фортрана и предназначенных для управления ОМ;
- ActiveX®-процедуры библиотек Avis2D и AvisGrid;
- дополнительные визуальные средства.

Массив, переданный ОМ, отображается в двух видах:

1) в виде числовой таблицы, выводимой в верхней части окна ОМ;
2) в графическом виде как трехмерное изображение (3D-вид), или как цветовая карта, или как векторный граф, или как рисунок на плоскости.

Процедуры библиотеки Aview позволяют приложениям CVF или Visual C++ отображать (посредством OLE-автоматизации) данные массива, применяя ОМ. Также данные массива можно сохранить в виде файла, который загружается в ОМ в процессе его автономного использования.

ActiveX®-процедуры (OCX) библиотек Avis2D и AvisGrid могут быть использованы любой поддерживающей автоматизацию средой, например Visual C++, Visual Basic® или CVF, для отображения массивов в разнообразных графических видах. Процедуры Avis2D обеспечивают при выполнении графического вывода более 100 свойств, методов и событий; процедуры AvisGrid применяются для создания представляющих массивы таблиц и предоставляют около 30 свойств, методов и событий.

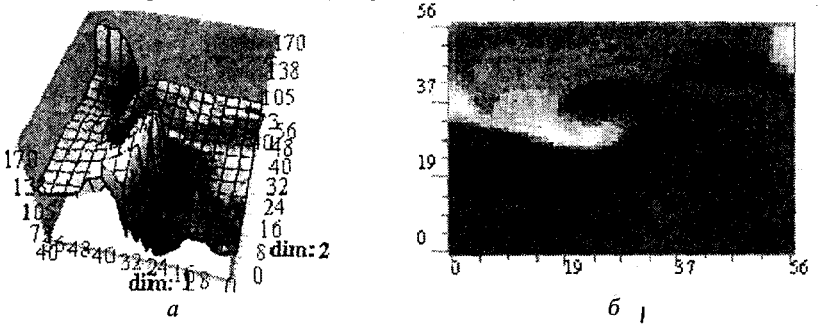
Возможны несколько вариантов употребления ОМ. Они, а также присущие им преимущества и недостатки перечислены в табл. П.-1.1.

Таблица П.-1.1. Варианты применения ОМ

Вариант	*Преимущества*	*Недостатки*
Загрузка agl-файла, созданного ранее выполненным приложением	Не требует написания специального кода для вызова ОМ	Нет возможности автоматизировать изменение отображаемых данных
Использование отладчика CVF (порядок употребления см. в главе документации, посвященной отладчику)	Не требует написания специального кода; работает с проектом любого типа	Требует ручного задания свойств массива и настройки ОМ; не может быть использован в Visual C++ или Visual Basic, а также в Release-режиме CVF
Использование fagl-подпрограмм или в случае СИ - agl-функций	Небольшое число процедур и, следовательно, небольшие затраты на программирование; процедуры работают с проектами любого типа и в Debug-, и в Release-режиме	Требует ручного задания свойств массива и настройки ОМ
Использование fagl- и fav-подпрограмм или в случае СИ++ - agl-функций и функций класса CAViewer	Можно программно задавать свойства массива и выполнять настройки ОМ; процедуры работают с проектами любого типа; последовательно в одном экземпляре ОМ можно отображать несколько массивов	Потребуется освоить большое число процедур (более 100); функции класса CAViewer нельзя применять в СИ (необходим СИ++)
Использование ActiveX-процедур библиотек Avis2D и/или AvisGrid	Дает возможность выводить создаваемые ОМ графические образы и таблицы данных без вызова ОМ; обеспечивает более быстрое воспроизведение образов и больше возможностей для настройки параметров	Употребляется только в Windows-приложениях Фортрана или MFC в случае Visual C++. Заметим, что в Visual Basic большинство EXE-проектов могут использовать процедуры библиотек Avis2D и AvisGrid; потребуется освоить большое число Avis2D/AvisGrid-процедур; Avis2D и AvisGrid процедуры не могут отображать HDF и текстовые файлы

П.-1.2. ОТОБРАЖЕНИЕ МАССИВОВ

Массивы отображаются на двумерной сетке (рис. П.-1.1, *а*).



а *б*

Рис. П.-1.1. Отображение массива:
а - двумерная сетка ОМ; б - растровая карта

При этом ОМ предоставляет следующие видовые режимы:

- 3D-вид, или Height Plot (приведен на рис. П.-1.1, *а*);
- растровая карта, или Image Map (построенному на рис. П.-1.1, а изображению соответствует приведенная на рис. П.-1.1, *б* карта);
- векторный граф, или Vector Graph (рассмотрен ниже);
- с, или Plane View (рассмотрен ниже).

Перечисленные режимы могут быть заданы как в ОМ непосредственно, так и в программе, из которой ОМ запускается. В ОМ переключение режима выполняется либо из меню (рис. П.-1.2, *а*), либо в результате выбора соответствующей иконки (рис. П.-1.2, *б*).

а *б*

Рис. П.-1.2. Выбор видового режима: а - подпункты меню View;
б - соответствующие им иконки

Рассмотрим пример употребления ОМ для вывода 3D-фигуры.

Пусть необходимо вывести параболоид, задаваемый уравнением

$$z = x^2 + y^2.$$

Создадим для этого двумерный массив *bo*, применив код

```
subroutine frame(bo, m, n)      ! Подпрограмма формирует массив bo,
```

```
 implicit none                    ! содержащий точки параболоида
 integer(4) :: m, n
 integer(4) :: i, j
 real(4) :: bo(m, n), x, y
 do j = 1, n                      ! Формирование массива bo
  do i = 1, m
  ! x- и y-координаты точки параболоида
  x = float(i - m /2)
  y = float(j - n /2)
   bo(i, j) = x * x + y * y       ! bo(i,j) содержит z-координату точки параболоида
  end do
 end do
end subroutine frame
```

Отображение массива обеспечит программа

```
program para
 use avdef
 use avviewer
 use dflib
 implicit none                    ! Данные о параболоиде z = x^2 + y^2
! Протяженности массива bo по первому и второму измерениям
 integer(4) :: m = 20, n = 20
 real(4), allocatable :: bo(:, :) ! Массив точек параболоида
 !dec$attributes array_visualizer :: bo
 integer(4) hv, status
 character(1) :: key
 allocate (bo(m, n))
 call frame(bo, m, n)             ! Формирование параболоида
 call faglStartWatch(bo, status)  ! Сообщаем ОМ имя отображаемого массива
 print *, "Starting Array Viewer"
 call favStartViewer(hv, status)  ! Создаем экземпляр ОМ
 call favSetArray(hv, bo, status) ! Отображаем массив bo
 ! Задаем имя, выводимое ОМ при отображении массива bo
 call favSetArrayName(hv, "Paraboloid", status)
 ! Показываем ОМ на экране
 call favShowWindow(hv, AV_TRUE, status)
 print *, "Press any key to close down the viewer"
 key = getcharqq( )
 call favEndViewer(hv, status)    ! Закрываем ОМ
 call faglEndWatch(bo, status)    ! Удаляем массив из списка наблюдения
 deallocate(bo)
end program para                  ! Результат приведен на рис. П.-1.3
```

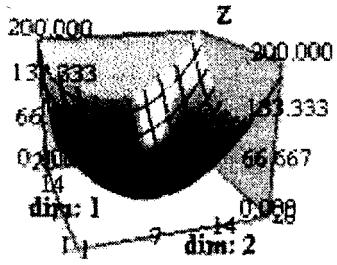
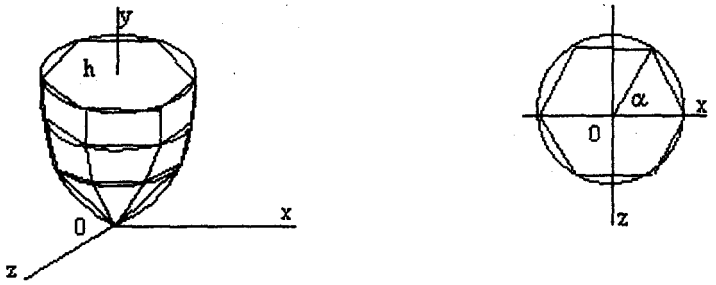


Рис. П.-1.3. Параболоид подпрограммы frame

Сформированный подпрограммой *frame* массив содержит фрагмент параболоида, пересекаемого вертикальными плоскостями. Чтобы получить более целостную картину, сформируем параболоид, обратившись к подпрограмме *frame2*, которая возвращает приведенную на рис. П.-1.4 каркасную модель параболоида.



а *б*

Рис. П.-1.4. Представление параболоида в виде граней: а - каркасная модель; в - угол разбиения

```
subroutine frame2(bo, nd, ch)          ! Подпрограмма формирует массив bo,
 implicit none                         ! содержащий точки параболоида
 integer(4) :: nd, ch
 integer(4) :: i, j, k
 real(4) :: bo(4, nd * ch)
 real(4) :: al, dal, dh, hp
 real(4) :: hcover = 20.0              ! Высота параболоида
 ! Формируем массив bo, содержащий точки параболоида
 dh = hcover / real(ch - 1)            ! Расстояние между сечениями
 ! dal - приращение угла разбиения al, используемое при переходе
 ! от одной точки разбиения сечения к другой (см. рис. П.-1.4, б)
 dal = 4.0 * asin(1.0) / real(nd)      ! dal = 2π/n
 hp = 0.0                              ! Низ параболоида в начале координат
 k = 0
```

```
 do i = 1, ch                          ! Формирование массива bo
  al = 0.0                             ! Начальный угол разбиения
  do j = 1, nd
   k = k + 1
   ! Используем параметрическое уравнение окружности
   ! x- и y-координаты точки параболоида
   bo(1:2, k) = sqrt(hp) * (/ cos(al), -sin(al) /)
   bo(3, k) = hp                       ! z-координата
   al = al + dal
  end do
  hp = hp + dh
 end do
  bo(4, :) = 0.0                       ! Цвет вывода - самый темный
end subroutine frame2
```

Вызов *frame*2 выполним в программе *para*, заменив им вызов подпрограммы *frame*:

```
integer(4) :: nd = 24                  ! Число разбиений одного сечения параболоида
integer(4) :: ch = 7                   ! Число пересекающих параболоид плоскостей
. . .
allocate(bo(4, nd * ch))
call frame2(bo, nd, ch)
```

Переключившись после отображения массива в режим векторного графа и погасив вывод осей, получим приведенное на рис. П.-1.5 изображение.

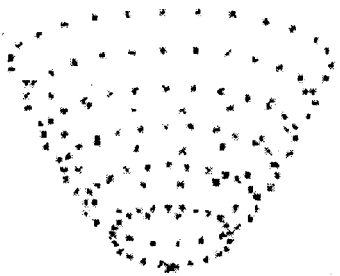
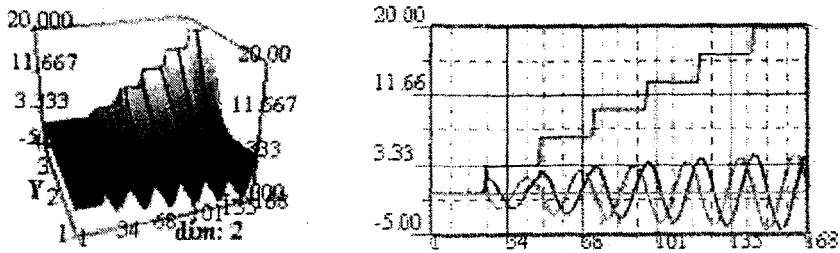


Рис. П.-1.5. Параболоид из frame2 в режиме векторного графа

Фигура, нарисованная в режиме векторного графа, похожа на параболоид, но представляется только в виде точек. В 3D-режиме ОМ образ массива *bo* представляется рис. П.-1.6, *а*, плоский вид - рис. П.-1.6, *б*.

Иными словами, на этих рисунках мы наблюдаем закономерности изменения координат принадлежащих параболоиду точек, размещенных в массиве *bo* подпрограммой *frame*2.



а *б*

Рис. П.-1.6. Представление массива bo: а - 3D-режим; б - плоский вид

Чтобы получить тонированное 3D-изображение параболоида, подобное приведенному на рис. П.-1.5, воспользуемся подпрограммой *frame*3, в которой точки рассчитываются таким же образом, как и в подпрограмме *frame*2, и размещаются в массиве *botemp*, а затем для каждой точки параболоида отыскиваются соответствующие ей индексы в массиве *bo* формы (*m*, *n*). Эти индексы вычисляются из предположения, что нижняя точка параболоида имеет индексы *m*/2 и *n*/2.

```
subroutine frame3(bo, m, n, nd, ch)     ! Подпрограмма формирует массив bo,
 implicit none                          ! содержащий точки параболоида
 integer(4) :: m, n, nd, ch
 integer(4) :: i, j, k, ibo, jbo
 real(4) :: bo(m, n), botemp(3, nd * ch)
 ! Максимальные значения координат параболоида
 real(4):: xma, yma
 real(4):: x, y, z, bomax
 real(4) :: al, dal, dh, hp
 real(4) :: hcover = 20.0               ! Высота параболоида
 ! Формируем массив botemp, содержащий координаты точек параболоида
 dh = hcover / real(ch - 1)             ! Расстояние между сечениями
 ! dal - приращение угла разбиения al, используемое при переходе
 ! от одной точки разбиения сечения к другой (см. рис. П.-1.4, б)
 dal = 4.0 * asin(1.0) / real(nd)       ! dal = 2π/n
 hp = 1.0e-7 * hcover                   ! Низ параболоида в начале координат
 k = 0
 do i = 1, ch                           ! Формирование массива botemp
  al = 0.0                              ! Начальный угол разбиения
  do j = 1, nd
   k = k + 1
   ! Используем параметрическое уравнение окружности
   ! x- и y-координаты точки параболоида
   botemp(1:2, k) = sqrt(hp) * (/ cos(al), -sin(al) /)
```

```
  botemp(3, k) = hp                    ! z-координата
  al = al + dal
 end do
 hp = hp + dh
end do
bo = -1.0                              ! Инициализация массива bo
xma = maxval(botemp(1, :)); yma = maxval(botemp(2, :))
do i = 1, k                            ! Формируем теперь массив bo
 x = botemp(1, i); y = botemp(2, i)
 ibo = m / 2 * (1.0 + x / xma); jbo = n / 2 * (1.0 + y / yma)
 ibo = min(m, ibo); ibo = max(1, ibo)
 jbo = min(n, jbo); jbo = max(1, jbo)
 z = botemp(3, i)
 bo(ibo, jbo) = z
end do
! Не все элементы массива bo могут быть определены в предшествующем цикле
! Заменим каждый отрицательный элемент массива bo
! его наибольшим элементом
bomax = maxval(bo)
where(bo < 0.0) bo = bomax
end subroutine frame3
```

Вызов *frame*3 выполним в программе *para*, заменив им вызов подпрограммы *frame*:

```
integer(4) :: nd = 100                 ! Число разбиений одного сечения параболоида
integer(4) :: ch = 50                  ! Число пересекающих параболоид плоскостей
integer(4) :: m = 20, n = 20           ! Протяженность массива bo
...
allocate(bo(m, n))
call frame3(bo, m, n, nd, ch)          ! Результат приведен на рис. П.-1.7
```

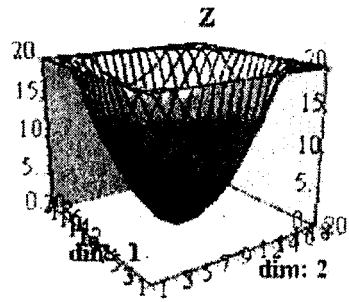


Рис. П.-1.7. 3D-вид созданного в frame3 параболоида

Для всех видов графиков, кроме векторного, строка массива ассоциируется с осью *x*, столбец - с осью *y*. Значение соответствующего элемента массива - с осью *z*.

П.-1.3. УПРАВЛЕНИЕ ИЗОБРАЖЕНИЕМ

Характер воспроизведения изображения можно изменять, работая непосредственно в ОМ. Для этой цели существуют команды, доступ к которым осуществляется через меню (рис. П.-1.8).

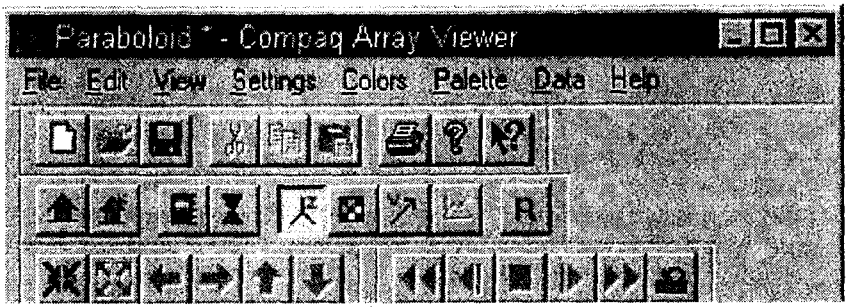


Рис. П.-1.8. Меню ОМ

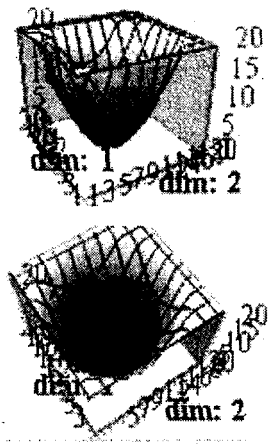
Помимо реализуемых через меню возможностей в ОМ можно, применяя мышь, вращать изображение и выделять требуемый диапазон данных в содержащей их таблице (рис. П.-1.9).

dim 2

dim 1	1	2	3	4
1	19.999996	19.999996	19.999996	19.999996
2	19.999996	19.999996	19.999996	19.183670
3	19.999996	19.999996	18.367344	16.326529
4	19.999996	19.183670	16.326529	13.469386
5	19.999996	17.142855	14.285712	11.836734

Рис. П.-1.9. Таблица данных ОМ

Для вращения изображения достаточно разместить мышь на поле графического вывода, нажать правую кнопку мыши и затем, оставаясь на поле вывода, перемещать мышь в произвольном направлении (рис. П.-1.10).



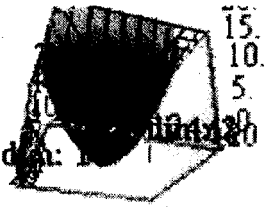
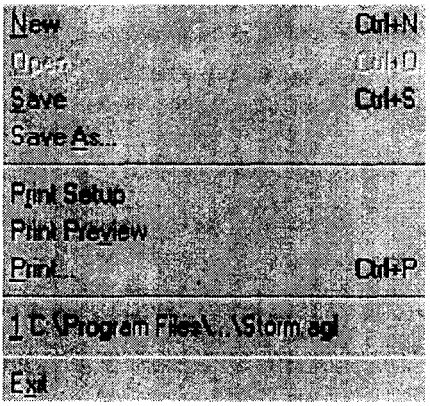


Рис. П.-1.10. Три проекции параболоида

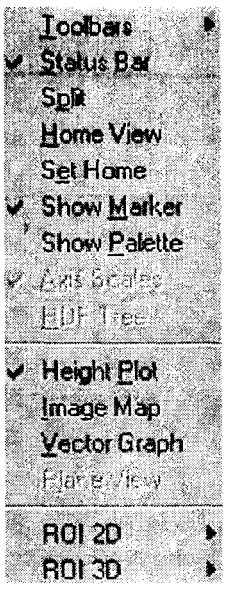
Многие предоставляемые ОМ возможности, доступ к которым осуществляется через пункты меню File, Edit или Help (рис. П.-1.11), традиционны и не требуют специальных пояснений.



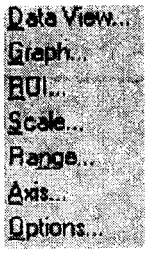
а *б* *в*

Рис. П.-1.11. Пункты меню ОМ: а - File; б - Edit; в - Help

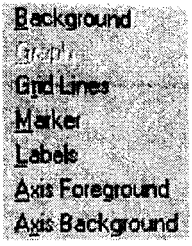
Доступ к иным операциям ОМ обеспечивают пункты меню View, Settings, Colors, Palette и Data (рис. П.-1.12).



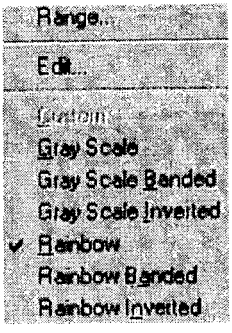
а



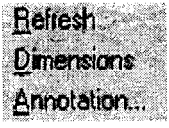
б



в



г



д

Рис. П.-1.12. Пункты меню ОМ: а - View; б - Settings; в - Colors; г - Palette; д - Data

П.-1.3.1. КОМАНДЫ МЕНЮ VIEW И PALETTE

Выбор подпункта Home View вернет повернутое изображение либо в заданное по умолчанию положение, либо в положение установленное в результате действия подпункта Set Home.

Те же результаты получатся при работе с иконками  и  .

Установить соответствие между точкой на поверхности и ячейкой таблицы данных позволяет маркер, который появится на изображении после выбора Show Marker. Когда маркер присутствует на рисунке, то каждый двойной удар мыши по ячейке таблицы вызовет его перемещение в соответствующую точку изображения и, наоборот, каждый двойной удар мыши по точке на фигуре вызовет, во-первых, перемещение в эту точку маркера и, во-вторых, выделение соответствующей ячейки данных (рис. П.-1.13).

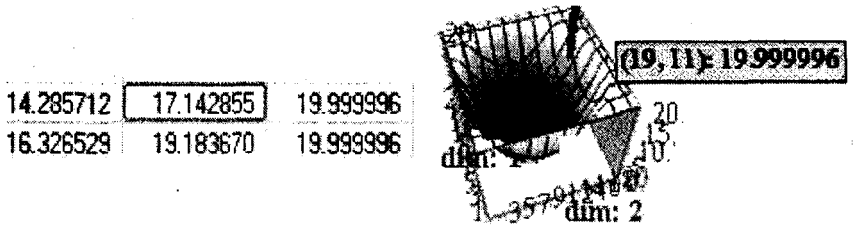


Рис. П.-1.13. Маркер и таблица данных

Замечания:

1. Позиционирование мыши на точке поверхности приводит к появлению на рисунке следующей информации: координаты соответствующей ячейки в таблице данных и *z*-координаты точки поверхности (рис. П.-1.13). Появившиеся данные не связаны с отображаемым маркером.
2. Для работы с маркером в ОМ предусмотрена иконка  .

Выбор Show Palette приведет к появлению столбца, отображающего используемую палитру цветов, с указанием соответствия оттенок - *z*-координата объекта (рис. П.-1.14).

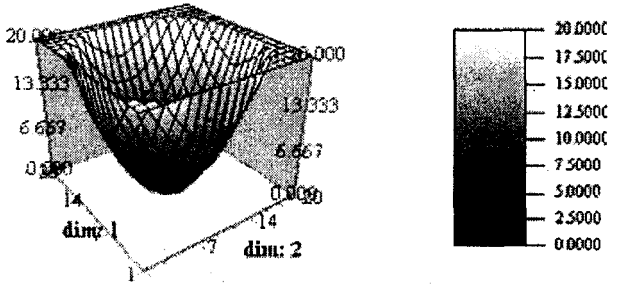


Рис. П.-1.14. Фигура и палитра

Вывод и удаление столбика с палитрой выполняется кнопкой  .

Вид палитры выбирается в меню Palette из списка:

- Gray Scale (шкала оттенков серого цвета, в которой минимальному значению *z*-координаты соответствует черный цвет, а максимальному - белый);
- Gray Scale Banded (шкала оттенков серого цвета с границами);
- Gray Scale Inverted (инвертированная шкала оттенков серого цвета);
- Rainbow (шкала цветов радуги);

- Rainbow Banded (шкала цветов радуги с границами);
- Rainbow Inverted (инвертированная шкала цветов радуги.

По умолчанию используется система цветов RGB, в которой результирующий цвет определяется как смесь красного, зеленого и синего компонентов. После выбора в меню Palette пункта Edit появится окно (рис. П.-1.15), в котором можно перейти к системе цветов HSV (оттенок - насыщение - величина) и выполнить также иные операции по настройке палитры.

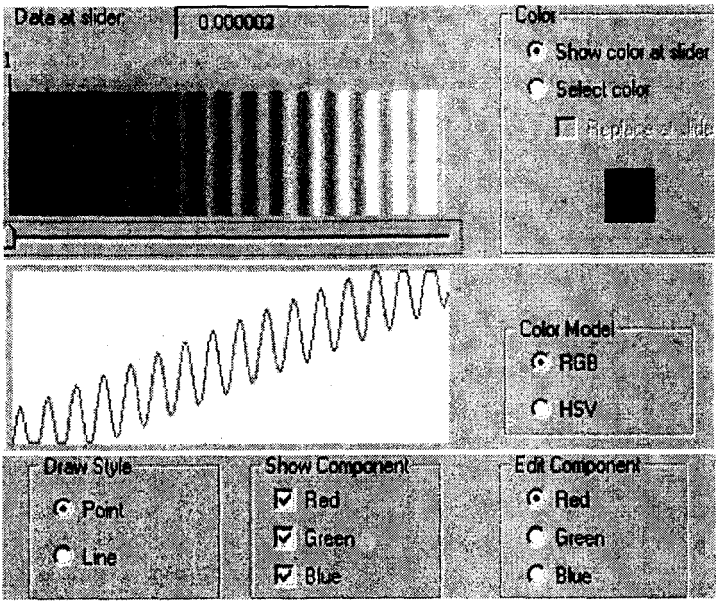


Рис. П.-1.15. Окно настройки палитры цветов

П.-1.3.2. ЗАДАНИЕ ЗОНЫ ВЫВОДА

Зона вывода (ЗВ) - это отображаемый диапазон таблицы данных. По умолчанию отображается вся таблица (т. е. весь массив). Так, в рассматриваемом параболоиде протяженность каждого измерения равна 20, а нижняя граница - единице. ЗВ, однако, можно изменить, указав нижние границы и протяженность каждого измерения массива. Установим, к примеру, нижнюю границу каждого измерения равной двум, а протяженность равной пяти. Используем для этого приведенный на рис. П.-1.16 диалог, запускаемый в результате выполнения цепочки Settings - ROI.

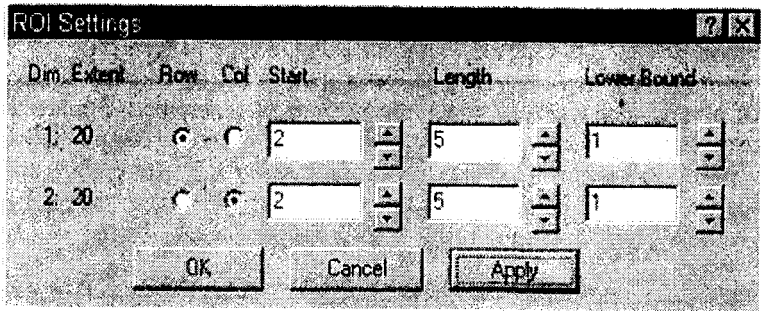
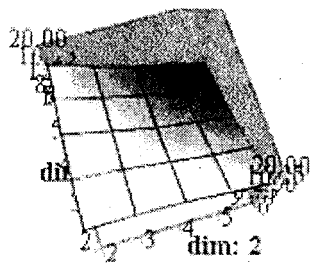


Рис. П.-1.16. Задание 3В

Нажатие OK приведет к изменению таблицы данных (рис. П.-1.17, *а*) и изображения (рис. П.-1.17, *б*).

	2	3	4	5	6	7
0%	19.999996	19.999996	19.999996	19.999996	19.999996	19.9999
0%	19.999996	19.999996	19.183670	17.142855	15.510201	13.9775
0%	19.999996	18.367344	16.326529	14.285712	12.653060	11.0204
0%	19.183670	16.326529	13.469386	11.836734	9.795918	8.5714
0%	17.142855	14.285712	11.836734	9.387755	7.755103	6.1224
0%	15.510201	12.653060	9.795918	7.755103	5.714287	4.4897
0%	13.877549	11.020408	8.571429	6.122450	4.489793	3.2653
0%	13.061223	10.204082	7.755103	5.714287	3.673472	2.4489

а



б

Рис. П.-1.17. Таблица данных и изображение после изменения 3В

Далее, используя приведенный на рис. П.-1.18 навигатор, можно либо перемещаться по изображению, сохраняя заданный размер 3В, либо выполнять ее увеличение или уменьшение.

а *б*

Рис. П.-1.18. Навигатор: а - просмотр изображения; б - изменение ЗВ

Кроме того, приведенное на рис. П.-1.16 окно позволяет менять, используя радиокнопки Row и Col, порядок строк и столбцов (в массивах ранга 2 в случае Фортрана строки задаются первым измерением, а столбцы - вторым; в массивах СИ и Бейсика порядок задание строки и столбцов обратный) и задавать величину нижней границы (поле Lower Bound), устанавливая ее, например, равной значению, заданному в программе. По умолчанию нижняя граница каждого измерения равна единице.

Заметим, что в заданной ЗВ ограничить область вывода можно, выделив в таблице прямоугольник данных и нажав на левую кнопку из рис. П.-1.18, *б*.

П.-1.3.3. РЕДАКТИРОВАНИЕ ТАБЛИЦЫ ДАННЫХ

Становится возможным после активизации пункта Cell Edit Enabled в приведенном на рис. П.-1.19 окне и нажатия кнопки OK.

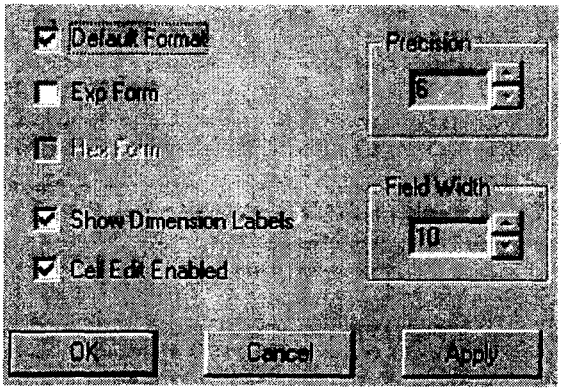


Рис. П.-1.19. Способ вывода числовых данных

Окно появляется после выбора Settings - Data View. Собственно редактирование данных становится возможным после двойного щелчка мышью по клетке таблицы данных. Ввод новых данных и нажатие на Enter вызовет перестройку изображения.

Другие возможности окна следуют из имеющихся на нем надписей.

П.-1.3.4. СПОСОБ ВЫВОДА ИЗОБРАЖЕНИЯ

Задается в окне Settings - Graph (рис. П.-1.20).

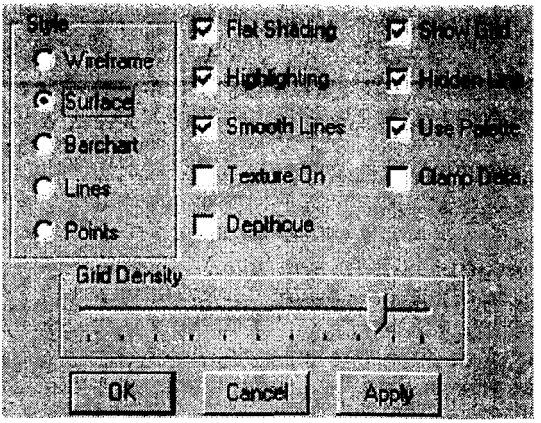
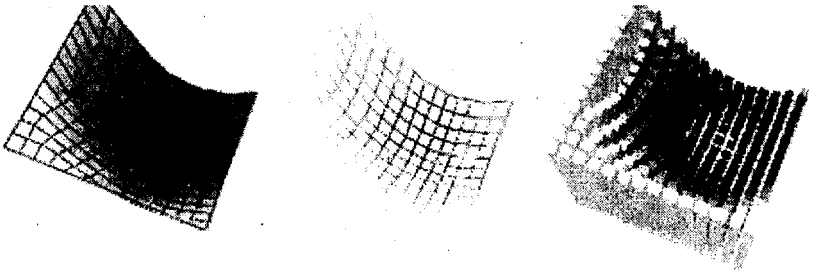


Рис. П.-1.20. Способ вывода изображения

Установленная по умолчанию конфигурация (приведена на рис. П.-1.20) предусматривает в режиме 3D-вида вывод поверхности (Surface):

- с плавным переходом (интерполяцией) цветов (Float Shading);
- с бликами (Highlighting), проявляющимися при вращении объекта;
- со сглаженными линиями (Smooth Lines).

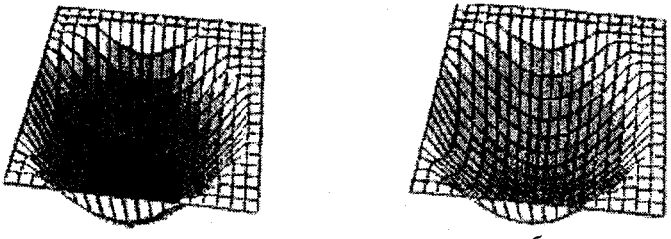
При этом показывается сетка (Show Grid), удаляются невидимые линии (Hidden Line) и используется цветовая палитра (Use Palette). Манипуляция имеющимися на окне кнопками приведет к изменению способа вывода изображения. Так, выбор Wireframe вызовет переход к каркасной модели, а Barchart - к диаграмме (рис. П.-1.21).



а *б* *в*

Рис. П.-1.21. Часть параболоида: а - тонированная поверхность; б - каркасная модель; в - диаграмма

Активизация Texture On вызовет наложение на объект текстуры, а - Depthcue приведет к изменению способа достижения эффекта глубины - снижается контрастность более глубоких участков фигуры (рис. П.-1.22).



а *б*

Рис. П.-1.22. Эффект глубины: а - режим Depthcue отключен; б - режим Depthcue включен

Перейдем теперь к описанию поставляемых с ОМ процедур, встраивание которых в приложение позволяет запускать ОМ, передавать ему массив, управлять изображением, выполняя при необходимости пересчет отображаемых данных и обновление отвечающей им фигуры.

П.-1.4. FAGL-ПОДПРОГРАММЫ

Вызов и некоторые функции управления ОМ обеспечивают приведенные в табл. П.-1.2 fagl-подпрограммы. Параметры подпрограмм содержатся в табл. П.-1.3.

Таблица П.-1.2. Fagl- подпрограммы ОМ

Синтаксис подпрограммы	*Описание*
faglClose(*array*, *status*)	Закрывает экземпляр ОМ. Если затем возникнет потребность отобразить массив *array* снова, то достаточно вызвать лишь faglShow; вызова faglStartWatch выполнять не нужно
faglEnd-Watch(*array*, *status*)	Удаляет массив *array* из списка отображаемых массивов и освобождает ресурсы, связанные с *array* и используемые подпрограммами библиотеки. Чтобы впоследствии отобразить массив, необходимо вызвать как faglStartWatch, так и faglShow
faglGetShare-Name(*array*, *file-name*, *status*)	Строка *filename*, возвращаемая faglGetShareName, может быть передана процедурам Avis2D и AvisGrid как свойство File-Name. Это позволяет отображать данные массива в приложении, использующем эти процедуры без сохранения массива в виде внешнего файла

faglHide(*array*, *status*)	Делает экземпляр ОМ невидимым. Экземпляр ОМ станет видимым, если затем вызвать faglShow. Однако если экземпляр ОМ создан посредством favStartViewer, то вместо faglShow следует употреблять favShowWindow
faglLBound(*array*, *lbnd*, *status*)	Устанавливает левые границы измерений отображаемого массива в видах Data или Graph Views ОМ. По умолчанию массив отображается с левыми границами, равными единице
faglName(*array*, *title*, *status*)	Размещает строку *title* на полосе заголовка экземпляра ОМ. Если, однако, экземпляр ОМ создан посредством favStartViewer, то вместо faglName следует употреблять favSetArrayName
faglSaveAs-File(*array*, *filename*, *status*)	Сохраняет текущий массив в файле с расширением AGL. Такой файл может быть загружен и отображен в ОМ
faglShow(*array*, *status*)	Создает экземпляр ОМ и отображает данные массива *array*. Также faglShow делает видимым экземпляр ОМ, скрытый командой faglHide. Перед вызовом faglShow можно задать заголовок, применив faglName. Если же экземпляр ОМ создан посредством favStartViewer, то вместо faglShow следует употреблять favShowWindow
faglStartWatch (*array*, *status*)	Добавляет массив *array* в список отображаемых массивов и возвращает дескриптор *hv*, который используется для доступа к массиву другими подпрограммами библиотеки. Фактически faglStartWatch использует системные ресурсы для приведения *array* к виду, необходимому для faglShow. Чтобы освободить эти ресурсы, следует вызвать faglEndWatch
faglUpdate(*array*, *status*)	Приводит в соответствие изображение с данными, хранящимися в массиве *array*. Употребляется, если приложение изменило отображаемый массив *array* с момента последнего вызова faglUpdate или faglShow и если есть необходимость обновить изображение. Если же экземпляр ОМ ассоциируется с массивом, созданным favStartViewer, а не faglShow, то вместо faglUpdate нужно вызывать favUpdate

Таблица П.-1.3. Параметры fagl-подпрограмм

Имя	*Смысл/вид*	*Тип*
array	Имя отображаемого массива. Должно быть прежде использовано при вызове faglStartWatch / *входной*	Числовой
status	Статус вызова fagl-подпрограммы. При отсутствии ошибок вызова равен нулю / *выходной*	INTEGER(4)

filename	Строка, возвращаемая faglGetShareName. Длина строки должна равняться AV_SHARENAME_LEN / *выходной*	CHARACTER(*)
"	Имя файла без расширения, если файл пишется в директорию, из которой вызвано приложение, либо полное имя файла, включающее путь к файлу (случай faglSaveAsFile) / *входной*	"
lbnd	Массив ранга 1, размер которого равен рангу отображаемого массива / *выходной*	INTEGER(4)
title	Строка, отображаемая на полосе заголовка заданного экземпляра ОМ / *входной*	CHARACTER(*)

Для вызова приведенных в табл. П.-1.2 подпрограмм в использующем их программном компоненте следует выполнить ссылку

```
use avdef                    ! Ссылка на ...ArrayVisualizer\include\avdef.f90
```

Модуль AVDEF содержит интерфейсы fagl-подпрограмм.

Перечисленные подпрограммы обычно используются следующим образом:

- до отображения массива *array* вызовите faglStartWatch с параметром *array*;
- если необходимо отображать массив, имея нижнюю левую границу, отличную от единицы, примените faglLBound;
- для отображения сообщения, сопровождающего выводимые данные, вызовите faglName;
- для запуска ОМ и отображения массива *array* вызовите faglShow с первым параметром, равным *array*. ОМ будет функционировать до тех пор, пока не выполнена команда faglClose;
- если хранимые массивом данные подверглись изменениям, то для их отображения вызовите faglUpdate;
- для сохранения массива *array* в виде файла с расширением AGL (Array Graphing Language) вызовите faglSaveAsFile, использовав *array* в качестве первого параметра. При этом ОМ может быть неактивным;
- при необходимости можно вызвать faglGetShareName и получить строку *filename*, позволяющую процедурам Avis2D и AvisGrid осуществлять доступ к области памяти, занятую массивом;
- после завершения просмотра массива вызовите faglEndWatch.

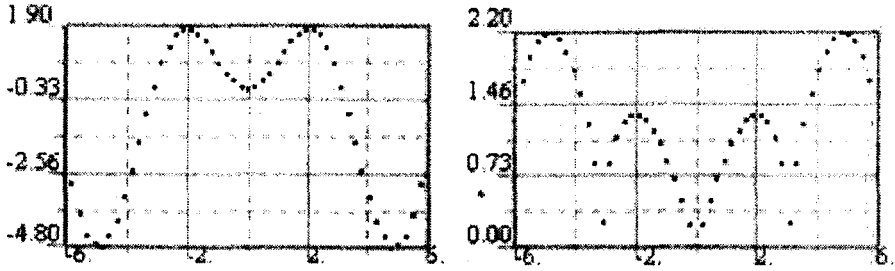
Дополнительно для выполнения действий, обычно производимых интерактивно, можно применять fav-процедуры.

Пример. Выводится в режиме векторного графа график функции $y = x\sin x$ на отрезке [-6, 6], а вслед после некоторой задержки - график

$y = \sqrt{|x \sin x|}$. Обновление изображения выполняет подпрограмма faglUpdate.

```
program sinx
 use avdef
 use avviewer
 use dflib
 implicit none
 integer(4) :: n = 50                    ! Число разбиений отрезка
 real(4), allocatable :: fun(:, :)       ! Массив значений функции y = xsinx
 !dec$attributes array_visualizer :: fun
 real(4) :: a = -6.0, b = 6.0, x, dx
 integer(4) i, hv, status, nError
 character(1) :: key
 allocate(fun(2, n))
 dx = (b - a) / float(n)                 ! Шаг изменения x
 x = a
 do i = 1, n                             ! Формирование массива данных
  fun(1, i) = x
  fun(2, i) = x * sin(x)
  x = x + dx
 end do
 call faglStartWatch(fun, status)        ! Сообщаем ОМ имя отображаемого массива
 print *, "Starting Array Viewer"
 ! Запуск ОМ с использованием fav-подпрограммы
 call favStartViewer(hv, status)
 if(status /= 0) then
  call favGetErrorNo(hv, nError, status)
  if(nError /= 0) then
   print *, "Array Viewer reports error ", nError; stop
  end if
 end if
 ! Передаем ОМ данные подлежащего отображению массива
 call favSetArray(hv, fun, status)
 ! Задаем заголовок
 call favSetArrayName(hv, "y = x * sin(x)", status)
 ! Задаем отображение массива в виде векторного графа
 call favSetGraphType(hv, VectorGraph, status)
 call favShowWindow(hv, av_true, status)    ! Показываем ОМ на экране
 call sleepqq(5000)                         ! Задержка в 5000 миллисекунд
 ! Задаем новый заголовок
 call favSetArrayName(hv, "y = sqrt(abs(x * sin(x)))", status)
 fun(2, :) = sqrt(abs(fun(2, :)))        ! Вносим изменения в массив
 call favUpdate(hv, 0, status)           ! и отображаем их на графике
```

```
print *, "Press any key to close down the viewer"
key = getcharqq( )
call favEndViewer(hv, status)          ! Закрываем ОМ
call faglEndWatch(fun, status)         ! Освобождаем ресурсы
deallocate(fun)
end program sinx                       ! Результаты приведены на рис. П.-1.23
```



а *б*

Рис. П.-1.23. Два векторных графа: а - y = xsinx; б - $y = \sqrt{|x \sin x|}$

В этом и более ранних примерах употреблен DEC-атрибут ARRAY_VISUALIZER:

```
real(4), allocatable :: MyArray(:, :)
!dec$attributes array_visualizer :: MyArray
```

Его действие таково: память, занимаемая массивом *MyArray*, используется и ОМ и приложением. При отсутствии атрибута будут созданы две области для данных *MyArray* и при каждом обновлении массива после вызова faglUpdate данные будут копироваться из области, принадлежащей приложению, в область, используемую ОМ.

Заметим, что DEC-атрибут ARRAY_VISUALIZER может быть применен только с динамическими объектами, т. е. объектами, обладающими атрибутом POINTER или ALLOCATABLE.

П.-1.5. FAV-ПОДПРОГРАММЫ

П.-1.5.1. ВВЕДЕНИЕ

Fav-подпрограммы управляют ОМ и обеспечивают взаимодействие ОМ с приложением. Как правило, fav- и fagl-подпрограммы употребляются совместно. Fav-подпрограммы подразделяются на группы, имеющие названия:

- запуск ОМ;
- доступ к данным;
- зона вывода;

- фильтрация данных;
- палитра;
- оси координат;
- выбор;
- виды изображений;
- 3D-вид;
- растровая карта;
- векторный граф;
- отображение данных;
- камера;
- маркер;
- разное.

Работа с fav-подпрограммами станет возможной после выполнения ссылки

```
use avviewer
```

в которой модуль AVVIEWER содержит интерфейсы и константы подпрограмм.

П.-1.5.2. ДЕЙСТВИЕ FAV-ПОДПРОГРАММ

Вызываемые fav-подпрограммами действия приведены в табл. П.-1.4. Описание выполнено в соответствии с перечисленными выше группами.

Таблица П.-1.4. Fav-подпрограммы

Подпрограмма	*Что выполняет*
	Запуск ОМ
favStartViewer	Запускает экземпляр ОМ
favEndViewer	Завершает работу экземпляра ОМ
	Доступ к данным
favSetFileName	Загружает и отображает в ОМ заданный файл
favSetArray	Отображает в ОМ заданный массив
	Зона вывода
favSetRoi	Устанавливает начало и конец ЗВ по заданному массиву
favGetRoiLb	Возвращает начало ЗВ по заданному массиву
favGetRoiUb	Возвращает конец ЗВ по заданному массиву
favSetRowColDim	Устанавливает способ интерпретации измерения массива – как строку или как столбец
favGetRowColDim	Устанавливает способ интерпретации измерений массива
favSetRoi2D	Устанавливает ЗВ в виде 2D-сечения массива

Фильтрация данных	
favSetDataClamp	Устанавливает, будут ли в режиме векторного графа выводиться данные, расположенные за пределами заданного диапазона
favGetDataClamp	Возвращает заданный favSetDataClamp режим
favSetXClamp	Устанавливает верхнюю и нижнюю границы *x*-координат в режиме видового графа
favGetXClamp	Возвращает верхнюю и нижнюю границы *x*-координат в режиме видового графа
favSetYClamp	Устанавливает верхнюю и нижнюю границы *y*-координат в режиме видового графа
favGetYClamp	Возвращает верхнюю и нижнюю границы *y*-координат в режиме видового графа
favSetZClamp	Устанавливает верхнюю и нижнюю границы *z*-координат в режиме видового графа
favGetZClamp	Возвращает верхнюю и нижнюю границы *z*-координат в режиме видового графа
favSetDataRefresh Enable	Делает активным/неактивным пункт меню Data-Refresh
favGetDataRe-freshEnable	Возвращает состояние пункта меню Data-Refresh
favUpdate	Вызывается, когда нужно обновить изображение, созданное ОМ, чтобы отобразить изменения, произошедшие с момента последнего обновления данных или начальной загрузки
Палитра	
favSetCustom Palette	Создает пользовательскую палитру цветов
favSetPaletteId	Задает цветовую палитру, используя определенные в модуле AVVIEWER именованные константы: • GREYSCALE = 1; • GREYSCALE_BANDED = 2; • GREYSCALE_INVERTED = 3; • RAINBOW = 4; • RAINBOW_BANDED = 5; • RAINBOW_INVERTED = 6
favGetPaletteId	Возвращает идентификатор текущей цветовой палитры
favSetPaletteRange	Задает диапазон данных, с которым будет ассоциироваться цветовая палитра
favGetPaletteRange	Возвращает диапазон данных, с которым ассоциируется цветовая палитра

favSetPaletteAuto Adjust	Включает/отключает автоматическую адаптацию диапазона палитры к диапазону отображаемых данных
favSetUseColor Palette	Включает/отключает режим использования цветовой палитры
favGetUseColor Palette	Возвращает заданный favSetUseColorPalette режим
favSetShowPalette	Включает/отключает воспроизведение рядом с фигурой цветовой палитры
favGetShowPalette	Возвращает заданный favSetShowPalette режим
Оси координат	
favSetAxisAuto Scale	Включает/отключает автоматическую разметку осей координат
favGetAxisAuto Scale	Возвращает заданную favSetAxisAutoScale установку
favSetDimScale	Ассоциирует ось координат с массивом, содержащим разметку оси
favSetShowAxis	Отображает/скрывает оси координат
favGetShowAxis	Возвращает состояние, заданное favSetShowAxis
favSetAxisLabel	Задает имя указанной оси координат. Символьные переменные, применяемые для задания имени, должны иметь длину, равную AV_MAX_LABEL_LEN
favGetAxisLabel	Возвращает имя указанной оси координат
favSetUseAxis Label	Включает/отключает вывод заданных пользователем имен осей координат
favGetUseAxis Label	Возвращает состояние, заданное favSetUseAxisLabel
favSetFontAuto Scale	Включает/отключает режим автоматического изменения размера шрифта, используемого для отображения имен осей координат, при изменении размеров окна вывода
favGetFontAuto Scale	Возвращает режим, заданный favSetFontAutoScale
favSetAxisStyle	Задает стиль вывода осей координат
favGetAxisStyle	Возвращает стиль вывода осей координат
favSetAxisAuto Detail	Устанавливает, будут ли большие и маленькие разметки на осях координат формироваться автоматически или задаваться явно в программе
favGetAxisAuto Detail	Возвращает заданный favSetAxisAutoDetail режим

favSetNumMajor Tickmarks	Устанавливает число больших разметок заданной оси координат (рядом с такими разметками проставляются соответствующие им значения)
favGetNumMajor Tickmarks	Возвращает число больших разметок заданной оси координат
favSetNumMinor Tickmarks	Устанавливает число маленьких разметок заданной оси координат
favGetNumMinor Tickmarks	Возвращает число маленьких разметок заданной оси координат
Выбор	
favSetDataSelect Enable	Включает/отключает режим выбора данных, выполняемого в результате двойного удара мыши по точке поверхности, в результате которого при включенном режиме происходит перемещение маркера
favGetDataSelect Enable	Возвращает режим, заданный favSetDataSelectEnable
Виды изображений	
favSetGraphType	Задает видовой режим (3D-вид, растровая карта, векторный граф, плоский вид)
favGetGraphType	Возвращает текущий видовой режим
favSetGraphStyle	Задает способ вывода изображения (в виде сетки, поверхности, диаграммы, линий или точек)
favGetGraphStyle	Возвращает способ вывода изображения
favSetGridDensity	Задает значение, определяющее густоту сетки - число пикселей между соседними линиями сетки
favGetGridDensity	Возвращает значение, определяющее густоту сетки
favSetDepthcue	Включает/отключает режим Depthcue
favGetDepthcue	Возвращает режим, заданный favSetDepthcue
favSetShowGrid	Включает/отключает отображение сетки на рисунке
favGetShowGrid	Возвращает режим, заданный favSetShowGrid
favSetLineSmooth	Включает/отключает сглаживание линий
favGetLineSmooth	Возвращает режим, установленный favSetLineSmooth
3D-вид	
favSetZScale	Задает высоту *z*-оси координат относительно размеров *x*- и *y*-осей
favGetZScale	Возвращает значение, установленное favSetZScale
favSetTextureMode	Включает/отключает режим наложения одномерной текстуры
favGetTextureMode	Возвращает режим, заданный favSetTextureMode

favSetShading	Включает/отключает интерполяцию цветов
favGetShading	Возвращает заданный favSetShading режим
favSetHighLight	Включает/отключает блики
favGetHighLight	Возвращает заданный favSetHighLight режим
favSetHiddenLine	Включает/отключает отображение невидимых линий
favGetHiddenLine	Возвращает заданный favSetHiddenLine режим
Растровая карта	
favSetImage Orientation	Устанавливает ориентацию растровой карты. По умолчанию первый элемент массива располагается в нижнем левом углу карты (ориентация IDENTITY). При другой ориентации (XFLIP, YFLIP или XYFLIP) порядок отображения элементов массива на растровой карте изменяется
favGetImage Orientation	Возвращает заданный favSetImageOrientation режим
favSetFixedAspect	Устанавливает фиксированное соотношение размеров сторон карты
favGetFixedAspect	Возвращает заданный favSetFixedAspect режим
favSetImageFilter	Включает линейный фильтр, используемый при выводе карты
favGetImageFilter	Возвращает заданный favSetImageFilter режим
Векторный граф	
favSetCompIndex	Задает индексы, используемые для задания *x*-, *y*-, *z*- и *w*-компонентов векторного графа (в режиме векторного графа данные выводятся как вектор *x*, *y*, *z*, *w*, в котором каждый компонент извлекается из массива в последовательности, заданной favSetCompIndex). По умолчанию в случае массива формы (4, *) для *x*-компонента используется первый элемент каждого столбца, для *y* - второй, для *z* - третий, а для *w* - четвёртый
favGetCompIndex	Возвращает заданный favSetCompIndex режим
Отображение данных	
favSetPrecision	Устанавливает точность, с которой числовые данные отображаются в таблице данных ОМ
favGetPrecision	Возвращает установленную favSetPrecision точность
favSetCellEdit Enabled	Включает/отключает режим редактирования таблицы данных ОМ
favGetCellEdit Enabled	Возвращает заданный favSetCellEditEnabled режим

favGetDefault Format	Возвращает заданный по умолчанию формат (длина поля и точность), используемый для представления числовых данных в таблице данных
favSetUseDefault Format	Задает/отключает режим использования заданного по умолчанию формата представления числовых данных. Если режим не задан, то формат регулируется подпрограммами favSetUseHex, favSetFieldWidth и favSetUseExp
favGetUseDefault Format	Возвращает заданный favSetUseDefaultFormat режим
favSetDimName	Дает имя указанному измерению массива. Длина строки, задающей имя, должна равняться AV_MAX_LABEL_LEN
favGetDimName	Возвращает имя, ассоциированное с заданным измерением массива
favSetShowDim Labels	Включает/отключает режим вывода на рисунке имен строк и столбцов массива
favGetShowDim Labels	Возвращает заданный favSetShowDimLabels режим
favSetUseHex	Включает/отключает режим отображения числовых данных в таблице данных ОМ в шестнадцатеричном виде
favGetUseHex	Возвращает заданный favSetUseHex режим
favSetUseExp	Включает/отключает режим отображения числовых данных в таблице данных ОМ в научном (экспоненциальном) формате
favGetUseExp	Возвращает заданный favSetUseExp режим
favSetFieldWidth	Задает длину поля вывода числовых данных в таблице данных ОМ
favGetFieldWidth	Возвращает текущую длину поля вывода числовых данных
Камера	
favSetCamera Position	Задает координаты камеры
favGetCamera Position	Возвращает координаты камеры
favSetCameraCoi	Задает координаты точки наблюдения, т. е. точки, на которую направляется камера. Устанавливаемые координаты должны находиться в диапазоне [0.0 - 1.0]. По умолчанию координаты этой точки равны (0.5, 0.5, 0.3)
favGetCameraCoi	Возвращает координаты точки, на которую направлена камера
favSetHomePosition	Устанавливает текущую позицию камеры в качестве исходной
favToHomePosition	Устанавливает камеру в исходную позицию, т. е. заданную подпрограммой favSetHomePosition

Маркер	
favSetRowCol	Задает позицию ячейки таблицы данных. При использовании маркера изменение позиции ячейки приводит к перемещению маркера и выделению заданной позиции
favGetRowCol	Возвращает позицию ячейки таблицы данных
favSetShowMarker	Показывает/скрывает маркер
favGetShowMarker	Возвращает заданный favSetShowMarker режим
Разное	
favSetArrayName	Задает текст, выводимый на заголовочной полосе ОМ
favGetErrorNo	Возвращает номер возникшей ошибки
favShowWindow	Отображает/скрывает окно вывода ОМ
favSetAnnotation	Задает текст аннотации

Замечание. Подпрограмма favGet*, если соответствующая favSet*-подпрограмма не вызывалась, вернет заданное по умолчанию значение, опрос которого favGet* выполняет.

В комплекте поставки ОМ имеются примеры его употребления. Они расположены в директории ...\ArrayVisualizer\Samples\Fortran\. Описание примеров можно просмотреть в Web-браузере, загрузив ...\ArrayVisualizer\Samples \Samples.htm.

Пример. Выводится 3D-вид параболоида, создаваемый подпрограммой *frame*3. При этом:

- изменяются имена осей координат: *dim*1 - на *x*, а *dim*2 - на *y* (команда favSetAxisLabel);
- задается отличная от установленной по умолчанию позиция камеры, равная (-2.3, -2.3, 2.0) (команда favSetCameraPosition), и позиция точки наблюдения - (0.0, 0.5, 0.0) (команда favSetCameraCoi);
- число больших разметок на каждой из осей устанавливается равным трем, а маленьких - единице (команды favSetNumMajorTickmarks и favSetNumMinorTickmarks);
- при выводе изображения применяются оттенки серого цвета (команда favSetPaletteId).

```
program para2
 use avdef
 use avviewer
 use dflib
 implicit none                          ! Данные о параболоиде z = x^2 + y^2
! Протяженности массива bo по первому и второму измерениям
 integer(4) :: m = 20, n = 20
```

```
integer(4) :: nd = 100                    ! Число разбиений одного сечения параболоида
integer(4) :: ch = 50                     ! Число пересекающих параболоид плоскостей
real(4), allocatable :: bo(:, :)          ! Массив точек параболоида
!dec$attributes array_visualizer :: bo
integer(4) hv, status
character(1) :: key
character(av_max_label_len) :: xLabel = 'x', yLabel = 'y'
allocate (bo(m, n))
call frame3(bo, m, n, nd, ch)             ! Формирование параболоида
call faglStartWatch(bo, status)           ! Сообщаем ОМ имя отображаемого массива
print *, "Starting Array Viewer"
call favStartViewer(hv, status)           ! Создаем экземпляр ОМ
call favSetArray(hv, bo, status)          ! Отображаем массив bo
! Задаем имя, выводимое ОМ при отображении массива bo
call favSetArrayName(hv, "Paraboloid", status)
! Задаем режим вывода заданных пользователем имен осей координат
call favSetUseAxisLabel(hv, x_axis, 1, status)
call favSetUseAxisLabel(hv, y_axis, 1, status)
! Новые имена x- и y-осей координат. Длина переменных, задающих имена осей,
! равна AV_MAX_LABEL_LEN
call favSetAxisLabel(hv, x_axis, xLabel, status)
call favSetAxisLabel(hv, y_axis, yLabel, status)
! Позиция точки наблюдения
call favSetCameraCoi(hv, 0.0, 0.5, 0.0, status)
! Позиция точки камеры
call favSetCameraPosition(hv, -2.3, -2.3, 2.0, status)
! Устанавливаем режим явного задания разметок координатных осей
call favSetAxisAutoDetail(hv, 0, status)
! Число больших разметок на координатных осях
call favSetNumMajorTickmarks(hv, x_axis, 3, status)
call favSetNumMajorTickmarks(hv, y_axis, 3, status)
call favSetNumMajorTickmarks(hv, z_axis, 3, status)
! Число малых разметок на координатных осях
call favSetNumMinorTickmarks(hv, x_axis, 1, status)
call favSetNumMinorTickmarks(hv, y_axis, 1, status)
call favSetNumMinorTickmarks(hv, z_axis, 1, status)
! Задание палитры из оттенков серого цвета
call favSetPaletteId(hv, greyscale, status)
! Показываем ОМ на экране
call favShowWindow(hv, AV_TRUE, status)
print *, "Press any key to close down the viewer"
key = getcharqq( )
call favEndViewer(hv, status)             ! Закрываем ОМ
```

```
call faglEndWatch(bo, status)          ! Удаляем массив из списка наблюдения
deallocate(bo)
end program para2                      ! Результат приведен на рис. П.-1.24
```

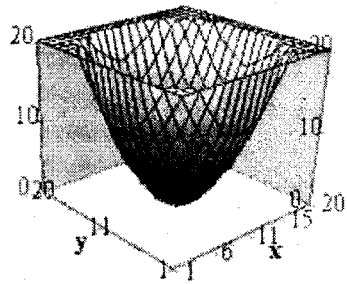


Рис. П.-1.24. Параболоид из программы para2

П.-1.6. РАСПРОСТРАНЕНИЕ КОМПОНЕНТОВ ОМ

Библиотека Aviewxxx.DLL (Aview110.DLL для версии 1.1 ОМ) должна поставляться с приложением, использующим процедуры ОМ. Причем DLL должна находиться в той же директории, что и приложение, либо приложению должен быть известен путь к DLL.

Если приложение использует Avis2D- и/или AvisGrid-процедуру, перенесите Avis2D.ocx- и/или AvisGrid.ocx-файл на компьютер, где будет запускаться приложение, и зарегистрируйте процедуры, выполнив команду

```
regsvr32 Avis2D.ocx
```

и/или команду

```
regsvr32 AvisGrid.ocx
```

Если на эксплуатируемом компьютере ОМ не установлен, то потребуется выгрузить файл Aviewer.exe с CVF-сайта http://www.compaq.com/fortran/ и выполнить его установку.

Приложение 2. ВЫВОД ГРАФИКОВ И ПОВЕРХНОСТЕЙ

П.-2.1. ВЫВОД ГРАФИКОВ ФУНКЦИЙ ОДНОЙ ПЕРЕМЕННОЙ

П.-2.1.1. ВЫВОД В DOS-ОКНО

IMSL содержит подпрограмму PLOTP (DPLOTP), позволяющую выводить в DOS-окно до 10 наборов данных. Ее вызов:

CALL PLOTP(*ndata*, *nfun*, *x*, *a*, *Lda*, *inc*, *range*, *symbol*, *xtitle*, *ytitle*, *title*)

Параметры подпрограммы PLOTP:

Все параметры подпрограммы являются *входными*.

ndata - число задающих график точек.

nfun - число наборов данных; не должно быть больше 10. Каждый набор данных задает один график функции.

x - вектор размера *ndata*, содержащий значения переменных, откладываемых по оси *x*.

a - массив формы (*Lda*, *nfun*), содержащий *nfun* наборов данных.

Lda - ведущий размер массива *a*; обычно *Lda* = *ndata*.

inc - шаг между используемыми элементами; PLOTP выводит точки *x*(1 + (*i* - 1) * *inc*), где *i* = 1, 2, ..., *ndata*.

range - вектор размера 4, задающий минимальный и максимальный *x* и минимальный и максимальный *y*; если минимальные и максимальные значения заданы равными, то они будут вычислены PLOTP.

symbol - символьная строка длины *nfun*; символ *symbol*(*i*:*i*) используется для вывода графика с номером *i*.

xtitle, *ytitle* и *title* - символьные строки, применяемые для обозначения соответственно осей *x*, *y* и вывода заголовка рисунка.

При работе с PLOTP могут возникать приведенные в табл. П.-2.1 ошибки.

Таблица П.-2.1. Информационные ошибки подпрограммы PLOTP

Тип	*Код*	*Причина ошибки*
3	7	Значение *nfun* больше 10; будут выведены только первые 10 графиков
3	8	Строка *title* слишком велика и будет усечена с правой стороны
3	9	Строка *ytitle* слишком велика и будет усечена с правой стороны
3	10	Строка *xtitle* слишком велика и будет усечена с правой стороны

Замечания:

1. Строки *ytitle* и *title* автоматически центрируются.
2. При выводе нескольких графиков буква M выводится на позицию, занимаемую более одним набором данных.
3. Устройство вывода задается UMACH.
4. Заданные по умолчанию размеры страницы в символах: ширина - 78; длина - 60. Размеры изменяются PGOPT.

Пример. Выводится график функции $y = 2e^{-x}\sin(2x) - 0.5$. Результат приведен на рис. П.-2.1.

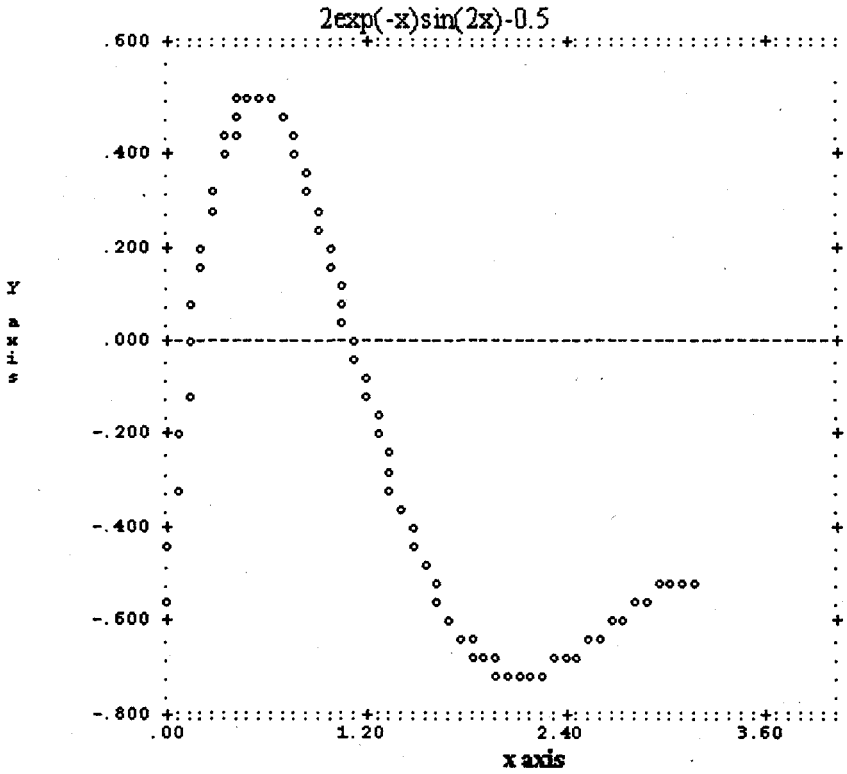


Рис. П.-2.1. График функции $y = 2e^{-x}\sin(2x) - 0.5$

```
program drt
 use msimsl
 integer(4) :: i, inc, Lda, ndata, nfun
 real(4) :: a(200, 1), delx, pi, range(4), x(200), fun2
```

```
character(1) :: symbol = 'o'
range = (/ 0.0, 4.0, -0.8, 0.8 /)
ndata  = 200; nfun = 1; Lda = 200; inc = 1
pi = 3.14159
delx = 2.* pi / float(Lda - 1)
do i = 1, Lda
 x(i)  = -pi + float(i - 1) * delx
 a(i, 1) = fun2(x(i))
end do
call pgopt(-1, 78)                    ! Размеры страницы
call pgopt(-2, 50)
open(1, file = 'a.txt')               ! Подсоединяем файл к устройству 1
call umach(-2, 1)                     ! Вывод выполняется в файл a.txt
call plotp(ndata, nfun, x, a, Lda, inc, range, symbol,                    &
     'x axis', 'y axis', ' 2exp(-x)sin(2x)-0.5')
end program drt

function fun2(x)                      ! Отображаемая функция
 real(4) :: fun2, x
 fun2 = 2.0 * exp(-x) * sin(2.0 * x) - 0.5
end function fun2                     ! Результат приведен на рис. П.-2.1
```

П.-2.1.2. ВЫВОД В ОКНО OPENGL

П.-2.1.2.1. Описание процедуры

Обеспечиваемое PLOTP качество неудовлетворительно. Рассмотрим поэтому подпрограмму *drawCurve*, позволяющую выводить до 10 графиков функций одной переменной средствами OpenGL. Вызов подпрограммы:

```
CALL drawCurve(npx, xmi, xma, f1,
     [f2, f3, f4, f5, f6, f7, f8, f9, f10,                                &
     color1, color2, color3, color4, color5,                              &
     color6, color7, color8, color9, color10, grid, coords])
```

Заключенные в квадратные скобки параметры имеют атрибут OPTIONAL, т. е. являются необязательными. Все параметры являются *входными*. Тип вещественных параметров *xmi*, *xma*, *fi* и *colori* (*i* = 1, 2, ..., 10) - REAL(4).

npx - число точек, используемых при выводе графика; реально график выводится в виде *npx* - 1 смежных отрезков прямой линии. Тип *npx* - INTEGER(4).

xmi, *xma* - диапазон изменения аргумента *x*.

fi - имя функции с номером *i*. Параметры *f*2 - *f*10 являются необязательными. Должны обладать атрибутом EXTERNAL.

colori - цвет, которым выводится функция с номером *i*; задается в виде массива или его конструктора, элементы которого - вещественные числа в диапазоне [0.0, 1.0]. Например, *color*1 = (/ 1.0, 1.0, 0.0 /). Все параметры *colori*, если они присутствуют и если число выводимых функций меньше 10, должны задаваться с ключевыми словами. Если *colori* для графика *i* не задан, то он выводится черным цветом.

Замечание. Графики функций выводятся на белом фоне, поэтому задание цвета *colori* = (/ 1.0, 1.0, 1.0 /) недопустимо.

grid - параметр типа LOGICAL(4), равный .TRUE., если нужно отобразить координатную сетку, и .FALSE. - в противном случае. Параметр является необязательным и должен быть задан с ключевым словом, например *grid* = .TRUE., если число выводимых функций и число заданных цветов меньше 10. Отсутствие параметра равносильно его заданию со значением .TRUE..

coords - параметр типа LOGICAL(4), равный .TRUE., если нужно вывести координаты сетки, и .FALSE. - в противном случае. Параметр является необязательным и должен быть задан с ключевым словом, если не задан хотя бы один из предшествующих необязательных параметров. Отсутствие параметра равносильно его заданию со значением .TRUE.. Заметим, однако, что координаты сетки выводятся, если одновременно и *coords* = .TRUE., и *grid* = .TRUE..

В программном компоненте, из которого вызывается *drawCurve*, должна быть ссылка на модуль *alib*, содержащий интерфейс подпрограммы *drawCurve*.

Подпрограмма вычисляет диапазон изменения координаты *y*; отображает оси координат и сетку; последняя отображается пунктиром. Приводимый вариант подпрограммы является упрощенным, так как не содержит анализа возможных ошибок и в нем не предусмотрен вывод символьных данных. Пользователь при необходимости может устранить эти (и другие) недостатки. Также он может создать вариант подпрограммы, получающей в качестве параметров не имена функций, а массив с задающими эти функции данными.

Пример. Выводятся графики функций

$$y = e^{-x}\sin(4x)$$

и

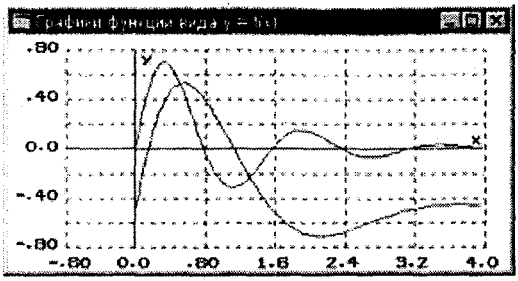
$$y = 2e^{-x}\sin(2x) - 0.5.$$

Первый график выводится красным, а второй - синим цветом. Графики выводит команда

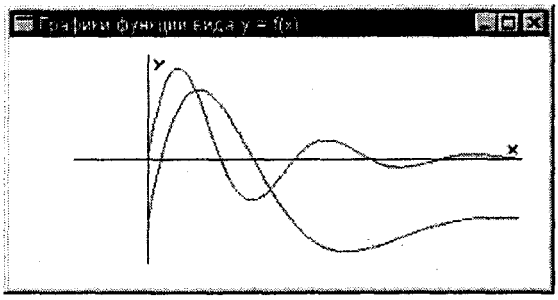
```
call drawCurve(100, 0.0, 4.0, f1 = fun1, f2 = fun2,                    &
```

```
color1 = (/ 1.0, 0.0, 0.0 /),                                    &
color2 = (/ 0.0, 0.0, 1.0 /), grid = .true., coords = .true.)
```

Код функций *fun*1 и *fun*2 приведен в следующем разделе.



а



б

Рис. П.-2.2. Графики функций $y = e^{-x}\sin(4x)$ *и* $y = 2e^{-x}\sin(2x) - 0.5$*:*
а - grid = .TRUE.; б - grid = .FALSE

П.-2.1.2.2. Программа вывода графиков функций $y = f(x)$

Приводимые процедуры ссылаются на модули *alib*, *points* и *GLface*, которые также использует и программа вывода поверхности - графика функции $y = f(x, y)$. Текст этих модулей, а также модуля *figures*, ссылка на который выполняется в *points*, приведены в разд. П.-2.3. Программы вывода поверхности даются в разд. П.-2.2.

Чтобы воспользоваться приводимыми программами, нужно создать консоль-проект, разместив в нем все перечисленные модули и либо приводимый ниже код, если нужно вывести графики функций $y = f(x)$, либо код из разд. П.-2.2.2, если отображается поверхность.

```
! Подпрограмма вывода графиков функций одной переменной
subroutine drawCurve(npx, xmi, xma,                              &
```

```
        f1, f2, f3, f4, f5, f6, f7, f8, f9, f10,                    &
        color1, color2, color3, color4, color5,                    &
        color6, color7, color8, color9, color10, grid, coords)
use msfwin
use points
use GLface                          ! Содержит интерфейсы myReshape1, display1,
integer(4) :: npx                   ! myReshape2 и display2
real(4) :: xmi, xma, ymi, yma       ! Диапазоны изменения координат
real(4), external :: f1
real(4), external, optional :: f2, f3, f4, f5, f6, f7, f8, f9, f10
real(4), dimension(3), optional ::                                  &
        color1, color2, color3, color4, color5,                    &
        color6, color7, color8, color9, color10
logical(4), optional :: grid, coords  ! grid = .TRUE., когда отображается сетка
real(4) :: x, y, dx                   ! coords = .TRUE., когда выводятся координаты
integer(4) :: i, j
! Число точек, используемых при выводе графика
npox = npx
ifun = 0; ifun(1) = 1
! ifun(i) = 1, если выводится график под номером i
if(present(f2)) ifun(2) = 1
if(present(f3)) ifun(3) = 1
if(present(f4)) ifun(4) = 1
if(present(f5)) ifun(5) = 1
if(present(f6)) ifun(6) = 1
if(present(f7)) ifun(7) = 1
if(present(f8)) ifun(8) = 1
if(present(f9)) ifun(9) = 1
if(present(f10)) ifun(10) = 1
cols = 0.0
if(present(color1)) cols(:, 1) = color1  ! Формируем массив цветов
if(present(color2)) cols(:, 2) = color2
if(present(color3)) cols(:, 3) = color3
if(present(color4)) cols(:, 4) = color4
if(present(color5)) cols(:, 5) = color5
if(present(color6)) cols(:, 6) = color6
if(present(color7)) cols(:, 7) = color7
if(present(color8)) cols(:, 8) = color8
if(present(color9)) cols(:, 9) = color9
if(present(color10)) cols(:, 10) = color10
gridval = .true.                    ! По умолчанию gridval = .TRUE.
if(present(grid)) gridval = grid    ! gridval = .TRUE., если отображается сетка
coordval = gridval
! coordval = .TRUE., если выводятся координаты
if(present(coords)) coordval = coords
```

```
 if(.not. allocated(px)) allocate(px(npox, nfmax), py(npox, nfmax))
 yma = -huge(yma); ymi = huge(ymi); dx = (xma - xmi) / float(npox)
 do j = 1, nfmax                          ! Находим диапазон изменения y - [ymi, yma]
  if(ifun(j) == 0) cycle
  x = xmi
  do i = 1, npox
   select case(j)
    case(1); y = f1(x)
    case(2); y = f2(x)
    case(3); y = f3(x)
    case(4); y = f4(x)
    case(5); y = f5(x)
    case(6); y = f6(x)
    case(7); y = f7(x)
    case(8); y = f8(x)
    case(9); y = f9(x)
    case(10); y = f10(x)
   end select
   px(i, j) = x; py(i, j) = y
   x = x + dx
   yma = max(yma, y); ymi = min(ymi, y)
  end do
 end do
 call initGL(2, sum(ifun))                ! 2 - выводятся графики вида y = f(x)
 ! Формирование списков команд, пригодных для вывода текста
 call makeFigures(2)
 ! Вычисляем протяженность осей координат и шаги координатной сетки
 call setVars(xmi, xma, ymi, yma)
 ! Смотрим, нужно ли выводить 0.0 по оси y (не нужно, если нижний левый угол
 ! графика - начало координат)
 if(abs(xmi1) < tiny(xmi1) .and. abs(ymi1) < tiny(ymi1)) fzer = .false.
 ! При изменении размеров окна вызывается подпрограмма myReshape1
 call fauxReshapeFunc(loc(myReshape1))
 ! Подпрограмма display1 выполняет роль оконной процедуры
 ! Вызывается каждый раз при перемещении и изменении размеров окна вывода
 ! и выводит оси координат, координатную сетку и графики функций
 call fauxMainLoop(loc(display1))
 deallocate(px, py)
end subroutine drawCurve

subroutine myReshape1(w, h)
 !ms$ attributes stdcall, alias : '_myReshape1@8' :: myReshape1
 use points                               ! Подпрограмма, формирующая матрицу
 integer(4) :: w, h                       ! проецирования
 call fglViewport(0, 0, w, h)             ! Задание видового порта
 call fglMatrixMode(gl_projection); call fglLoadIdentity( )
```

```
 call fgluOrtho2D(xl, xr, yb, yt)
end subroutine myReshape1

! Выводит оси координат, координатную сетку и графики функций
subroutine display1( )                    ! Вывод изображения
 !ms$ attributes stdcall, alias : '_display1@0' :: display1
 use points
 integer(4) :: i, j
 real(4) :: cc(3)
 call fglClearColor(1.0, 1.0, 1.0, 1.0)   ! Цвет фона - белый
 call fglClear(gl_color_buffer_bit)       ! Очистка буфера цвета
 call fglColor3f(0.0, 0.0, 0.0)           ! Текущий цвет - черный
 if(gridval) then                         ! Вывод координатной сетки по образцу
  call fglLineStipple(2, pattern)         ! Повторяем каждый бит образца 2 раза
  ! Теперь вывод линии будет выполняться с применением образца
  call fglEnable(gl_line_stipple)
  call gridt(xmi1, xma1, ymi1, yma1, 0.0, 0.0, 'x', gridx, 2)
  call gridt(ymi1, yma1, xmi1, xma1, 0.0, 0.0, 'y', gridy, 2)
  call fglDisable(gl_line_stipple)
 end if
 call fglBegin(gl_lines)                  ! Вывод осей координат
  call fglVertex2f(xmi1, 0.0)
  call fglVertex2f(xma1, 0.0)
  call fglVertex2f(0.0, ymi1)
  call fglVertex2f(0.0, yma1)
 call fglEnd( )
 call fglRasterPos2f(xma1, 0.0)           ! Вывод имени оси x
 letter = rasterfont(15, :)
 call fglBitmap(spx, spy, 10.0, -4.0, 0.0, 0.0, loc(letter))
 call fglRasterPos2f(0.0, yma1)           ! Вывод имени оси y
 letter = rasterfont(16, :)
 call fglBitmap(spx, spy, -4.0, 10.0, 0.0, 0.0, loc(letter))
 call fglFlush( )                         ! Вывод на экран осей координат и сетки
 do j = 1, nfmax                          ! Вывод графиков функций
  if(ifun(j) == 0) cycle
  cc = cols(:, j)
  call fglColor3fv(loc(cc))               ! cc - текущий цвет
  call fglBegin(gl_line_strip)
   do i = 1, npox
    call fglVertex2f(px(i, j), py(i, j))
   end do
  call fglEnd( )
  call fglFlush( )                        ! Отображение графиков функций на экране
 end do
end subroutine display1
```

```
! Драйвер программы вывода графиков функций y = f(x)
program curve
 ! Модуль с интерфейсом подпрограммы drawCurve
 use alib
 real(4), external :: fun1, fun2
 ! Рисуем графики двух функций
 ! Вывод выполняется на белом фоне; оси координат выводятся черным цветом
 call drawCurve(100, 0.0, 4.0, f1 = fun1, f2 = fun2,                    &
       color1 = (/ 1.0, 0.0, 0.0 /),                                    &
       color2 = (/ 0.0, 0.0, 1.0 /), grid = .true., coords = .true.)
end program curve

function fun1(x)                          ! Отображаемые функции
 real(4) :: fun1, x
 fun1 = sin(4.0 * x) * exp(-x)
end function fun1

function fun2(x)
 real(4) :: fun2, x
 fun2 = 2.0 * sin(2.0 * x) * exp(-x) - 0.5
end function fun2
```

П.-2.2. ВЫВОД ГРАФИКА ФУНКЦИИ ДВУХ ПЕРЕМЕННЫХ

П.-2.2.1. ОПИСАНИЕ ПРОЦЕДУРЫ

График функции $z = f(x, y)$ выполняется подпрограммой

```
CALL drawSurf(npx, npy, xmi, xma, ymi, yma, f,                          &
     [color, grid, coords, cullFace, shading, normals])
```

Все параметры являются *входными*. Параметры, ограниченные квадратными скобками, являются необязательными. Тип параметров *npx* и *npy* - INTEGER(4); *xmi*, *xma*, *ymi*, *yma*, *f* и *color* - REAL(4); *grid*, *coords*, *cullFace*, *shading*, *normals* - LOGICAL(4).

npx и *npy* - соответственно размеры сетки по осям *x* и *y*.

xmi и *xma* - соответственно минимальная и максимальная *x*-координаты графика.

ymi и *yma* - соответственно минимальная и максимальная *y*-координаты графика.

f - функция $z = f(x, y)$, график которой выводится. В вызывающей процедуре должна иметь атрибут EXTERNAL.

color - вектор размера 3, содержащий RGB-компоненты используемого при выводе поверхности цвета. Диапазон изменения каждого компонента - [0, 1]. Например, вектор *color* = (/ 0.2, 0.2, 0.2 /) задает оттенок серого цвета. Если выводится каркасная модель поверхности, то заданный массивом *color*

цвет применяется для отображения ее ребер. При выводе тоновой модели *color* определяет цвет, зеркально отражаемый поверхностью. Поскольку вывод выполняется на белом фоне, то в случае каркасной модели *color* не должен задавать белый цвет - (/ 1.0, 1.0, 1.0 /). При выводе тоновой модели такое значение *color* допустимо.

grid - логический параметр, регулирующий вывод координатной сетки. Она выводится, если *grid* = .TRUE.. Значение по умолчанию - .TRUE..

coords - логический параметр, управляющий выводом координат. Они выводятся, если *coords* = .TRUE.. Значение по умолчанию - .TRUE.. Параметр игнорируется, когда *grid* = .FALSE..

cullFace - логический параметр, включающий или отключающий режим вывода нелицевых сторон граней поверхности. Нелицевые стороны отображаются, когда *cullFace* = .FALSE.. Игнорируется, если *shading* = .TRUE.. Значение по умолчанию - .TRUE..

shading - логический параметр, задающий вид используемой для ввода модели поверхности. Если *shading* = .TRUE., то выводится тоновая модель поверхности. В противном случае - каркасная. Значение по умолчанию - .FALSE..

normals - логический параметр, обеспечивающий, если *normals* = .TRUE., отображение поля нормалей к вершинам выводимой поверхности. По умолчанию *normals* = .FALSE., т. е. нормали не выводятся. Параметр игнорируется, если *shading* = .TRUE..

Рассмотрим в качестве примеров вывод поверхностей

$$z = \sin(r)/r, \text{ где } r = \sqrt{x^2 + y^2},$$

и

$$z = xe^{-x^2-y^2},$$

отображая их с различными значениями параметров *drawSurf*.

В первом случае *z* возвращает функция

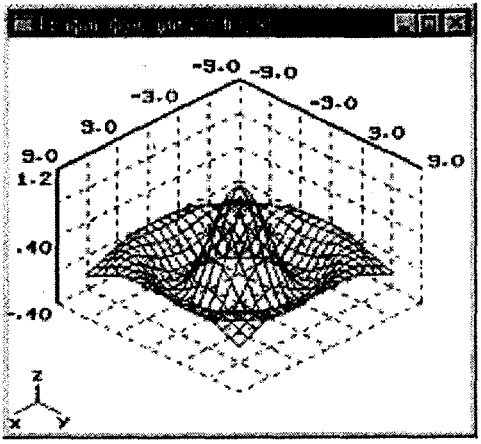
```
function fun1(x, y)                    ! Отображаемая поверхность; строится
 real(4) :: fun1, x, y, r              ! в квадрате -8.0 ≤ x ≤ 8.0; -8.0 ≤ y ≤ 8.0
 r = sqrt(x**2 + y**2) + epsilon(r)    ! на сетке 20×20
 fun1 =  sin(r) / r
end function fun1
```

а во втором -

```
function fun2(x, y)                    ! Отображаемая поверхность; строится
 real(4) :: fun2, x, y                 ! в прямоугольнике -2.0 ≤ x ≤ 2.0; -1.0 ≤ y ≤ 1.0
 fun2 = x * exp(-x * x - y * y)        ! на сетке 20×10
end function fun2
```

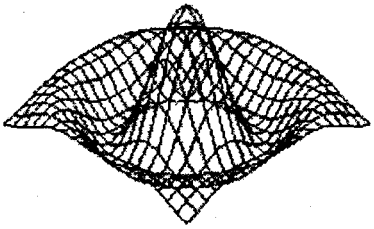
Пример 1:

```
call drawSurf(20, 20, -8.0, 8.0, -8.0, 8.0, f = fun1,                    &
     color = (/ 0.0, 0.0, 0.0 /))
```



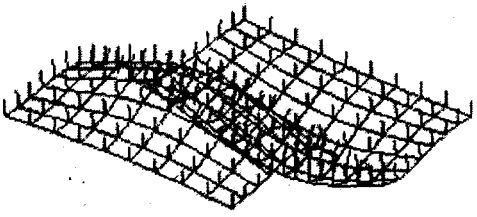
Пример 2:

```
call drawSurf(20, 20, -8.0, 8.0, -8.0, 8.0, f = fun1,                    &
     color = (/ 0.0, 0.0, 0.0 /), grid = .false.)
```



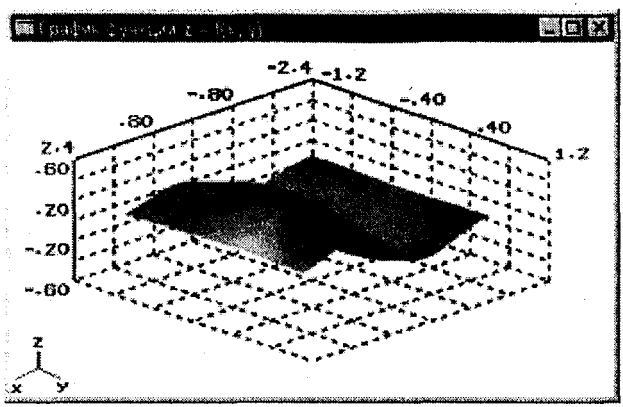
Пример 3:

```
call drawSurf(20, 10, -2.0, 2.0, -1.0, 1.0, f = fun2,                    &
     color = (/ 0.0, 0.0, 0.0 /), normals = .true.)
```



Пример 4:

```
call drawSurf(20, 20, -8.0, 8.0, -8.0, 8.0, f = fun2,                    &
     color = (/ 1.0, 1.0, 1.0 /) , shading = .true., cullFace = .false.)
```



П.-2.2.2. ПРОГРАММА ВЫВОДА ГРАФИКА ФУНКЦИИ $z = f(x, y)$

Раздел содержит коды модулей и их процедур, обеспечивающих (вместе с модулями из разд. П.-2.3) вывод поверхности, задаваемой функцией $z = f(x, y)$.

```
! Подпрограммы расчета нормалей к вершинам поверхности
module norms
 use points                          ! Код модуля points должен быть размещен
contains                             ! ранее кода модуля norms
! Подпрограмма вызывается при расчете освещенности поверхности
 subroutine norm( )                  ! Формирует нормали к граням
  integer(4) :: i, j
  real(4) :: a(3), b(3)              ! Массивы координат векторов a и b
  do j = 1, npoy - 1                 ! Расчет нормалей к граням
   y = py(j, 1)
   y2 = py(j + 1, 1)
   b(1) = 0.0
   a(2) = 0.0; b(2) = y2 - y
   do i = 1, npox - 1                ! Обходим грани против часовой стрелки
    a(1) = px(i + 1, 1) - px(i, 1)
    a(3) = pz(i + 1, j) - pz(i, j)
    b(3) = pz(i, j + 1) - pz(i, j)
    sn(i, j, 1) = a(2)*b(3) - b(2)*a(3)   ! Координаты вектора внешней нормали
    sn(i, j, 2) = a(3)*b(1) - b(3)*a(1)   ! к боковой грани
    sn(i, j, 3) = a(1)*b(2) - b(1)*a(2)
```

```
  ! Нормализация
  sn(i, j, :) = sn(i, j, :) / sqrt(sum(sn(i, j, :) * sn(i, j, :)))
 end do
end do
! Полагаем, что крайние вершины имеют те же нормали, что и предшествующие
sn(npox, :, :) = sn(npox - 1, :, :); sn(:, npoy, :) = sn(:, npoy - 1, :)
call vnorm( )                    ! Расчет координат векторов нормалей к вершинам
contains
subroutine vnorm( )              ! Формирует массив sn с нормалями к вершинам
 integer(4) :: i, j
 real(4) :: sx, sy, sz1, sz2, sz3    ! Смещение векторов нормалей соседних граней
 real(4), allocatable, dimension(:, :, :) :: sn2
 allocate(sn2(npox, npoy, 3))
 sn2 = sn
 do j = 2, npoy - 1                  ! Расчет нормалей к вершинам
  sy = py(j, 1) - py(j - 1, 1)       ! Смещение по y
  do i = 2, npox - 1                 ! Обходим грани против часовой стрелки
   sx = px(i, 1) - px(i - 1, 1)      ! Смещение по x
   sz1 = pz(i, j) - pz(i, j - 1)     ! Смещение по z
   sz2 = pz(i, j) - pz(i - 1, j)
   sz3 = pz(i, j) - pz(i - 1, j - 1)
   sn(i, j, :) = sn2(i, j, :) +                                    &
                 sn2(i, j - 1, :) + sy + sz1 +                     &
                 sn2(i - 1, j, :) + sx + sz2 +                     &
                 sn2(i - 1, j - 1, :) + sx + sy + sz3
   ! Нормализация
   sn(i, j, :) = sn(i, j, :) / sqrt(sum(sn(i, j, :) * sn(i, j, :)))
  end do
 end do
 deallocate(sn2)
 ! Полагаем, что первые вершины имеют те же нормали, что и вторые
 sn(1, :, :) = sn(2, :, :); sn(:, 1, :) = sn(:, 2, :)
 ! Полагаем, что последние вершины имеют те же нормали, что и предпоследние
 sn(npox, :, :) = sn(npox - 1, :, :); sn(:, npoy, :) = sn(:, npoy - 1, :)
end subroutine vnorm

end subroutine norm

subroutine showNorm(p)           ! Отображение нормалей к вершинам
 use opengl
 integer(4) :: i
 real(4) :: y, p                 ! p - масштабный коэффициент
 real(4) :: v1(3), v2(3)         ! Координаты нормали
 call fglDisable(gl_lighting)    ! Отключаем расчет освещенности
 call fglColor3f(0.0, 0.0, 0.0)  ! Текущий цвет - черный
```

```
 call fglBegin(gl_lines)
 do j = 1, npoy                          ! Вывод нормалей
  y = py(j, 1)
  do i = 1, npox
    v1 = (/ px(i, 1), y, pz(i, j) /);    call fglVertex3fv(loc(v1))
    v2 = v1 + p * sn(i, j, :);           call fglVertex3fv(loc(v2))
  end do
 end do
 call fglEnd( )
 call fglFlush( )                        ! Отображение нормалей
end subroutine showNorm

end module norms

! Характеристики материала, источника света и сцены
module matLight
 ! Параметры материала, определяющие
 ! RGBA-компоненты отражения фонового света
 real(4), dimension(4) :: mat_ambient = (/ 0.2, 0.2, 0.2, 1.0 /)
 ! Параметры материала, определяющие RGBA-компоненты диффузного отражения
 real(4), dimension(4) :: mat_diffuse = (/ 0.6, 0.6, 0.6, 1.0 /)
 ! Параметры материала, определяющие RGBA-компоненты
 ! зеркально отражаемого света лицевых граней
 real(4), dimension(4) :: mat_specular_front = (/ 0.9, 0.9, 0.9, 1.0 /)
 ! Параметры материала, определяющие RGBA-компоненты
 ! зеркально отражаемого света нелицевых граней
 real(4), dimension(4) :: mat_specular_back = (/ 0.0, 0.9, 0.0, 1.0 /)
 ! Параметры источника, определяющие
 ! RGBA-компоненты фонового света источника
 real(4), dimension(4) :: light_ambient = (/ 0.2, 0.2, 0.2, 1.0 /)
 ! Параметры источника, определяющие
 ! RGBA-компоненты диффузного света источника
 real(4), dimension(4) :: light_diffuse = (/ 0.5, 0.5, 0.5, 1.0 /)
 ! Параметры источника, определяющие
 ! RGBA-компоненты зеркального света источника
 real(4), dimension(4) :: light_specular = (/ 0.9, 0.9, 0.9, 1.0 /)
 ! Координаты источника света (задаются после вычислений xma1, yma1, zma1)
 real(4), dimension(4) :: light0_position = 0.0
 ! Параметры, определяющие фоновый свет всей сцены
 real(4), dimension(4) :: lmodel_ambient = (/ 0.4, 0.4, 0.4, 1.0 /)

contains

 ! Задание характеристик материала, источника света и сцены
 subroutine setScene( )
  use opengl
  ! Задаем свойства материала для расчета освещенности объекта
```

```
  ! Первые 2 свойства одинаковы для лицевых и нелицевых сторон граней
  call fglMaterialfv(gl_front_and_back, gl_ambient, loc(mat_ambient))
  call fglMaterialfv(gl_front_and_back, gl_diffuse, loc(mat_diffuse))
  ! Таким образом зеркально отражают свет лицевые стороны граней
  call fglMaterialfv(gl_front, gl_specular, loc(mat_specular_front))
  ! Степень ослабления зеркального отражаемого света
  call fglMaterialf(gl_front_and_back, gl_shininess, 3)
  ! Таким образом зеркально отражают свет нелицевые стороны граней
  call fglMaterialfv(gl_back, gl_specular, loc(mat_specular_back))
  ! Задаем свойства источника света GL_LIGHT0 и его координаты (GL_POSITION)
  call fglLightfv(gl_light0, gl_ambient, loc(light_ambient))
  call fglLightfv(gl_light0, gl_diffuse, loc(light_diffuse))
  call fglLightfv(gl_light0, gl_specular, loc(light_specular))
  call fglLightfv(gl_light0, gl_position, loc(light0_position))
  ! Включаем в уравнение освещенности источник GL_LIGHT0
  call fglEnable(gl_light0)
  ! Задание модели освещенности всей сцены
  call fglLightModelfv(gl_light_model_ambient, loc(lmodel_ambient))
  call fglLightModelf(gl_light_model_two_side, gl_true)
  call fglLightModelf(gl_light_model_local_viewer, gl_true)
  call fglEnable(gl_depth_test)                    ! Активизируем тест глубины
  ! Выводятся пиксели с наименьшими z-координатами
  call fglDepthFunc(gl_less)
  call fglClearColor(1.0, 1.0, 1.0, 1.0)           ! Цвет фона - ярко-белый
  ! Задаем значение, каким очищается буфер глубины
  call fglClearDepth(1.0)
 end subroutine setScene
end module matLight

! Подпрограмма вывода поверхности (тонированной или каркасной)
subroutine drawSurf(npx, npy, xmi, xma, ymi, yma, f,                    &
      color, grid, coords, cullFace, shading, normals)
 use msfwin
 use points                          ! Содержит ссылку на модуль figures
 ! Задает свойства материалов, источников света и сцены
 use matLight
 use GLface                          ! Содержит интерфейсы myReshape1, display1,
                                     ! myReshape2 и display2
 use norms                           ! Ссылка на модуль, содержащий процедуры
 integer(4) :: npx, npy              ! для расчета и вывода нормалей
 ! Диапазоны изменения координат
 real(4) :: xmi, xma, ymi, yma, zmi, zma
 real(4), external :: f
 real(4), dimension(3), optional :: color
 ! grid = .TRUE., когда отображается сетка
 ! coords = .TRUE., когда выводятся координаты
```

```
logical(4), optional :: grid, coords
! shading = .TRUE., когда выполняется расчет освещенности объекта
! normals = .TRUE., когда выводится поле нормалей к вершинам
! cullFace = .TRUE., когда не выводятся нелицевые стороны граней
logical(4), optional :: shading, normals, cullFace
real(4) :: x, y, z, dx, dy
integer(4) :: i, j
! Число точек, используемых при выводе графика
npox = npx
npoy = npy
cols = 0.0
if(present(color)) cols(:, 1) = color      ! Формируем массив цветов
gridval = .true.
if(present(grid)) gridval = grid           ! gridval = .TRUE., если отображается сетка
coordval = gridval
! coordval = .TRUE., если выводятся координаты
if(present(coords)) coordval = coords
! shadval = .TRUE., когда выполняется расчет освещенности объекта
! normval = .TRUE., когда выводится поле нормалей к вершинам
! cullFace = .TRUE., когда не выводятся нелицевые стороны граней
shadval = .false.; normval = .false.; cullval = .true.
if(present(shading)) shadval = shading
if(present(normals)) normval = normals
if(present(cullFace)) cullval = cullFace
! При выводе каркасной модели цвет не должен быть близок к цвету фона
if(present(color) .and. .not. shadval .and. all(cols(:, 1) > 0.7)) cols(:, 1) = 0.0
if(.not. allocated(pz))                                                    &
      allocate(px(npox, 1), py(npoy, 1), pz(npox, npoy), sn(npox, npoy, 3))
zma = tiny(zma); zmi = huge(zmi)
dx = (xma - xmi) / float(npox); dy = (yma - ymi) / float(npoy)
y = ymi
do j = 1, npoy                             ! Находим [zmi, zma] - диапазон изменения z
 x = xmi
 py(j, 1) = y
 do i = 1, npox
  z = f(x, y)
  pz(i, j) = z
  x = x + dx
  zma = max(zma, z); zmi = min(zmi, z)
 end do
 y = y + dy
end do
x = xmi
do i = 1, npox
  px(i, 1) = x
```

```
  x = x + dx
 end do
 ! Если выполняется тонирование или выводится поле нормалей
 if(shadval .or. normval) call norm( )   ! Вычисляем нормали к вершинам
 call initGL(3, 1)                       ! 3 - выводится график z = f(x, y)
 ! Формирование списков команд, пригодных для вывода текста
 call makeFigures(3)
 ! Вычисляем протяженности осей координат и шаги координатной сетки
 call setVars(xmi, xma, ymi, yma, zmi, zma)
 call fglMatrixMode(gl_modelview); call fglLoadIdentity( )
 call fglRotatef(-90.0, 1.0, 0.0, 0.0)   ! Выполняем поворот в МСК
 call fglRotatef(180.0, 0.0, 0.0, 1.0)   ! и ориентируем ось z вверх
 ! Координаты источника света привязываем
 ! к максимальным размерам области вывода
 light0_position = (/ 3.0 * xma1, yma1, 5.0 * zma1, 0.0 /)
 ! mat_specular_front - свет, зеркально отражаемый материалом
 mat_specular_front(1:3) = color         ! color - задаваемый на входе цвет
 call setScene( )                        ! Задаем параметры сцены
 ! При изменении размеров окна вызывается подпрограмма myReshape2
 call fauxReshapeFunc(loc(myReshape2))
 ! Подпрограмма display2 выполняет роль оконной процедуры
 ! Вызывается каждый раз при перемещении и изменении размеров окна вывода
 ! и выводит оси координат, координатную сетку и поверхность
 call fauxMainLoop(loc(display2))
 deallocate(px, py, pz, sn)
end subroutine drawSurf

subroutine myReshape2(w, h)
 !ms$ attributes stdcall, alias : '_myReshape2@8' :: myReshape2
 use points                              ! Подпрограмма, формирующая матрицу
 integer(4) :: w, h                      ! проецирования
 real(4) :: pfi, psi
 call fglViewport(0, 0, w, h)            ! Задание видового порта
 call fglMatrixMode(gl_projection); call fglLoadIdentity( )
 pfi = asind(sqrt(1.0 / 3.0 )); psi = asind(sqrt(0.5))
 call fglRotatef(-pfi, 1.0, 0.0, .0)     ! Выполняем изометрическое проецирование
 call fglRotatef(-psi, 0.0, 1.0, 0.0)
 call fglOrtho(xl, xr, yb, yt, zn, zf)
 call fglTranslatef(0.0, (yb + yt), 0.0) ! Перемещение в область видимости
end subroutine myReshape2

subroutine display2( )                   ! Вывод изображения
 !ms$ attributes stdcall, alias : '_display2@0' :: display2
 use points
 use norms                               ! Ссылка на модуль, содержащий процедуры
 integer(4) :: i, j                      ! для расчета и вывода нормалей
```

```
real(4) :: y, cc(3), v(3)
! Очистка буфера цвета и буфера глубины
call fglClear(gl_color_buffer_bit .or. gl_depth_buffer_bit)
call fglColor3f(0.0, 0.0, 0.0)          ! Текущий цвет – черный
! Отключаем расчет освещенности для вывода сетки
call fglDisable(gl_lighting)
if(gridval) then                        ! Вывод координатной сетки по образцу
 call fglLineStipple(2, pattern)        ! Повторяем каждый бит образца 2 раза
 call fglEnable(gl_line_stipple)
 ! Теперь вывод линии будет выполняться с применением образца
 call gridt(xmi1, xma1, ymi1, yma1, zmi1, zma1, 'x', gridx, 3)
 call gridt(ymi1, yma1, xmi1, xma1, zmi1, zma1, 'y', gridy, 3)
 call gridt(zmi1, zma1, xmi1, xma1, ymi1, yma1, 'z', gridz, 3)
 call fglDisable(gl_line_stipple)
end if
call fglLineWidth(2.0)
call fglBegin(gl_lines)                 ! Вывод границ области вывода
 call fglVertex3f(xmi1, ymi1, zma1)     ! Граница по x
 call fglVertex3f(xma1, ymi1, zma1)
 call fglVertex3f(xmi1, ymi1, zma1)     ! Граница по y
 call fglVertex3f(xmi1, yma1, zma1)
 call fglVertex3f(xma1, ymi1, zmi1)     ! Граница по z
 call fglVertex3f(xma1, ymi1, zma1)
call fglEnd( )
! Выведем пиктограмму в нижнем левом углу видового порта
! Подготовка к выводу пиктограммы осей координат
call fglPushMatrix( )                   ! Запоминаем матрицу проецирования
call fglMatrixMode(gl_modelview)        ! Текущей стала видовая матрица
call fglPushMatrix( )                   ! Запоминаем и эту матрицу
call fglLoadIdentity( )                 ! Видовая матрица равна единичной
call fglMatrixMode(gl_projection)       ! Текущей стала матрица проецирования
call fglLoadIdentity( )                 ! Матрица проецирования равна единичной
call fglPixelStorei(gl_unpack_alignment, 1)
call fglRasterPos2i(-1, -1)             ! Вывод пиктограммы осей координат
call fglBitmap(32, 32, -4.0, -3.0, 0.0, 0.0, loc(axis))
letter = rasterfont(15, :)              ! Вывод имени оси x
call fglBitmap(spx, spy, -1.0, -3.0, 32.0, 0.0, loc(letter))
letter = rasterfont(16, :)              ! Вывод имени оси y
call fglBitmap(spx, spy, -1.0, -3.0, -16.0, 35.0, loc(letter))
letter = rasterfont(17, :)              ! Вывод имени оси z
call fglBitmap(spx, spy, 0.0, 0.0, 0.0, 0.0, loc(letter))
call fglPopMatrix( )                    ! Извлекаем из стека матрицу проецирования
call fglMatrixMode(gl_modelview)        ! Текущей стала видовая матрица
call fglPopMatrix( )                    ! Восстанавливаем эту матрицу
call fglFlush( )                        ! Вывод на экран осей координат и сетки
```

```
 if(shadval) then                    ! Выводится тонированная поверхность
  call fglShadeModel(gl_smooth)      ! Выполняем интерполяцию цветов
 ! Активизируем заданные параметры освещенности
  call fglEnable(gl_lighting)
 else
  call fglPolygonMode(gl_back, gl_line)
  call fglPolygonMode(gl_front, gl_line)
  cc = cols(:, 1)                    ! cc - текущий цвет
  call fglColor3fv(loc(cc))
  call fglLineWidth(1.0)
  call fglShadeModel(gl_flat)
 end if
 if(cullval) then
  call fglCullFace(gl_back)          ! Выводим только лицевые стороны граней
  call fglEnable(gl_cull_face)
 end if
 call fglBegin(gl_quads)             ! Вывод независимых четырехугольников
  do j = 1, npoy - 1                 ! Вывод поверхности
   y = py(j, 1)
   y2 = py(j + 1, 1)                 ! Обходим грани против часовой стрелки
   do i = 1, nprox - 1               ! Строим очередную грань
    ! Нормаль к вершине 1
    ! Задается, если выполняется тонирование или вывод нормалей
    if(shadval .or. normval) v = sn(i, j, :); call fglNormal3fv(loc(v))
    call fglVertex3f(px(i, 1), y, pz(i, j))
    if(shadval .or. normval)v = sn(i + 1, j, :); call fglNormal3fv(loc(v))
    call fglVertex3f(px(i + 1, 1), y, pz(i + 1, j))
    if(shadval .or. normval)v = sn(i + 1, j + 1, :); call fglNormal3fv(loc(v))
    call fglVertex3f(px(i + 1, 1), y2, pz(i + 1, j + 1))
    if(shadval .or. normval)v = sn(i, j + 1, :); call fglNormal3fv(loc(v))
    call fglVertex3f(px(i, 1), y2, pz(i, j + 1))
   end do
  end do
  call fglEnd( )
  call fglFlush( )                   ! Отображение поверхности
  if(normval) then                   ! Если выводится поле нормалей к вершинам
   ! Параметр - масштабный коэффициент
   call showNorm(abs((zma1 - zmi1) / 10.0))
  end if
end subroutine display2

program surf                         ! Драйвер программы вывода 3D-поверхности
 use alib                            ! Модуль с интерфейсом подпрограммы drawSurf
 real(4), external :: fun1, fun2
 ! Рисуем график поверхности, заданной функцией fun
 ! Выводим координатную сетку (grid = .TRUE.) и координаты (coords = .TRUE.)
```

```
! Задаем вывод тонированной модели (shading = .TRUE.)
! Если normals = .TRUE., то выводится поле нормалей к вершинам
! cullFace = .TRUE., когда не выводятся нелицевые стороны граней
! Вывод выполняется на белом фоне; оси координат выводятся черным цветом
call drawSurf(20, 10, -2.0, 2.0, -1.0, 1.0,                                    &
      f = fun2, color = (/ 1.0, 0.0, 0.0 /),                                   &
      grid = .true., coords = .true.,                                          &
      cullFace = .false.,                                                      &
      shading = .true., normals = .false.)
end program surf

function fun1(x, y)                    ! Отображаемая поверхность; строится
 real(4) :: fun1, x, y, r              ! в квадрате -8.0 ≤ x ≤ 8.0; -8.0 ≤ y ≤ 8.0
 r = sqrt(x**2 + y**2) + epsilon(r)    ! на сетке 20×20
 fun1 =  sin(r) / r
end function fun1

function fun2(x, y)                    ! Отображаемая поверхность; строится
 real(4) :: fun2, x, y                 ! в прямоугольнике -2.0 ≤ x ≤ 2.0; -1.0 ≤ y ≤ 1.0
 fun2 = x * exp(-x * x - y * y)        ! на сетке 20×10
end function fun2
```

П.-2.3. МОДУЛИ, ПРИМЕНЯЕМЫЕ ПРИ ВЫВОДЕ ГРАФИКОВ ФУНКЦИЙ

На приводимые модули ссылаются как процедуры, обеспечивающие вывод графиков функций вида $y = f(x)$, так и процедуры, рисующие поверхность $z = f(x, y)$.

```
module figures                         ! Модуль, задающий цифры, точку,
 use opengl                            ! знаки + и - и буквы x, y, z и e
 integer(4), parameter :: mli = 18, nli = 8
! mli - число символов (битовых образов) в массиве rasterfont
! nli - размер знакоместа
 integer(1), dimension(nli) :: letter  ! Массив для одного символа
 integer(1) :: pointplace = 12         ! Позиция точки в массиве
! Точка выводится более компактно. Для ее поиска применяется pointplace
! rasterfont - массив символов с ASCII-кодами цифр и букв x, y, z и e
 integer(1), dimension(mli, nli) :: rasterfont = reshape((/ &
 #3c, #66, #c3, #c3, #c3, #c3, #66, #3c,                   &    ! 0   (48)
 #7e, #18, #18, #18, #18, #78, #38, #18,                   &    ! 1   (49)
 #ff, #60, #30, #18, #0c, #06, #e7, #7e,                   &    ! 2   ...
  #7e, #e7, #03, #03, #3e, #03, #e7, #7e,                  &    ! 3
 #0c, #0c, #0c, #ff, #cc, #6c, #3c, #1c,                   &    ! 4
 #7e, #c3, #03, #03, #fe, #c0, #c0, #ff,                   &    ! 5
 #7e, #c3, #c3, #c3, #fe, #c0, #c3, #7e,                   &    ! 6
```

```
 #30, #30, #30, #18, #0c, #06, #03, #ff,          &     ! 7
 #7e, #c3, #c3, #c3, #3c, #c3, #c3, #7e,          &     ! 8
 #7e, #e7, #03, #03, #7f, #c3, #c3, #7e,          &     ! 9   (57)
 #00, #00, #00, #3f, #3f, #00, #00, #00,          &     ! -   (45)
 #00, #38, #38, #00, #00, #00, #00, #00,          &     ! .   (46)
 #7c, #c2, #c0, #fc, #c6, #c6, #7c, #00,          &     ! e   (101 = 69 + 32)
 #00, #18, #18, #ff, #ff, #18, #18, #00,          &     ! +   (43)
 #c6, #6c, #38, #38, #6c, #c6, #00, #00,          &     ! x   (120)
 #60, #30, #18, #1c, #36, #63, #c3, #00,          &     ! y   (121)
 #fe, #60, #30, #18, #0c, #fe, #00, #00,          &     ! z   (122)
 #00, #00, #00, #00, #00, #00, #00, #00           &     ! ''  (32)
 /), shape = (/ mli, nli /), order = (/ 2, 1 /))
real(4) :: spx = 8.0, spy = 8.0          ! x и y - размеры знакоместа
! axis - массив с 3D-пиктограммой осей координат;
! применяется при выводе поверхностей
integer(1), dimension(128) :: axis = (/                                    &
#00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, &
#00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, &
#00, #00, #00, #00, #80, #00, #00, #01, #60, #00, #00, #06, #18, #00, #00, #18, &
#06, #00, #00, #60, #01, #80, #01, #80, #00, #60, #06, #00, #00, #18, #18, #00, &
#00, #06, #60, #00, #00, #01, #80, #00, #00, #01, #80, #00, #00, #01, #80, #00, &
#00, #01, #80, #00, #00, #01, #80, #00, #00, #01, #80, #00, #00, #01, #80, #00, &
#00, #01, #80, #00, #00, #01, #80, #00, #00, #01, #80, #00, #00, #01, #80, #00, &
#00, #01, #80, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00, #00 /)
contains

! Модуль figures включает процедуры:
! makeFigures - формирует списки команд вывода текста
! printString - выводит строку текста

! Создаем списки команд, используемые затем при выводе наборов символов
! Выводимый набор символов задается в виде строки,
! например: '0123456789xyz+-.e'
subroutine makeFigures(k23)
 integer(4) :: k23, i, sg
 ! Начало координат битового образа
 real(4) :: xorigx, yorigx, xorigy, yorigy
 real(4) :: xorigz, yorigz
 ! Выравнивание по одному байту
 call fglPixelStorei(gl_unpack_alignment, 1)
 ! Списки команд для вывода x-, y- и z-координат
 ! Генерируем 3*m списков команд - по 3 списка на каждый заданный
 ! массивом rasterfont символ. Первый список для x-координат,
 ! второй - для y-координат, и третий - для z-координат
 if(k23 == 2) then
```

```
  xorigx = 14.0; yorigx = 15.0
  xorigy = 14.0; yorigy = 1.0
  sg = 1
 else
  xorigx = 8.0; yorigx = -5.0
  xorigy = -5.0; yorigy = -1.0
  xorigz = 12.0; yorigz = 10.0
  sg = -1
 end if
 do i = 1, mli
  letter = rasterfont(i, :)
  call details(i, sg, xorigx, yorigx)
  call details(i + mli, -sg, xorigy, yorigy)
  if(k23 == 3) call details(i + mli + mli, sg, xorigz, yorigz)
 end do

contains

 ! Внутренняя процедура подпрограммы makeFigures
 ! Детализирует различия в способах вывода x- и y-координат
 subroutine details(k, sg, xorig, yorig)
  integer(4) :: k, sg                    ! k - номер списка; sg = +1 или -1
  real(4) :: xorig, yorig                ! Начало координат битового образа
  call fglNewList(k, gl_compile)
  if(i == pointplace) then
   ! Способ вывода точки
   call fglBitmap(spx, spy, xorig + sg, yorig, sg * (spx - 1.0), 0.0, loc(letter))
  else
   ! Способ вывода x-координат, если sg = +1, и y-координат, если sg = -1
   call fglBitmap(spx, spy, xorig, yorig, sg * (spx + 1.0), 0.0, loc(letter))
  end if
 call fglEndList( )
 end subroutine details

end subroutine makeFigures

subroutine printString(s, gt, k23)          ! Вывод строки текста
 character(*) s
 character(1) gt                            ! gt = 'x', 'y' или 'z'
 integer(4) :: k23                          ! k23 = 2, если выводится 2D-график,
 integer(2), dimension(len_trim(s)) :: ars  ! иначе k23 = 3
 integer(2) :: lens, i, k
 k23 = k23
 lens = len_trim(s)
 do i = 1, lens                  ! Формируем массив номеров команд, которые
  k= int2(ichar(s(i:i)))         ! нужно выполнить для вывода строки с текстом
  if(k > 47 .and. k < 58) then   ! Цифры 0, 1, ..., 9
```

```
    ars(i) = k - 47
   else if(k == 45 .or. k == 46) then
    ars(i) = k - 34                    ! Знаки - и .
   else if(k == 69) then               ! Буква e
    ars(i) = k - 56
   else if(k == 43) then
    ars(i) = k - 29                    ! Знак +
   else if(k > 119 .and. k < 123) then ! Буквы x, y и z
    ars(i) = k - 105
   else
    ars(i) = 18                        ! Пробел
   end if
  end do
  if(gt == 'y') ars = ars + mli
  if(gt == 'z') ars = ars + mli + mli
  call fglPushAttrib(gl_list_bit)      ! Сохраняем атрибуты изображения
  ! Номер выполняемой команды будет равен ars(i)
  call fglCallLists(lens, gl_short, loc(ars))
  call fglPopAttrib( )                 ! Восстанавливаем атрибуты изображения
 end subroutine printString
end module figures

module points
 use figures                           ! Содержит ссылку на модуль opengl.mod
 integer(4), parameter :: nfmax = 10   ! Максимальное число выводимых графиков
 ! Число используемых для вывода графика точек
 integer(4) :: npox, npoy
 integer(4) :: ifun(nfmax)
 logical(4) :: gridval                 ! gridval = .TRUE., если сетка отображается
 real(4) :: gridx, gridy, gridz        ! Шаги координатной сетки по осям x, y и z
 ! coordval = .TRUE., если выводятся координаты
 logical(4) :: coordval
 ! shadval = .TRUE., когда выполняется расчет освещенности объекта
 ! normval = .TRUE., когда выводится поле нормалей к вершинам
 ! cullval = .TRUE., когда не выводятся нелицевые стороны граней
 logical(4) :: shadval, normval, cullval
 ! Число шагов координатной сетки по осям x, y и z
 integer(4), parameter :: ngx = 5, ngy = ngx, ngz = ngx - 2
 ! px, py - массивы, задающие выводимые кривые
 real(4), allocatable, dimension(:, :) :: px, py
 ! Массив для вывода поверхности
 real(4), allocatable, dimension(:, :) :: pz
 ! coordval = .TRUE., если выводятся координаты
 real(4), allocatable, dimension(:, :, :) :: sn
 real(4) :: xl, xr, yb, yt, zn, zf     ! Протяженность осей координат
```

```
 real(4) :: cols(3, nfmax)              ! Массив цветов для графиков функций
 real(4) :: xmi1, xma1, ymi1, yma1, zmi1, zma1
 logical(4) :: fzer = .true.            ! fzer = .false., когда 0.0 не выводится по оси y
 ! Образец для вывода координатной сетки (пунктир)
 integer(2) :: pattern = 2#0011001100110011
contains
! Модуль points содержит процедуры:
! initGL - выполняет инициализацию окна OpenGL
! setVars - вычисляет протяженность осей координат и шаги координатной сетки
! fgrid - находит шаг координатной сетки и область ее отображения по оси x, y или z
! gridt - подпрограмма, выводящая линии координатной сетки и их координаты
 subroutine initGL(k23, nf)     ! Инициализация окна OpenGL
  integer(4) :: k23, nf         ! k23 = 2, если работаем в 2D, k23 = 3, если - в 3D
  ! w, h - начальные размеры окна вывода
  integer(4) :: result = 0, w = 400, h = 400
  character(50) :: title
  if(k23 == 2) then
   call fauxInitDisplayMode(aux_single .or. aux_rgb)
  else
   call fauxInitDisplayMode(aux_single .or. aux_rgb .or. aux_depth16)
  end if
  call fauxInitPosition(100, 50, w, h)
  if(k23 == 2) then
   if(nf == 1) then                 ! Число выводимых функций
    title = 'График функции y = f(x)' // char(0)
   else
    title = 'Графики функции вида y = f(x)' // char(0)
   end if
   call fglShadeModel(gl_flat)      ! Вывод без интерполяции цветов
  else
   title = 'График функции z = f(x, y)' // char(0)
  end if
  result = fauxInitWindow(title)
 end subroutine initGL

 subroutine setVars(xmi, xma, ymi, yma, zmi, zma)
  real(4) :: xmi, xma, ymi, yma
  real(4), optional :: zmi, zma
  real(4) :: addx, addy, rk
  real(4) :: ptmi(4), ptma(4)       ! Точки вершин объема вывода
  real(4) :: xyzi(4), xyza(4), addxyz(4), rkxyz(4)
  real(4), dimension(4, 4) :: rx, ry ! Матрицы поворота вокруг осей x и y
  ! Матрица поворота вокруг оси x на -90°;
  ! вращение выполняется против часовой стрелки
```

```
rx = reshape((/      1.0,   0.0,   0.0,   0.0,      &
                     0.0,   0.0,   1.0,   0.0,      &
                     0.0,  -1.0,   0.0,   0.0,      &
                     0.0,   0.0,   0.0,   1.0 /),   shape = (/4, 4 /))
! Матрица поворота вокруг оси y на 180°
ry = reshape((/     -1.0,   0.0,   0.0,   0.0,      &
                     0.0,   1.0,   0.0,   0.0,      &
                     0.0,   0.0,  -1.0,   0.0,      &
                     0.0,   0.0,   0.0,   1.0 /),   shape = (/4, 4 /))
! Вычисляем gridx, gridy и gridz - шаги координатной сетки по осям x, y и z
xmi1 = xmi; xma1 = xma; ymi1 = ymi; yma1 = yma
gridx = fgrid(xmi1, xma1, ngx)          ! gridx - шаг координатной сетки по оси x
! xmi1, xma1 - область отображения координатной сетки по оси x
gridy = fgrid(ymi1, yma1, ngy)
! Координаты xmi1, xma1, ymi1, yma1 задают границы координатной сетки
! Координаты xl, xr, yb, yt используем для задания области вывода
if(present(zmi)) then                   ! Работаем с осью z
  zmi1 = zmi; zma1 = zma
  gridz = fgrid(zmi1, zma1, ngz)
  ptmi = (/ xmi1, ymi1, zmi1, 1.0 /)
  ptma = (/ xma1, yma1, zma1, 1.0 /)
  rx = matmul(ry, rx)
  ptmi =  matmul(rx, ptmi)
  ptma =  matmul(rx, ptma)
  xyzi = min(ptmi, ptma)
  xyza = max(ptmi, ptma)
  rkxyz = (/ 0.4, 0.7, 0.4, 1.0 /)
  addxyz = rkxyz * (xyza - xyzi)
  xyzi = xyzi - addxyz; xyza = xyza + addxyz;
  xl = xyzi(1); xr = xyza(1)
  yb = xyzi(2); yt = xyza(2)
  zn = xyzi(3); zf = xyza(3)
 else ! Работаем в 2D
  rk = 0.1
  addx = rk * (xma1 - xmi1)             ! Увеличиваем протяженность осей координат
  addy = rk * (yma1 - ymi1)             ! x и y соответственно на 2 * addx и 2 * addy
  xl = xmi1 - 1.4 * addx; xr = xma1 + 0.6 * addx
  yb = ymi1 - 1.2 * addy; yt = yma1 + 0.8 * addy
 end if
end subroutine setVars

! fgrid - функция модуля points; содержит функцию tita, возвращающую
! границу области отображения координатной сетки
function fgrid(ti, ta, ng)              ! Находит шаг координатной сетки и область
 real(4) :: fgrid, ti, ta               ! ее отображения по оси x, y или z
 integer(4) ng, k
```

```
 fgrid = (ta - ti) / float(ng)
 k = 0
 if(fgrid < 1.0) then
  do while(fgrid <= 1.0)
   k = k + 1
   fgrid = fgrid * 10.0
  end do
  fgrid = aint(fgrid) / float(10 ** k)
 else if(fgrid > 10.0) then
  do while(fgrid > 10.0)
   k = k + 1
   fgrid = fgrid / 10.0
  end do
  fgrid = aint(fgrid) * float(10 ** k)
 else
  fgrid = aint(fgrid)
 end if
 ti = tita(ti, 1)                    ! ti, ta - границы области отображения
 ta = tita(ta, 2)                    ! координатной сетки
 contains                            ! tita - процедура функции fgrid
  function tita(t, k)                ! ti и ta должны быть кратны fgrid
   real(4) :: tita, t                ! t - граница области отображения сетки
   integer(4) :: k                   ! k = 1, если рассматривается ti
   tita = 0.0                        ! k = 2, если - ta
   if(t < 0.0) then
    do while(tita > t)
     tita = tita - fgrid
    end do
    if(k == 2) tita = tita + fgrid
   else
    do while(tita < t)
     tita = tita + fgrid
    end do
    if(k == 1) tita = tita - fgrid
   end if
  end function tita
end function fgrid

! Подпрограмма, выводящая линии координатной сетки и их координаты
! Содержит процедуры:
! fgs - находит шаг координатной сетки в оконных координатах
! outString - формирует строку strin,
! которая содержит координаты выводимой линии
! makeStrin - формирует строку, которая используется для вывода координат сетки
subroutine gridt(ti1, ta1, ti2, ta2, ti3, ta3, gt, grid, k23)
```

```
real(4) :: ti1, ta1, ti2, ta2, ti3, ta3      ! Протяженность осей координат
character(1) :: gt                           ! Вид линий: 'x', 'y' или 'z'
! strin - строка, содержащая координату линии координатной сетки
character(10) :: strin
! grid - шаг координатной сетки
real(4) :: t, pt1(3), pt2(3), pt3(3), grid
! k23 = 2, если вывод на плоскости, иначе k23 = 3
integer(4) :: k23
logical(4) :: fl
integer(4) :: gs                             ! Шаг сетки в оконных координатах
integer(4) :: tcur, tlast                    ! tlast - координата последнего вывода текста
! Применяется для предотвращения наложения строк,
! содержащих координаты сетки. Перекрытие строк может возникнуть при
! незначительной ширине (высоте) окна вывода
gs = fgs( )
tlast = 0
tcur = 0                                     ! Текущая позиция вывода текста
! Находим положение первой выводимой линии координатной сетки
t = ti1 - grid
do while(t < ta1)                            ! Вывод линий координатной сетки
 fl = .true.                                 ! Истина, если линии выводятся
 t = t + grid                                ! Если gt = 'x', то выводятся x-линии
 if(k23 == 2) then                           ! Если gt = 'y', то выводятся y-линии
  if(abs(t) < 0.9 * grid) fl = .false.       ! Если gt = 'z', то выводятся z-линии
 else
  if(t > ta1) exit
 end if
 if(gt == 'x') then
  pt1 = (/ t, ti2, ti3 /)
  pt2 = (/ t, ta2, ti3 /)
  if(k23 == 3) pt3 = (/ t, ti2, ta3 /)
 else if(gt == 'y') then
   pt1 = (/ ti2, t, ti3 /)
   pt2 = (/ ta2, t, ti3 /)
   if(k23 == 3) pt3 = (/ ti2, t, ta3 /)
 else if(gt == 'z') then
   pt1 = (/ ti2, ti3, t /)
   pt2 = (/ ta2, ti3, t /)
   pt3 = (/ ti2, ta3, t /)
 end if
 if(coordval) call outString( )              ! Вывод координат или имени оси координат
 if(fl) then
  call fglBegin(gl_lines)                    ! Вывод очередной линии координатной сетки
  call fglVertex3fv(loc(pt1))
  call fglVertex3fv(loc(pt2))
```

```
    if(k23 == 3) then                    ! Проводим вертикальную линию
     call fglVertex3fv(loc(pt1))
     call fglVertex3fv(loc(pt3))
    end if
   call fglEnd( )
  end if
 end do
contains
! fgs, outString и makeStrin - внутренние процедуры подпрограммы gridt
! fgs - находит шаг координатной сетки в оконных координатах
function fgs( )
 integer(4) :: fgs, k                    ! k - номер измерения
 real(8), dimension(4, 4) :: modelMatrix, projMatrix
 integer(4), dimension(4) :: viewport
 real(8) :: win1(2), win2(2), pt(4), ptn(4)
 call fglGetDoublev(gl_modelview_matrix, loc(modelMatrix))
 call fglGetDoublev(gl_projection_matrix, loc(projMatrix))
 call fglGetIntegerv(gl_viewport, loc(viewport))
 select case(gt)
 case('x'); k = 1; pt = (/ ti1, ti2, ti3, 1.0 /)
 case('y'); k = 2; pt = (/ ti2, ti1, ti3, 1.0 /)
 case('z'); k = 3; pt = (/ ti2, ti3, ti1, 1.0 /)
 end select
 ptn = matmul(matmul(projMatrix, modelMatrix), pt)
 win1 = (ptn(1:2) + 1.0_8) * dfloat((viewport(3:4) - viewport(1:2)) / 2)
 pt(k) = pt(k) + real(grid, 8)
 ptn = matmul(matmul(projMatrix, modelMatrix), pt)
 win2 = (ptn(1:2) + 1.0_8) * dfloat((viewport(3:4) - viewport(1:2)) / 2)
 win1 = win2 - win1
 fgs = int(sqrt(sum(win1 * win1)))
end function fgs
! outString - внутренняя процедура подпрограммы gridt
subroutine outString( )                  ! Формирует и выводит строку с текстом
 integer(4) :: slen                      ! Длина строки с выводимым текстом
 ! Формируем строку strin, которая содержит координаты выводимой линии
 if(abs(t) > 0.1 * grid) then
  call makeStrin(strin, t, gt)
 else                                    ! Выводится координата 0.0
  strin = '0.0'
 end if
 if(k23 == 2 .or. gt == 'z') then
  tcur = tcur + gs                       ! Текущая позиция вывода текста
 else
  tcur = tcur + sqrt(2.0) / 2.0 * gs
```

```
  end if
  ! Если не нужен вывод 0.0 по оси y
  if(.not. fzer .and. gt == 'y' .and. strin == '0.0' .and. k23 == 2) return
  if(tlast > 0) then                    ! Если выводится не первая координата
   slen = len_trim(strin)               ! Проверим, можно ли разместить новый текст
   if(k23 == 2) then                    ! без перекрытий с предшествующим
    if(gt == 'x') then
     if(tcur - 2 * slen * spx < tlast) return
    else                                ! gt = 'y'
     if(tcur - 3 * spy < tlast) return
    end if
   else                                 ! k23 = 3
    if(gt == 'x' .or. gt == 'y') then
     if(tcur - 3 * spy < tlast) return
    else                                ! gt = 'z'
     if(tcur - 3 * spy < tlast) return
    end if
   end if
  end if
  tlast = tcur                          ! Текст можно разместить без перекрытий
  ! Вывод сформированной строки strin
  if(k23 == 2) then                     ! Если работаем в 2D
   ! Растровая позиция первой цифры строки strin
   call fglRasterPos2fv(loc(pt1))
  else
   if(gt == 'z') then
    call fglRasterPos3fv(loc(pt2))
   else
    call fglRasterPos3fv(loc(pt3))
   end if
  end if
  ! Вывод координат
  call printString(trim(strin), gt, k23)
end subroutine outString

! makeStrin - внутренняя процедура подпрограммы gridt
subroutine makeStrin(strin, t, gt)      ! Формирует строку, которая используется
 character(*) strin, gt                 ! для вывода координат сетки
 real(4) :: t
 ! Максимальное число позиций для координат - 5; минимальное - 3
 real(4) :: ctpmin = 0.0099, ctmmax = -0.099
 ! Если 0.0010 <= t <= 9999., то используется формат F
 ! Если -999. <= t <= -0.010, то также используется формат F
 ! Если 9999. < t <= .9e+9), используется формат E5.1e1
 ! Если t = 0, то возвращается '0.0'
 ! Во всех остальных случаях возвращается пробел
```

```
integer(4), parameter :: np= 7, nm = 5
integer(4) :: k, i, j                    ! Номер выбранного в массиве fmt формата
character(1) :: ch
character(10), dimension(np) ::          &
     fmtp = (/ '(F5.4)', '(F4.3)', '(F3.2)', '(F3.1)', '(F3.0)', '(F4.0)', '(F5.0)' /) .
character(10), dimension(nm) :: fmtm = (/ '(F5.3)', '(F4.2)', '(F4.1)', '(F4.0)', '(F5.0)' /)
character(10) :: fmt
fmt = ' '
if(t >= 0.00095 .and. t < 10000. .or. t >= -999. .and.  t <= -0.0095) then
 k = 1
 if(t > 0) then
  ! (Минимальное положительное значение) * 10 для формата F
  ct = ctpmin
  do while(ct <= t .and. k < np)
   k = k + 1
   ct = ct * 10
  end do
 else
  ! (Максимальное отрицательное значение) / 10 для формата F
  ct = ctmmax
  do while(ct > t .and. k < nm)
   k = k + 1
   ct = ct * 10
  end do
 end if
 !  if(t >= 0.0010 .and. t < 0.0100)          fmt = '(F5.4)'
 !  if(t >= 0.010 .and. t < 0.100)            fmt = '(F4.3)'
 !  if(t >= 0.10 .and. t < 1.00)              fmt = '(F3.2)'
 !  if(t >= 1.0 .and. t < 10.0)               fmt = '(F3.1)'
 !  if(t >= 10.0 .and. t < 100.0)             fmt = '(F3.0)'
 !  if(t >= 100.0 .and. t < 1000.0)           fmt = '(F4.0)'
 !  if(t >= 1000.0 .and. t < 10000.0)         fmt = '(F5.0)'
 !
 !  if(t <= -0.010 .and. t > -0.100)          fmt = '(F5.3)'
 !  if(t <= -0.10 .and. t > -1.00)            fmt = '(F5.2)'
 !  if(t <= -1.0 .and. t > -10.0)             fmt = '(F4.1)'
 !  if(t <= -10. .and. t > -100.)             fmt = '(F4.0)'
 !  if(t <= -100. .and. t > -1000.)           fmt = '(F5.0)'
 if(t > 0) then
  fmt = fmtp(k)
 else
  fmt = fmtm(k)
 end if
else if(t > 0.0 .and. t <= .9e+9) then
  fmt = '(e5.1e1)'
```

```
  else if(abs(t) < tiny(t)) then
    strin = '0.0'
    return
  else
    strin = ' '
    return
  end if
  write(strin, fmt) t                    ! Преобразование число - строка
  ! Меняем порядок следования символов в строке
  if(gt == 'y' .and. k23 == 2 .or. gt == 'x' .and.                    &
       k23 == 3 .or. gt == 'z' .and. k23 == 3) then
   k = len_trim(strin)
   do i = 1, k / 2
    ch = strin(i:i)
    j = k - i + 1
    strin(i:i) = strin(j:j)
    strin(j:j) = ch
   end do
  end if
 end subroutine makeStrin

 end subroutine gridt

end module points

! Содержит интерфейсы подпрограмм myReshape1, display1, myReshape2 и display2
module GLface                        ! Подпрограммы myReshape1, display1,
 interface                           ! myReshape2 и display2 должны обладать
  subroutine myReshape1(w, h)        ! атрибутом EXTENAL
   !ms$ attributes stdcall, alias : '_myReshape1@8' :: myReshape1
   integer(4) :: w, h
  end subroutine myReshape1
  subroutine display1( )
   !ms$ attributes stdcall, alias : '_display1@0' :: display1
  end subroutine display1
  subroutine myReshape2(w, h)
   !ms$ attributes stdcall, alias : '_myReshape2@8' :: myReshape2
   integer(4) :: w, h
  end subroutine myReshape2
  subroutine display2( )
   !ms$ attributes stdcall, alias : '_display2@0' :: display2
  end subroutine display2
 end interface
end module GLface

! Модуль с интерфейсами программ вывода
```

```
! графиков 2D-функций и 3D-поверхности
module alib
 ! Параметры с атрибутом OPTIONAL - необязательные
 interface
  subroutine drawCurve(npx, xmi, xma,                                    &
             f1, f2, f3, f4, f5, f6, f7, f8, f9, f10,                    &
             color1, color2, color3, color4, color5,                     &
             color6, color7, color8, color9, color10, grid, coords)
   ! Число точек, используемых для вывода кривой
   integer(4) :: npx
   real(4) :: xmi, xma                    ! Диапазон изменения аргумента x
   real(4), external :: f1                ! Задание одной функции обязательно
   real(4), external, optional :: f2, f3, f4, f5, f6, f7, f8, f9, f10
   real(4), dimension(3), optional ::                                     &
             color1, color2, color3, color4, color5,                     &
             color6, color7, color8, color9, color10
   logical(4), optional :: grid, coords   ! grid = .TRUE., если сетка отображается
  end subroutine drawCurve                ! coords = .TRUE., когда выводятся координаты
 end interface

 interface
  subroutine drawSurf(npx, npy, xmi, xma, ymi, yma, f,                   &
       color, grid, coords, cullFace, shading, normals)
   integer(4) :: npx, npy
   real(4) :: xmi, xma, ymi, yma          ! Диапазоны изменения координат
   real(4), external :: f
   real(4), dimension(3), optional :: color
  ! grid = .TRUE., когда отображается сетка;
  ! coords = .TRUE., когда выводятся координаты
   logical(4), optional :: grid, coords
  ! shading = .TRUE., когда выполняется расчет освещенности объекта
  ! normals = .TRUE., когда выводится поле нормалей к вершинам
  ! cullFace = .TRUE., когда не выводятся нелицевые стороны граней
   logical(4), optional :: shading, normals, cullFace
  end subroutine drawSurf
 end interface
end module alib
```

П.-2.4. СОЗДАНИЕ РАСТРОВОГО ШРИФТА

В графиках, создаваемых подпрограммами drawCurve и drawSurf, для вывода символов используются битовые образы, задаваемые в массиве *rasterfont* модуля *figures*. Размер каждого битового образа - 8×8 пикселей. Его вывод в окно OpenGL осуществляется командой fglBitmap.

П.-2.4.1. СОЗДАНИЕ БИТОВОГО ОБРАЗА ОДНОГО СИМВОЛА

При создании битового образа одной цифры, буквы или иного символа можно придерживаться такой последовательности:

- создать чертеж буквы в сетке размером 8×8 (рис. П.-2.3). Каждая ячейка сетки отвечает одному пикселю, т. е. вся буква займет 64 пикселя. Обновляемому на экране пикселю соответствует единичное значение ячейки, необновляемому - нуль.

0	0	0	0	0	0	0	0
0	1	1	1	1	1	0	0
1	1	0	0	0	1	1	0
1	1	0	0	0	1	1	0
1	1	1	1	1	1	0	0
1	1	0	0	0	0	0	0
1	1	0	0	0	0	1	0
0	1	1	1	1	1	0	0

Рис. П.-2.3. Проект строчной буквы ***е***

- разместить в целочисленный массив *isymb*(1:8) значения, содержащиеся в строках сетки, использовав для записи строки двоичное представление числа, и выполнить программу

```
program figure
 integer(4), parameter :: n = 8
 integer(1) :: isymb(n), i          ! isymb - массив, содержащий одну букву
 isymb = (/                         &
  2#00000000,                       &              ! #00
  2#01111100,                       &              ! #7C
  2#11000110,                       &              ! #C6
  2#11000110,                       &              ! #C6
  2#11111100,                       &              ! #FC
  2#11000000,                       &              ! #C0
  2#11000010,                       &              ! #C2
  2#01111100 /)                                    ! #7C
 ! Меняем порядок следования элементов. Это необходимо, поскольку вывод
 ! битового образа выполняется с первого элемента массива снизу вверх
 isymb = isymb(n:1:-1)
 do i = 1, n                        ! Вывод без продвижения
  write(*, fmt = '(1x, a, z2.2, a)', advance = 'no')  '#', isymb(i), ','
  ! Результат: #7C, #C2, #C0, #FC, #C6, #C6, #7C, #00,
 end do
end program figure
```

- выведенный результат использовать для задания значения элемента массива *rasterfont* модуля *figures*.

Замечание. Переход к представлению чисел в шестнадцатеричной системе счисления не является необходимым (можно задать символы и в двоичной системе счисления), а выполняется с целью снижения размера файла с исходным кодом.

П.-2.4.2. ВЫВОД ПОСЛЕДОВАТЕЛЬНОСТИ СИМВОЛОВ

Приводимая программа используется для проверки созданных символов, а ее отдельные фрагменты - в программах вывода графиков функций. При необходимости можно расширить приведенный набор символов (битовых образов) и изменить их размеры и геометрию.

Собственно вывод строки символов выполняется подпрограммой

call printString(*строка символов*)

Подпрограмма *printString* по заданной строке для каждого ее символа находит номер команды, обеспечивающей вывод символа. Найденный номер заносится в массив *ars*, который затем употребляется в качестве параметра команды fglCallLists, активизирующей необходимые для вывода заданных символов операторы. Последние оформлены подпрограммой *makeFigures* в виде списка команд.

Если подаваемая в качестве параметра строка содержит символ, отсутствующий в массиве *rasterfont*, то на его месте выводится пробел.

```
module fitest                          ! Модуль, задающий цифры, точку, знаки + и -
 use opengl                            ! и буквы x, y и z
integer(4), parameter :: mli = 18, nli = 8
! mli - число символов (битовых образов) в массиве rasterfont
! nli - размер знакоместа
integer(1), dimension(nli) :: letter      ! Массив для одного символа
! rasterfont - массив символов с ASCII-кодами цифр и букв x, y, z и e
integer(1), dimension(mli, nli) :: rasterfont = reshape((/      &
 #3c, #66, #c3, #c3, #c3, #c3, #66, #3c,                  &     ! 0
 #7e, #18, #18, #18, #18, #78, #38, #18,                  &     ! 1
 #ff, #60, #30, #18, #0c, #06, #e7, #7e,                  &     ! 2
 #7e, #e7, #03, #03, #3e, #03, #e7, #7e,                  &     ! 3
 #0c, #0c, #0c, #ff, #cc, #6c, #3c, #1c,                  &     ! 4
 #7e, #c3, #03, #03, #fe, #c0, #c0, #ff,                  &     ! 5
 #7e, #c3, #c3, #c3, #fe, #c0, #c3, #7e,                  &     ! 6
 #30, #30, #30, #18, #0c, #06, #03, #ff,                  &     ! 7
 #7e, #c3, #c3, #c3, #3c, #c3, #c3, #7e,                  &     ! 8
 #7e, #e7, #03, #03, #7f, #c3, #c3, #7e,                  &     ! 9
 #c6, #6c, #38, #38, #6c, #c6, #00, #00,                  &     ! x
 #c0, #60, #30, #38, #6c, #c6, #00, #00,                  &     ! y
 #fe, #60, #30, #18, #0c, #fe, #00, #00,                  &     ! z
```

```
 #00, #18, #18, #ff, #ff, #18, #18, #00,                    &     ! +
 #00, #00, #00, #00, #00, #00, #00, #00,                    &     ! Пробел
 #00, #00, #00, #ff, #ff, #00, #00, #00,                    &     ! -
 #00, #38, #38, #00, #00, #00, #00, #00,                    &     ! .
#7c, #c2, #c0, #fc, #c6, #c6, #7c, #00                      &     ! e
/), shape = (/ mli, nli /), order = (/ 2, 1 /))
contains
! Создаем списки команд, используемых затем при выводе наборов символов
! Выводимый набор символов задается в виде строки,
! например: '0123456789xyz+-.e'
subroutine makeFigures( )
 integer(4) :: i
 ! Выравнивание по одному байту
 call fglPixelStorei(gl_unpack_alignment, 1)
 do i = 1, mli                          ! Генерируем m списков команд - по одному
  call fglNewList(i, gl_compile)        ! списку на каждый заданный массивом
   letter = rasterfont(i, :)            ! rasterfont символ
   call fglBitmap(8, 8, 0.0, 8.0, 10.0, 0.0, loc(letter))
  call fglEndList( )
 end do
end subroutine makeFigures
subroutine printString(s)               ! Вывод строки текста
 character(*) s
 integer(2), dimension(len_trim(s)) :: ars
 integer(2) :: lens, i, k
 lens = len_trim(s)
 do i = 1, lens                         ! Формируем массив номеров команд, которые
  k= int2(ichar(s(i:i)))                ! нужно выполнить для вывода строки с текстом
  if(k > 47 .and. k < 58) then          ! Цифры 0, 1, ..., 9
   ars(i) = k - 47
  else if(k > 119 .and. k < 123) then   ! Буквы x, y и z
   ars(i) = k - 109
  else if(k == 43 .or. k == 45 .or. k == 46) then
   ars(i) = k - 29                      ! Знаки +, - и .
  else if(k == 101) then                ! Буква e
   ars(i) = k - 83
  else
   ars(i) = 15                          ! Пробел
  end if
 end do
 call fglPushAttrib(gl_list_bit)        ! Сохраняем атрибуты изображения
 ! Номер выполняемой команды будет равен ars(i)
 call fglCallLists(lens, gl_short, loc(ars))
 call fglPopAttrib( )                   ! Восстанавливаем атрибуты изображения
```

```
 end subroutine printString

end module fitest

subroutine myinit( )
 use fitest
 call fglShadeModel(gl_flat)              ! Вывод без интерполяции цветов
 call makeFigures( )                      ! Формирование списков команд, пригодных
end subroutine myinit                     ! для вывода текста

subroutine display( )
 !ms$ attributes stdcall, alias : '_display@0' :: display
 use fitest                               ! Подготовка к отображению строки текста
 real(4) :: black(3) = (/ 0.0, 0.0, 0.0 /)  ! и вызов подпрограммы вывода строки
 call fglClearColor(1.0, 1.0, 1.0, 1.0)   ! Цвет фона - белый
 call fglClear(gl_color_buffer_bit)       ! Очистка буфера цвета
 call fglColor3fv(loc(black))
 ! Меняем позицию вывода (растровую позицию)
 call fglRasterPos2i(20, 20)
 call printString("01234567890 xyz+-.e")
 call fglFlush( )
end subroutine display

subroutine myReshape(w, h)
 !ms$ attributes stdcall, alias : '_myReshape@8' :: myReshape
 use opengl
 integer(4) :: w, h
 call fglViewport(0, 0, w, h)
 call fglMatrixMode(gl_projection); call fglLoadIdentity( )
 call fglOrtho(0.0, w, 0.0, h, -1.0, 1.0)
end subroutine myReshape

program outtext                           ! Задание окна, режима воспроизведения,
 use opengl                               ! инициализация и обработка событий
 use msfwin                               ! Событие - это изменение размеров окна
 integer(4) :: result
 interface                                ! Подпрограммы myReshape и display
  subroutine myReshape(w, h)              ! должны обладать атрибутом EXTENAL
   !ms$ attributes stdcall, alias : '_myReshape@8' :: myReshape
   integer(4) :: w, h
  end subroutine myReshape
  subroutine display( )
   !ms$ attributes stdcall, alias : '_display@0' :: display
  end subroutine display
 end interface
 call fauxInitDisplayMode(aux_single .or. aux_rgb)
 call fauxInitPosition(0, 0, 300, 100)
```

```
result = fauxInitWindow("Проба текста "C)
call myinit( )                          ! Формирование списка команд
call fauxReshapeFunc(loc(myReshape))
call fauxMainLoop(loc(display))
end program outtext                     ! Результат приведен на рис. П.-2.4
```

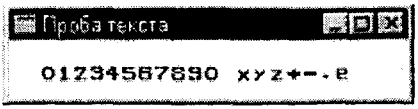


Рис. П.-2.4. Символы модуля fitest

П.-2.4.3. СОЗДАНИЕ ОБРАЗЦА

Образец, в соответствии с которым заполняются выводимые многоугольники, должен быть предварительно определен.

В качестве образца OpenGL использует битовый массив, содержащий 32×32 бита. Для его представления в программе задается массив типа INTEGER(1) из 128 элементов. Каждые 4 элемента массива содержат 32 бита и, следовательно, определяют одну строчку образца.

Чтобы задать образец, нарисуем его отображающий незаполненный шаблон - 32 строки, содержащие по 32 нуля каждая (рис. П.-2.5), и в клеточках, которые должны заполняться, проставим число 1.

Чтобы перевести созданный макет образца в элементы массива, учтем, что каждая строчка макета должна быть представлена четырьмя числами типа INTEGER(1). Тогда для представления, например, трех первых строчек, потребуется задать последовательность чисел (используем двоичную систему счисления):

```
2#00000000,2#00000000,2#00000000,2#00000000,
2#00000000,2#00000001,2#10000000,2#00000000,
2#00000000,2#00000110,2#01100000,2#00000000
```

```
00000000000000000000000000000000
00000000000000011000000000000000
00000000000001100110000000000000
00000000001100000011000000000000
00000000110000000000110000000000
00000000001100000011000000000000
00000000000001100110000000000000
00000000000000011000000000000000
00000000000000011000000000000000
10000000000000011000000000000001
01100000000000011000000000000110
00011000000111111111100000011000
00000110000000000000000001100000
00000011000000000000000110000000
00000000110000000000110000000000
00000000001100000011000000000000
00000000000001100110000000000000
00000000000000011000000000000000
00000000000000011000000000000000
00000000000000011000000000000000
00001111111100011000111111110000
```

Продолжая таким же образом, можно выписать все 128 элементов образца. Однако код будет более компактным, если вместо двоичного представления использовать, например, шестнадцатеричное. Тогда приведенные 3 строчки запишутся так:

#00, #00, #00, #00, #00, #01, #80, #00, #00,
#06, #60, #00

```
00000001100000011000000110000000
00000001100000011000000110000000
00000001100000011000000110000000
00000001100000011000000110000000
00000001100000011000000110000000
00000110011000011000011001100000
00011000000110011001100000011000
00000110011000000000011001100000
00000001100000000000000110000000
00000000000000000000000000000000
```

Рис. П.-2.5. Макет битового образца

Программа, переводящая двоичное представление образца в шестнадцатеричное, может выглядеть так:

```
program pattern
 integer(4), parameter :: n = 128          ! axis - массив с макетом битового образца
 integer(1), dimension(n) :: axis = (/                                          &
 2#00000000, 2#00000000, 2#00000000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#00000000, 2#00000110, 2#01100000, 2#00000000,                                &
 2#00000000, 2#00011000, 2#00011000, 2#00000000,                                &
 2#00000000, 2#01100000, 2#00000110, 2#00000000,                                &
 2#00000000, 2#00011000, 2#00011000, 2#00000000,                                &
 2#00000000, 2#00000110, 2#01100000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#10000000, 2#00000001, 2#10000000, 2#00000001,                                &
 2#01100000, 2#00000001, 2#10000000, 2#00000110,                                &
 2#00011000, 2#00001111, 2#11110000, 2#00011000,                                &
 2#00000110, 2#00000000, 2#00000000, 2#01100000,                                &
 2#00000001, 2#10000000, 2#00000001, 2#10000000,                                &
 2#00000000, 2#01100000, 2#00000110, 2#00000000,                                &
 2#00000000, 2#00011000, 2#00011000, 2#00000000,                                &
 2#00000000, 2#00000110, 2#01100000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#00000000, 2#00000001, 2#10000000, 2#00000000,                                &
 2#00001111, 2#11110001, 2#10001111, 2#11110000,                                &
 2#00000001, 2#10000001, 2#10000001, 2#10000000,                                &
 2#00000001, 2#10000001, 2#10000001, 2#10000000,                                &
```

```
2#00000001, 2#10000001, 2#10000001, 2#10000000,                    &
2#00000001, 2#10000001, 2#10000001, 2#10000000,                    &
2#00000001, 2#10000001, 2#10000001, 2#10000000,                    &
2#00000110, 2#01100001, 2#10000110, 2#01100000,                    &
2#00011000, 2#00011001, 2#10011000, 2#00011000,                    &
2#00000110, 2#01100000, 2#00000110, 2#01100000,                    &
2#00000001, 2#10000000, 2#00000001, 2#10000000,                    &
2#00000000, 2#00000000, 2#00000000, 2#00000000 /)
do i = 1, n                                  ! Вывод без продвижения
 write(*, fmt = '(1x, a, z2.2, a)', advance = 'no') '#', axis(i), ','
end do
end program pattern
```

По результатам ее работы сформируем модуль *pattern*, содержащий в массиве *mask* описание приведенного на рис. П.-2.5 образца.

```
module pattern
 integer(1), dimension(128) :: mask = (/                                  &
 #00, #00, #00, #00, #00, #01, #80, #00, #00, #06, #60, #00, #00, #18, #18, #00, &
 #00, #60, #06, #00, #00, #18, #18, #00, #00, #06, #60, #00, #00, #01, #80, #00, &
 #00, #01, #80, #00, #80, #01, #80, #01, #60, #01, #80, #06, #18, #0F, #F0, #18, &
 #06, #00, #00, #60, #01, #80, #01, #80, #00, #60, #06, #00, #00, #18, #18, #00, &
 #00, #06, #60, #00, #00, #01, #80, #00, #00, #01, #80, #00, #00, #01, #80, #00, &
 #00, #01, #80, #00, #0F, #F1, #8F, #F0, #01, #81, #81, #80, #01, #81, #81, #80, &
 #01, #81, #81, #80, #01, #81, #81, #80, #01, #81, #81, #80, #06, #61, #86, #60, &
 #18, #19, #98, #18, #06, #60, #06, #60, #01, #80, #01, #80, #00, #00, #00, #00 /)
end module pattern
```

Для проверки созданного образца можно воспользоваться программой *patternTest*, заполняющей прямоугольник по содержащемуся в образце шаблону. Ее текст:

```
program patternTest
 use opengl
 use msfwin
 use pattern                                 ! Ссылка на модуль, содержащий образец
 integer(4) :: result, w = 210, h = 110      ! Размеры окна вывода
 ! Координаты вершин четырехугольника
 real(4), dimension(3) ::   v11 = (/ -100.0, -50.0, -50.0 /),          &
                            v12= (/ 100.0, -50.0, 50.0 /),             &
                            v13 = (/ 100.0, 50.0, 50.0 /),             &
                            v14 = (/ -100.0, 50.0, -50.0 /)
 ! Работаем с одним буфером (AUX_SINGLE) в режиме RGBA (AUX_RGBA)
 call fauxInitDisplayMode(ior(aux_single, aux_rgba))
 call fauxInitPosition(200, 200, w, h)       ! Начальное окно вывода
 result = fauxInitWindow('Проверка образца 'C)
 call fglClearColor(1.0, 1.0, 1.0, 1.0)      ! Цвет фона - ярко-белый
```

```
call fglClear(gl_color_buffer_bit)          ! Очистка буфера цвета
call fglShadeModel(gl_flat)                 ! Вывод без интерполяции цветов
call fglMatrixMode(gl_modelview)            ! Текущей стала видовая матрица
call fglLoadIdentity( )                     ! Ее инициализация
call fglMatrixMode(gl_projection)           ! Текущей стала матрица проецирования
call fglLoadIdentity( )                     ! Инициализация матрицы проецирования
! Задаем трехмерную область видимости
call fglOrtho(-w / 2, w / 2, -h / 2, h / 2, -w / 2, w / 2)
call fglColor3f(0.0, 0.0, 0.0)              ! Текущий цвет - черный
! Лицевые стороны выводятся заполненными
call fglPolygonMode(gl_front, gl_fill)
! Задаем образец (он содержится в массиве mask)
call fglPolygonStipple(loc(mask))
! Активизируем режим заполнения по образцу
call fglEnable(gl_polygon_stipple)
call fglBegin(gl_quads)
 ! Вывод прямоугольника. Вершины обходятся против часовой стрелки
 call fglVertex3fv(loc(v11));    call fglVertex3fv(loc(v12))
 call fglVertex3fv(loc(v13));    call fglVertex3fv(loc(v14))
call fglEnd( )
call fglFlush( )                            ! Отображаем буфер кадра на экране
call fauxMainLoop(null)                     ! Оставляем окно открытым
end program patternTest                     ! Результат приведен на рис. П.-2.6
```

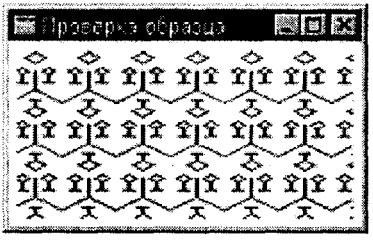


Рис. П.-2.6. Проверка образца mask

Замечание. Выводимый прямоугольник заполняется по образцу, заданному массивом *mask*, поскольку в программе присутствуют команды

```
! Задаем образец (он содержится в массиве mask)
call fglPolygonStipple(loc(mask))
! Активизируем режим заполнения по образцу
call fglEnable(gl_polygon_stipple)
```

и ссылка на модуль *pattern*, содержащий образец.

Приложение 3. ДЛЯ ПОЛЬЗОВАТЕЛЕЙ QUICKWIN

В случае многооконного графического вывода QuickWin при обработке событий, например связанных с мышью, происходящих на фоне продолжительных вычислений, необходимо для процедуры, реализующей вычисления, и для процедуры обработки событий выделить различные нити. В противном случае, если нити не используются, процедура, выполняющая вычисления, блокирует процедуру обработки событий, т. е. делает приложение неработоспособным. На это обстоятельство обратил внимание сотрудник Ленинградской АЭС Пименов А. (email ort2-pan@laes.sbor.ru).

Рассмотрим возможный вариант реализации приведенных положений на *примере*, в котором открываются два окна QuickWin. В первом можно строить полилинии, указывая мышью очередную точку, со вторым связан некоторый вычислитель, выполняющий помимо расчетов вывод результатов в текстовой файл; о работе вычислителя в окне выводятся сообщения. Построение полилинии можно осуществлять после выбора пункта меню "Новая полилиния", а запуск процедуры-вычислителя выполняется в результате выбора пункта меню "Работа с файлом". Полилиния строится по точкам, координаты которых вводятся мышью в окне "Построение полилинии". Возможное состояние окон отображено на рис. П.-3.1.

Рис. П.-3.1. Приложение с двумя нитями

Для обеспечения нормального функционирования приложения процедура *newpline*, в которой обрабатываются события - нажатия левой кнопки мыши, и процедура-вычислитель *runfile* реализуются в виде отдельных нитей; для доступа к последовательному ресурсу - экрану монитора - употреб-

ляются критические секции. Заметим, что реализация процедуры-вычислителя без выделения нити сделает приложение неработоспособным.

Текст программы двуоконного приложения с нитями и критическими секциями:

```
module dat
 use dflib
 use mt                                  ! Модуль, содержащий тип rtl_critical_section
 integer(4) :: npoints                   ! Число введенных точек
 integer(4) :: res, h1, h2, j1, j2, obj
 integer(4), parameter :: unit1 = 10, unit2 = 20
 type(rtl_critical_section) rts
end module dat

program go
 use dat
 external points
 call InitializeCriticalSection(loc(rts))
 obj = CreateMutex(0, .true., 0)         ! Создаем объект-исключение
 npoints = 0
 open(unit1, file ='user', title = 'Построение полилинии')
 ! Оператор включен для отображения дочернего окна
 call clearscreen($gclearscreen)
 open(unit2, file ='user', title = 'Работа с файлом')
 call clearscreen($gclearscreen)         ! Отобразим окно устройства unit2
 rest = clickmenuqq(qwin$tile)           ! Имитация выбора подпункта меню Tile
 h1 = CreateThread(0, 0, points, %val(unit1), create_suspended, j1)
 ! Предотвратим немедленное завершение программы
 do
  call sleepqq(1000000)
 end do
end program go

function initialsettings( )              ! Инициализация QuickWin - формирование
 use dflib                               ! пользовательского меню
 logical(4) :: initialsettings, flag
 external newpline, wfile                ! Связанные с пунктами меню подпрограммы
 flag = appendmenuqq(1, $menuenabled, 'Новая полилиния'c, newpline)
 flag = appendmenuqq(2, $menuenabled, 'Работа с файлом'c, wfile)
 flag = appendmenuqq(3, $menuenabled, 'Выход'c, winexit)
 initialsettings = .true.
end function initialsettings

! fl - обязательный параметр для процедуры меню
subroutine newpline(fl)
 use dat                                 ! Новая полилиния
```

```
  logical(4) :: fl, ltemp
  ! Пункт меню - неактивен
  res = modifymenuflagsqq(1, 0, $menugrayed)
  res = ResumeThread(h1)                    ! Запуск 1-й нити
  ltemp = fl                                ! Предотвратим сообщения компилятора
end subroutine newpline

subroutine points(u1)
  use dat
  integer(4) :: u1
  !dec$attributes value :: u1
  external plines
  res = registermouseevent(u1, mouse$lbuttondown, plines)
  npoints = 0
  res = setactiveqq(u1)                     ! Направим вывод на окно unit1
  res = focusqq(u1)                         ! Переместим окно unit1 в фокус
  ! Заполнение окна цветом фона (черным цветом)
  call clearscreen($gclearscreen)
end subroutine points

subroutine plines(u1, mevent, keystate, mxpos, mypos)
  use dat                                   ! Вывод отрезка прямой
  integer(4) :: u1, mevent, keystate, mxpos, mypos
  integer(2) :: status2, mpixel = 2
  ! Координаты для вывода вершины
  integer(2) :: lcx, lcy, rcx, rcy
  type(xycoord) pt                          ! Для выполнения подпрограммы MOVETO
  res = mevent; res = keystate              ! Для подавления предупреждений компилятора
  res = setactiveqq(u1)                     ! Направляем вывод на окно unit
  lcx = max(mxpos - mpixel, 0); lcy = max(mypos - mpixel, 0)
  rcx = mxpos + mpixel; rcy = mypos + mpixel
  ! Точку - красным цветом
  res = setcolorrgb(rgbtointeger(255, 0, 0))
  ! Вывод вершины. Обеспечиваем безконфликтный доступ к экрану монитора
  call EnterCriticalSection(loc(rts))
  status2 = ellipse($gfillinterior, lcx, lcy, rcx, rcy)
  ! Примитив - ярко-белым цветом
  res = setcolorrgb(rgbtointeger(255, 255, 255))
  npoints = npoints + 1                     ! Число введенных точек
  if(npoints > 1) then
   ! Вывод отрезка прямой
   status2 = lineto(int2(mxpos), int2(mypos))
  else
   ! Перемещение позиции вывода в первую введенную точку
   call moveto(int2(mxpos), int2(mypos), pt)
  end if
```

```
  call LeaveCriticalSection(loc(rts))
 end subroutine plines

! fl - обязательный параметр для процедуры меню
subroutine wfile(fl)
  use dflib
  use dat
  logical(4) :: fl, ltemp                    ! Работа с файлом
  type(xycoord) :: pos
  external runfile
  ! Пункт меню - неактивен
  res = modifymenuflagsqq(2, 0, $menugrayed)
  h2 = CreateThread(0, 0, runfile, %val(unit2), 0, j2)
  ltemp = fl                                 ! Предотвратим сообщения компилятора
end subroutine wfile

subroutine runfile(u2)       ! Вычислитель, связанный с окном 1,
  use dflib                  ! выполняющий достаточно трудоемкие операции
  use dat
  integer(4) :: u2
  !dec$attributes value :: u2
  real(4) :: tm                              ! Время работы с файлом в минутах
  integer(4) :: i, j, k, n =10
  real(4) :: et, st, m1(10, 10) = 1, m2(10, 10) = 1, m3(10, 10) = 3
  character(30) :: say, iter
  logical(4) :: lret
  type(xycoord) :: pos
  res = setactiveqq(u2); res = focusqq(u2)
  call EnterCriticalSection(loc(rts))        ! Обеспечиваем бесконфликтный доступ
  call clearscreen($gclearscreen)            ! к экрану монитора
  call LeaveCriticalSection(loc(rts))
  numfonts = initializefonts( )              ! Инициализация шрифтов
  fontnum = setfont('t"arial"h15w8i')        ! Установим шрифт ARIAL
  call cpu_time(st)
  do i = 1, n
   open(1, file = 'a.txt')
   write(iter, *) i
   say = 'Writing ' // adjustl(iter)
   call EnterCriticalSection(loc(rts))       ! Обеспечиваем бесконфликтный доступ
   call clearscreen($gclearscreen)           ! к экрану монитора
   call moveto(int2(5), int2(30), pos)
   call outgtext(trim(say))                  ! Вывод сообщения, например Writing 1
   call LeaveCriticalSection(loc(rts))
   call sleepqq(100)
   do j = 1, 1000                            ! Некоторые вычисления
    m3 = matmul(m1, m2)
```

```
    rewind 1
    write(1, *) m3
   end do
   say = 'Reading ' // adjustl(iter)
   call EnterCriticalSection(loc(rts))     ! Обеспечиваем бесконфликтный доступ
   call clearscreen($gclearscreen)         ! к экрану монитора
   call moveto(int2(5), int2(30), pos)
   call outgtext(trim(say))                ! Вывод сообщения, например Reading 1
   call LeaveCriticalSection(loc(rts))
   call sleepqq(100)
   do j = 1, 1000
    rewind 1
    read(1, *) m3
   end do
   close(1)
  end do
  call cpu_time(et)
  open(1, file = 'a.txt')
  tm = (et - st) / 60.0                    ! Время вычислений
  write(1, *) tm
  close(1)
  write(say, '(f8.2)') tm
  say = trim(say) // ' ' // 'mn'
  call EnterCriticalSection(loc(rts))      ! Обеспечиваем бесконфликтный доступ
  call clearscreen($gclearscreen)          ! к экрану монитора
  call moveto(int2(5), int2(30), pos)
  call outgtext(say)                       ! Вывод времени, затраченного на вычисления
  call LeaveCriticalSection(loc(rts))
  res = modifymenuflagsqq(2, 0, $menuenabled)
  lret = CloseHandle(h2)
end subroutine runfile
```

Приложение 4. ВЫЗОВ ФОРТРАНА ИЗ VISUAL BASIC

Описанный в [2] механизм соединения Фортрана с Visual Basic непригоден при работе с Microsoft Visual Basic версий 5 и выше. На это обратил внимание Луговский Алексей (email Lugovsky@online.ru). Ниже приводится его письмо, в котором излагается техника построения приложения Visual Basic версии 6.0 (VB6), вызывающего процедуры Фортрана.

"Здравствуйте, уважаемый Олег Васильевич! С удовольствием делюсь своими знаниями:

- создаем в FPS4.0 проект dll;
- после названия функции или процедуры вставляем строки

```
! Используем знак | для обозначения "или"
!ms$attributes dllexport :: <имя функции | процедуры>
!ms$attributes alias:'<имя функции | процедуры>' :: <имя функции | процедуры>
```

- в начале модуле VB6, где объявляются глобальные переменные, пишем:

```
Declare Function(Sub) <имя функции или процедуры> Lib _
  "<путь к dll>" (<список переменных>) [As <тип функции>]
```

Создадим в качестве примера dll-проект *primer* с файлом primer.f90. В файле primer.f90 пишем:

```
function primer1(x, t)
 !ms$attributes dllexport :: primer1
 !ms$attributes alias:'primer1' :: primer1
 real(8) primer1, x, t
 . . .
 primer1 = ...
end function primer1

subroutine primer2(u, n)
 !ms$attributes dllexport :: primer2
 !ms$attributes alias:'primer2' :: primer2
 integer n
 real(8) u(0:n)
 ...
end subroutine primer2
```

Компилируем dll-проект и создаем файл primer.dll; в соответствующем модуле VB6 пишем:

```
Declare Function primer1 Lib "c:\projects\primer\debug\primer.dll" _
    (x As Double, t As Double) As Double
Declare Sub primer2 Lib "c:\projects\primer\debug\primer.dll" _
    (u As Double, n As Long)
```

Путь, конечно, можно указать свой.
Луговский Алексей."

ЛИТЕРАТУРА

1. Амосов А. А., Дубинский Ю. А., Копченова Н. В. Вычислительные методы для инженеров. - М.: Высш. шк., 1994. - 544 с.
2. Бартеньев О. В. Visual Fortran: Новые возможности. - М.: Диалог-МИФИ, 1999. - 288 с.
3. Он же. Графика OpenGL: программирование на Фортране. - М.: Диалог-МИФИ, 2000. - 368 с.
4. Он же. Современный Фортран. - М.: Диалог-МИФИ, 2000. - 448 с.
5. Он же. Фортран для профессионалов. Математическая библиотека IMSL: (Ч. 1). - М.: Диалог-МИФИ, 2000. - 448 с.
6. Дэннис Дж., мл., Шнабель Р. Численные методы безусловной оптимизации и решения нелинейных уравнений. - М.: Мир, 1988. - 440 с.
7. Каханер Д., Моулер К., Нэш С. Численные методы и математическое обеспечение. - М.: Мир, 1988. - 575 с.
8. Aird T. J., Rice J. R. Systematic search in high dimensional sets//SIAM Journal on Numerical Analysis. № 14. P. 296-312, 1977.
9. Brent R. P. An algorithm with guaranteed convergence for finding a zero of a function, The Computer Journal. № 14. P. 422-425, 1971.
10. Cooley J. W., Tukey J. W. An algorithm for the machine computation of complex Fourier series. Mathematics of Computation, 19, 297-301, 1965.
11. Crump K. S. Numerical inversion of Laplace transforms using a Fourier series approximation. Journal of the Association for Computing Machinery, 23, 89-96, 1976.
12. Garbow B. S., Giunta G., Lyness J. N., Murli A. Software for an implementation of Weeks' method for the inverse Laplace transform problem. ACM Transactions of Mathematical Software, 14, 163-170, 1988.
13. Gay D. M. Computing optimal locally constrained steps, SIAM Journal on Scientific and Statistical Computing , 2, 186-197, 1981.
14. Gill P. E., Murray W. Minimization subject to bounds on the variables, NPL Report NAC 72, National Physical Laboratory, England, 1976.
15. Gill P. E., Murray W., Wright M. H. Practical Optimization. New York: Academic Press, 1981.
16. Gill P. E., Murray W., Saunders M. A., Wright M. H. Model building and practical aspects of nonlinear programming, in Computational Mathematical Programming, (edited by Schittkowski K.), NATO ASI Series, 15, Springer-Verlag, Berlin, Germany, 1985.
17. Goldfarb D., Idnani A. A numerically stable dual method for solving strictly convex quadratic programs, Mathematical Programming, 27, 1-33, 1983.
18. Griffin R., Redish K. A. Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, Communications of the ACM, 13, 54, 1970.

19. Hanson R. J. Least squares with bounds and linear constraints, SIAM Journal on Scientific and Statistical Computing, 7; № 3, 1986.
20. Hoog F. R., Knight J. H., Stokes A. N. An improved method for numerical inversion of Laplace transforms. SIAM Journal on Scientific and Statistical Computing, 3, 357-366, 1982.
21. Jenkins, M. A. Algorithm 493: Zeros of a real polynomial, ACM Transactions on Mathematical Software, 1, 178-189, 1975.
22. Jenkins M. A., Traub. J. F. A three-stage algorithm for real polynomials using quadratic iteration, SIAM Journal on Numerical Analysis, 7, 545-566, 1970.
23. Jenkins M. A., Traub. J. F. Zeros of a complex polynomial, Communications of the ACM, 15, 97-99, 1972.
24. Knuth D. E. The Art of Computer Programming. V. 3: Sorting and Searching. Addison-Wesley Publishing Company, Reading, Mass, 1973.
25. Levenberg K. A method for the solution of certain problems in least squares, Quarterly of Applied Mathematics, 2, 164-168, 1944.
26. Lyness J.N. Giunta G. A modification of the Weeks Method for numerical inversion of the Laplace transform. Mathmetics of Computation, 47, 313-322, 1986.
27. Marquardt D. An algorithm for least-squares estimation of nonlinear parameters, SIAM Journal on Applied Mathematics, 11, 431-441, 1963.
28. More J., Burton G., Kenneth H. User guide for MINPACK-1, Argonne National Labs Report ANL-80-74, Argonne, Illinois, 1980.
29. Muller D. E. A method for solving algebraic equations using an automatic computer, Mathematical Tables and Aids to Computation, 10, 208-215, 1965.
30. Murtagh B. A. Advanced Linear Programming: Computation and Practice. New York: McGraw-Hill, , 1981.
31. Murty K. G. Linear Programming. New York: John Wiley and Sons, 1983.
32. Nelder J. A., Mead R. A simplex method for function minimization, Computer Journal 7, 308-313, 1965.
33. Petro R. Remark on Algorithm 347: An efficient algorithm for sorting with minimal storage, Communications of the ACM, 13, 624, 1970.
34. Powell M. J. D. Restart procedures for the conjugate gradient method, Mathematical Programming, 12, 241-254, 1977.
35. Powell M. J. D. A fast algorithm for nonlinearly constrained optimization calculations, in Numerical Analysis Proceedings, Dundee 1977, Lecture Notes in Mathematics, (edited by Watson G. A.), 630, Springer-Verlag, Berlin, Germany, 144-157, 1978.
36. Powell M. J. D. ZQPCVX a FORTRAN subroutine for convex quadratic programming, DAMTP Report NA17, Cambridge, England, 1983.

37. Powell M. J. D. On the quadratic programming algorithm of Goldfarb and Idnani, Mathematical Programming Study, 25, 46-61, 1985.
38. Powell M. J. D. A tolerant algorithm for linearly constrained optimization calculations, DAMTP Report NA17, University of Cambridge, England, 1988.
39. Powell M. J. D. TOLMIN: A fortran package for linearly constrained optimization calculations, DAMTP Report NA2, University of Cambridge, England, 1989.
40. Schittkowski K. On the convergence of a sequential quadratic programming method with an augmented Lagrangian line search function, Mathematik Operationsforschung und Statistik, Serie Optimization, 14, 197-216, 1983.
41. Schittkowski K. NLPQL: A FORTRAN subroutine solving constrained nonlinear programming problems, (edited by Clyde L. M.), Annals of Operations Research, 5, 485-500, 1986.
42. Schittkowski K. More test examples for nonlinear programming codes, SpringerVerlag, Berlin, 74, 1987.
43. Schnabel R. B. Finite Difference Derivatives - Theory and Practice, Report, National Bureau of Standards, Boulder, Colorado, 1985.
44. Singleton R.C. Algorithm 347: An efficient algorithm for sorting with minimal storage, Communications of the ACM, 12, 185-187, 1969.
45. Smith B. T. ZERPOL. A Zero Finding Algorithm for Polynomials Using Laguerre's Method, Department of Computer Science, University of Toronto, 1967.
46. Stoer J. Principles of sequential quadratic programming methods for solving nonlinear programs, in Computational Mathematical Programming, (edited by Schittkowski K.), NATO ASI Series, 15, Springer-Verlag, Berlin, Germany, 1985.
47. Weeks W. T. Numerical inversion of Laplace transforms using Laguerre functions. J. ACM, 13, 419-429, 1966.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

P

C

ОГЛАВЛЕНИЕ