Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»

Факультет информационных технологий и управления

Кафедра вычислительных методов и программирования

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ (ЯЗЫК С/С++). ЛАБОРАТОРНЫЙ ПРАКТИКУМ

В двух частях
Часть 2

Рекомендовано УМО по образованию в области информатики и радиоэлектроники в качестве учебно – методического пособия для всех специальностей I ступени высшего образования, закрепленных за УМО

Минск БГУИР 2018

УДК 004.42(076.5)
ББК 32.973.26-018.2 я 73
О–75

Авторы:

С. А. Беспалов, А. В. Гуревич, Т. М. Кривоносова, Т. А. Рак,
В. Л. Смирнов, О. О. Шатилова, В. П. Шестакович

Рецензенты:

кафедра экономической информатики государственного учреждения образования «Белорусский государственный экономический университет» (протокол №7 от 16.02.2017);

доцент кафедры автоматизированных систем управления производством учреждения образования «Белорусский государственный аграрный технический университет», кандидат технических наук, доцент И.П. Матвеенко.

О–75 **Основы алгоритмизации и программирования (язык С/С++). лабораторный практикум в 2 ч. Ч. 2 : учеб. – метод. пособие / Беспалов С. А. [и др.] . – Минск: БГУИР, 2018. – 114 с.: ил.**

ISBN 978-985-543-388-1.

Приведены краткие теоретические сведения по алгоритмам обработки динамических структур данных (линейные и нелинейные списки), алгоритмам сортировки и поиска, некоторые методы приближенных вычислений, а также примеры их реализации на языке С/С++. Состоит из 12 лабораторных работ и индивидуальных заданий к ним.

Часть 1 издана в БГУИР в 2017 году (авторы: С. А. Беспалов, И. Е. Зайцева, Т. М. Кривоносова, Т. А. Рак, В. Л. Смирнов, О. О. Шатилова, В. П. Шестакович).

УДК 004.42(076.5)
ББК 32.973.26-018.2я73

ISBN 978-985-543-388-1(ч.2)
ISBN 978-985-543-241-9

© УО «Белорусский государственный университет информатики и радиоэлектроники», 2018

СОДЕРЖАНИЕ

Лабораторная работа №1. Рекурсивные функции

Цель работы: изучить способы реализации алгоритмов с использованием рекурсии.

1.1. Краткие теоретические сведения

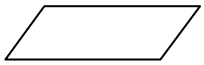
Рекурсия – это способ организации вычислительного процесса, при котором функция в ходе выполнения входящих в нее операторов обращается сама к себе. Классическим примером является вычисление факториала *n*! (*n*>0):

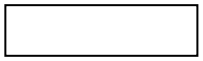
```
double Faktorial_R (int n)  {
 if (n < 2) return 1;                        // Условие окончания рекурсии
 else
return n* Faktorial_R (n – 1);    // Рекурсивное обращение к функции
}
```

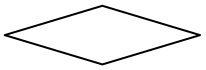
При выполнении правильно организованной рекурсивной функции осуществляется последовательный переход от текущего уровня организации алгоритма к нижнему уровню, в котором будет получено решение задачи (в приведенном примере при *n* < 2), не требующее дальнейшего обращения к функции (не рекурсивное).

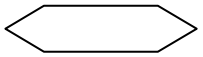
При описании алгоритмов используем следующие стандартные фигуры блок-схем:

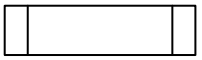

– начало функции (заголовок) и конец функции (выход и возвращение в точку вызова);


– ввод/вывод данных;


– вычислительный процесс (операция присваивания);


– блок сравнения (операторы *if, switch-case*);


– заголовок оператора цикла;


– обращение к функции пользователя.

1.2. Пример выполнения задания

Написать программу вычисления факториала ***положительного*** числа ***n***, содержащую функции пользователя с рекурсией и без рекурсии (рис. 1.1).

```
//------------------ Функция без рекурсии ---------------------------------------
double Faktorial(int n)  {
   double f = 1;
for (int i = 1; i <= n; i++) f  *=  i;
return f;
}
//------------------- Рекурсивная функция ----------------------------------------
```

```
double Faktorial_R(int n)  {
    if (n < 2) return 1;
    else
return n*Faktorial_R(n-1);
}
double Faktorial(int);
double Faktorial_R(int);
void main(void)
{
int n, kod;
while(true) {                    // Бесконечный цикл с выходом по default
cout << "\n\tInput n " ;
cin >> n;
cout << "\n Recurs – 0\n Simple – 1\n Else – Exit" << endl ;
cin >> kod;
switch(kod) {
            case 0:
                  cout << "\t Recurs = " << Faktorial_R(n) << endl;
            break;
            case 1:
                  cout << "\t Simple = " << Faktorial(n) << endl;
            break;
            default: return;
      }
}
```

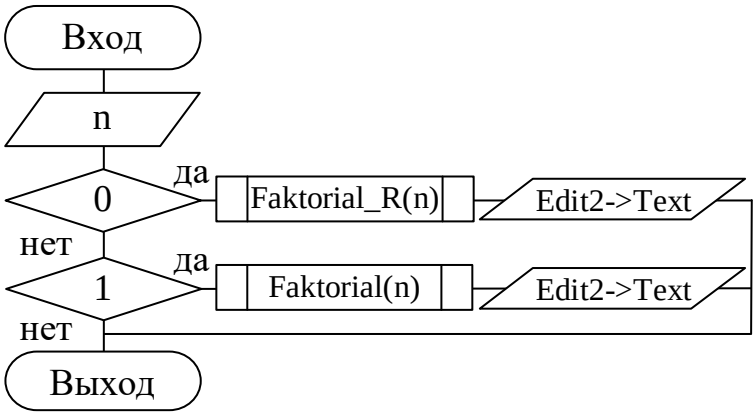


Рис. 1.1. Блок – схема вызова рекурсивной и нерекурсивной функции

Блок-схемы функций пользователя *Faktorial_R* и *Faktorial* представлены на рис. 1.2.

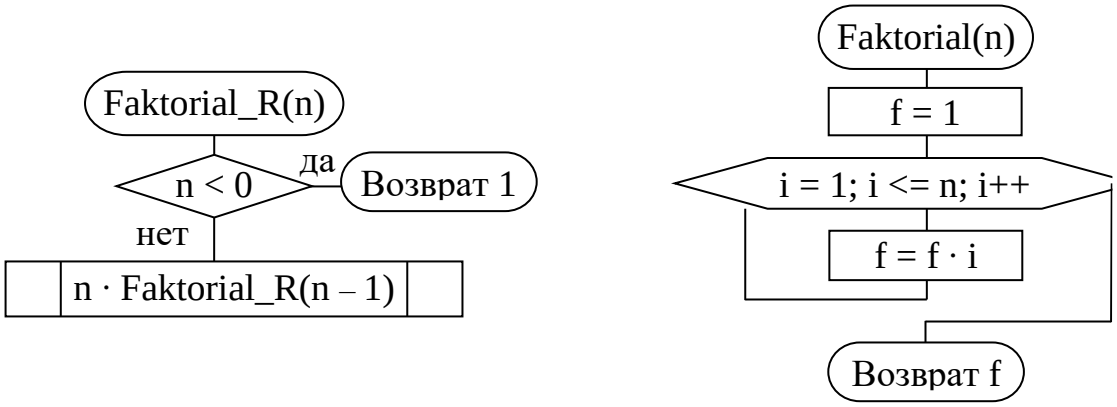


Рис. 1.2

1.3. Индивидуальные задания

Составить алгоритм в виде блок-схемы, написать и отладить поставленную задачу с использованием рекурсивной и обычной функций. Сравнить полученные результаты.

1. Для заданного целого десятичного числа N получить его представление в p-ичной системе счисления ($p < 10$).

2. В упорядоченном массиве целых чисел a_i ($i = 1, ..., n$) найти номер находящегося в массиве элемента c, используя метод двоичного поиска.

3. Найти наибольший общий делитель чисел M и N, используя теорему Эйлера: если M делится на N, то $НОД\ (N, M) = N$, иначе $НОД\ (N, M) = (M\%N, N)$.

4. Числа Фибоначчи определяются следующим образом: $Fb(0) = 0$; $Fb(1) = 1$; $Fb(n) = Fb(n – 1) + Fb(n – 2)$. Определить $Fb(n)$.

5. Найти значение функции Аккермана $A(m, n)$, которое определяется для всех неотрицательных целых аргументов m и n следующим образом:

$A(0, n) = n + 1$;

$A(m, 0) = A(m – 1, 1)$ при $m > 0$;

$A(m, n) = A(m – 1, A(m, n – 1))$ при $m > 0$ и $n > 0$.

6. Найти методом деления отрезка пополам минимум функции $f(x) = 7\sin^2(x)$ на отрезке [2, 6] с заданной точностью ε (например, 0.01).

7. Вычислить значение $x = \sqrt{a}$, используя рекуррентную формулу $x_n = \frac{1}{2}\left(x_{n-1} + \frac{a}{x_{n-1}} \right)$, в качестве начального значения использовать $x_0 = 0{,}5(1 + a)$.

8. Найти максимальный элемент в массиве a_i (i=1, …, n), используя очевидное соотношение $\max(a_1, \ldots, a_n) = \max[\max(a_1, \ldots, a_{n-1}), a_n]$.

9. Вычислить значение $y(n) = \sqrt{1+\sqrt{2+\ldots+\sqrt{n}}}$.

10. Найти максимальный элемент в массиве a_i (i=1, …, n), используя метод деления пополам $\max(a_1,\ldots, a_n) = \max[\max(a_1,\ldots, a_{n/2}), \max(a_{n/2+1}, \ldots, a_n)]$.

11. Вычислить значение $y(n) = \cfrac{1}{n+\cfrac{1}{(n-1)+\cfrac{1}{(n-2)+\cfrac{1}{\cfrac{\ldots}{\ldots+\cfrac{1}{1+\cfrac{1}{2}}}}}}}$.

12. Вычислить произведение четного количества n ($n \geq 2$) сомножителей следующего вида: $y = \left(\frac{2}{1}\cdot\frac{2}{3}\right)\cdot\left(\frac{4}{3}\cdot\frac{4}{5}\right)\cdot\left(\frac{6}{5}\cdot\frac{6}{7}\right)\cdot\ldots$.

13. Вычислить $y = x^n$ по следующему правилу: $y = (x^{n/2})^2$, если n четное, и $y = x \cdot y^{n-1}$, если n нечетное.

14. Вычислить значение $C_n^k = \frac{n!}{k!(n-k)!}$ (значение 0! = 1).

15. Вычислить $y(n) = \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\ldots}}}}$, n задает число ступеней.

16. В заданном массиве заменить все числа, граничащие с цифрой «1», нулями.

1.4. Контрольные вопросы

1. Какая функция называется рекурсивной?

2. Может ли в реализации рекурсивной функции существовать несколько операторов передачи управления *return*?

Лабораторная работа №2. Алгоритмы поиска и сортировки в массивах

Цель работы: изучить способы сортировки и поиска в массивах структур и файлах.

2.1. Краткие теоретические сведения

При обработке баз данных часто применяются массивы структур. Обычно база данных накапливается и хранится на диске в файле. К ней часто приходится обращаться, обновлять, перегруппировывать. Работа с базой может быть организована двумя способами.

1. Внесение изменений и поиск осуществляются прямо на диске, с использованием специфической техники работы со структурами в файлах. При этом временные затраты на обработку данных (поиск, сортировку) значительно возрастают, но нет ограничений на использование оперативной памяти.

2. Считывание всей базы (или необходимой ее части) в массив структур. При этом обработка производится в оперативной памяти, что значительно увеличивает скорость, однако требует больших затрат памяти.

Наиболее частыми операциями при работе с базами данных являются «поиск» и «сортировка». При этом алгоритмы решения этих задач существенно зависят от того, организованы структуры в массивы или размещены на диске.

Обычно элемент данных (структура) содержит некое ключевое поле (ключ), по которому его можно найти. Ключом может служить любое поле структуры, например, фамилия, номер телефона или адрес. Основное требование к ключу в задачах поиска состоит в том, чтобы операция проверки на равенство была корректной, поэтому при поиске данных по ключу, имеющему вещественное значение, следует указывать не его конкретное значение, а интервал, в который это значение попадает.

2.1.1. Алгоритмы поиска

Предположим, что у нас имеется следующая структура:

```
struct Ttype {
type key;          // Ключевое поле типа type
     . . .         // Описание других полей структуры
} *a;                    // Указатель для динамического массива структур
```

Задача поиска требуемого элемента в массиве структур *a* (размер *n* – задается при выполнении программы) заключается в нахождении индекса *i_key*, удовлетворяющего условию *a*[*i_key*].*key* = *f_key*, где *key* – интересующее нас поле структуры данных, *f_key* – искомое значение того же типа что и *key*. После нахождения индекса *i_key* обеспечивается доступ ко всем другим полям найденной структуры *a*[*i_key*].

Линейный поиск используется, когда нет никакой дополнительной информации о разыскиваемых данных, и представляет собой последовательный

перебор всех элементов массива. Если поле поиска является уникальным, то поиск выполняется до обнаружения требуемого ключа или до конца, если ключ не обнаружен. Если же поле поиска не уникальное, приходится перебирать все данные до конца массива:

```
int i_key = 0, kod = 0;
for (i = 1; i < n; i++)
if (a[i].key == f_key) {
        kod = 1;
// Обработка найденного элемента, например, вывод
        i_key = i;
                                // break;      – если поле поиска уникальное
}
if (kod == 0)                          // Вывод сообщения, что элемент не найден
```

Поиск делением пополам используется, если данные упорядочены по возрастанию (по убыванию) ключа *key*. Алгоритм поиска осуществляется следующим образом:

– берется средний элемент *m*;

– если элемент массива *a*[*m*].*key* < *f_key*, то все элементы $i \le m$ исключаются из дальнейшего поиска, иначе – исключаются все элементы с индексами $i > m$.

Приведем пример, реализующий этот алгоритм:

```
int i_key = 0, j = n – 1, m;
while(i_key < j) {
        m = (i_key + j)/2;
        if (a[m].key < f_key) i_key = m + 1;
        else  j = m;
}
if (a[i_key].key != key) return –1;              // Элемент не найден
            else return i;
```

Проверка совпадения *a*[*m*].*k* = *f_key* в этом алгоритме внутри цикла отсутствует, т. к. тестирование показало, что в среднем выигрыш от уменьшения количества проверок превосходит потери от нескольких «лишних» вычислений до выполнения условия *i_key* = *j*,

2.1.2. Алгоритмы сортировки

Под сортировкой понимается процесс перегруппировки элементов массива, приводящий к их упорядоченному расположению относительно ключа.

Цель сортировки – облегчить последующий поиск элементов. Метод сортировки называется устойчивым, если в процессе перегруппировки относительное расположение элементов с равными ключами не изменяется. Основное условие при сортировке массивов – это не вводить дополнительных массивов, т. е. все перестановки элементов должны выполняться в исходном массиве. Сортировку массивов принято называть ***внутренней***, а сортировку файлов – ***внешней***.

Методы внутренней сортировки классифицируются по времени их работы. Хорошей мерой эффективности может быть число операций сравнений ключей и число пересылок (перестановок) элементов.

Прямые методы имеют небольшой код и просто программируются, быстрые, усложненные методы требуют меньшего числа действий, но эти действия обычно более сложные, чем в прямых методах, поэтому для достаточно малых значений n ($\boldsymbol{n \le 50}$) прямые методы работают быстрее. Значительное преимущество быстрых методов начинает проявляться при $\boldsymbol{n \ge 100}$.

Среди *простых* методов наиболее популярны следующие.

1. ***Метод прямого обмена*** (пузырьковая сортировка):

```
for (i = 0; i < n – 1; i++)
        for (j = i + 1; j < n; j++)
                if (a[i].key > a[j].key) {          // Переставляем элементы
                r = a[i];
                a[i] = a[j];
                a[j] = r;
          }
```

2. ***Метод прямого выбора***:

```
for (i = 0; i < n – 1; i++) {
            m = i;
            for (j = i + 1; j < n; j++)
                    if (a[j].key < a[m].key) m = j;
            r = a[m];                                  // Переставляем элементы
     a[m] = a[i];
     a[i] = r;
   }
```

3) Сортировка с помощью прямого (двоичного) включения.

4) *Шейкерная* сортировка (модификация пузырьковой).

Примечание: методы 3 и 4 используются реже.

К *улучшенным* методам сортировки относятся следующие.

1. ***Метод Д. Шелла*** (1959), усовершенствование метода прямого включения.

2. Сортировка ***с помощью дерева***, метод ***HeapSort***, Д. Уильямсона (1964).

3. Сортировка ***с помощью разделения***, метод ***QuickSort***, Ч. Хоара (1962), улучшенная версия пузырьковой сортировки, являющийся на сегодняшний день самым эффективным методом.

Идея метода разделения *QuickSort* заключается в следующем. Выбирается значение ключа среднего m-го элемента $x = a[m].key$. Массив просматривается слева направо до тех пор, пока не будет обнаружен элемент $a[i].key > x$. Затем массив просматривается справа налево, пока не будет обнаружен элемент $a[j].key < x$. Элементы $a[i]$ и $a[j]$ меняются местами. Процесс просмотра и обмена продолжается до тех пор, пока i не станет больше j. В результате массив оказывается разбитым на левую часть $a[L]$, где $0 \le L \le j$, с ключами меньше (или равными) x и правую $a[R]$, где $i \le R < n$, с ключами больше (или равными) x.

Алгоритм такого разделения очень прост и эффективен:

```
i = 0; j = n – 1; x = a[(L + R)/2].key;
while (i <= j) {
        while (a[i].key < x)  i++;
        while (a[j].key > x)  j--;
        if (i <= j) {
                r = a[i];                          // Переставляем элементы
                a[i] = a[j];
                a[j] = r;
                 i++;          j--;
        }
}
```

Чтобы отсортировать массив, остается применять алгоритм разделения к левой и правой частям, затем к частям частей и так до тех пор, пока каждая из частей не будет состоять из единственного элемента. Алгоритм получается итерационным, на каждом этапе которого стоят две задачи по разделению. К решению одной из них можно приступить сразу, для другой следует запомнить начальные условия (номер разделения, границы) и отложить ее решение до момента окончания сортировки выбранной половины.

Сравнение методов сортировок показывает, что при $n > 100$ наихудшим является метод пузырька, метод *QuickSort* в 2–3 раза лучше, чем *HeapSort*, и в 3–7 раз, чем метод Шелла.

2.2. Индивидуальные задания

Написать программу обработки файла данных, состоящих из структур, в которой реализованы следующие функции: стандартная обработка файла (создание, просмотр, добавление); линейный поиск в файле; сортировка массива (файла) методами прямого выбора и *QuickSort*; двоичный поиск в отсортированном массиве.

1. В магазине формируется список лиц, записавшихся на покупку товара. Вид списка: номер, ФИО, домашний адрес, дата учета. Удалить из списка все повторяющиеся записи, проверяя ФИО и адрес. Ключ: дата постановки на учет.

2. Список товаров на складе включает: наименование товара, количество единиц товара, цену единицы и дату поступления товара на склад. Вывести в алфавитном порядке список товаров, хранящихся больше месяца, стоимость которых превышает 100 000 р. Ключ: наименование товара.

3. Для получения места в общежитии формируется список: ФИО студента, группа, средний балл, доход на каждого члена семьи. Общежитие в первую очередь предоставляется тем, у кого доход меньше двух минимальных зарплат, затем остальным в порядке уменьшения среднего балла. Вывести список очередности. Ключ: доход на каждого члена семьи.

4. В справочной автовокзала хранится расписание движения автобусов. Для каждого рейса указаны его номер, пункт назначения, время отправления и

прибытия. Вывести информацию о рейсах, которыми можно воспользоваться для прибытия в пункт назначения раньше заданного времени. Ключ: время прибытия.

5. На междугородной АТС информация о разговорах содержит дату разговора, код и название города, время разговора, тариф, номер телефона в этом городе и номер телефона абонента. Вывести по каждому городу общее время разговоров с ним и сумму. Ключ: общее время разговоров.

6. Информация о сотрудниках фирмы включает: ФИО, табельный номер, количество отработанных часов за месяц, почасовой тариф. Рабочее время свыше 144 ч считается сверхурочным и оплачивается в двойном размере. Вывести размер заработной платы каждого сотрудника фирмы за вычетом подоходного налога (12 % от суммы заработка). Ключ: размер заработной платы.

7. Информация об участниках спортивных соревнований содержит: ФИО игрока, игровой номер, возраст, рост, вес, наименование страны, название команды. Вывести информацию о самой молодой команде. Ключ: возраст.

8. Для книг, хранящихся в библиотеке, задаются: номер книги, автор, название, год издания, наименование издательства и количество страниц. Вывести список книг с фамилиями авторов в алфавитном порядке, изданных после заданного года. Ключ: автор.

9. Различные цеха завода выпускают продукцию нескольких наименований. Сведения о продукции включают: наименование, количество, номер цеха. Для заданного цеха необходимо вывести изделия по каждому наименованию в порядке убывания их количества. Ключ: количество выпущенных изделий.

10. Информация о сотрудниках предприятия содержит: ФИО, номер отдела, наименование должности, дату начала работы. Вывести списки сотрудников по отделам в порядке убывания стажа. Ключ: дата начала работы.

11. Ведомость абитуриентов, сдавших вступительные экзамены в университет, содержит: ФИО, номер группы, адрес, оценки. Определить количество абитуриентов, проживающих в г. Минске и сдавших экзамены со средним баллом не ниже 8,5, вывести их фамилии в алфавитном порядке. Ключ: ФИО

12. В справочной аэропорта хранится расписание вылетов самолетов на следующие сутки: номер рейса, тип самолета, пункт назначения, время вылета. Вывести информацию для заданного пункта назначения в порядке возрастания времени вылета. Ключ: пункт назначения.

13. В кассе хранится информация о поездах на ближайшую неделю: дата выезда, пункт назначения, время отправления, число свободных мест. Необходимо зарезервировать m мест до города N на k-й день недели с временем отправления поезда не позднее t часов. Вывести время отправления или сообщение о невозможности выполнить заказ. Ключ: число свободных мест.

14. Ведомость абитуриентов, сдавших вступительные экзамены в университет, содержит: ФИО абитуриента, четыре отметки. Определить средний балл по университету и вывести список абитуриентов, средний балл которых выше среднего балла по университету в порядке убывания балла. Ключ: средний балл.

15. В ателье хранятся квитанции о сданной в ремонт аппаратуре: наименование группы изделий (телевизор, радиоприемник и т. п.), марку изделия, дата приемки, состояние готовности заказа (выполнен, не выполнен). Вывести информацию о состоянии заказов на текущие сутки по группам изделий. Ключ: дата приемки в ремонт.

16. Информация о сотрудниках института содержит: ФИО, наименование факультета, кафедры, должности, объем нагрузки (часов). Вывести списки сотрудников по кафедрам в порядке убывания нагрузки. Ключ: объем нагрузки.

2.3. Контрольные вопросы

1. Устойчив ли интерполяционный поиск на неравномерном объеме информации?

2. Какая сложность у быстрой сортировки? За счет чего она достигается?

Лабораторная работа №3. Динамическая структура СТЕК

Цель работы: изучить алгоритмы работы с динамическими структурами данных в виде стека.

3.1. Краткие теоретические сведения

Стек (*Stack*)– структура типа ***LIFO*** (*Last In, First Out*) – последним вошел, первым выйдет. Элементы в стек можно добавлять или извлекать только через его вершину. Программно стек реализуется в виде однонаправленного списка с одной точкой входа – вершиной стека.

Максимальное число элементов стека не ограничивается, т. е. по мере добавления в стек нового элемента память под него должна запрашиваться, а при удалении – освобождаться. Таким образом, стек – динамическая структура данных, состоящая из ***переменного*** числа элементов.

Для работы со стеком введем следующую структуру (вместо приведенного типа *Stack* может быть любой другой идентификатор):

```
struct Stack {
        int info;       // Информационная часть элемента, например, int
        Stack *next;    // Адресная часть – указатель на следующий элемент
} *begin;               // Указатель вершины стека
```

При работе со стеком обычно выполняются следующие операции:
– формирование стека (добавление элемента в стек);
– обработка элементов стека (просмотр, поиск, удаление);
– освобождение памяти, занятой стеком.

Приведем примеры основных функций для работы со стеком, взяв для простоты в качестве информационной части целые числа, т. е. объявленную выше структуру типа *Stack*.

Функция формирования элемента стека

Простейший вид функции (*push*), в которую в качестве параметров передаются указатель на вершину и введенная информация, а измененное значение вершины возвращается в точку вызова оператором *return*:

```
Stack* InStack(Stack *p, int in) {
Stack *t = new Stack;                   // Захватываем память для элемента
t -> info = in;                 // Формируем информационную часть
        t -> next = p;                  // Формируем адресную часть
return t;
}
```

Обращение к этой функции для добавления нового элемента «*a*» в стек, вершиной которого является указатель *begin*: begin = InStack(begin, a);

Алгоритм просмотра стека (без извлечения его элементов, т. е. без сдвига вершины):

1. Устанавливаем текущий указатель на начало списка: *t = begin*;

2. Начинаем цикл, работающий до тех пор, пока указатель *t* не равен *NULL* (признак окончания списка).

3. Выводим информационную часть текущего элемента *t -> info* на экран.

4. Текущий указатель переставляем на следующий элемент, адрес которого находится в поле *next* текущего элемента: *t = t -> next*;

5. Конец цикла.

Функция, реализующая рассмотренный алгоритм:

```
void View(Stack *p) {
Stack *t = p;
while( t != NULL) {
// Вывод на экран информационной части, например, cout <<  t -> info << endl;
                t = t -> Next;
        }
}
```

Обращение к этой функции: View(begin);

Блок-схема функции *View* представлена на рис. 3.1.

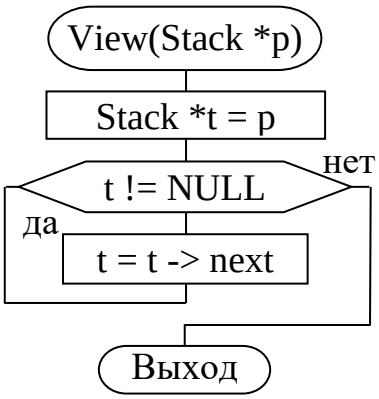


Рис. 3.1

Функция получения информации из вершины стека с извлечением:

```
Stack* OutStack(Stack* p, int *out)  {
Stack *t = p;                  // Устанавливаем указатель t на вершину p
*out = p -> info;
p = p -> next;                 // Переставляем вершину p на следующий элемент
delete t;                      // Удаляем бывшую вершину t
return p;                      // Возвращаем новую вершину p
}
```

Обращение к этой функции: begin = OutStack(begin, &a); информацией является переданное по адресу значение «*a*».

Функция освобождения памяти, занятой стеком:

```
void Del_All(Stack **p)  {
Stack *t;
```

```
while( *p != NULL) {
t = *p;
*p = (*p) -> Next;
delete t;
        }
}
```

Обращение к этой функции: Del_All(&begin); после ее выполнения указатель на вершину *begin* будет равен *NULL*.

Сортировка однонаправленных списков

Для ускорения поиска информации в списке обычно при выводе данных список упорядочивают (сортируют) по ключу.

Проще всего использовать метод сортировки, основанный на перестановке местами двух соседних элементов, если это необходимо. Существует два способа перестановки элементов: обмен адресами и обмен информацией.

1. Первый способ – перестановка адресов двух соседних элементов, следующих за элементом с известным указателем. Первый элемент стека в этом случае ***не сортируется***. Для того чтобы и первый элемент оказался отсортированным, следует перед обращением к функции сортировки добавить один (любой) элемент в стек, а после сортировки – удалить его.

Функция сортировки для этого случая имеет вид:

```
void Sort_p(Stack **p) {
        Stack *t = NULL, *t1, *r;
        if ((*p) -> next -> next == NULL) return;
        do {
                for (t1=*p; t1-> next->next  != t; t1=t1-> next)
                        if (t1->next->info  >  t1-> next-> next-> info){
                                r = t1->next->next;
                                t1 -> next -> next = r -> next;
                                r-> next =t1-> next;
                                t1-> next = r;
                        }
                t= t1-> next;
        } while ((*p)-> next -> next != t);
}
```

Обращение к этой функции: Sort_p(&begin);

2. Второй способ – обмен информацей между текущим и следующим элементами. Функция сортировки для этого случая будет иметь вид:

```
void Sort_info(Stack *p) {
        Stack *t = NULL, *t1;
        int r;
        do {
                for (t1=p; t1 -> next != t; t1 = t1-> next)
                        if (t1-> info > t1-> next -> info)  {
```

```
                        r = t1-> info;
                        t1-> info = t1-> next -> info;
                        t1-> next -> info = r;
                    }
                t = t1;
        } while (p -> next != t);
}
```

3.2. Пример выполнения задания

Написать программу, содержащую основные функции обработки однонаправленного списка, организованного в виде стека, информационная часть которого представляет собой случайные целые числа от 0 до 20.

```
. . .
 struct Stack {                          // Декларация структурного типа
        int info;
        Stack * next;
} *begin, *t;
//------------ Декларации прототипов функций пользователя ----------
Stack* InStack(Stack*, int);
void View(Stack*);
void Del_All(Stack **);
//--------------- Функция добавления элемента в Стек ------------------------
Stack* InStack(Stack *p, int in) {
        Stack *t = new Stack;
        t -> info = in;
        t -> next = p;
        return t;
}
//----------------- Функция просмотра Стека---------------------------------
void View(Stack *p) {
        Stack *t = p;
        while( t != NULL) {
        cout << "   " << t->info << endl;
                t = t -> next;
        }
}
//----------------------- Функция освобождения памяти ----------------------
void Del_All(Stack **p) {
        while(*p != NULL) {
                t = *p;
                *p = (*p) -> next;
                delete t;
        }
}

void main()
{
int i, in, n, kod;
while(true){
   cout << "\n\tCreat - 1.\n\tAdd - 2.\n\tView - 3.\n\tDel - 4.\n\tEXIT – 0.  :  " ;
```

```
    cin >> kod;
    switch(kod) {
      case 1: case 2:
         if(kod == 1 && begin != NULL){
// Если создаем новый стек, должны освободить память, занятую предыдущим
                    cout << "Clear Memory!" << endl;
                    break;
                }
        cout << "Input kol = ";        cin >> n;
        for(i = 1; i <= n; i++)  {
            in = random(20);
            begin = InStack(begin, in);
        }
        if (kod == 1) cout << "Create " << n << endl;
        else cout << "Add " << n << endl;
      break;
      case 3:       if(!begin){
                        cout << "Stack Pyst!" << endl;
                        break;
                    }
        cout << "--- Stack ---" << endl;
        View(begin);
      break;
      case 4:
        Del_All(&begin);
        cout<<"Memory Free!"<<endl;
      break;
      case 0:
        if(begin != NULL)
         Del_All(&begin);
        return;          // Выход – EXIT
}
    }
}
```

3.3. Индивидуальные задания

Написать программу по созданию, добавлению, просмотру и решению приведенных далее задач (в рассмотренных примерах это действие отсутствует) для однонаправленного линейного списка типа *Stack*. Реализовать сортировку стека методами, рассмотренными в подразделе 3.1.

Решение поставленной задачи описать в виде блок-схемы.

Во всех заданиях создать список из положительных и отрицательных случайных целых чисел.

1. Разделить созданный список на два: в первом – положительные числа, во втором – отрицательные.

2. Удалить из созданного списка элементы с четными числами.

3. Удалить из созданного списка отрицательные элементы.

4. В созданном списке поменять местами крайние элементы.

5. Из созданного списка удалить элементы, заканчивающиеся на цифру 5.

6. В созданном списке поменять местами элементы, содержащие максимальное и минимальное значения.

7. Перенести из созданного списка в новый список все элементы, находящиеся между вершиной и максимальным элементом.

8. Перенести из созданного списка в новый список все элементы, находящиеся между вершиной и элементом с минимальным значением.

9. В созданном списке определить количество и удалить все элементы, находящиеся между минимальным и максимальным элементами.

10. В созданном списке определить количество элементов, имеющих значения, меньше среднего значения от всех элементов, и удалить эти элементы.

11. В созданном списке вычислить среднее арифметическое и заменить им первый элемент.

12. Созданный список разделить на два: в первый поместить четные, а во второй – нечетные числа.

13. В созданном списке определить максимальное значение и удалить его.

14. Из созданного списка удалить каждый второй элемент.

15. Из созданного списка удалить каждый нечетный элемент.

16. В созданном списке вычислить среднее арифметическое и заменить им все четные значения элементов.

3.4. Контрольные вопросы

1. Что такое стек?
2. Что такое рекурсивный тип данных?
3. Возможно ли удалить из стека элемент, не являющийся вершиной, при этом не потеряв основной список?

Лабораторная работа №4. Динамическая структура ОЧЕРЕДЬ

Цель работы: изучить возможности работы со списками, организованными в виде очереди.

4.1. Краткие теоретические сведения

Очередь – линейный список, в котором извлечение данных происходит из начала, а добавление – в конец, т. е. это структура, организованная по принципу ***FIFO*** (*First In, First Out*) – первым вошел, первым выйдет.

При работе с очередью используют ***два указателя*** – на первый элемент (начало – ***begin***) и на последний (конец – ***end***). Очереди организуются в виде односвязных или двухсвязных списков, в зависимости от количества связей (указателей) в адресной части элемента структуры.

Односвязные списки

Шаблон элемента структуры, информационной частью которого является целое число (аналогично стеку), будет иметь следующий вид:

```
struct Spis1 {
        int info;
        Spis1 *next;
} *begin, *end;                          // Указатели на начало и на конец
```

Основные операции с очередью следующие:
– формирование очереди;
– обработка очереди (просмотр, поиск, удаление);
– освобождение занятой памяти.

Формирование очереди состоит из двух этапов: создание первого элемента и добавление нового элемента в конец очереди.

Функция формирования очереди из данных объявленного выше типа *Spis*1 с добавлением новых элементов в конец может иметь следующий вид (*b* – начало очереди, *e* – конец):

```
void Create(Spis1 **b, Spis1 **e, int in)  {   // in – переданная информация
Spis1 *t = new Spis1;
        t -> info = in;                 // Формирование информационной части
t -> next = NULL;           // Формирование адресной части
if(*b == NULL)              // Формирование первого элемента
*b = *e = t;
else {                      // Добавление элемента в конец
                (*e) -> next = t;
                *e = t;
}
}
```

Обращение к функции Create(&begin, &end, in);

Алгоритмы просмотра и освобождения памяти выполняются аналогично рассмотренным ранее для Стека (см. лабораторную работу № 3).

Двухсвязные списки

Двухсвязным (двунаправленным) является список, в адресную часть которого кроме указателя на следующий элемент включен и указатель на предыдущий.

Зададим структуру, в которой адресная часть состоит из указателей на предыдущий (*prev*) и следующий (*next*) элементы:

```
struct Spis2    {
int info;
Spis2  *prev, *next;
} *begin, *end;
```

Формирование двунаправленного списка проводится в два этапа: формирование первого элемента и добавление нового, причем добавление может выполняться как в начало (*begin*), так и в конец (*end*) списка.

Создание первого элемента

1. Захватываем память под элемент: *Spis2 *t = new Spis2*;
2. Формируем информационную часть (*t -> info = StrToInt*(*Edit1->Text*));
3. Формируем адресные части: *t -> next = t -> prev = NULL*;
4. Устанавливаем указатели начала и конца списка на первый элемент: *begin = end = t*;

Добавление элемента

Добавить в список новый элемент можно как в начало, так и в конец. Захват памяти и формирование информационной части выполняются аналогично предыдущему алгоритму (п.п. 1 – 2).

Если элемент добавляется в начало списка, то выполняется следующая последовательность действий:

```
t -> prev = NULL;       // Предыдущего нет
t -> next = begin;      // Связываем новый элемент с первым
begin -> prev = t;      // Изменяем адрес prev бывшего первого
begin = t;              // Переставляем указатель begin на новый
```

В конец элемент добавляется следующим образом:

```
t -> next = NULL;       // Следующего нет
t -> prev = end;        // Связываем новый с последним
end -> next = t;        // Изменяем адрес next бывшего последнего
end = t;                // Изменяем указатель end
```

Просмотр списка

Просмотр списка можно выполнять с начала или с конца списка. Просмотр с начала выполняется так же, как для однонаправленного списка, только

в функции *View*() необходимо изменить структурный тип (см. лабораторную работу № 3).

Просмотр списка с конца

1. Устанавливаем текущий указатель на конец списка: *t = end*;
2. Начинаем цикл, работающий до тех пор, пока *t* != *NULL*.
3. Информационную часть текущего элемента *t -> info* выводим на экран.
4. Переставляем текущий указатель на предыдущий элемент, адрес которого находится в поле *prev* текущего элемента: *t = t -> prev*;
5. Конец цикла.

Алгоритм удаления элемента в списке по ключу

Удалить из списка элемент, информационная часть (ключ) которого совпадает со значением, введенным с клавиатуры.

Решение данной задачи проводим в два этапа – поиск и удаление.

Первый этап – поиск

Алгоритм поиска аналогичен просмотру списка с начала. Введем дополнительный указатель и присвоим ему значение *NULL*: *Spis2 *key = NULL*. Введем с клавиатуры искомое значение *i_p* (ключ поиска).

1. Установим текущий указатель на начало списка: *t = begin*;
2. Начало цикла (выполнять пока *t* != *NULL*).
3. Сравниваем информационную часть текущего элемента с искомым, если они совпадают (*t -> info = i_p*), то запоминаем адрес найденного элемента: *key = t*; (если ключ поиска *уникален*, то завершаем поиск – *break*).
4. Переставляем текущий указатель на следующий элемент: *t = t -> next*;
5. Конец цикла.

Выполняем контроль, если *key = NULL* , т. е. искомый элемент не найден, то сообщаем о неудаче и этап удаления не выполняем (*return*).

Второй этап – удаление

Если элемент найден (*key* != *NULL*), то выполняем удаление элемента из списка, в зависимости от его местонахождения.

1. Если удаляемый элемент находится в начале списка, т. е. *key* равен *begin*, то первым элементом списка становится второй элемент:

а) указатель начала списка переставляем на следующий (второй) элемент: *begin = begin -> next*;

б) адресу *prev* присваиваем значение *NULL*, т. е. теперь предыдущего нет *begin -> prev = NULL*;

2. Если удаляемый элемент в конце списка, т. е. *key* равен *end*, то последним элементом в списке должен стать предпоследний:

а) указатель конца списка переставляем на предыдущий элемент: *end = end -> prev*;

б) обнуляем адрес *next* нового последнего элемента:

end -> next = NULL;

3. Если удаляемый элемент находится в середине списка, нужно обеспечить связь предыдущего и следующего элементов (рис. 4.1):

а) от *k*-го элемента с адресом *key* обратимся к предыдущему (*k* – 1)-му элементу, адрес которого key -> prev, и в его поле *next* [(*key->prev*)->*next*] запишем адрес (*k* + 1)-го элемента, значение которого *key->next*:

(*key -> prev*) -> *next = key -> next*;

б) аналогично, в поле *prev* (*k*+1)-го элемента, с адресом key->next запишем адрес (*k*–1)-го элемента:

(key -> next) -> prev = key -> prev;

4. Освобождаем память, занятую удаленным элементом.

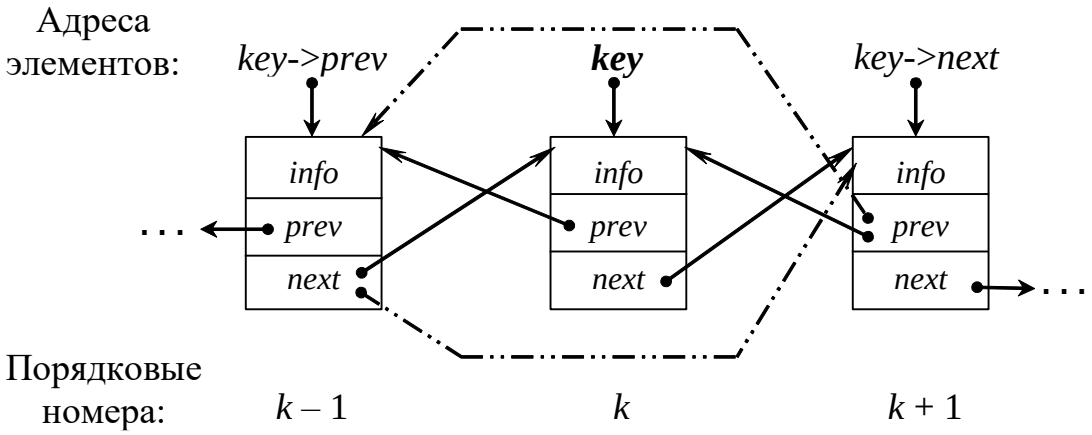


Рис. 4.1.

Алгоритм освобождения памяти, занятой двунаправленным списком, аналогичен рассмотренному алгоритму для стека (см. лабораторную работу № 3).

4.2. Пример выполнения задания

Написать программу, содержащую основные функции обработки двунаправленного списка, информационная часть которого представляет собой целые числа.

```
struct Spis2 {
      int info;
      Spis2 *next, *prev;
 } *begin, *end, *t;
//--------------------------------------------------------------------
void Create_Spis2(Spis2**, Spis2**, int);
void Add_Spis2(int, Spis2**, Spis2**, int);
void View_Spis2(int, Spis2*);
void Del_All(Spis2**);
//------------------- Создание первого элемента ----------------------------
void Create_Spis2(Spis2 **b, Spis2 **e, int in)  {
```

```
        t = new Spis2;
        t -> info = in;
        t -> next = t -> prev = NULL;
*b = *e = t;
}
//------------------- Добавление элемента в список -------------------------
void Add_Spis2(int kod, Spis2 **b, Spis2 **e, int in)  {
      t = new Spis2;
      t -> info = in;
      if(kod == 0){
            t -> prev = NULL;
            t -> next = *b;
            (*b) -> prev = t;
            *b = t;
      }
      else {
            t -> next = NULL;
            t -> prev = *e;
            (*e) -> next = t;
            *e = t;
      }
}
//---------------------- Просмотр элементов списка --------------------------
void View_Spis2(int kod, Spis2 *t)  {
      while(t != NULL) {
            Form1->Memo1->Lines->Add(t->info);
// В консольном приложении:          cout << t->info << endl;
            if(kod == 0) t = t->next;
            else  t = t->prev;
      }
}

void main()
{
int i, in, n, kod, kod1;
char Str[2][10] = {"Begin ", "End "};
        while(true){
        cout << "\n\tCreat - 1\n\tAdd - 2\n\tView - 3\n\tDel - 4\n\tEXIT - 0  :  " ;
        cin >> kod;
        switch(kod) {
                              case 1:       if(begin != NULL){
                                    cout << "Clear Memory!" << endl;
                                    break;
                  }
        cout << "Begin Info = ";     cin >> in;
        Create_Spis2(&begin, &end, in);
        cout << "Creat Begin = " << begin -> info << endl;
   break;
   case 2:
        cout << "Info = ";       cin >> in;
        cout << "Add Begin - 0, Add End - 1 :  ";      cin >> kod1;
        Add_Spis2(kod1, &begin, &end, in);
        if(kod1 == 0) t = begin;
        else t = end;
        cout << "Add to " << Str[kod1] << "  " << t -> info << endl;
```

```
   break;
   case 3:   if(!begin){
                  cout << "Stack Pyst!" << endl;
                  break;
       }
 cout<<"View Begin-0,View End-1:";
 cin >> kod1;
 if(kod1 == 0) {
         t = begin;
         cout <<"-- Begin --" << endl;
 }
  else {
         t = end;
         cout <<"--- End --" << endl;
  }
  View_Spis2(kod1, t);
    break;
    case 4:
   Del_All(&begin);
   cout <<"Memory Free!"<< endl;
   break;
    case 0:  if(begin != NULL)
          Del_All(&begin);
      return;
    }
  }
}
```

4.3. Индивидуальные задания

Написать программу по созданию, добавлению (в начало, в конец), просмотру (с начала, с конца) и решению приведенной в подразделе 3.3 задачи для двунаправленных линейных списков.

4.4. Контрольные вопросы

1. Что такое однонаправленная очередь?
2. Что такое двунаправленная очередь?
3. Где, на ваш взгляд, удобнее всего использовать динамические линейный списки?

Лабораторная работа №5. Обратная польская запись

Цель работы: изучить правила формирования постфиксной записи арифметических выражений с использованием стека.

5.1. Краткие теоретические сведения

Одной из задач при разработке трансляторов является задача расшифровки арифметических выражений.

Выражение ***a+b*** записано в ***инфиксной*** форме, ***+ab*** – в ***префиксной***, ***ab+*** – в ***постфиксной***. В наиболее распространенной инфиксной форме для указания последовательности выполнения операций необходимо расставлять скобки. Польский математик Я. Лукашевич использовал тот факт, что при записи постфиксной формы скобки не нужны, а последовательность операндов и операций удобна для расшифровки. Поэтому постфиксная запись выражений получила название ***обратной польской записи*** **(*ОПЗ*)**. Например, в ОПЗ выражение

$$r = (a + b) \cdot (c + d) - e;$$

выглядит следующим образом:

$$r = ab + cd + \cdot e - .$$

Алгоритм преобразования выражения из инфиксной формы в форму ОПЗ был предложен Эдсгер Вибе Дейкстрой. При его реализации вводится понятие стекового приоритета операций.

Рассмотрим алгоритм получения ОПЗ из исходной строки символов, в которой записано выражение в ***инфиксной форме***.

1. Символы-операнды переписываются в выходную строку, в которой формируется постфиксная форма выражения.

2. Открывающая скобка записывается в стек.

3. Очередная операция выталкивает в выходную строку все операции из стека с большим или равным приоритетом.

4. Закрывающая скобка выталкивает все операции из стека до ближайшей открывающей скобки в выходную строку, открывающая скобка удаляется из стека, а закрывающая – игнорируется.

5. Если после просмотра последнего символа исходной строки в стеке остались операции, то все они выталкиваются в выходную строку.

5.2. Пример выполнения задания

Написать программу расшифровки и вычисления арифметических выражений с использованием стека.

Вид формы и полученные результаты представлены на рис. 5.1.

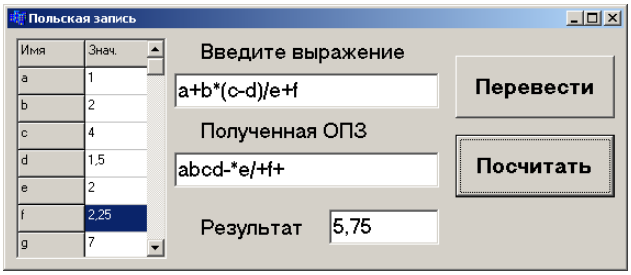


Рис. 5.1

Приведем только тексты используемых функций-обработчиков и созданных функций пользователя (тексты функций *InStack* и *OutStack* взять из лабораторной работы №3, заменив тип *int* на *char*):

```
. . .
struct Stack {
char info;
Stack *next;
} *begin;
int Prior (char);
Stack* InStack( Stack*,char);
Stack* OutStack( Stack*,char*);
double Rezult(String);
double mas[201];                                  // Массив для вычисления
Set  <char, 0, 255> znak;                  // Множество символов-знаков
int Kol = 8;
//--------------- Текст функции-обработчика FormCreate -------------
  Edit1->Text = "a+b*(c-d)/e";              Edit2->Text = "";
  char a = 'a';
  StringGrid1->Cells[0][0] ="Имя";
  StringGrid1->Cells[1][0] ="Знач.";
  for(int i = 1; i<Kol; i++) {
       StringGrid1->Cells[0][i] = a++;       StringGrid1->Cells[1][i] = i;
  }
//---------------------- Текст функции-обработчика кнопки Перевести --------
Stack *t;
begin = NULL;                                  // Стек операций пуст
char ss, a;
String InStr, OutStr;                       // Входная и выходная строки
OutStr = "";  Edit2->Text = "";
InStr = Edit1->Text;
znak << '*' << '/' << '+' << '-' << '^';
int len = InStr.Length(), k;
for (k = 1; k <= len; k++) {
ss = InStr[k];
// Открывающую скобку записываем в стек
if ( ss == '(' ) begin = InStack(begin, ss);
if ( ss == ')' ) {
// Выталкиваем из стека все знаки операций до открывающей скобки
             while ( (begin -> info) != '(' ) {
                 begin = OutStack(begin,&a);    // Считываем элемент из стека
                 OutStr += a;                          // Записываем в строку
```

```
                }
                begin = OutStack(begin,&a);         // Удаляем из стека '(' скобку
        }
        // Букву (операнд) заносим в выходную строку
        if (ss >=  'a' && ss <= 'z' )  OutStr += ss;
        /* Если знак операции, то переписываем из стека в выходную строку все
операции с большим или равным приоритетом */
        if (znak.Contains(ss)) {
                while ( begin != NULL && Prior (begin -> info) >= Prior (ss) ) {
                      begin = OutStack(begin, &a);
                      OutStr += a;
                }
                begin = InStack(begin, ss);
        }}
        // Если стек не пуст, переписываем все операции в выходную строку
        while ( begin != NULL){
                begin = OutStack(begin, &a);
                OutStr += a;
         }
          Edit2->Text = OutStr;             // Выводим полученную строку
        }
        //---------------------- Текст функции-обработчика кнопки Посчитать -------
        char ch;
        String OutStr = Edit2->Text;
        for (int i=1; i<Kol; i++) {
              ch = StringGrid1->Cells[0][i][1];
              mas[int(ch)]=StrToFloat(StringGrid1->Cells[1][i]);
        }
        Edit3->Text=FloatToStr(Rezult(OutStr));
        //-------- Функция реализации приоритета операций----------
        int Prior ( char a ){
        switch ( a ) {
                case '^':                       return 4;
                case '*':  case '/':            return 3;
                case '-':  case '+':            return 2;
                case '(':                       return 1;
        }
        return 0;}
        //---------------- Расчет арифметического выражения ---------------------------
        double Rezult(String Str) {
           char ch, ch1, ch2;
           double op1, op2, rez;
           znak << '*' << '/' << '+' << '-' << '^';
           char chr = 'z'+1;
           for (int i=1; i <= Str.Length(); i++){
                 ch=Str[i];
                 if (! znak.Contains(ch)) begin = InStack(begin, ch);
                 else {
                         begin = OutStack(begin,&ch1);
                         begin = OutStack(begin,&ch2);
                         op1 = mas[int (ch1)];
                         op2 = mas[int (ch2)];
```

```
                switch (ch){
                        case '+' :      rez=op2+op1;            break;
                        case '-' :      rez=op2-op1;            break;
                        case '*' :      rez=op2*op1;            break;
                        case '/' :      rez=op2/op1;            break;
                        case '^' :      rez=pow(op2,op1);       break;
                }
        mas[int (chr)] = rez;
        begin = InStack(begin,chr);
        chr++;
        }
   }
   return rez;
}
```

5.3. Индивидуальные задания

Написать программу формирования ОПЗ и расчета полученного выражения. Разработать удобный интерфейс ввода исходных данных и вывода результатов. Работу программы проверить на конкретном примере (табл. 5.1).

Таблица 5.1

Номер варианта	*Выражение*	*a*	*b*	*c*	*d*	*e*	*Результат*
1	$a / (b - c) \cdot (d + e)$	8.6	2.4	5.1	0.3	7.9	– 26.12
2	$(a + b) \cdot (c - d) / e$	7.4	3.6	2.8	9.5	0.9	– 81.89
3	$a - (b + c \cdot d) / e$	3.1	5.4	0.2	9.6	7.8	2.16
4	$a / b - ((c + d) \cdot e)$	1.2	0.7	9.3	6.5	8.4	– 131.006
5	$a \cdot (b - c + d) / e$	9.7	8.2	3.6	4.1	0.5	168.78
6	$(a + b) \cdot (c - d) / e$	0.8	4.1	7.9	6.2	3.5	2.38
7	$a \cdot (b - c) / (d + e)$	1.6	4.9	5.7	0.8	2.3	– 0.413
8	$a / (b \cdot (c + d)) - e$	8.5	0.3	2.4	7.9	1.6	1.151
9	$(a + (b / c - d)) \cdot e$	5.6	7.4	8.9	3.1	0.2	0.666
10	$a \cdot (b + c) / (d - e)$	0.4	2.3	6.7	5.8	9.1	– 1.091
11	$a - (b / c \cdot (d + e))$	5.6	3.2	0.9	1.7	4.8	– 17.51
12	$(a - b) / (c + d) \cdot e$	0.3	6.7	8.4	9.5	1.2	– 0.429
13	$a / (b + c - d \cdot e)$	7.6	4.8	3.5	9.1	0.2	1.173
14	$a \cdot (b - c) / (d + e)$	0.5	6.1	8.9	2.4	7.3	– 0.144
15	$(a + b \cdot c) / (d - e)$	9.1	0.6	2.4	3.7	8.5	– 2.196
16	$a - b / (c \cdot (d - e))$	1.4	9.5	0.8	6.3	7.2	14.594

5.4. Контрольные вопросы

1. Что такое постфиксная запись?
2. Что такое инфиксная запись?
3. Где, на ваш взгляд, могут быть применены алгоритмы, реализующие обратную польскую запись?

Лабораторная работа № 6. Нелинейные списки

Цель работы: изучить алгоритмы обработки данных с использованием нелинейных структур в виде дерева.

6.1. Краткие теоретические сведения

Представление динамических данных в виде древовидных структур оказывается довольно удобным и эффективным для решения задач быстрого поиска информации.

Дерево состоит из элементов, называемых ***узлами*** (вершинами), которые соединены между собой направленными дугами (рис. 6.1). В случае $X \rightarrow Y$ вершина X называется ***предком*** (*родителем*), а Y – ***потомком*** (*сыном*, *дочерью*).

У дерева есть единственный узел, не имеющий предков (ссылок на этот узел), который называется ***корнем***. У любого другого узла есть ровно один предок, т. е. на каждый узел дерева имеется ровно одна ссылка. Узел, у которого нет сыновей, называется ***листом*** (например, узел *Y*).

Внутренний узел – это узел, не являющийся ни листом, ни корнем. ***Порядок узла*** равен количеству его узлов-сыновей. ***Степень дерева*** – максимальный порядок его узлов. ***Высота (глубина) узла*** равна числу его предков плюс один. ***Высота дерева*** – это наибольшая высота его узлов.

Бинарное дерево поиска

Наиболее часто для работы со списками используются бинарные (имеющие степень 2) деревья (рис. 6.1).

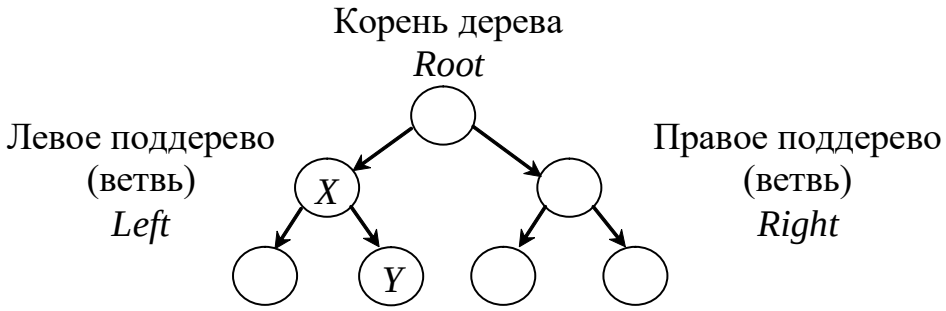


Рис. 6.1

В дереве поиска ключи расположены таким образом, что значение ключа левого сына меньше, чем значение предка, а правого сына – больше.

Сбалансированными, или *AVL*-деревьями, называются деревья, для каждого узла которых высóты его поддеревьев различаются не более чем на единицу.

Дерево по своей организации является рекурсивной структурой данных, поскольку каждое его поддерево также является деревом. В связи с этим дей-

ствия с такими структурами чаще всего описываются с помощью рекурсивных алгоритмов.

При работе с бинарным деревом простейшего вида, т. е. ключами которого являются целые числа (*уникальные*, не повторяющиеся), необходимо использовать структуру следующего вида:

```
struct Tree {
int info;
Tree *left, *right;
} *root;                          // root – указатель корня
```

В общем случае при работе с деревьями необходимо уметь:

– сформировать дерево (добавить новый элемент);

– обойти все элементы дерева (например, для просмотра или выполнения некоторой операции);

– выполнить поиск элемента с указанным значением в узле;

– удалить заданный элемент.

Формирование дерева поиска состоит из двух этапов: создание корня, являющегося листом, и добавление нового элемента (листа) в найденное место. Для этого используется функция формирования листа:

```
Tree* List(int inf) {
Tree *t = new Tree;               // Захват памяти
t -> info = inf;                  // Формирование информационной части
t -> left = t -> right = NULL;    // Формирование адресных частей
return t;                         // Возврат созданного указателя
}
```

1. Первоначально (*root = NULL*) создаем корень (***первый лист дерева***):

```
root = List (StrToInt(Edit1->Text));
```

2. Иначе (*root* != *NULL*) добавляем информацию (*key*) в нужное место:

```
void Add_List(Tree *root, int key)       {
Tree *prev, *t;                   // prev – указатель предка нового листа
 bool find = true;
t = root;
while ( t && find) {
        prev = t;
        if( key == t->info) {
                        find = false;         // Ключ должен быть уникален
                        ShowMessage("Dublucate Key!");
        }
        else
                if ( key < t -> info ) t = t -> left;
                else   t = t -> right;
}
if (find) {                                   // Нашли нужное место
        t = List(key);                        // Создаем новый лист
        if ( key < prev -> info ) prev -> left = t;
        else    prev -> right = t;
}
}
```

Функция просмотра элементов дерева

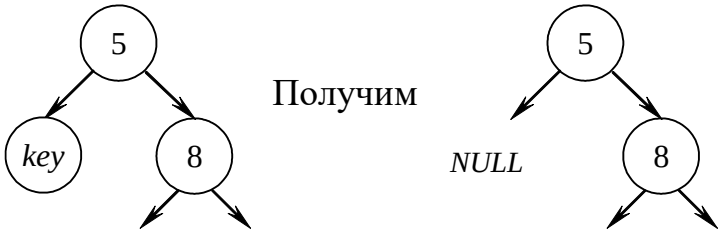
```
void View_Tree(Tree *p, int level )     {
  String str;
if ( p )  {
        View_Tree (p -> right , level+1);                  // Правое поддерево
        for ( int i=0; i<level; i++) str = str + "    ";
                Form1->Memo1->Lines->Add(str + IntToStr(p->info));
        View_Tree(p -> left , level+1);                    // Левое поддерево
   }
}
```

Обращение к функции *View* будет иметь вид View(root, 0);

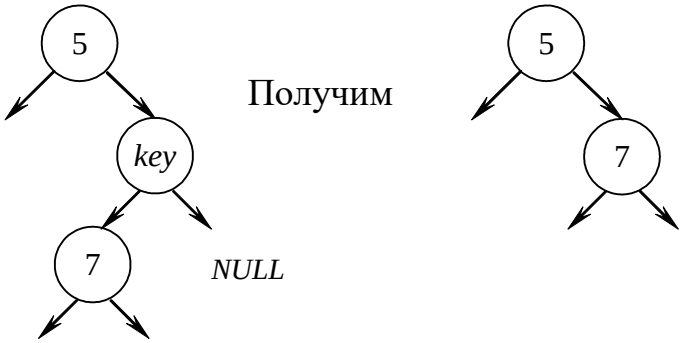
Вторым параметром функции является переменная, определяющая, на каком уровне (*level*) находится узел (у корня уровень «0»). Строка *str* используется для получения пробелов, необходимых для вывода значения на соответствующем уровне.

Удаление узла с заданным ключом из дерева поиска с сохранением его свойств, выполняется в зависимости от того, сколько сыновей (потомков) имеет удаляемый узел.

1. Удаляемый узел является листом – просто удаляем ссылку на него. Приведем пример схемы удаления листа с ключом *key*:

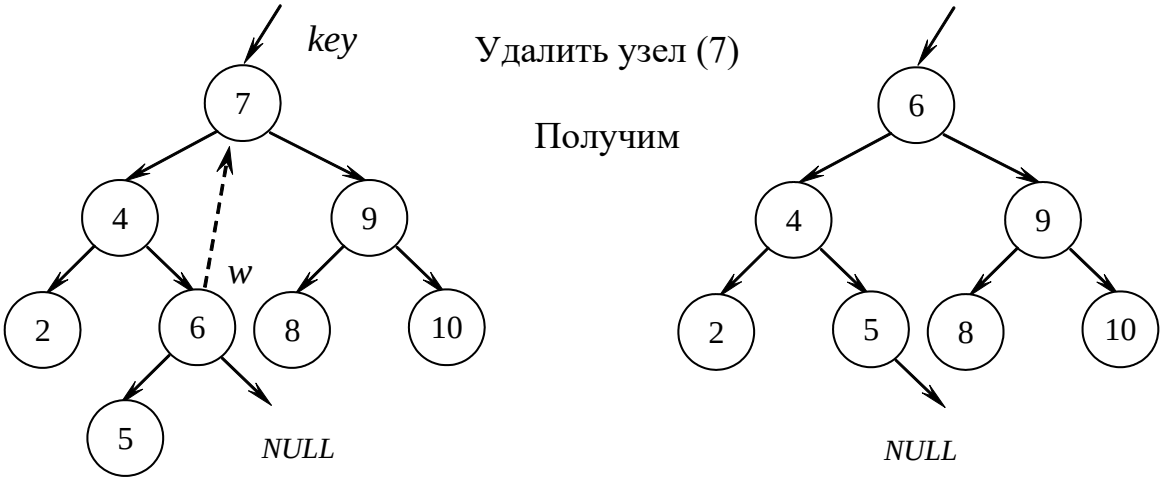


2. Удаляемый узел имеет только одного потомка, т. е. из удаляемого узла выходит ровно одна ветвь. Пример схемы удаления узла *key*, имеющего одного сына:



3. Удаление узла, имеющего двух потомков, значительно сложнее приведенных вариантов. Если *key* – удаляемый узел, то его следует заменить узлом *w*, который содержит либо наибольший ключ (самый правый, у которого указатель *Right* равен *NULL*) в левом поддереве, либо наименьший ключ (самый левый, у которого указатель *Left* равен *NULL*) в правом поддереве.

Используя первое условие, находим узел *w*, который является самым правым узлом поддерева *key*, у него имеется только левый потомок:



Функция удаления узла по заданному ключу key может иметь вид

```
Tree* Del_Info(Tree *root, int key)  {
Tree *Del, *Prev_Del, *R, *Prev_R;
// Del, Prev_Del – удаляемый узел и его предыдущий (предок);
// R, Prev_R – элемент, на который заменяется удаленный узел, и его пре-
док;
Del = root;
Prev_Del = NULL;
//-------- Поиск удаляемого элемента и его предка по ключу key ---------
while (Del != NULL && Del -> info != key) {
Prev_Del = Del;
if (Del->info > key)  Del = Del->left;
        else Del = Del->right;
}
if (Del == NULL)  {                              // Элемент не найден
        ShowMessage ( "NOT Key!");
        return root;
}
//--------------------- Поиск элемента R для замены -------------------------------
if (Del -> right == NULL) R = Del->left;
else
if (Del -> left == NULL) R = Del->right;
else {
//----------------- Ищем самый правый узел в левом поддереве -----------------
        Prev_R = Del;
        R = Del->left;
        while (R->right != NULL) {
                Prev_R = R;
                R = R->right;
        }
//------------ Нашли элемент для замены R и его предка Prev_R -------------
        if( Prev_R == Del)  R->right = Del->right;
        else {
                R->right = Del->right;
                Prev_R->right = R->left;
```

```
                R->left = Prev_R;
        }
 }
if (Del == root) root = R;                  // Удаляя корень, заменяем его на R
else
//------- Поддерево R присоединяем к предку удаляемого узла ----------
if (Del->info < Prev_Del->info)
        Prev_Del->left = R;                          // На левую ветвь
else    Prev_Del->right = R;                         // На правую ветвь
delete Del;
return root;
}
```

Поиск узла с минимальным (максимальным) ключом:

```
Tree* Min_Key(Tree *p) {                    // Tree* Max_Key(Tree *p)
   while (p->left != NULL) p = p->left;     // p=p->right;
   return p;
  }
```

Тогда для получения минимального ключа p_min -> info:

```
                        Tree *p_min = Min_Key(root);
```

Построение сбалансированного дерева поиска для заданного (созданного) массива ключей «***a***» можно осуществить, если этот массив предварительно отсортирован в порядке возрастания ключа с помощью следующей рекурсивной процедуры (при обращении $n = 0$, k – размер массива):

```
void Make_Blns(Tree **p, int n, int k, int *a) {
  if (n == k) { *p = NULL;
                return;
  }
  else {
        int m=(n+k)/2;
        *p = new Tree;
        (*p)->info = a[m];
        Make_Blns( &(*p)->left, n, m, a);
        Make_Blns( &(*p)->right, m+1, k, a);
  }
}
```

Алгоритмы обхода дерева

Существуют три алгоритма обхода деревьев, которые естественно следуют из самой структуры дерева.

1. Обход слева направо: *Left-Root-Right* (сначала посещаем левое поддерево, затем – корень и, наконец, правое поддерево).

2. Обход сверху вниз: *Root-Left-Right* (посещаем корень до поддеревьев).

3. Обход снизу вверх: *Left-Right-Root* (посещаем корень после поддеревьев).

Интересно проследить результаты этих трех обходов на примере записи формулы в виде дерева, т. к. как они и позволяют получить различные формы записи арифметических выражений.

Пусть для операндов *A* и *B* выполняется операция сложения. Привычная форма записи в виде *A* + *B* называется ***инфиксной.*** Форма записи, в которой знак операции следует перед операндами +*AB*, называется ***префиксной***, если же операция записывается после операндов *AB*+ – ***постфиксной***.

Рассмотрим небольшой пример, пусть задано выражение *A*+*B***C*. Т. к.умножение имеет более высокий приоритет, то данное выражение можно переписать в виде *A* + (*B* · *C*). Для записи выражения в постфиксной форме сначала преобразуем ту часть выражения, которая вычисляется первой, в результате получим: *A* + (*BC*·).

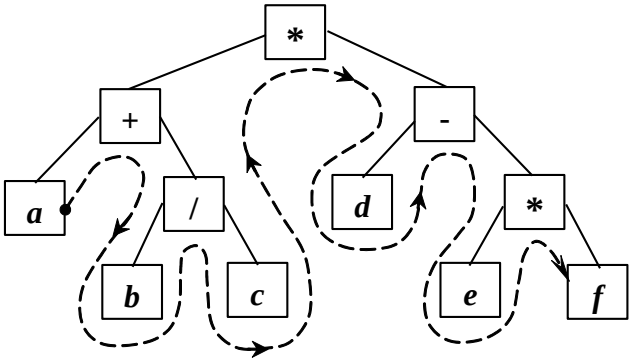
Теперь запишем в постфиксной форме операцию сложения между операндами А и (*BC**): *ABC**+.

Таким образом, выражение *A* + *B* * *C* в постфиксном виде *ABC**+, префиксная форма записи будет иметь вид +**ABC*.

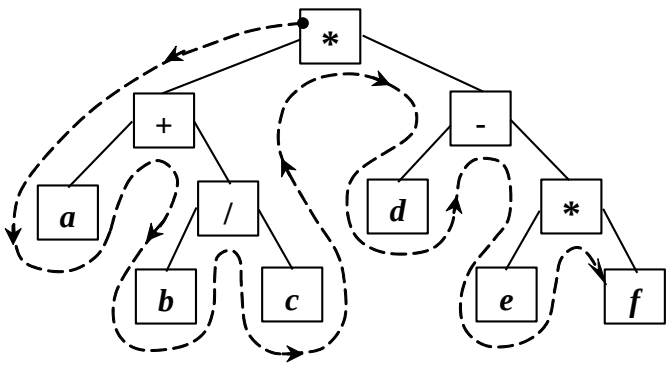
Рассмотрим различные обходы дерева на примере формулы: ((*a*+*b*/*c*) * (*d*– *e***f*)). Дерево формируется по двум принципам выполнится последней;

– узлы – это операции, операнды – это листья дерева.

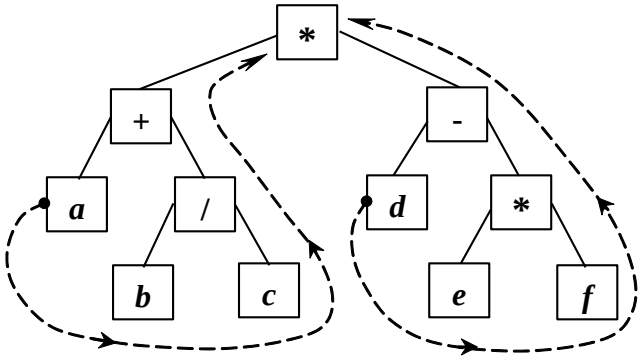
Обход 1 (*Left-Root-Right*) дает обычную инфиксную запись выражения (без скобок): *a* + *b* / *c* * *d* – *e* * *f* .



Обход 2 (*Root-Left-Right*) – имеет префиксную запись выражения (без скобок): * + *a* / *b c* – *d* * *e f* .



Обход 3 (*Left-Right-Root*) – имеет постфиксную запись выражения:
*a b c / + d e f * – * .*



Функция освобождения памяти, занятой деревом

```
void Del_Tree(Tree *t) {
        if ( t != NULL)  {
                Del_Tree ( t -> left);                  // На левую ветвь
                Del_Tree ( t -> right);                 // На правую ветвь
        delete t;
        }
}
```

6.2. Пример выполнения задания

В качестве примера рассмотрим проект (для последовательно введенных ключей 10 (корень), 25, 20, 6, 21, 8, 1, 30), который создает дерево, отображает его в *Memo*, удаляет элемент по ключу и удаляет дерево. Панель диалога будет иметь вид, представленный на рис. 6.2.

Как и в примерах из предыдущих лабораторных работ, приведем только тексты функций-обработчиков соответствующих кнопок:

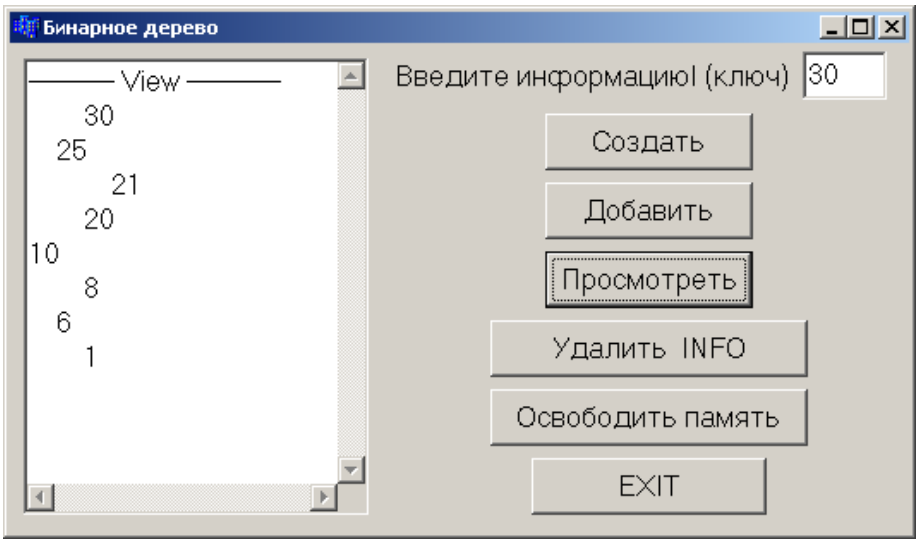


Рис. 6.2

```
//--------------------- Шаблон структуры ---------------------------------------------
struct Tree {
        int info;
        Tree *left, *right;
}*root;                                                         // Корень
//----------------- Декларации прототипов функций работы с деревом ---------
-------
void Add_List(Tree*, int);
void View_Tree (Tree*, int);
Tree* Del_Info(Tree*, int);
void Del_Tree(Tree*);
Tree* List(int);
```

//---------- Текст функции-обработчика кнопки ***Создать*** ----------

```
if(root != NULL) Del_Tree(root);
root = List (StrToInt(Edit1->Text));
```

//---------- Текст функции-обработчика кнопки ***Просмотреть*** -----

```
if( root == NULL ) ShowMessage(" Create TREE !");
else {
    Memo1->Lines->Add("---------- View -----------");
    View_Tree(root, 0);
}
```

//---------- Текст функции-обработчика кнопки ***Добавить*** ----------

```
if(root == NULL) root = List (StrToInt(Edit1->Text));
else Add_List (root, StrToInt(Edit1->Text));
```

//---------- Текст функции-обработчика кнопки ***Удалить INFO*** ----

```
int b = StrToInt(Form1->Edit1->Text);
root = Del_Info(root, b);
```

//---------- Текст функции-обработчика кнопки ***ОЧИСТИТЬ*** -------

```
Del_Tree(root);
ShowMessage(" Tree Delete!");
root = NULL;
```

//---------- Текст функции-обработчика кнопки ***EXIT*** ----------

```
if(root!=NULL){
  Del_Tree(root);
  ShowMessage(" Tree Delete!");
```

```
}
Close();
```

6.3. Индивидуальные задания

Разработать проект для работы с деревом поиска, содержащий следующие обработчики, которые должны:

– ввести информацию из компоненты *StringGrid* в массив. Каждый элемент массива должен содержать строку текста и целочисленный ключ (например, ФИО и номер паспорта);

– внести информацию из массива в дерево поиска;

– сбалансировать дерево поиска;

– добавить в дерево поиска новую запись;

– по заданному ключу найти информацию и отобразить ее;

– удалить из дерева поиска информацию с заданным ключом;

– распечатать информацию прямым, обратным обходом и в порядке возрастания ключа;

– решить одну из поставленных задач и решение оформить в виде блок-схемы.

1. Поменять местами информацию, содержащую максимальный и минимальный ключи.

2. Подсчитать число листьев в дереве (лист – это узел, из которого нет ссылок на другие узлы дерева).

3. Удалить из дерева ветвь с вершиной, имеющей заданный ключ.

4. Определить максимальную глубину дерева, т. е. число узлов в самом длинном пути от корня дерева до листьев.

5. Определить число узлов на каждом уровне дерева.

6. Удалить из левой ветви дерева узел с максимальным значением ключа и все связанные с ним узлы.

7. Определить количество символов во всех строках дерева.

8. Определить число листьев на каждом уровне дерева.

9. Определить число узлов в дереве, у которых есть только один сын.

10. Определить число узлов в дереве, у которых есть две дочери.

11. Определить количество записей в дереве начинающихся с определенной буквы (например, «а»).

12. Найти среднее значение всех ключей дерева и найти строку, имеющую ближайший к этому значению ключ.

13. Между максимальным и минимальным значениями ключей найти запись с ключом, ближайшим к среднему значению.

14. Определить количество записей в левой ветви дерева.

15. Определить количество записей в правой ветви дерева.

16. Определить число листьев в левой ветви дерева.

6.4. Контрольные вопросы

1. Что такое дерево?
2. Что называется глубиной дерева?
3. Какие существуют алгоритмы обхода деревьев?
4. Какое дерево называется бинарным?
5. Что такое сбалансированное дерево?

Лабораторная работа № 7. Решение систем линейных алгебраических уравнений

Цель работы: изучить прямые и численные методы нахождения корней системы линейных алгебраических уравнений (СЛАУ).

7.1. Основные понятия и определения

Выделяют четыре основные задачи линейной алгебры: решение СЛАУ, вычисление определителя матрицы, нахождение обратной матрицы, определение собственных значений и собственных векторов матрицы.

Задача отыскания решения СЛАУ с n неизвестными – одна из наиболее часто встречающихся в практике вычислительных задач, т. к. большинство методов решения сложных задач основано на сведении их к решению некоторой последовательности СЛАУ.

Обычно СЛАУ записывается в виде

$$\sum_{j=1}^{n} a_{i,j} x_j = b_j;\ 1 \le i \le n \quad \text{или коротко}$$

$$A\vec{x} = \vec{b},\ A = \begin{bmatrix} a_{1,1}\ a_{1,2}\ \dots\ a_{1,n} \\ a_{2,1}\ a_{2,2}\ \dots\ a_{2,n} \\ \dots\dots\dots\dots\dots \\ a_{n,1}\ a_{n,2}\ \dots\ a_{n,n} \end{bmatrix};\ \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{bmatrix};\ \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}. \qquad (7.1)$$

Здесь A и $\vec{b}$ заданы, требуется найти $\vec{x}$*, удовлетворяющий (7.1).

Известно, что если определитель матрицы $|A| \neq 0$, то СЛАУ имеет единственное решение. В противном случае либо решение отсутствует (если $\vec{b} \neq 0$), либо имеется бесконечное множество решений (если $\vec{b} = 0$). При решении систем, кроме условия $|A| \neq 0$, важно чтобы задача была ***корректной,*** т. е. чтобы при малых погрешностях правой части $\Delta\vec{b}$ и (или) коэффициентов $\Delta a_{i,j}$ погрешность решения $\Delta\vec{x}^*$ также оставалась малой. Признаком некорректности, или плохой обусловленности, является близость к нулю определителя матрицы.

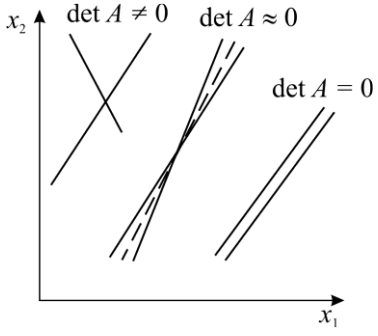


Рис. 7.1

Плохо обусловленная система двух уравнений геометрически соответствует почти параллельным прямым (рис. 7.1). Точка пересечения таких прямых (решение) при малейшей погрешности коэффициентов резко сдвигается. Обусловленность (корректность) СЛАУ характеризуется числом $\chi = \|A\| \cdot \|A^{-1}\| \ge 1$. Чем дальше χ от 1, тем ху-

же обусловлена система. Обычно при $\chi > 10^3$ система некорректна и требует специальных методов решения – методов регуляризации.

Методы решения СЛАУ делятся на прямые и итерационные и применимы только для корректных систем.

Прямые методы дают достаточно точное решение (если не учитывать ошибки округления) за конечное число арифметических операций. Для хорошо обусловленных СЛАУ небольшого порядка $n \le 10^3 - 10^4$ применяются практически только прямые методы.

Наибольшее распространение среди прямых методов получили **метод Гаусса** для СЛАУ общего вида, его модификация для трехдиагональной матрицы – **метод прогонки** и **метод квадратного корня** для СЛАУ с симметричной матрицей.

Итерационные методы основаны на построении сходящейся к точному решению $\vec{x}^*$ рекуррентной последовательности векторов $\vec{x}^0, \vec{x}^1, \vec{x}^2, ..., \vec{x}^k \xrightarrow[k\to\infty]{} \vec{x}^*$.Итерации выполняют до тех пор, пока норма разности $\delta_k = \left\| \vec{x}^k - \vec{x}^{k-1} \right\| = \max \left| x_i^k - x_i^{k-1} \right| \le \varepsilon$. Последнее $\vec{x}^k$ берут в качестве приближенного решения.

Итерационные методы выгодны для систем большого порядка $n > 1000$, а также для решения плохо обусловленных систем. Многообразие итерационных методов решения СЛАУ объясняется возможностью большого выбора рекуррентных последовательностей, определяющих метод. Наибольшее распространение среди итерационных методов получили одношаговые **методы простой итерации и Зейделя** с использованием **релаксации**.

Для контроля полезно найти **невязку** полученного решения $\vec{x}^k$:

$$\Delta = \max_{1 \le i \le n} \left| b_i - \sum_{j=1}^{n} a_{i,j} x_j^k \right|$$

если Δ велико, то это указывает на грубую ошибку в расчете.

Далее приведено описание алгоритмов указанных методов решения СЛАУ.

7.2. Прямые методы решения СЛАУ

Метод Гаусса (MG)

Метод основан на приведении с помощью преобразований, не меняющих решение, исходной СЛАУ (7.1) с произвольной матрицей к СЛАУ с верхней треугольной матрицей вида

$$\begin{aligned} a'_{1,1}x_1 + a'_{1,2}x_2 + \ldots + a'_{1,n}x_n &= b'_1, \\ a'_{2,2}x_2 + \ldots + a'_{2,n}x_n &= b'_2, \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots & \\ a'_{n,n}x_n &= b'_n. \end{aligned} \quad . \qquad (7.2)$$

Этап приведения к системе с треугольной матрицей называется ***прямым ходом метода Гаусса*.**

Решение системы с верхней треугольной матрицей (7.2), как легко видеть, находится по формулам, называемым ***обратным ходом метода Гаусса*:**

$$x_n = b'_n / a'_{n,n};\ x_k = \frac{\left[b'_k - \sum_{i=k+1}^{n} a'_{k,i}x_i \right]}{a'_{k,k}}, k = n-1, n-2, \ldots, 1. \qquad (7.3)$$

Прямой ход метода Гаусса осуществляется следующим образом: вычтем из каждого *m*-го уравнения *(m = 2 ... n)* первое уравнение, умноженное на $a_{m,1} / a_{1,1}$ и вместо *m*-го уравнения оставим полученное. В результате в матрице системы исключаются все коэффициенты первого столбца ниже диагонального. Затем, используя второе полученное уравнение, аналогично исключим элементы второго столбца *(m = 3, ..., n)* ниже диагонального и т. д. Такое исключение называется ***циклом метода Гаусса***. Проделывая последовательно эту операцию с расположенными ниже *k*-го уравнениями (*k*=1, 2, ..., *n* – 1), приходим к системе вида (7.2). При указанных операциях решение СЛАУ не изменяется.

На каждом *k*-м шаге преобразований прямого хода элементы матриц изменяются по ***формулам прямого хода метода Гаусса***:

$$\begin{aligned} a_{m,i} &= a_{m,i} - a_{k,i}\frac{a_{m,k}}{a_{k,k}},\ k = 1, n-1,\ i = k, n; \\ b_m &= b_m - b_k\frac{a_{m,k}}{a_{k,k}},\ m = k+1, n. \end{aligned} \qquad (7.4)$$

Элементы $a_{k,k}$ называются *главными*. Заметим, что если в ходе расчетов по данному алгоритму на главной диагонали окажется нулевой элемент $a_{k,k} = 0$, то произойдет сбой в ЭВМ. Для того чтобы этого избежать, следует каждый цикл по *k* начинать с перестановки строк: среди элементов *k*-го столбца $a_{m,k}$, $k \le m \le n$ находят номер *p* главного, т. е. наибольшего по модулю, и меняют местами строки *k* и *p*. Такой выбор главного элемента значительно повышает устойчивость алгоритма к ошибкам округления, т. к. в формулах (7.4) при этом производится умножение на числа $a_{m,k} / a_{k,k}$, меньшие единицы, и погрешность, возникшая ранее, уменьшается.

Схема алгоритма с выбором главного элемента приведена на рис. 7.2.

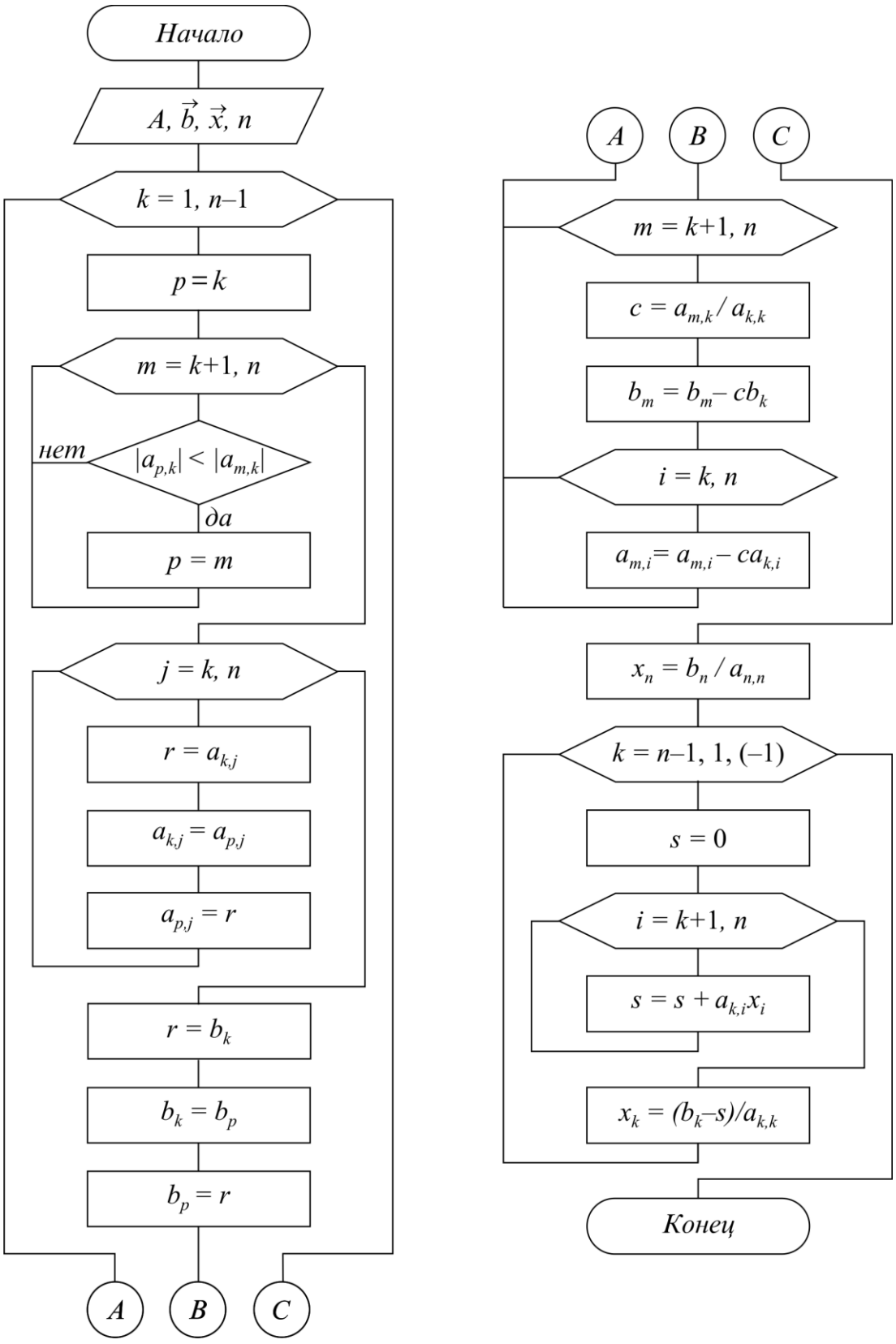

Начало
A, b, x, n
k = 1, n–1
p = k
m = k+1, n
нет
|a_p,k| < |a_m,k|
да
p = m
j = k, n
r = a_k,j
a_k,j = a_p,j
a_p,j = r
r = b_k
b_k = b_p
b_p = r
A
B
C
A
B
C
m = k+1, n
c = a_m,k / a_k,k
b_m = b_m – cb_k
i = k, n
a_m,i = a_m,i – ca_k,i
x_n = b_n / a_n,n
k = n–1, 1, (–1)
s = 0
i = k+1, n
s = s + a_k,i x_i
x_k = (b_k–s)/a_k,k
Конец

Рис. 7.2

Проиллюстрируем метод Гаусса на решении СЛАУ третьего порядка:

$$2x_1 + x_2 - x_3 = 1;$$
$$4x_1 + 6_2 + 2x_3 = 6;$$
$$6x_1 + 5x_2 + 8x_3 = 14.$$

Первый цикл: вычтем из второго уравнения первое, умноженное на $a_{2,1} / a_{1,1} = 2$, а из третьего – первое, умноженное на $a_{3,1} / a_{1,1} = 3$ получим

$$2x_1 + x_2 - x_3 = 1;$$
$$0 + 4_2 + 4x_3 = 4;$$
$$0 + 2x_2 + 11x_3 = 11.$$

Второй цикл: вычтем из третьего уравнения второе, умноженное на $a_{3,2} / a_{2,2} = 0.5$; получим систему с треугольной матрицей вида (7.2):

$$2x_1 + x_2 - x_3 = 1;$$
$$0 + 4_2 + 4x_3 = 4;$$
$$0 + 0 \ + 9x_3 = 9.$$

Обратный ход: из последнего уравнения находим $x_3 = 1$, подставляя его во второе, находим $x_2 = 0$, подставляя его в первое, находим $x_1 = 1$.

Таким образом, получен вектор решения $\vec{x}^* = (x_1^*, x_2^*, x_3^*) = (1, 0, 1)$.

Метод прогонки (МР)

Многие задачи (например, решение дифференциальных уравнений второго порядка) приводят к необходимости решения СЛАУ с трехдиагональной матрицей:

$$\begin{vmatrix} q_1 & r_1 & 0 & 0 & \dots & 0 & 0 & 0 \\ p_2 & q_2 & r_2 & 0 & \dots & 0 & 0 & 0 \\ 0 & p_3 & q_3 & r_3 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & 0 & \dots & p_n & q_n & r_n \\ 0 & 0 & 0 & 0 & \dots & 0 & q_{n1} & r_{n1} \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_n \\ x_{n1} \end{vmatrix} = \begin{vmatrix} d_1 \\ d_2 \\ d_3 \\ \dots \\ d_n \\ d_{n1} \end{vmatrix}. \qquad (7.5)$$

или коротко эту систему записывают в виде

$$\begin{aligned} & q_1 x_1 + r_1 x_2 = d_1; \\ & p_i x_{i-1} + q_i x_i + r_i x_{i+1} = d_i; \\ & p_{n1} x_n + q_{n1} x_{n1} = d_{n1}; \\ & n1 = n+1, \ 2 \le i \le n. \end{aligned} \qquad (7.6)$$

В этом случае расчетные формулы метода Гаусса значительно упрощаются. После исключения поддиагональных элементов в результате прямого хода метода Гаусса и последующего деления каждого уравнения на диагональный элемент систему (7.5) можно привести к виду

$$\begin{vmatrix} 1 & -\xi_1 & 0 & \dots & 0 & \dots & 0 & 0 \\ 0 & 1 & -\xi_2 & \dots & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & 0 & \dots & 1 & -\xi_n \\ 0 & 0 & 0 & \dots & 0 & \dots & 0 & 1 \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ \dots \\ x_n \\ x_{n1} \end{vmatrix} = \begin{vmatrix} \eta_1 \\ \eta_2 \\ \dots \\ \eta_n \\ \eta_{n1} \end{vmatrix} . \qquad (7.7)$$

При этом ***формулы прямого хода*** для вычисления ξ_i, η_i имеют вид

$$\xi_1 = -r_1 / q_1, \qquad \eta_1 = d_1 / q_1,$$
$$\xi_i = -r_i / (q_i + p_i\xi_{i-1}); \qquad \eta_i = (d_i - p_i\eta_{i-1}) / (q_1 + p_i\xi_{i-1}); \qquad (7.8)$$
$$i = 2, ..., n.$$

Когда такое преобразование (прямой ход) сделано, ***формулы обратного хода*** метода Гаусса получают в виде

$$x_{n1} = (d_{n1} - p_{n1}\eta_n) / (q_{n1} + p_{n1}\xi_n);$$
$$x_i = \xi_i x_{i+1} + \eta_i; \qquad (7.9)$$
$$i = n, n-1, ..., 1.$$

Расчетные формулы (7.8), (7.9) получили название «метод прогонки». Достаточным условием того, что в формулах метода прогонки не произойдет деления на нуль и расчет будет устойчив относительно погрешностей округления, является выполнение неравенства $|q_i| \geq |p_i| + |r_i|$ (хотя бы для одного i должно быть строгое неравенство).

Схема алгоритма метода прогонки представлена на рис. 7.3.

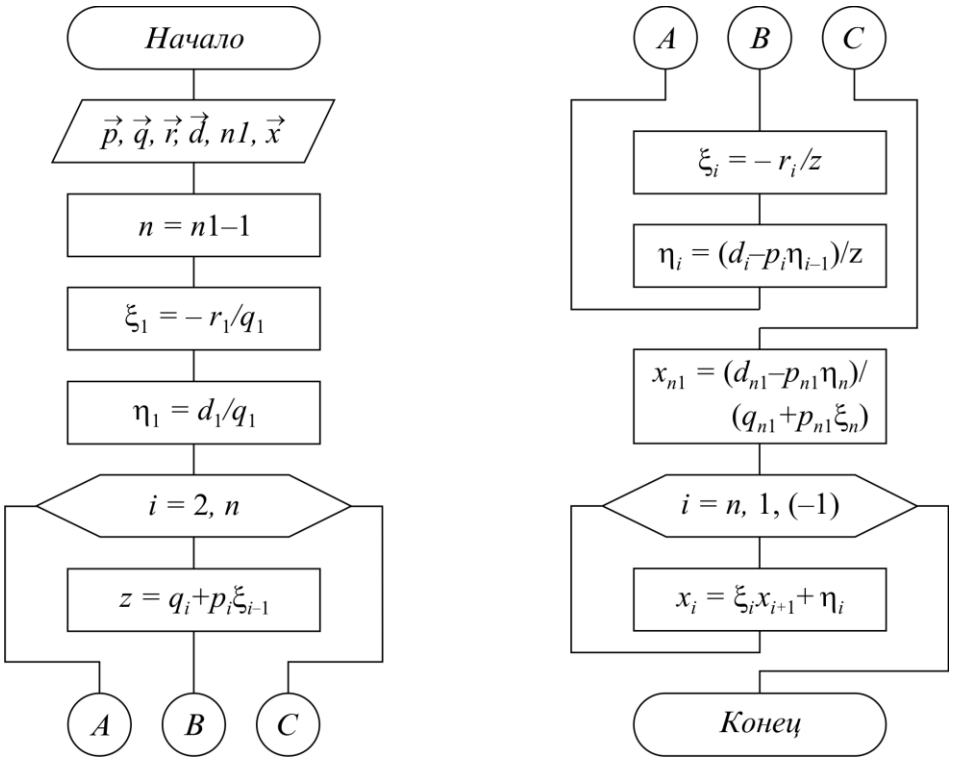


Рис. 7.3

Метод квадратного корня (MQ)

Метод предназначен для решения СЛАУ с симметричной матрицей и основан на представлении такой матрицы в виде произведения трех матриц: $A = S^T \cdot D \cdot S$, где D – диагональная с элементами d_i=±1; S – верхняя треугольная ($s_{i,k} = 0$, если $i>k$, причем $s_{i,i} > 0$); S^T – транспонированная нижняя треугольная. Матрицу S можно по аналогии с числами трактовать как корень квадратный из матрицы A, отсюда и название метода.

Если S и D известны, то решение исходной системы $A \cdot \vec{x} = S^T \cdot D \cdot S \cdot \vec{x} = \vec{b}$ сводится к последовательному решению трех систем – двух треугольных и одной диагональной:

$$S^T \cdot \vec{z} = \vec{b};\ D \cdot \vec{y} = \vec{z};\ \vec{S} \cdot \vec{x} = \vec{y}. \qquad (7.10)$$

Здесь $\vec{z} = D \cdot S \cdot \vec{x},\ \vec{y} = S \cdot \vec{x}$.

Решение систем (7.10) ввиду треугольности матрицы S осуществляется по формулам, аналогичным обратному ходу метода Гаусса:

$$y_1 = b_1 / s_{1,1} d_1;\ \ y_i = (b_i - \sum_{k=1}^{i-1} d_k y_k s_{k,i}) / s_{i,i} d_i;\ i = 2, 3, ..., n;$$

$$x_n = y_n / s_{n,n};\ \ x_i = (y_i - \sum_{k=i+1}^{n} x_k s_{i,k}) / s_{i,i};\ i = n-1, n-2, ..., 1.$$

Нахождение элементов матрицы S (извлечение корня из А) осуществляется по рекуррентным формулам:

$$d_k = \text{sign}(a_{k,k} - \sum_{i=1}^{k-1} d_i \left| s_{i,k} \right|^2;$$

$$s_{k,k} = \sqrt{\left| a_{k,k} - \sum_{i=1}^{k-1} d_i \left| s_{i,k} \right|^2 \right|};\ k = 1, 2, ..., n; \qquad (7.11)$$

$$s_{k,j} = (a_{k,j} - \sum_{i=1}^{k-1} d_i s_{i,k} s_{i,j}) / (s_{k,k} d_k);\ j = k+1, k+2, ..., n.$$

В этих формулах сначала полагаем k = 1 и последовательно вычисляем $d_1 = \text{sign}(a_{1,1}),\ s_{1,1} = \sqrt{\left| a_{1,1} \right|}$ и все элементы первой строки матрицы $S(s_{1,j}, j > 1)$, затем полагаем k = 2, вычисляем $s_{2,2}$ и вторую строку $s_{1,j}$ для j > 2 и т. д.

Метод квадратного корня почти вдвое эффективнее метода Гаусса, т. к. полезно использует симметричность матрицы.

Схема алгоритма метода квадратного корня представлена на рис. 7.4. Функция **sign(*x*)** возвращает –1 для всех $x < 0$ и +1 для всех $x > 0$.

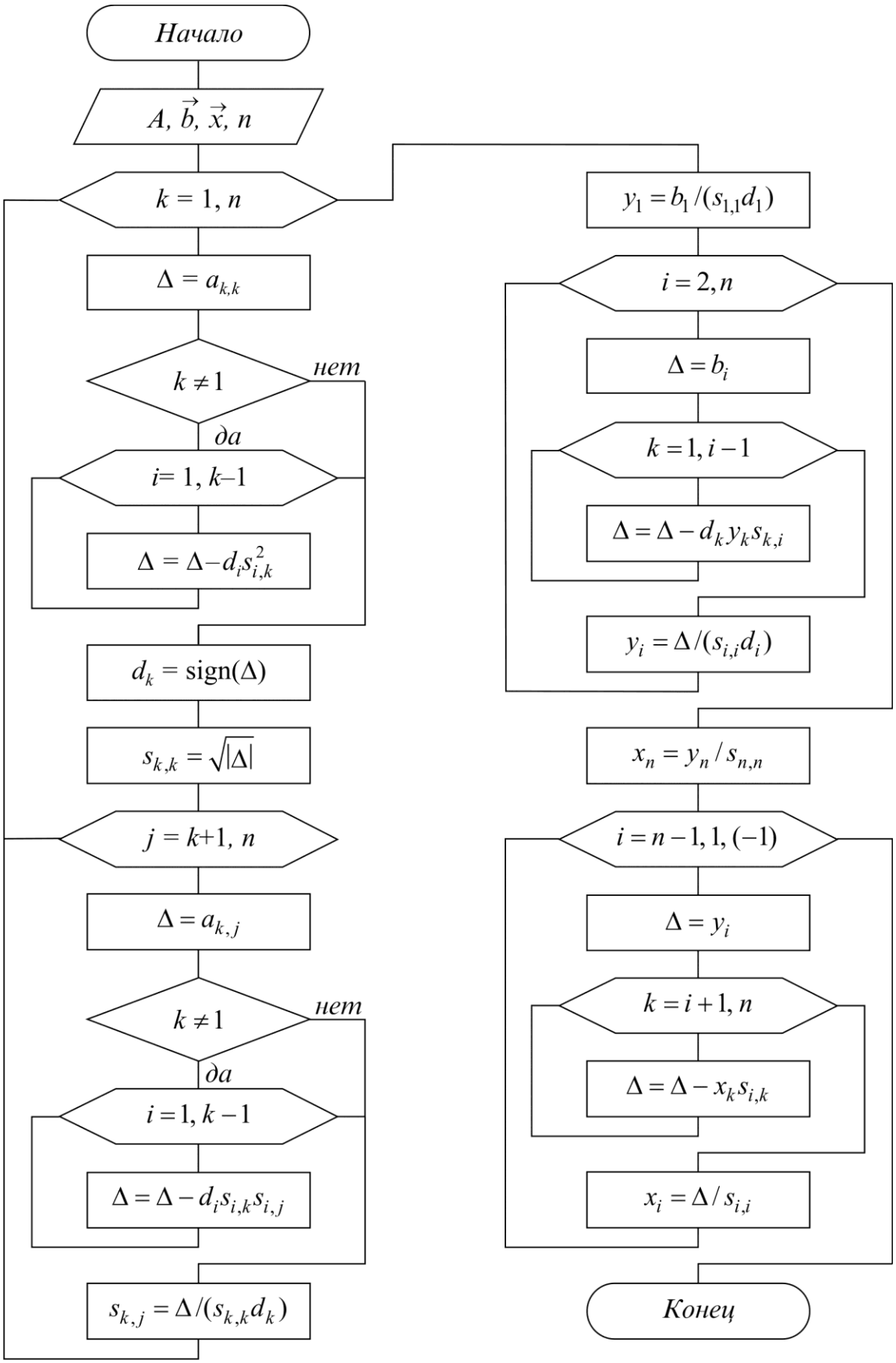

Начало
A, b, x, n
k = 1, n
Δ = a_{k,k}
k ≠ 1
нет
да
i= 1, k–1
Δ = Δ–d_i s²_{i,k}
d_k = sign(Δ)
s_{k,k} = √|Δ|
j = k+1, n
Δ = a_{k,j}
k ≠ 1
нет
да
i = 1, k – 1
Δ = Δ – d_i s_{i,k} s_{i,j}
s_{k,j} = Δ/(s_{k,k} d_k)
y_1 = b_1/(s_{1,1} d_1)
i = 2, n
Δ = b_i
k = 1, i – 1
Δ = Δ – d_k y_k s_{k,i}
y_i = Δ/(s_{i,i} d_i)
x_n = y_n / s_{n,n}
i = n – 1, 1, (–1)
Δ = y_i
k = i + 1, n
Δ = Δ – x_k s_{i,k}
x_i = Δ / s_{i,i}
Конец

Рис. 7.4

Проиллюстрируем метод квадратного корня, решая систему трех уравнений:

$$\begin{cases} x_1 + x_2 + x_3 = 3 \\ x_1 + 2x_2 + 2x_3 = 5 \\ x_1 + 2x_2 + 3x_3 = 6 \end{cases} \qquad A = \begin{bmatrix} 1\,1\,1 \\ 1\,2\,2 \\ 1\,2\,3 \end{bmatrix}, \qquad b = \begin{bmatrix} 3 \\ 5 \\ 6 \end{bmatrix}. \tag{7.12}$$

Нетрудно проверить, что матрица A есть произведение двух треугольных матриц (здесь $d_i = 1$):

$$A = \begin{bmatrix} 1\,1\,1 \\ 1\,2\,2 \\ 1\,2\,3 \end{bmatrix} = \begin{bmatrix} 1\,0\,0 \\ 1\,1\,0 \\ 1\,1\,1 \end{bmatrix} \times \begin{bmatrix} 1\,1\,1 \\ 0\,1\,1 \\ 0\,0\,1 \end{bmatrix} = S^T \cdot S.$$

Исходную систему запишем в виде

$$S^T \cdot S \cdot \vec{x} = \begin{bmatrix} 1\,0\,0 \\ 1\,1\,0 \\ 1\,1\,1 \end{bmatrix} \times \begin{bmatrix} 1\,1\,1 \\ 0\,1\,1 \\ 0\,0\,1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 6 \end{bmatrix}.$$

Обозначим

$$S \cdot \vec{x} = \begin{bmatrix} 1\,1\,1 \\ 0\,1\,1 \\ 0\,0\,1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}.$$

Тогда для вектора $\vec{y}$ получим систему $S^T \vec{y} = \vec{b_i}$:

$$\begin{bmatrix} 1\,0\,0 \\ 1\,1\,0 \\ 1\,1\,1 \end{bmatrix} \times \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 5 \\ 6 \end{bmatrix} \Rightarrow y_1 = 3,\ y_2 = 2,\ y_3 = 1.$$

Зная $\vec{y}$, решаем систему $S\vec{x} = \vec{y}$:

$$\begin{bmatrix} 1\,1\,1 \\ 0\,1\,1 \\ 0\,0\,1 \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \\ 1 \end{bmatrix} \Rightarrow x_3 = 1,\ x_2 = 1,\ x_1 = 1.$$

7.3. Итерационные методы решения СЛАУ

Метод простой итерации (МИ)

В соответствии с общей идеей итерационных методов исходная система (7.1) должна быть приведена к виду, разрешенному относительно $\vec{x}$:

$$\vec{x} = G\vec{x} + \vec{c} = \varphi(\vec{x}), \tag{7.13}$$

где G – матрица; $\vec{c}$ – столбец свободных членов.

При этом решение (7.13) должно совпадать с решением (7.1). Затем строится рекуррентная последовательность первого порядка в виде

$$\vec{x}^k = \varphi(\vec{x}^{k-1}) = G\vec{x}^{k-1} + \vec{c}, \quad k = 1,\ 2,\ \ldots .$$

Для начала вычислений задается некоторое начальное приближение $\vec{x}^0$ (например $x_1^0 = 1,\ \ldots,\ x_n^0 = 1$), для окончания – некоторое малое ε. Получаемая последовательность будет сходиться к точному решению, если норма матрицы $\|G\| < 1$.

Привести исходную систему к виду (7.13) можно различными способами, например:

$$\vec{x} = \vec{x} + \alpha(A\vec{x} - \vec{b}) = (E + \alpha A)\vec{x} - \alpha\vec{b} = G\vec{x} + \vec{c}.$$

Здесь E – единичная матрица; α – некоторый параметр, подбирая который можно добиться, чтобы $\|G\| = \|E + \alpha A\| < 1$.

В частном случае, если исходная матрица A имеет преобладающую главную диагональ, т. е. $|a_{i,i}| > \sum\limits_{\substack{k=1 \\ k\neq i}}^{n} |a_{i,k}|$, то преобразование (7.1) к (7.13) можно осуществить просто, решая каждое i-е уравнение относительно x_i. В результате получим следующую рекуррентную формулу:

$$x_i^k = -\frac{1}{a_{i,i}}\left[\sum_{\substack{j=1 \\ j\neq i}}^{n} a_{i,j}x_j^{k-1} - b_i\right] = \sum_{j=1}^{n} g_{i,j}x_j^{k-1} + c_i, \qquad (7.14)$$

$$g_{i,j} = -a_{i,j}/a_{i,i}; \quad g_{i,i} = 0; \quad c_i = b_i/a_{i,i}.$$

Схема алгоритма представлена на рис. 7.5.

Приведем пример решения методом простой итерации следующей системы трех уравнений:

$$\begin{cases} 4x_1 - x_2 - x_3 = 2; \\ x_1 + 5x_2 - 2x_3 = 4; \\ x_1 + x_2 + 4x_3 = 6. \end{cases} \qquad (7.15)$$

Преобразуем ее к виду (7.13), для чего из первого уравнения выразим x_1, из второго – x_2 и из третьего – x_3 и получим рекуррентную формулу

$$\begin{cases} x_1 = (2 + x_2 + x_3)/4 \\ x_2 = (4 - x_1 + 2x_3)/5 \\ x_3 = (6 - x_1 - x_2)/4 \end{cases} \Rightarrow \begin{cases} x_1^k = (2 + x_2^{k-1} + x_3^{k-1})/4; \\ x_2^k = (4 - x_1^{k-1} + 2x_3^{k-1})/5; \\ x_3^k = (6 - x_1^{k-1} - x_2^{k-1})/4. \end{cases} \qquad (7.16)$$

Зададим начальное приближение $\vec{x}^0 = (x_1^0,\ x_2^0,\ x_3^0) = (0,\ 0,\ 0)$, подставим в правую часть (k = 1), получим $\vec{x}^1$: $(x_1^1 = 0.5, \quad x_2^1 = 0.8, \quad x_3^1 = 1.5)$.

Подставляя полученные значения снова в (7.16) (k = 2), получим $\vec{x}^2$:

$$x_1^2 = (2 + 0.8 + 1.5)/4 = 1.075;$$
$$x_2^2 = (4 - 0.5 + 2 \cdot 1.5)/5 = 1.3;$$
$$x_3^2 = (6 - 0.5 - 0.8)/4 = 1.175;$$

т. е. $\vec{x}^2 = (1.075;\ 1.3;\ 1.175)$.

Вычислим погрешность на второй итерации:

$$\delta_2 = \left\| \vec{x}^2 - \vec{x}^1 \right\|_C = \max_{1 \le i \le 3} \left| x_i^2 - x_i^1 \right| = \max(0.575;\ 0.5;\ 0.325) = 0.575.$$

Если $\delta > \varepsilon$, то процесс продолжаем.

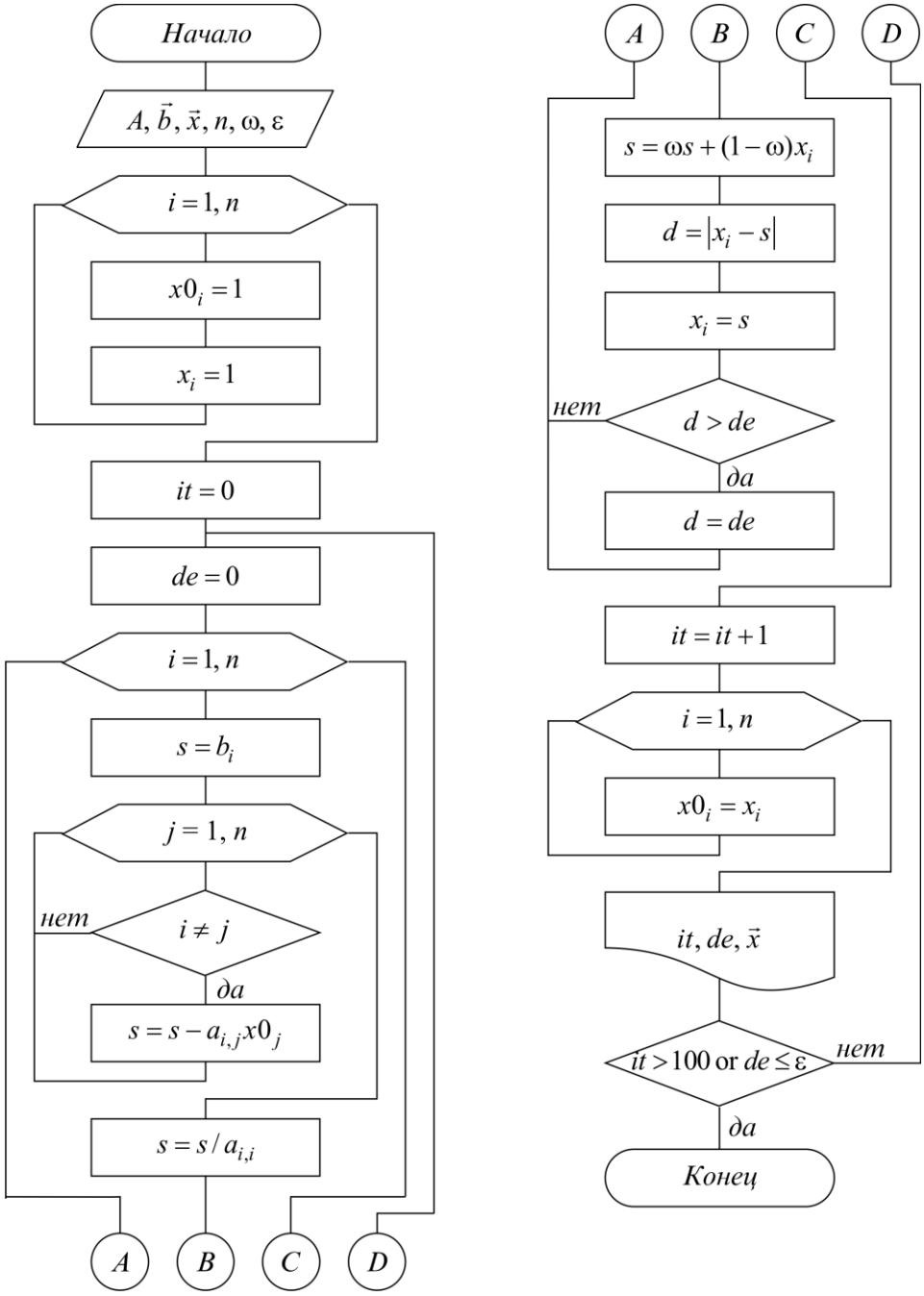


Рис. 7.5

Метод Зейделя (MZ)

Метод Зейделя является модификацией метода простой итерации. Суть его состоит в том, что при вычислении очередного приближения $x_i^k (2 \le i \le n)$ в формуле (7.14) используются вместо x_1^{k-1}, ..., x_{i-1}^{k-1} уже вычисленные ранее x_1^k, ..., x_{i-1}^k, т. е. (7.14) преобразуется к виду

$$x_i^k = \sum_{j=1}^{i-1} g_{i,j} x_j^k + \sum_{j=i+1}^{n} g_{i,j} x_j^{k-1} + c_i, \quad i = 1, ..., n. \tag{7.17}$$

Такое усовершенствование позволяет ускорить сходимость итераций почти в два раза. Кроме того, данный метод может быть реализован на ЭВМ без привлечения дополнительного массива, т. к. полученное новое x_i^k сразу засылается на место старого.

Схема алгоритма аналогична схеме метода простой итерации (см. рис. 7.5), если x_{0j} заменить на x_j и убрать строки $x_{0i}=1$, $x_{0i}=x_i$.

Для иллюстрации решим систему (7.15) методом Зейделя. При этом относительно $(x_1,\ x_2,\ x_3)$ аналогично (7.16) получаем рекуррентную формулу (7.17) вида

$$\begin{aligned} x_1^k &= (2 + x_2^{k-1} + x_3^{k-1})/4; \\ x_2^k &= (4 - x_1^k + 2x_3^{k-1})/5; \\ x_3^k &= (6 - x_1^k - x_2^k)/4. \end{aligned} \tag{7.18}$$

Зададим, как и в методе простой итерации, начальное условие $\vec{x}^0 = (0,\ 0,\ 0)$, подставим в первое уравнение, получаем $x_1^1 = 0.5$. При вычислении x_2^1 это значение сразу используем: $x_2^1 = (4 - 0.5 + 0)/4 = 0.7$, аналогично $x_3^1 = (6 - 0.5 - 0.7)/4 = 1.2$ получаем $\vec{x}^1 = (0.5;\ 0.7;\ 1.2)$. Для второй итерации:

$$\begin{aligned} x_1^2 &= (2 + 0.5 + 1.2)/4 = 0.925; \\ x_2^2 &= (4 - 0.925 + 2 \cdot 1.2)/5 = 1.085; \\ x_3^2 &= (6 - 0.925 - 1.085)/4 = 0.998; \\ \vec{x}^2 &= (0.925;\ 1.085;\ 0.998). \end{aligned}$$

Погрешность после двух итераций $\delta = \left\| \vec{x}^2 - \vec{x}^1 \right\| = 0.425$ меньше, чем в методе простой итерации.

7.4. Понятие релаксации

Методы простой итерации и Зейделя сходятся примерно так же, как геометрическая прогрессия со знаменателем $\|G\|$. Если норма матрицы G близка к единице, то сходимость очень медленная. Для ускорения сходимости исполь-

зуется метод релаксации. Суть его в том, что полученное по методу простой итерации или Зейделя очередное значение x_i^k пересчитывается по формуле

$$x_i^k = \omega x_i^k + (1-\omega)x_i^{k-1}. \qquad (7.19)$$

Здесь $0 < \omega \le 2$ – параметр релаксации.

Если $\omega < 1$ – *нижняя релаксация,* если $\omega > 1$ – *верхняя релаксация.* Параметр ω подбирают так, чтобы сходимость метода достигалась за минимальное число итераций.

Схема алгоритма на рис. 7.5 выполнена с использованием релаксации.

7.5. Индивидуальные задания

Составить программу решения СЛАУ порядка *n* и решить систему линейных уравнений пятого порядка с трехдиагональной симметричной матрицей вида

$$\begin{vmatrix} q & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & q \end{vmatrix} \times \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{vmatrix} = \begin{vmatrix} 0 \\ d \\ d \\ d \\ 0 \end{vmatrix}.$$

Значения *q* и *d* заданы в табл. 7.1.

В вариантах, использующих прямые методы, вычислить невязку (см. подразд. 7.1).

В вариантах, использующих итерационные методы, построить график зависимости количества итераций *it*, необходимых для достижения заданной точности, от параметра релаксации ω и установить, при каком значении ω число итераций минимально. Параметр релаксации менять от 0.2 до 2 с шагом 0.2.

Таблица 7.1

Индивидуальные задания.

Номер варианта	***Метод***	***d***	***q***	***ε***
1	MG	–1	–4.25	–
2	MI	1	–3.17	10^{-3}
3	MQ	–2	–3.23	–
4	MZ	2	–2.57	10^{-4}
5	MP	–3	–4.67	–
6	MI	3	–2.23	10^{-4}
7	MG	–4	–2.75	–

Окончание табл. 7.1

Номер варианта	***Метод***	***d***	***q***	***ε***
8	MI	4	–2.25	10^{-4}
9	MQ	–5	–2.85	–
10	MZ	5	–2.24	10^{-5}
11	MP	–6	–3.83	–
12	MZ	6	–2.17	10^{-4}
13	MG	–7	–2.86	–
14	MI	7	–3.14	10^{-3}
15	MQ	–8	–4.88	–

7.6. Контрольные вопросы

1. Что понимается под корректностью СЛАУ?
2. Решите по методу Гаусса заданную систему из трех уравнений.
3. В чем суть метода квадратного корня?
4. Когда используются методы прогонки и квадратного корня?
5. Решите заданную систему трех уравнений методом простой итерации и методом Зейделя.
6. Для чего нужна релаксация? Ее суть.

Лабораторная работа №8. Аппроксимация функций

Цель работы: изучить алгоритмы аппроксимации функций; освоить методику построения и использования алгебраических интерполяционных многочленов Лагранжа и Ньютона.

8.1. Задачи аппроксимации функций

В окружающем нас мире все взаимосвязано, поэтому установление характера зависимости между различными величинами позволяет по значению одной величины определить значение другой. Математической моделью зависимости одной величины от другой является функция $y=f(x)$.

В практике расчетов, связанных с обработкой экспериментальных данных, вычислением $f(x)$, разработкой вычислительных методов, встречаются два вопроса:

1. Как установить вид функции $y=f(x)$, если она неизвестна? Предполагается при этом, что задана таблица ее значений $\{ (x_i,\ y_i),\ i=1,\ m \}$, которая получена либо из экспериментальных измерений, либо из сложных расчетов.

2. Как упростить вычисление известной функции $f(x)$ или же ее характеристик ($f'(x)$, $\max f(x)$), если $f(x)$ слишком сложная?

Ответы на эти вопросы даются при помощи теории аппроксимации функций, ***основная задача*** которой состоит в нахождении функции $y = \varphi(x)$, близкой (т. е. аппроксимирующей) в некотором нормированном пространстве к исходной функции $y = f(x)$. Функцию $\varphi(x)$ при этом выбирают такой, чтобы она была максимально удобной для последующих расчетов.

Основной подход к решению этой задачи заключается в том, что $\varphi(x)$ выбирается зависящей от нескольких свободных параметров $\vec{\tilde{n}} = (\tilde{n}_1, \tilde{n}_2, ..., \tilde{n}_n)$, т. е. $y = \varphi(x) = \varphi(x, \tilde{n}_1, \tilde{n}_2, ..., \tilde{n}_n) = \varphi(x, \vec{\tilde{n}})$, значения которых подбираются из некоторого условия близости $f(x)$ и $\varphi(x)$.

Обоснование способов нахождения удачного вида функциональной зависимости $\varphi(x, \vec{\tilde{n}})$ и подбора параметров $\vec{\tilde{n}}$ составляет задачу ***теории аппроксимации функций.***

В зависимости от способа подбора параметров $\vec{n}$ получают различные ***методы аппроксимации***; наибольшее распространение среди них получили ***интерполяция*** и ***среднеквадратичное приближение,*** частным случаем которого является ***метод наименьших квадратов.***

Наиболее простой, хорошо изученной и нашедшей широкое применение в настоящее время является ***линейная аппроксимация,*** при которой выбирают функцию $\varphi(x, \vec{\tilde{n}})$, линейно зависящую от параметров $\vec{\tilde{n}}$, т. е. в виде обобщенного многочлена:

$$\varphi(x,\ \tilde{n}) = \tilde{n}_1\varphi_1(x) + ... + \tilde{n}_n\varphi_n(x) = \sum_{k=1}^{n} \tilde{n}_k\varphi_k(x). \qquad (8.1)$$

Здесь $\{\varphi_1(x),\ ...,\ \varphi_n(x)\}$ – известная система линейно независимых (базисных) функций. В качестве $\{\varphi_k(x)\}$ могут быть выбраны любые элементарные функции, например: тригонометрические, экспоненты, логарифмические или комбинации таких функций. Важно, чтобы система базисных функций была *полной*, т. е. обеспечивающей аппроксимацию *f(x)* многочленом (8.1) с заданной точностью при $n \to \infty$.

Приведем хорошо известные и часто используемые системы. При интерполяции обычно используется система линейно независимых функций $\{\varphi_k(x) = x^{k-1}\}$. Для среднеквадратичной аппроксимации удобнее в качестве $\varphi_k(x)$ брать ортогональные на интервале [–1, 1] многочлены Лежандра:

$$\{\varphi_1(x) = 1;\ \varphi_2(x) = x;\ \varphi_{k+1}(x) = [(2k+1)x\varphi_k(x) - k\varphi_{k-1}(x)],\ \ k = 2,3,\ ...,\ n\};$$

$$\int_{-1}^{1} \varphi_k(x)\cdot\varphi_l(x)\,dx = 0;\ \ k \neq l.$$

Заметим, что если функция $f(x)$ задана на отрезке [*a, b*], то при использовании этой системы необходимо предварительно осуществить преобразование координат $x' = \left(x - \frac{b+a}{2}\right)\frac{2}{b-a}$, приводящее интервал $a \le x \le b$ к интервалу $-1 \le x' \le 1$.

Для аппроксимации периодических функций используют ортогональную на отрезке [*a, b*] систему тригонометрических функций $\left\{\varphi_k(x) = \cos\left(2k\pi\frac{x-a}{b-a}\right), \varphi_k(x) = \sin\left(2k\pi\frac{x-a}{b-a}\right)\right\}$. В этом случае обобщенный многочлен (8.1) записывается в виде $y = \sum_{k=1}^{n} c_k\varphi_k(x) + d_k\psi_k(x)$.

8.2. Суть интерполяции

Интерполяция является одним из способов аппроксимации функций. Суть ее состоит в следующем. В области значений *x*, представляющей некоторый интервал [*a, b*], где функции *f* и φ должны быть близки, выбирают упорядоченную систему точек (узлов) $x_1 < x_2 < ... < x_n$ (обозначим $\vec{x} = (x_1,\ ...,\ x_n)$), число которых равно количеству искомых параметров $\tilde{n}_1,\ \tilde{n}_2,\ ...,\tilde{n}_n$. Далее параметры $\vec{\tilde{n}}$ подбирают такими, чтобы функция $\varphi(x,\vec{\tilde{n}})$ совпадала с $f(x)$ в этих узлах, $\varphi(x_i,\vec{\tilde{n}}) = f(x_i), i = 1, ..., n$ (рис. 8.1), для чего решают полученную систему из *n* алгебраических в общем случае нелинейных уравнений.

В случае линейной аппроксимации (8.1) система для нахождения коэффициентов $\vec{\tilde{n}}$ линейна и имеет следующий вид:

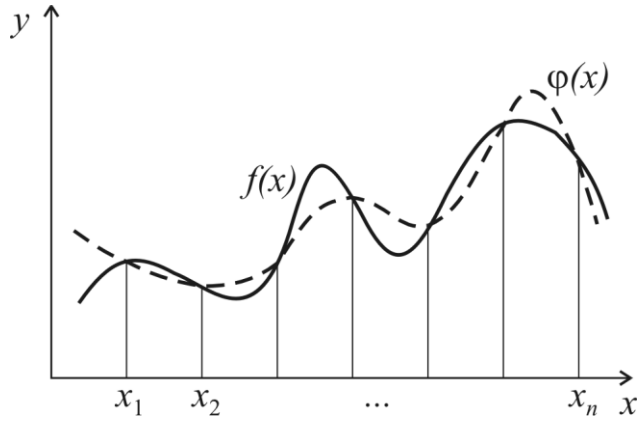


Рис. 8.1

$$\sum_{k=1}^{n} \tilde{n}_k \varphi_k(x_i) = y_i; \qquad (8.2)$$

$$i = 1,\ 2,\ ...,\ n;\ \ y_i = f(x_i).$$

Система базисных функций $\{\varphi_k(x)\}$, используемых для интерполяции, должна быть **чебышевской**, т. е. такой, чтобы определитель матрицы системы (8.2) был отличен от нуля и, следовательно, задача интерполяции имела единственное решение.

Для большинства практически важных приложений при интерполяции наиболее удобны обычные алгебраические многочлены, т. к. они легко обрабатываются.

Интерполяционным многочленом называют алгебраический многочлен степени $n-1$, совпадающий с аппроксимируемой функцией в выбранных n точках.

Общий вид алгебраического многочлена

$$\varphi(x,\ \vec{\tilde{n}}) = P_{n-1}(x) = \tilde{n}_1 + \tilde{n}_2 x + \tilde{n}_3 x^2 + ... + \tilde{n}_n x^{n-1} = \sum_{k=1}^{n} a_k x^{k-1}. \qquad (8.3)$$

Матрица системы (8.2) в этом случае имеет вид

$$G = \begin{bmatrix} 1 & x_1 & .. & x_1^{n-1} \\ 1 & x_2 & .. & x_2^{n-1} \\ .. & .. & .. & .. \\ 1 & x_n & .. & x_n^{n-1} \end{bmatrix}; \quad |G| = \prod_{n \ge k > m \ge 0} (x_k - x_m), \qquad (8.4)$$

и ее определитель (это определитель Вандермонда) отличен от нуля, если точки x_i разные. Поэтому задача (8.2) имеет единственное решение, т. е. для заданной системы различных точек существует единственный интерполяционный многочлен.

Погрешность аппроксимации функции $f(x)$ интерполяционным многочленом степени $n-1$, построенным по n точкам, можно оценить, если известна ее производная порядка n, по формуле

$$\varepsilon = \| f(x) - P_{n-1}(x) \|_{\tilde{N}} \le \sqrt{\frac{2}{(n-1)\pi}} \left\| \frac{d^n f(x)}{dx^n} \right\|_{\tilde{N}} \left(\frac{h}{2} \right)^n, \ \ h = \max_i | x_i - x_{i-1} |. \qquad (8.5)$$

Из (8.5) следует, что при $h \to 0$ порядок погрешности p при интерполяции алгебраическим многочленом равен количеству выбранных узлов $p = n$. Величина ε

может быть сделана малой как за счет увеличения *n*, так и уменьшения *h*. В практических расчетах используют, как правило, многочлены невысокого порядка ($n \le 6$), в связи с тем, что с ростом *n* резко возрастает погрешность вычисления самого многочлена из-за погрешностей округления.

8.3. Виды многочленов и способы интерполяции

Один и тот же многочлен можно записать по-разному, например $P_1(x) = 1 - 2x + x^2 = (x-1)^2$. Поэтому в зависимости от решаемых задач применяют различные виды представления интерполяционного многочлена и соответственно способы интерполяции.

Наряду с общим представлением (8.3) наиболее часто в приложениях используют интерполяционные многочлены в форме Лагранжа и Ньютона. Их особенность в том, что не надо находить параметры $\vec{c}$, т. к. многочлены в этой форме прямо записаны через значения таблицы $\{(x_i, y_i)\, i = 1, ..., n\}$.

Интерполяционный многочлен Ньютона (PN)

$$N_{n-1}(x_T) = y_1 + \sum_{k=1}^{n-1} (x_T - x_1)(x_T - x_2)...(x_T - x_k)\Delta_1^k . \qquad (8.6)$$

Здесь x_T – текущая точка, в которой надо вычислить значение многочлена; Δ_1^k – разделенные разности порядка *k*, которые вычисляются по следующим рекуррентным формулам:

$$\Delta_i^1 = \frac{y_i - y_{i+1}}{x_i - x_{i+1}},\ i = 1, ..., n;$$

$$\Delta_i^2 = \frac{\Delta_i^1 - \Delta_{i+1}^1}{x_i - x_{i+2}},\ i = 1, ..., n-2;$$

$$\Delta_i^k = \frac{\Delta_i^{k-1} - \Delta_{i+1}^{k-1}}{x_i - x_{i+k}},\ i = 1, ..., n-k.$$

Схема расчета многочлена Ньютона представлена на рис. 8.2.

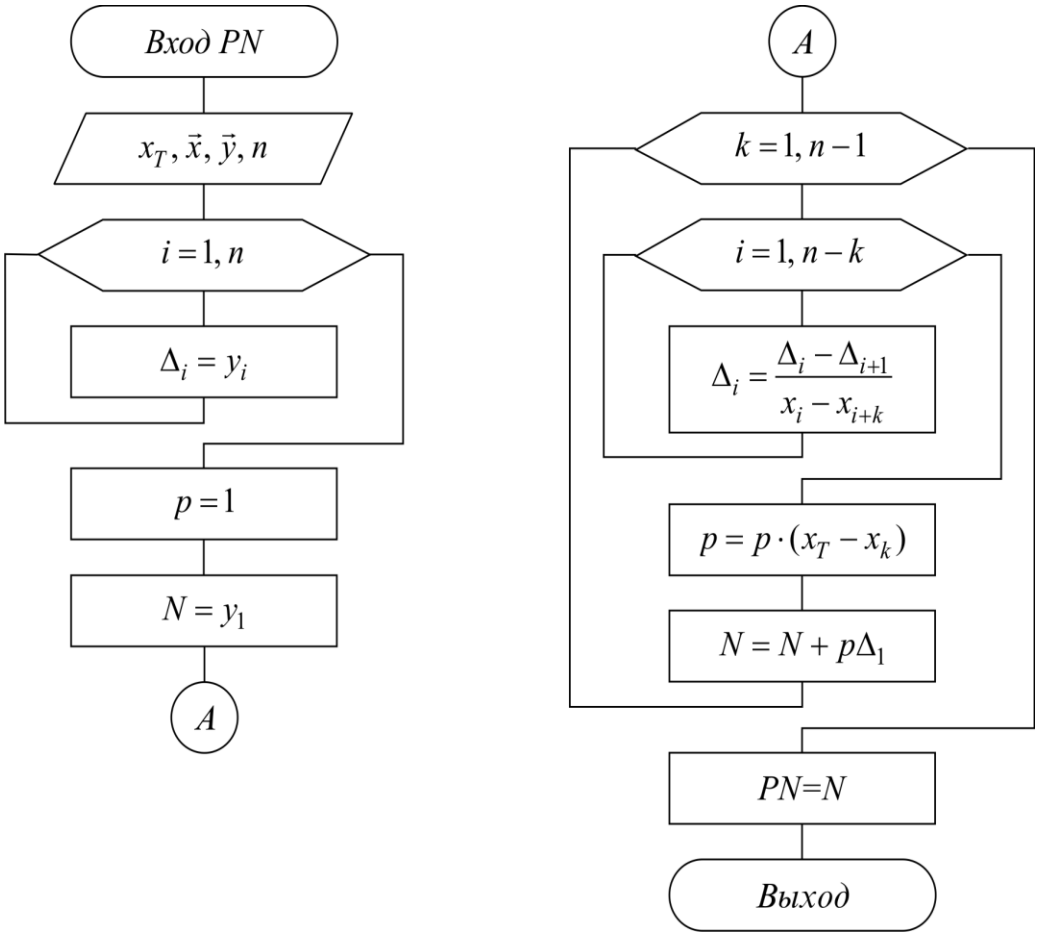


Рис. 8.2

Линейная (PNL) и квадратичная (PNS) интерполяция

Вычисления по интерполяционной формуле (8.6) для $n > 3$ используют редко. Обычно при интерполяции по заданной таблице из $m > 3$ точек применяют квадратичную $n = 3$ или линейную $n = 2$ интерполяцию. В этом случае для приближенного вычисления значения функции f в точке x находят в таблице ближайший к этой точке i-узел из общей таблицы, строят интерполяционный многочлен Ньютона первой или второй степени по формулам

$$N_1(x_T) = y_{i-1} + (x_T - x_{i-1})\frac{y_i - y_{i-1}}{x_i - x_{i-1}}, \quad x_{i-1} \le x_T \le x_i; \tag{8.7}$$

$$N_2(x_T) = N_1(x_T) + (x_T - x_{i-1})(x_T - x_i)\frac{\left(\frac{y_{i-1} - y_i}{x_{i-1} - x_i}\right) - \left(\frac{y_i - y_{i+1}}{x_i - x_{i+1}}\right)}{x_{i-1} - x_{i+1}}; \quad x_{i-1} \le x_T \le x_{i+1}$$

и за значение *f*(*x*) принимают $N_1(x)$ (*линейная интерполяция*) или $N_2(x)$ (*квадратичная интерполяция*). Схема расчета для линейной и квадратичной интерполяций приведена на рис. 8.3.

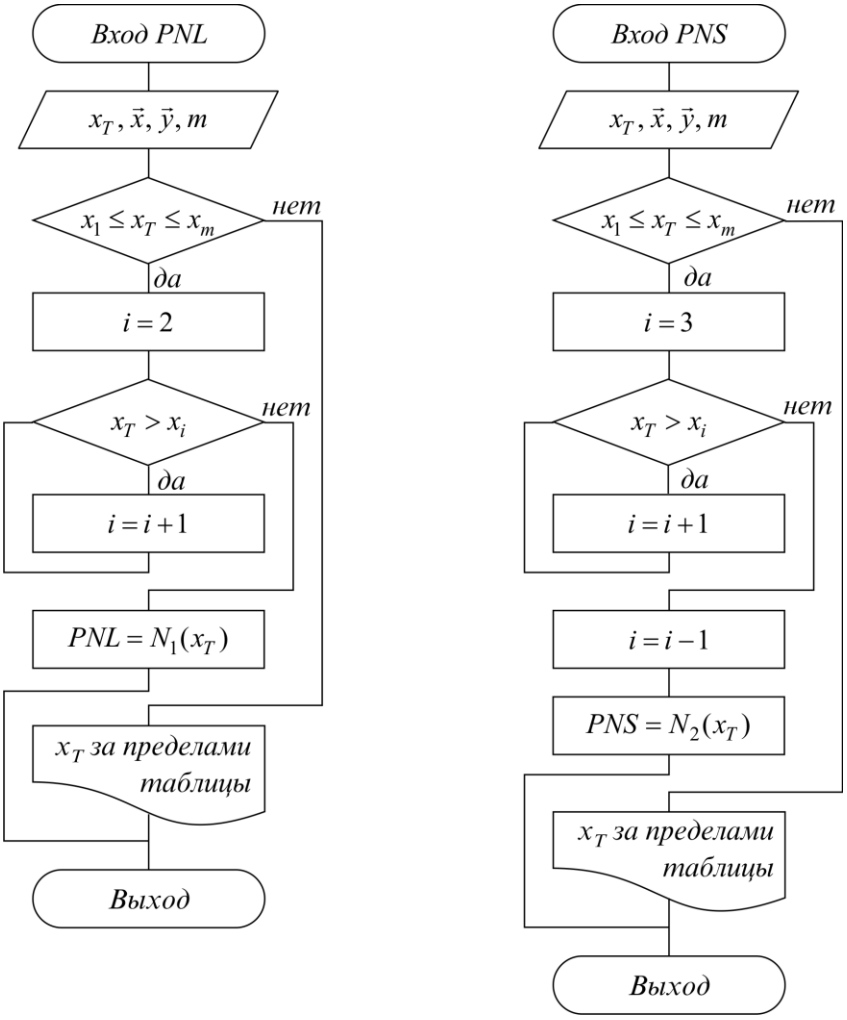


Рис. 8.3

Интерполяционный многочлен Лагранжа (PL)

$$L_{n-1}(x_T) = \sum_{k=1}^{n} y_k e_k(x_T); \quad e_k(x_T) = \prod_{\substack{i=1 \\ i \neq k}}^{n} \frac{x_T - x_i}{(x_k - x_{i)}} \; . \tag{8.8}$$

Многочлены $e_k^{n-1}(x_j)$ выбраны так, что во всех узлах, кроме *k*-го, они обращаются в нуль, в *k*-м узле они равны единице:

$$e_k^{n-1}(x_j) = \begin{cases} 1, \text{при } j = k; \\ 0, \text{при } j \neq k. \end{cases}$$

Поэтому из выражения (8.8) видно, что $L_{n-1}(x_i) = y_i$.

Схема расчета интерполяционного многочлена Лагранжа представлена на рис. 8.4.

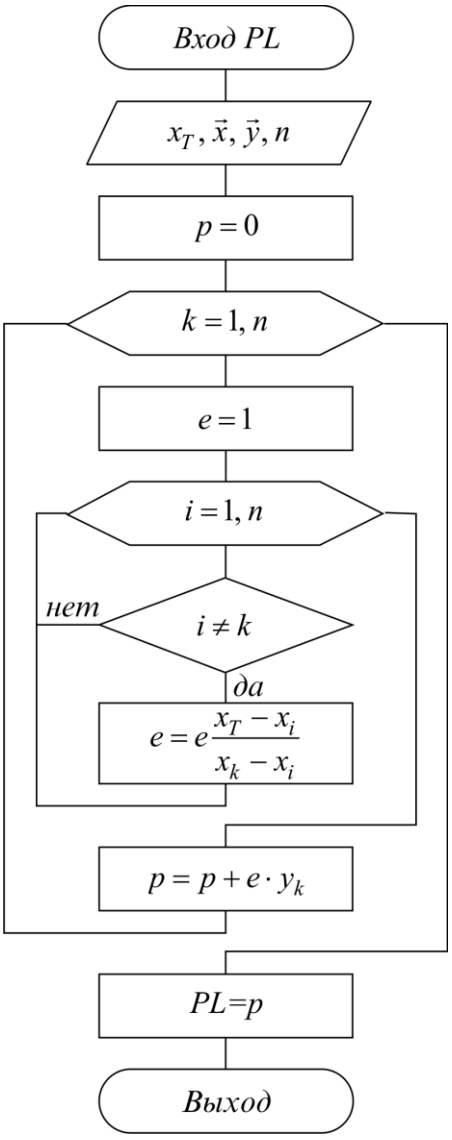


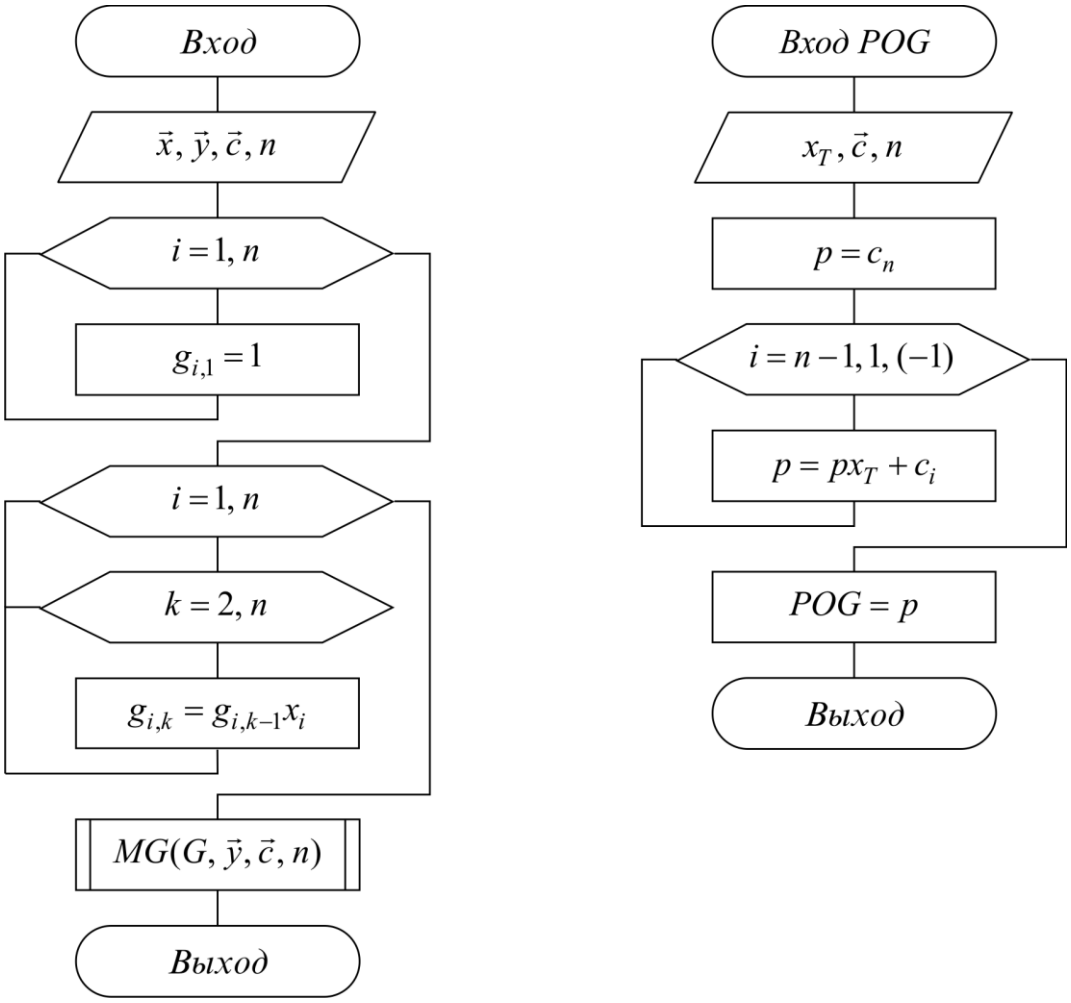
Рис. 8.4

Интерполяция общего вида, использующая прямое решение системы (8.2) методом Гаусса (POG)

Следует отметить, что ввиду громоздкости многочлены Ньютона и Лагранжа уступают по эффективности расчета многочлену общего вида (8.3), если предварительно найдены коэффициенты $\vec{\tilde{n}}$.

Поэтому, когда требуется производить много вычислений многочлена, построенного по одной таблице, оказывается выгодно вначале один раз найти коэффициенты $\vec{\tilde{n}}$ и затем использовать формулу (8.3). Коэффициенты $\vec{\tilde{n}}$ находят прямым решением системы (8.2) с матрицей (8.4), затем вычисляют его значения по экономно программируемой формуле (алгоритм Горнера)

$$P_{n-1}(x) = c_1 + x_T(c_2 + \ldots\ x_T(c_{n-2} + x_T(c_{n-1} + x_T c_n)\ \ldots)). \qquad (8.9)$$



а б

Рис. 8.5

Схема расчета интерполяционного многочлена общего вида по формуле (8.9) с прямым решением системы (8.2) приведена на рис. 8.5.

Интерполяция общего вида, использующая расчет коэффициентов многочлена (8.3) через многочлен Лагранжа (POL)

Находить коэффициенты $\vec{\tilde{n}}$ многочлена (8.3) можно, не решая прямо систему (8.2), а используя разложение коэффициентов Лагранжа (8.8):

$$e_k^{n-1}(x) = a_{k,1}^{n-1} + a_{k,2}^{n-1} \cdot x + \ \ldots \ + a_{k,n}^{n-1} \cdot x^{n-1}, \qquad c_i = \sum_{k=1}^{n} y_k \cdot a_{k,i}^{n-1}. \tag{8.10}$$

Рекуррентные формулы для нахождения коэффициентов $a_{k,j}^{n-1}$:

$$a_{k,1}^{m} = a_{k,1}^{m-1} \frac{-x_m}{x_k - x_m}; \quad a_{k,m+1}^{m} = a_{k,m}^{m-1} \frac{1}{x_k - x_m}; \tag{8.11}$$

$$a_{k,j}^{m} = \frac{a_{k,j-1}^{m-1} - a_{k,j}^{m-1} \cdot x_m}{x_k - x_m}; \quad 1 < j < m; \qquad m = 2, \ \ldots, \ n-1$$

получаются из вида многочленов $e_k^{n-1}(x)$, если использовать очевидное представление

$$e_k^m(x) = e_k^{m-1} \cdot \frac{x - x_{n-m}}{x_k - x_{n+m}}; \quad e_k^0 = a_{k,1}^0 = 1; \quad m = 1,\ 2,\ ...,\ n; \quad m \neq k.$$

Схема алгоритма вычисления коэффициентов многочлена общего вида по формулам (8.10), (8.11) представлена на рис. 8.6.

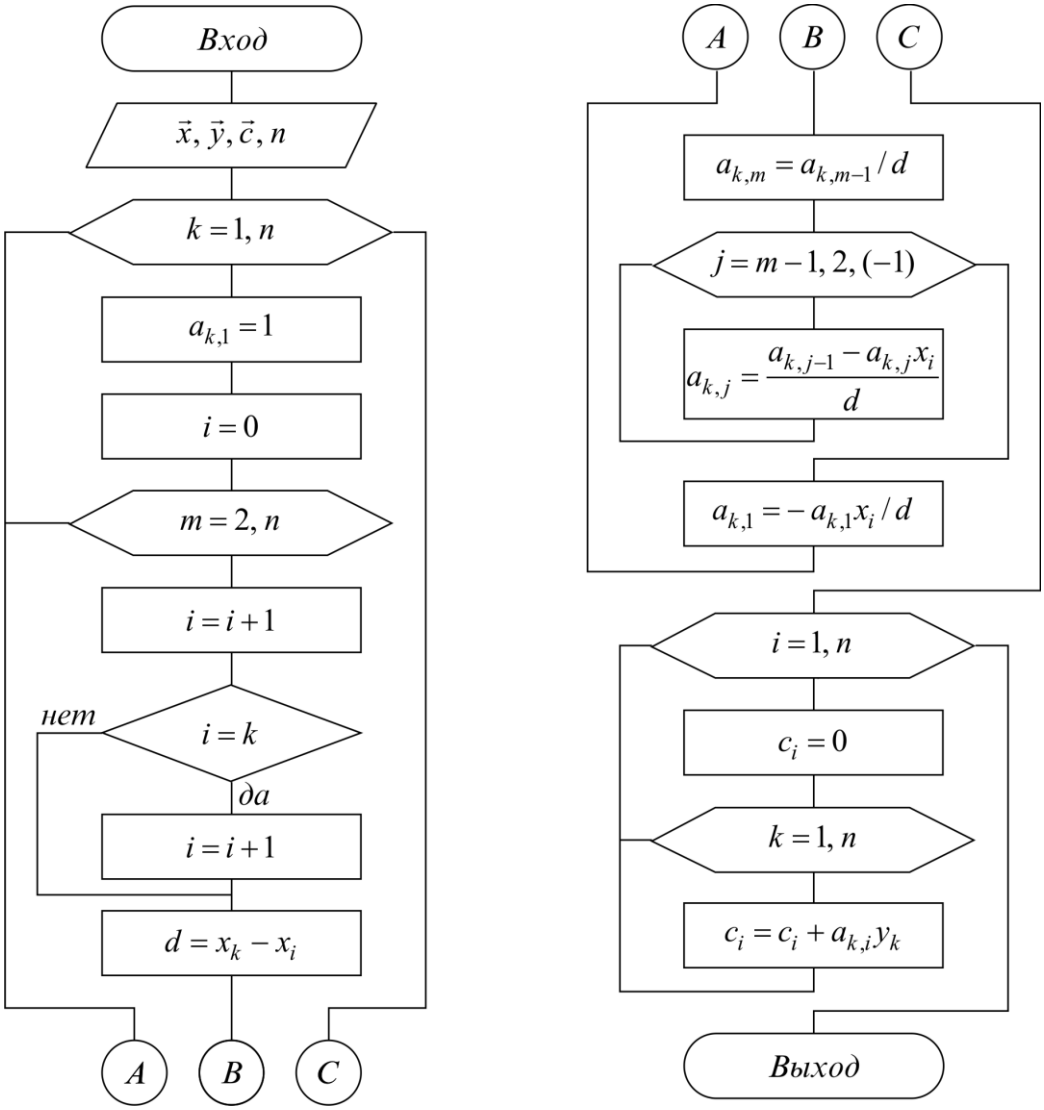


Рис. 8.6

8.4. Понятие среднеквадратичной аппроксимации

Суть среднеквадратичной аппроксимации заключается в том, что параметры $\vec{n}$ функции $\varphi(x, \vec{c})$ подбираются такими, чтобы обеспечить минимум

квадрата расстояния между функциями $f(x)$ и $\varphi(x,\vec{c})$ в пространстве $L_2[a,b]$, т. е. из условия

$$\min_{c_1,\ldots,c_n} \left\| f(x) - \varphi(x,\vec{c}) \right\|_{L_2}. \qquad (8.12)$$

В случае линейной аппроксимации (8.1) задача (8.12) сводится к решению СЛАУ для нахождения необходимых коэффициентов $\vec{\tilde{n}}$:

$$\sum_{k=1}^{n} (\varphi_i \varphi_k)_{L_2} \cdot c_k = (f, \varphi_i)_{L_2}; i = 1, \ldots, n. \qquad (8.13)$$

Здесь $(\varphi_i \varphi_k)_{L_2}$, $(f, \varphi_i)_{L_2}$ – скалярные произведения в L_2.

Матрица системы (8.13) симметричная, и ее следует решать методом квадратного корня. Особенно просто эта задача решается, если выбрана ***ортогональная система функций*** $\{\varphi_k(x)\}$, т. е. такая, что

$$(\varphi_i \ \ \varphi_k) = \begin{cases} 0, & i \neq k \\ \|\varphi_k\|^2, & i = k. \end{cases}$$

Тогда матрица СЛАУ (8.13) диагональная и параметры $\vec{c}$ находятся по формуле

$$c_k = \frac{(f, \varphi_k)}{\|\varphi_k\|^2}.$$

В этом случае представление (8.1) называется *обобщенным рядом Фурье*, а c_k называются *коэффициентами Фурье*.

Метод наименьших квадратов (МНК)

МНК является частным случаем среднеквадратичной аппроксимации. При использовании МНК в области значений x, представляющей некоторый интервал $[a, b]$, где функции f и φ должны быть близки, выбирают систему различных точек (узлов) $x_1, \ldots, x_m$, число которых обычно больше, чем количество искомых параметров $c_1, \ldots, c_n$, $m \geq n$. Далее, потребовав, чтобы сумма квадратов невязок во всех узлах была минимальна (рис. 8.7):

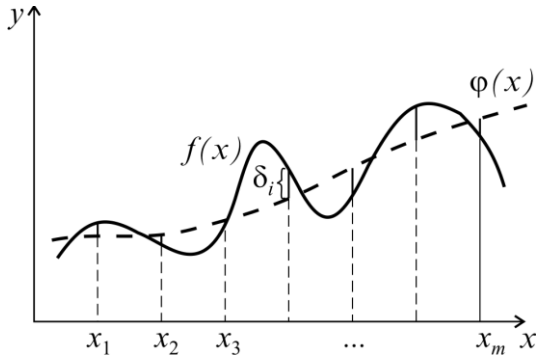


Рис. 8.7

$$\min_{\vec{c}} \sum_{i=1}^{m} [y_i - \varphi(x_i\ \vec{c})]^2 = \min_{\vec{c}} \sum_{i=1}^{m} \delta_i^2 = \min_{\vec{c}} \delta(\vec{c}),$$

находим из этого условия параметры $c_1, \ldots, c_n$.

В общем случае эта задача сложная и требует применения численных методов оптимизации. Однако в случае линейной аппроксимации (8.1), составляя условия минимума суммы квадратов невязок во всех точках $\delta(\vec{c})$ (в точке минимума все частные производные должны быть равны нулю):

$$\frac{\partial\delta(c_1, c_2, ..., c_n)}{\partial c_i} = 0, i = 1, ..., n \qquad (8.14)$$

получаем систему n линейных уравнений относительно n неизвестных $c_1, ..., c_n$ следующего вида:

$$\sum_{k=1}^{n}(\vec{\varphi}_i\ \vec{\varphi}_k)c_k = (\vec{y},\ \vec{\varphi}_i),\ \ i = 1,\ n\ \ \text{или}\ \ G\tilde{n} = \vec{b}\,. \qquad (8.15)$$

Здесь $\vec{\varphi}_i = \left(\varphi_i(x_1),\ \varphi_i(x_2),\ ...\ ,\ \varphi_i(x_m)\right),\ \vec{y} = \left(y_1,\ ...\ ,\ y_m\right)$ – векторы-таблицы функций. Элементы матрицы G и вектора $\vec{b}$ в (8.15) определяются выражениями

$$\left.\begin{aligned} g_{i,k} &= (\vec{\varphi}_i\ \ \vec{\varphi}_k) = \sum_{j=1}^{m}\varphi_i(x_j)\varphi_k(x_j); \\ b_i &= (\vec{y},\ \vec{\varphi}_i) = \sum_{j=1}^{m} y_j\varphi_i(x_j). \end{aligned}\right\}$$ – скалярные произведения векторов.

Система (8.15) имеет симметричную матрицу G и решается методом квадратного корня.

При аппроксимации по МНК обычно применяют такие функции $\{\varphi_i(x)\}$, которые используют особенности решаемой задачи и удобны для последующей обработки.

Схема расчета коэффициентов многочлена вида (8.3) по методу наименьших квадратов представлена на рис. 8.8.

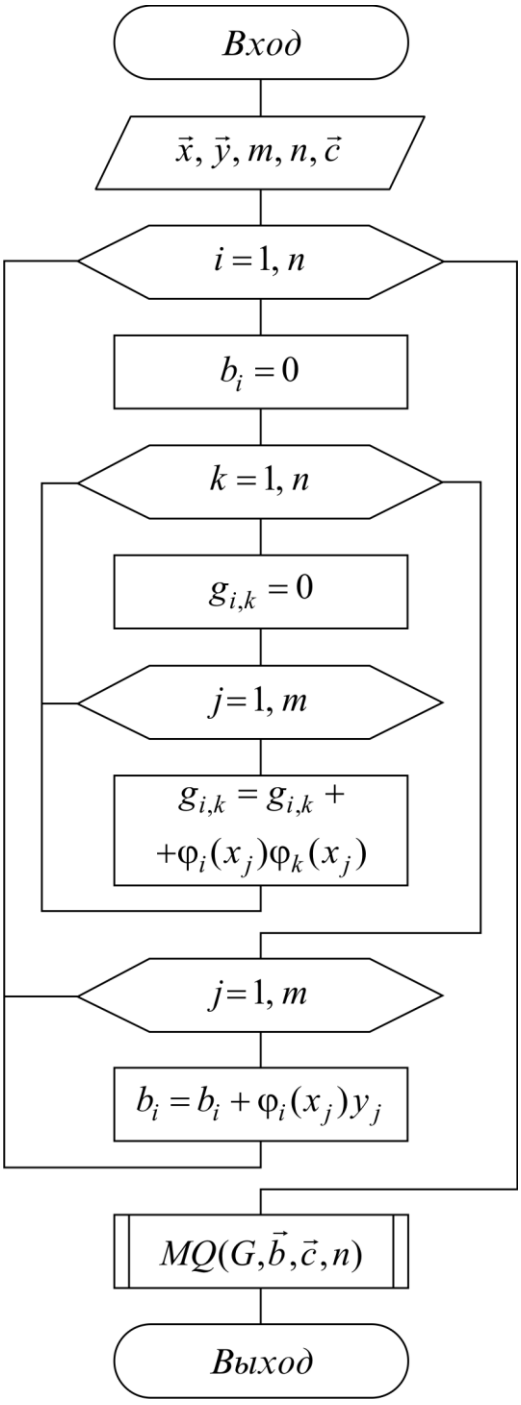


Рис. 8.8

Приведем пример аппроксимации по МНК. Предположим, что известна таблица значений $f(x)$: $\{x_1 = 1, y_1 = 0.5, x_2 = 2, y_2 = 1.2, x_3 = 3, y_3 = 0.8\}$, т. е. $m = 3$. Требуется найти параметры аппроксимирующей функции $\varphi(x, c_1, c_2)$ вида $\varphi = c_1 + c_2 \cdot x$ ($n = 2$).

Составляем сумму квадратов невязок:

$$\delta(\tilde{n}) = \sum_{i=0}^{3} (y_i - \tilde{n}_1 - \tilde{n}_2 \cdot x_i)^2 =$$

$$= (0.5 - \tilde{n}_1 - \tilde{n}_2 \cdot 1)^2 + (1.2 - \tilde{n}_1 - \tilde{n}_2 \cdot 2)^2 + (0.8 - \tilde{n}_1 - \tilde{n}_2 \cdot 3)^2.$$

Условия минимума (8.14):

$$\begin{cases} \dfrac{\partial \delta}{\partial \tilde{n}_1} = -2 \cdot (0.5 - \tilde{n}_1 - 1 \cdot \tilde{n}_2) - 2 \cdot (1.2 - \tilde{n}_1 - 2 \cdot \tilde{n}_2) - 2 \cdot (0.8 - \tilde{n}_1 - 3 \cdot \tilde{n}_2) = 0; \\ \dfrac{\partial \delta}{\partial \tilde{n}_2} = -2 \cdot (0.5 - \tilde{n}_1 - 1 \cdot \tilde{n}_2) - 4 \cdot (1.2 - \tilde{n}_1 - 2 \cdot \tilde{n}_2) - 6 \cdot (0.8 - \tilde{n}_1 - 3 \cdot \tilde{n}_2) = 0. \end{cases}$$

Приводя подобные члены, получим окончательно систему двух уравнений с симметричной матрицей относительно неизвестных c_1 и c_2:

$$\begin{cases} 3\tilde{n}_1 + 6\tilde{n}_2 = 2.5; \\ 6\tilde{n}_1 + 14\tilde{n}_2 = 5.3. \end{cases}$$

Решая ее, находим $\tilde{n}_1 \approx 0.53$, $\tilde{n}_2 = 0.15$.

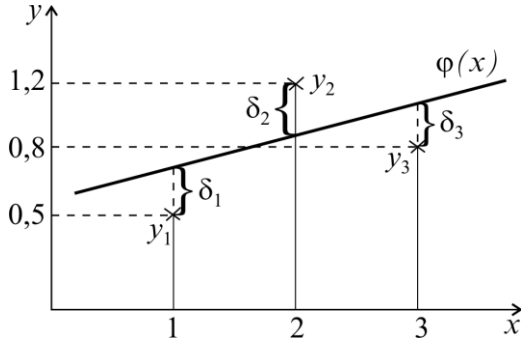


Рис. 8.9

На рис. 8.9 приведена таблица функции *f(x)* и полученная по МНК функция φ(*x*).

Порядок расчета по МНК следующий. Вначале по исходной таблице формируется матрица *G* и рассчитываются коэффициенты (см. рис. 8.8) (в качестве $\varphi_k(x)$ здесь берется функция x^{k-1}). Затем, используя полученные коэффициенты, рассчитывается значение функции в искомой точке (см. рис. 8.5, б).

8.5. Индивидуальные задания

Во всех вариантах (табл. 8.1.) требуется аппроксимировать заданную исходную функцию *f*(*x*) многочленом на интервале [*a, b*]. Задано количество неизвестных параметров *n*, вид аппроксимации и *m* – количество точек, в которых задана функция. Таблица исходной функции $y_i = f(x_i)$ вычисляется в точках $x_i = a + (i-1)(b-a)/(m-1)$, $i = 1, m$. Используя полученную таблицу (x_i, y_i), требуется вычислить значения функций $f(x_j)$, $\varphi(x_j, \vec{c})$ и погрешность $d(x_j) = f(x_j) - \varphi(x_j, \vec{c})$ в точках $x_j = a + (j-1)(b-a)/20$; $j = 1, 21$, построить графики и проанализировать качество полученной аппроксимации.

Таблица 8.1

N вар.	*Функция f(x)*	*a*	*b*	*m*	*n*	*Вид аппроксимации*
1	$4x-7\sin(x)$	–2	3	11	3	МНК
2	$x^2-10\sin^2(x)$	0	3	4	4	Ньютона PN
3	$\ln(x)-5\cos(x)$	1	8	4	4	Лагранжа PL
4	$e^x/x^3-\sin^3(x)$	4	7	4	4	Общего вида POG
5	$\sqrt{x}-\cos^2(x)$	5	8	4	4	Общего вида POL
6	$\ln(x)-5\sin^2(x)$	3	6	11	2	Линейная PNL
7	$x-5\sin^2(x)$	1	4	11	4	МНК
8	$\sin^2(x)-x/5$	0	4	11	3	Квадратичная PNS
9	x^3+10x^2	–8	2	5	5	Ньютона PN
10	x^3-5x^2	–2	5	5	5	Лагранжа PL
11	$x^3+6x^2-0.02e^x$	–5	3	5	5	Общего вида POG
12	$x^2+5\cos(x)$	–1	4	5	5	Общего вида POL
13	$\sin^2(x)-3\cos(x)$	1	7	11	5	МНК
14	$x^3-50\cos(x)$	–2	5	11	3	Квадратичная PNS
15	$0.1x^3+x^2-10\sin(x)$	–4	2	11	2	Линейная PNL

8.6. Контрольные вопросы

1. Что такое аппроксимация?
2. Что такое интерполяция?
3. Что такое экстраполяция?

Лабораторная работа № 9. Методы решения нелинейных уравнений

Цель работы: изучить численные алгоритмы нахождения корней нелинейных уравнений.

9.1. Решение нелинейных уравнений

Математической моделью многих физических процессов является функциональная зависимость $y = f(x)$. Поэтому задачи исследования различных свойств функции $f(x)$ часто возникают в инженерных расчетах. Одной из таких задач является нахождение значений x, при которых функция $f(x)$ обращается в нуль, т. е. решение уравнения

$$f(x) = 0. \qquad (9.1)$$

Точное решение удается получить в исключительных случаях, и обычно для нахождения корней уравнения применяются численные методы. Решение уравнения (9.1) при этом осуществляется в два этапа:

1. Приближенное определение местоположения, характер и выбор интересующего корня.

2. Вычисление выбранного корня с заданной точностью ε.

Первая задача решается графическим методом: на заданном отрезке $[a, b]$ вычисляется таблица значений функции с некоторым шагом h, строится ее график и определяются интервалы (α_i, β_i) длиной h, на которых находятся корни. На рис. 9.1 представлены три наиболее часто встречающиеся ситуации:

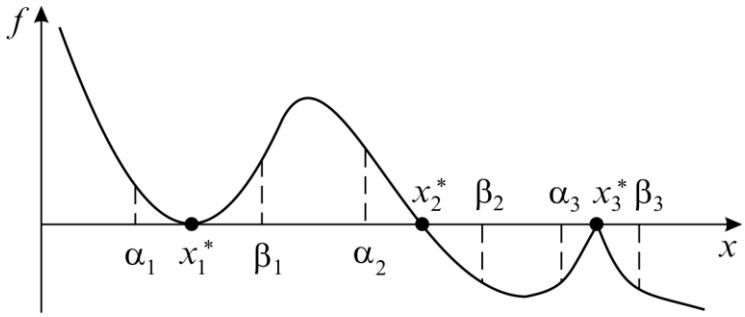


Рис. 9.1

а) кратный корень: $f'(x_1^*) = 0, \quad f(\alpha_1) \cdot f(\beta_1) > 0;$

б) простой корень: $f'(x_2^*) \neq 0, \quad f(\alpha_2) \cdot f(\beta_2) < 0;$

в) вырожденный корень: $f'(x_3^*)$ не существует, $f(\alpha_3) \cdot f(\beta_3) > 0$.

Как видно из рис. 9.1, в случаях «а» и «в» значение корня совпадает с точкой экстремума функции, и для нахождения таких корней можно использовать методы поиска минимума функции.

На втором этапе вычисление значения корня с заданной точностью осуществляется одним из итерационных методов. При этом в соответствии с общей методологией *m-шагового итерационного метода* (см. подразд. 8.5) на интервале (α, β), где находится интересующий нас корень x^*, выбирается m начальных

значений $x_0, x_1, \ldots, x_{m-1}$ (обычно $x_0 = \alpha$, $x_1 = \beta$), после чего последовательно находятся члены ($x_m, x_{m+1}, \ldots, x_{n-1}, x_n$) рекуррентной последовательности *порядка m* по правилу $x_k = \varphi(x_{k-1}, \ldots, x_{k-m})$ до тех пор, пока $|x_n - x_{n-1}| < \varepsilon$. Последнее x_n выбирается в качестве приближенного значения корня ($x^* \approx x_n$).

Многообразие методов определяется возможностью большого выбора значений φ. Наиболее часто используемые на практике методы описаны далее.

9.2. Итерационные методы уточнения корней

Метод простой итерации (MI)

Очень часто в практике вычислений встречается ситуация, когда уравнение (9.1) записано в виде, разрешенном относительно *x*:

$$x = \varphi(x). \tag{9.2}$$

Заметим, что переход от записи уравнения (9.1) к эквивалентной записи (9.2) можно сделать многими способами, например, положив

$$\varphi(x) = x + \psi(x) f(x), \tag{9.3}$$

где $\psi(x)$ – произвольная, непрерывная, знакопостоянная функция (часто достаточно выбрать ψ=const).

В этом случае корни уравнения (9.2) являются также корнями (9.1) и наоборот.

Исходя из записи (9.2) члены рекуррентной последовательности в методе простой итерации вычисляются по закону

$$x_k = \varphi(x_{k-1}), k = 1, 2, \ldots . \tag{9.4}$$

Метод является одношаговым, т. к. последовательность имеет первый порядок (m = 1) и для начала вычислений достаточно знать одно начальное приближение $x_0 = \alpha$, $x_0 = \beta$, или $x_0 = (\alpha + \beta)/2$.

Геометрическая иллюстрация сходимости и расходимости метода простой итерации представлена на рис. 9.2, из которого видно, что метод не всегда сходится к точному решению.

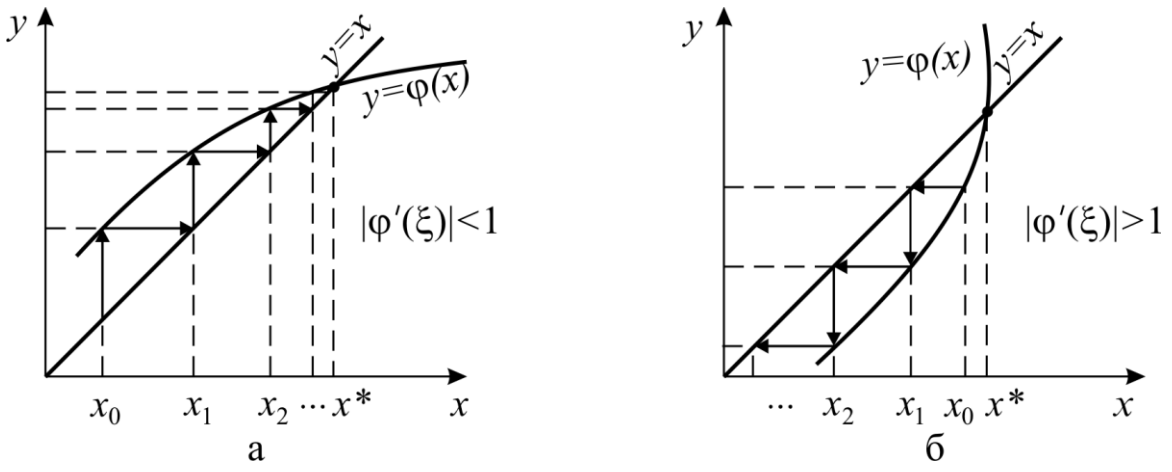


Рис. 9.2

Условием сходимости метода простой итерации, если $\varphi(x)$ дифференцируема, как показывает анализ графиков, является выполнение неравенства

$$|\varphi'(\xi)| < 1, \text{ для любого } \xi = (\alpha, \beta), \quad x^* \in (\alpha, \beta). \tag{9.5}$$

Максимальный интервал **(α, β),** для которого выполняется неравенство (9.5), называется ***областью сходимости***. При выполнении условия (9.5) метод сходится, если начальное приближение x_0 выбрано из области сходимости. При этом ***скорость сходимости погрешности*** $\varepsilon_k = |x^* - x_k|$ к нулю вблизи корня приблизительно такая же, как у геометрической прогрессии $\varepsilon_k \approx \varepsilon_{k-1} q$ со знаменателем $q \cong |\varphi'(x^*)|$, т. е. чем меньше q, тем быстрее сходимость, и наоборот. Поэтому при переходе от (9.1) к (9.2) функцию $\psi(x)$ в (9.3) выбирают так, чтобы выполнялось условие сходимости (9.5) для как можно большей области (α, β) и с наименьшим q. Удачный выбор этих условий гарантирует эффективность расчетов. Часто положительные результаты дает применение ***релаксации*** (см. подразд. 7.7). Схема алгоритма метода простой итерации представлена на рис. 9.3.

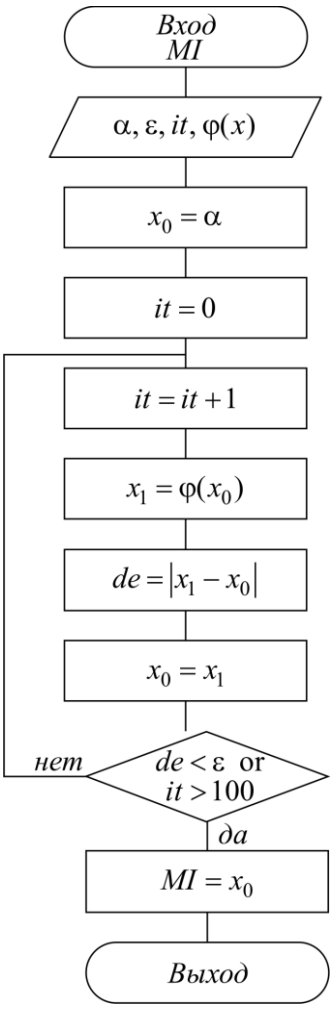


Рис. 9.3

Метод Ньютона (MN)

Этот метод является модификацией метода простой итерации и часто называется методом касательных. Если $f(x)$ имеет непрерывную производную, тогда, выбрав в (9.3) $\psi(x) = -1/f'(x)$, получаем эквивалентное уравнение $x = x - f(x)/f'(x) = \varphi(x)$, в котором $q \cong \varphi'(x^*) \equiv 0$. Поэтому скорость сходимости рекуррентной последовательности метода Ньютона

$$x_k = x_{k-1} - \frac{f(x_{k-1})}{f'(x_{k-1})} = \varphi(x_{k-1}) \qquad (9.6)$$

вблизи корня очень большая, погрешность очередного приближения примерно равна квадрату погрешности предыдущего $\varepsilon_k \cong |\varphi''(x^*)|\varepsilon_{k-1}^2$.

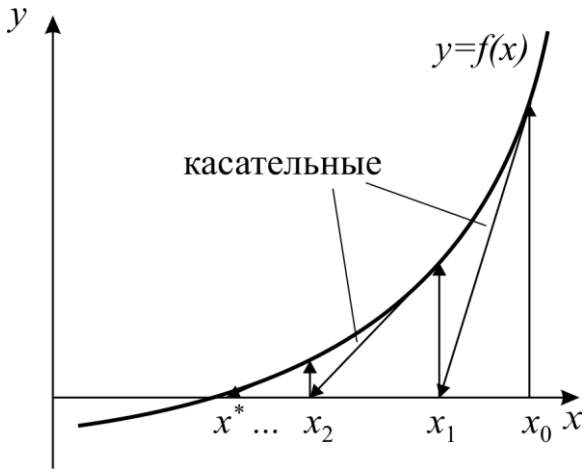


Рис. 9.4

Из (9.6) видно, что этот метод одношаговый (m = 1), и для начала вычислений требуется задать одно начальное приближение x_0 из ***области сходимости***, определяемой неравенством $|f \cdot f''|/(f')^2 < 1$. Метод Ньютона получил также второе название ***метод касательных*** благодаря геометрической иллюстрации его сходимости, представленной на рис. 9.4. Этот метод позволяет находить как простые, так и кратные корни. Основной его недостаток – малая область сходимости и необходимость вычисления производной $f'(x)$.

Метод секущих (MS)

Данный метод является модификацией метода Ньютона, позволяющей избавиться от явного вычисления производной путем ее замены приближенной формулой (9.2). Это эквивалентно тому, что вместо касательной на рис. 9.4 проводится секущая. Тогда вместо процесса (9.6) получаем

$$x_k = x_{k-1} - \frac{f(x_{k-1})h}{f(x_{k-1}) - f(x_{k-1} - h)} = \varphi(x_{k-1}). \qquad (9.7)$$

Здесь h – некоторый малый параметр метода, который подбирается из условия наиболее точного вычисления производной (см. лабораторную работу № 10).

Одношаговый метод (m = 1) и его условие сходимости при правильном выборе h такое же, как у метода Ньютона.

Метод Вегстейна (MV)

Этот метод является модификацией предыдущего метода секущих. В нем предлагается при расчете приближенного значения производной по разностной формуле использовать вместо точки $x_{k-1}-h$ в (9.7) точку x_{k-2}, полученную на предыдущей итерации (рис. 9.5). Расчетная формула метода Вегстейна:

$$x_k = x_{k-1} - \frac{f(x_{k-1})(x_{k-1}-x_{k-2})}{f(x_{k-1})-f(x_{k-2})} = \\ = \varphi(x_{k-1}, x_{k-2}). \tag{9.8}$$

Метод является двухшаговым (m = 2), и для начала вычислений требуется задать два начальных приближения x_0, x_1. Лучше всего $x_0=\alpha, x_1=\beta$ (см. рис. 9.1). Метод Вегстейна сходится медленнее метода секущих, однако требует в два раза меньшего числа вычислений $f(x)$ и за счет этого оказывается более эффективным.

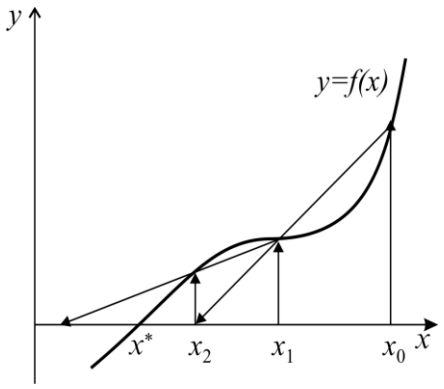


Рис. 9.5

Этот метод иногда называется улучшенным методом простой итерации и в применении к записи уравнения в форме (9.2) имеет вид

$$x_k = x_{k-1} - \frac{x_{k-1}-\varphi(x_{k-1})}{\left(1-\dfrac{\varphi(x_{k-1})-\varphi(x_{k-2})}{x_{k-1}-x_{k-2}}\right)}. \tag{9.9}$$

Схема алгоритма метода Вегстейна представлена на рис. 9.6.

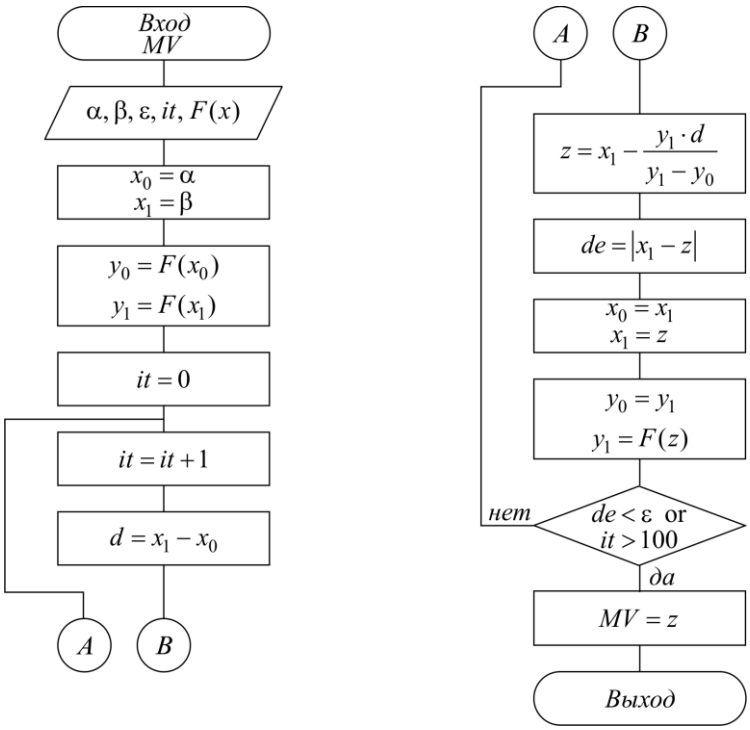


Рис. 9.6

Метод парабол (МР)

Предыдущие три метода (Ньютона, секущих, Вегстейна) фактически основаны на том, что исходная функция *f*(*x*) аппроксимируется линейной зависимостью вблизи корня и в качестве следующего приближения выбирается точка пересечения аппроксимирующей прямой с осью абсцисс. Ясно, что аппроксимация будет лучше, если вместо линейной зависимости использовать квадратичную. На этом и основан один из самых эффективных методов – метод парабол. Его суть: задаются три начальные точки $x_1,\ x_2,\ x_3$ (например, $x_1 = x_0 + h,$ $x_2 = x_0,\ \ x_3 = x_0 - h$), в этих точках рассчитываются три значения функции $y = f(x),\ \ y_1,\ y_2,\ y_3$ и строится интерполяционный многочлен второго порядка (рис. 9.7), который удобно записать в форме

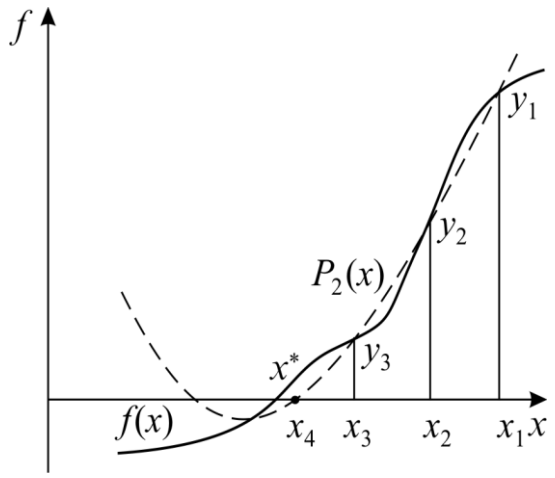


Рис. 9.7

$$P_2 = p(x - x_3)^2 + q(x - x_3) + r = pz^2 + qz + r\,. \qquad (9.10)$$

Коэффициенты этого многочлена вычисляются по формулам

$$z = x - x_3; \quad z_1 = x_1 - x_3; \quad z_2 = x_2 - x_3; \quad r = y_3;$$

$$p = \frac{(y_1 - y_3)z_2 - (y_2 - y_3)z_1}{z_1 z_2 (z_1 - z_2)};\ q = \frac{(y_1 - y_3)z_2^2 - (y_2 - y_3)z_1^2}{z_1 z_2 (z_2 - z_1)}\ . \qquad (9.11)$$

Полином (9.10) имеет два корня:

$$z_m^{1,\,2} = \frac{-q \pm \sqrt{q^2 - 4pr}}{2p},$$

из которых выбирается наименьший по модулю z_m и рассчитывается следующая точка $x_4 = x_3 + z_m,$ в результате получается рекуррентная формула метода парабол:

$$x_k = x_{k-1} + z_m(x_{k-1},\ x_{k-2},\ x_{k-3}) = \varphi(x_{k-1},\ x_{k-2},\ x_{k-3})\,. \qquad (9.12)$$

Метод парабол трехшаговый (m = 3). Скорость сходимости его больше, чем у метода Вегстейна, однако не лучше, чем у метода Ньютона вблизи корня. Схема алгоритма метода парабол представлена на рис. 9.8.

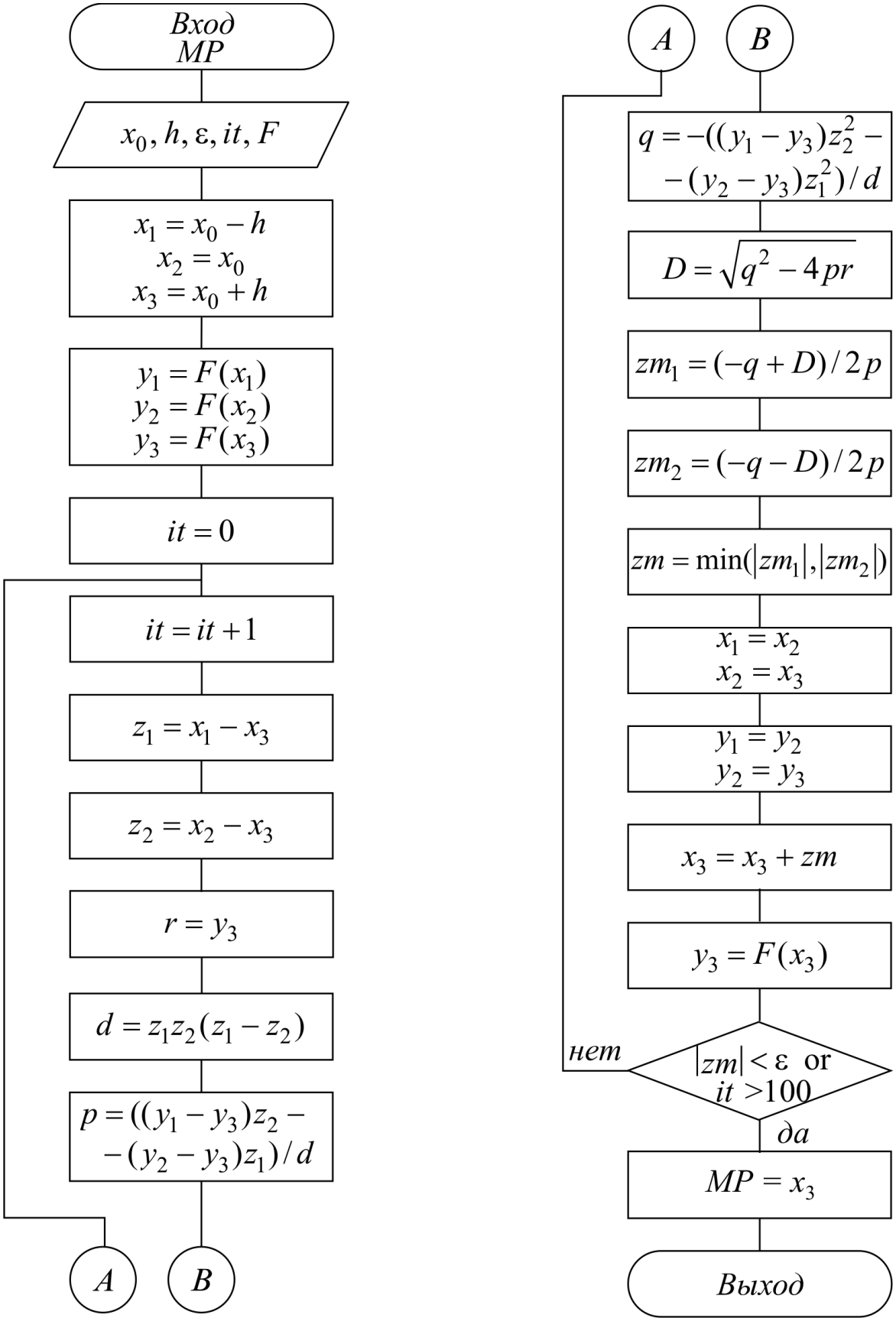

Вход
MP
$x_0, h, \varepsilon, it, F$
$x_1 = x_0 - h$
$x_2 = x_0$
$x_3 = x_0 + h$
$y_1 = F(x_1)$
$y_2 = F(x_2)$
$y_3 = F(x_3)$
$it = 0$
$it = it + 1$
$z_1 = x_1 - x_3$
$z_2 = x_2 - x_3$
$r = y_3$
$d = z_1 z_2 (z_1 - z_2)$
$p = ((y_1 - y_3)z_2 - (y_2 - y_3)z_1)/d$
A
B
A
B
$q = -((y_1 - y_3)z_2^2 - (y_2 - y_3)z_1^2)/d$
$D = \sqrt{q^2 - 4pr}$
$zm_1 = (-q + D)/2p$
$zm_2 = (-q - D)/2p$
$zm = \min(|zm_1|, |zm_2|)$
$x_1 = x_2$
$x_2 = x_3$
$y_1 = y_2$
$y_2 = y_3$
$x_3 = x_3 + zm$
$y_3 = F(x_3)$
нет
$|zm| < \varepsilon$ or $it > 100$
да
$MP = x_3$
Выход

Рис. 9.8

Метод деления отрезка пополам (MD)

Все вышеописанные методы могут работать, если функция *f*(*x*) является непрерывной и дифференцируемой вблизи искомого корня. В противном случае они не гарантируют получение решения.

Для разрывных функций, а также если не требуется быстрая сходимость, для нахождения *простого корня* на интервале (α, β) применяют надежный метод деления отрезка пополам. Его алгоритм основан на построении рекуррентной последовательности по следующему закону: в качестве начального приближения выбираются границы интервала, на котором точно имеется один простой корень $x_0 = \alpha$, $x_1 = \beta$, далее находится его середина $x_2 = \frac{x_0 + x_1}{2}$; очередная точка x_3 выбирается как середина того из смежных с x_2 интервалов $[x_0, x_2]$ или $[x_2, x_1]$, на котором находится корень. В результате получается следующий алгоритм метода деления отрезка пополам:

1. Вычисляем $y_0 = f(x_0)$, $y_1 = f(x_1)$.
2. Вычисляем $x_2 = (x_0 + x_1)/2$, $y_2 = f(x_2)$.
3. Если $y_0 \cdot y_2 > 0$, тогда $x_0 = x_2$, $y_0 = y_2$, иначе $x_1 = x_2$, $y_1 = y_2$.
4. Если $x_1 - x_0 > \varepsilon$, тогда повторять с п. 2.
5. Вычисляем $x^* = (x_0 + x_1)/2$.
6. Конец.

За одно вычисление функции погрешность уменьшается вдвое, т. е. скорость сходимости невелика, однако метод устойчив к ошибкам округления и всегда сходится.

9.3. Индивидуальные задания

По схеме, приведенной на рис. 9.9, отладить программу определения всех корней функции *f*(*x*) в указанном интервале [*a*, *b*], использовать метод в соответствии с полученным вариантом из табл. 9.1.

Таблица 9.1

Номер варианта	*f(x)*	*Интервал*		*Метод*
		a	*b*	
1	$4x - 7\sin(x)$	–2	2	MI
2	$x^2 - 10\sin^2(x) + 2$	–1	3	MN
3	$\ln(x) - 5\cos(x)$	1	8	MS
4	$e^x / x^3 - \sin^3(x) - 2$	4	7	MV
5	$\sqrt{x} - \cos^2(x) - 2$	4	8	MP
6	$\ln(x) - 5\sin^2(x)$	2	6	MD

Окончание таблицы 9.1

Номер варианта	*f(x)*	*Интервал*		*Метод*
		a	*b*	
7	$x-5\sin^2(x)-5$	3	9	MI
8	$\sin^2(x)-x/5-1$	–4	0	MN
9	x^3+10x^2-50	–12	5	MS
10	x^3-5x^2+12	–2	5	MV
11	$x^3+6x^2-0.02e^x-14$	–6	2	MP
12	$x^2+5\cos(x)-3$	–4	2	MD
13	$\sin^2(x)-3\cos(x)$	–7	3	MI
14	$x^3-50\cos(x)$	–4	3	MN
15	$0.1x^3+x^2-10\sin(x)-8$	–4	4	MS

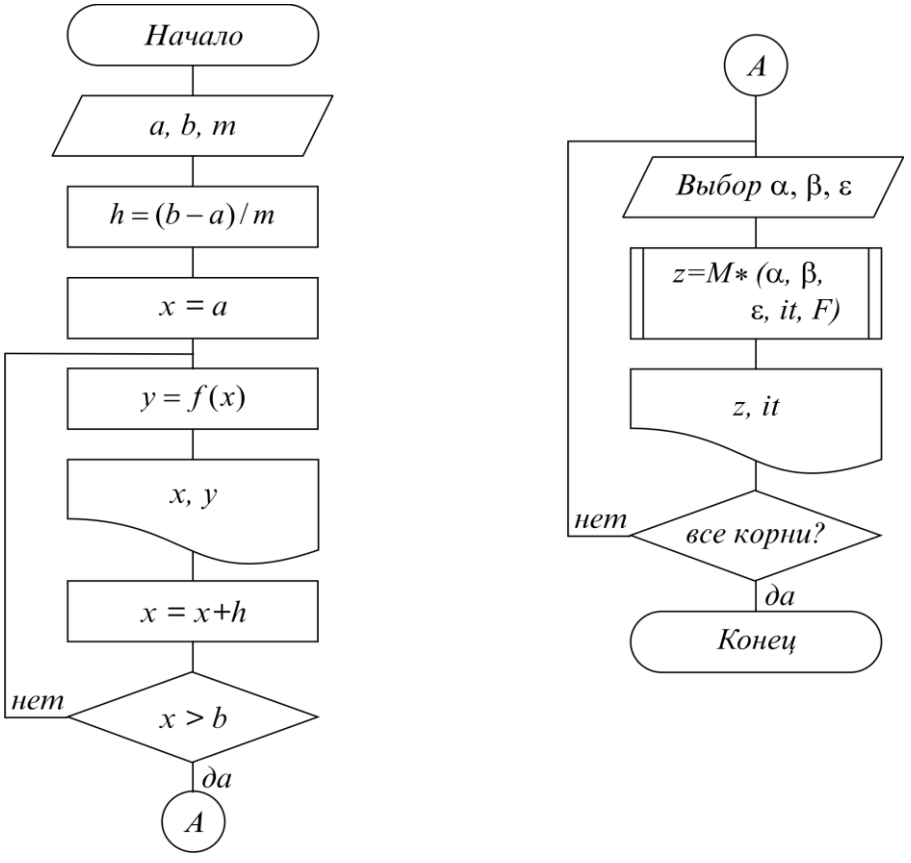


Рис. 9.9

Примечание. В табл. 9.1 все функции на указанном интервале имеют три корня.

Программа работает следующим образом: сначала на экран выдается таблица значений функции и делается запрос на ввод начального приближения (это может быть α, β или x_0) к тому корню, который надо получить с заданной точностью; после того как введены требуемые данные, идет обращение к подпрограмме и печать результатов.

Расчет функции, а также метод нахождения корня оформить в виде отдельных подпрограмм. Вариант метода и функции взять из таблицы. Выбрать точность $\varepsilon=10^{-4}$, а значение m по усмотрению.

После выполнения расчетов нарисовать график функции, а также график сходимости x_k к указанному корню x^*, для чего предусмотреть в процедуре нахождения корня возможность вывода значений x_k и $d_k = x_k - x_{k-1}$.

9.4. Контрольные вопросы

1. Что такое простой корень?
2. Что такое вырожденный корень?
3. Что объединяет методы простой итерации, Ньютона и Вегстейна?

Лабораторная работа № 10. Алгоритмы вычисления производных и интегралов

Цель работы: изучить численные алгоритмы нахождения значений производных и интегралов.

10.1. Краткие теоретические сведения

Формулы для вычисления интеграла $U = \int_a^b f(x)dx$ получают следующим образом. Область интегрирования $[a, b]$ разбивают на малые отрезки, тогда значение интеграла по всей области равно сумме интегралов на этих отрезках.

Выбирают на каждом отрезке $[x_i, x_{i+1}]$ 1–5 узлов и строят интерполяционный многочлен соответствующего порядка. Вычисляют интеграл от этого многочлена, и в результате получают формулу численного интегрирования через значения подынтегральной функции в выбранной системе точек. Такие выражения называют ***квадратурными формулами.***

Рассмотрим наиболее часто используемые квадратурные формулы для равных отрезков длиной $h = (b - a) / m$; $x_i = a + (i - 1)\cdot h$; $i = 1, 2, \ldots, m$; где m – количество разбиений отрезка интегрирования.

Формула средних

Формула средних получается, если на каждом i-м отрезке взять один центральный узел $x_{i+1/2} = (x_i + x_{i+1}) / 2$, соответствующий середине отрезка. Функция на каждом отрезке аппроксимируется многочленом нулевой степени (константой) $P_0(x) = y_{i+1/2} = f(x_{i+1/2})$. Заменяя площадь криволинейной фигуры площадью прямоугольника высотой $y_{i+1/2}$ и основанием h, получим формулу средних (рис. 10.1):

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} P_0(x)dx = h\sum_{i=1}^{m} y_{i+1/2} = \Phi_{CP}\,. \qquad (10.1)$$

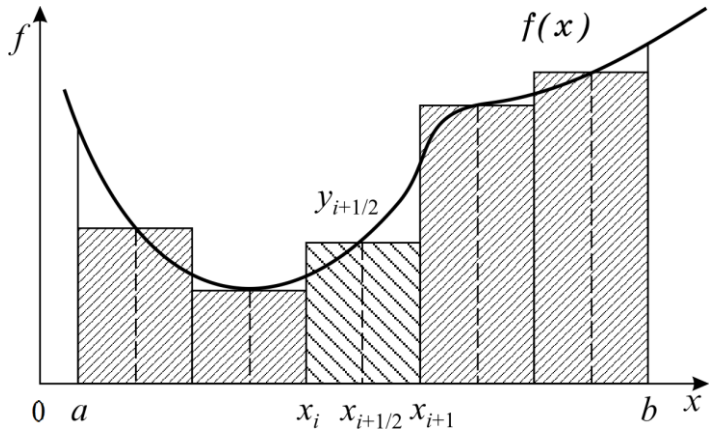


Рис. 10.1

Формула трапеций

Формула трапеций получается при аппроксимации функции $f(x)$ на каждом отрезке $[x_i, x_{i+1}]$ интерполяционным многочленом первого порядка, т. е. прямой, проходящей через точки (x_i, y_i), (x_{i+1}, y_{i+1}). Площадь криволинейной фигуры заменяется площадью трапеции с основаниями y_i, y_{i+1} и высотой h (рис. 10.2):

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} P_1(x)dx = h\sum_{i=1}^{m} \frac{y_i + y_{i+1}}{2} = h\left[\frac{y_1 + y_{m+1}}{2} + \sum_{i=2}^{m} y_i\right] = \Phi_{TP}\,. \quad (10.2)$$

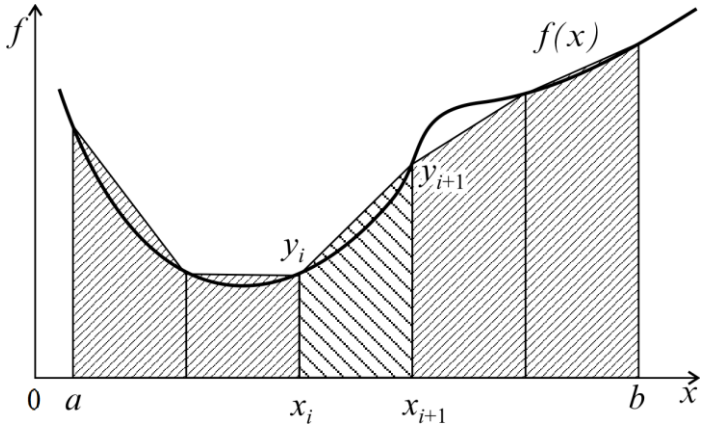


Рис. 10.2

Формула Симпсона

Формула Симпсона получается при аппроксимации функции $f(x)$ на каждом отрезке $[x_i, x_{i+1}]$ интерполяционным многочленом второго порядка (параболой) с узлами x_i, $x_{i+1/2}$, x_{i+1}. После интегрирования параболы получаем (рис. 10.3).

$$\int_a^b f(x)dx \approx \sum_{i=1}^{m} \int_{x_i}^{x_{i+1}} P_2(x)dx = \frac{h}{6}\sum_{i=1}^{m} (y_i + 4y_{i+0.5} + y_{i+1}) = \Phi_{CИМ}\,. \quad (10.3)$$

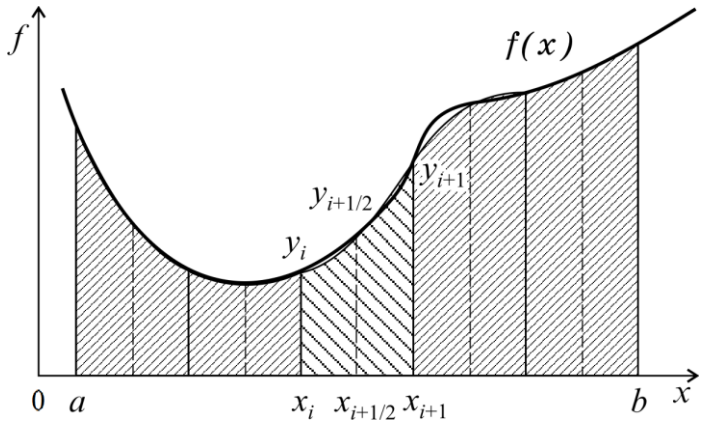


Рис. 10.3

После приведения подобных членов получаем более удобный для программирования вид:

$$\Phi_{\text{СИМ}} = \frac{h}{3} \cdot \left[\frac{y_1 + 4y_{1+0.5} + y_{m+1}}{2} + \sum_{i=2}^{m} \left(2y_{i+0.5} + y_i \right) \right].$$

Схема с автоматическим выбором шага по заданной точности

Метод 1. Одним из вариантов вычисления интеграла с заданной точностью является следующий.

1. Задают первоначальное число интервалов разбиения m и вычисляют приближенное значение интеграла S1 выбранным методом.

2. Число интервалов удваивают m = 2m.

3. Вычисляют значение интеграла S2.

4. Если |S1 – S2| ≥ ε (ε – заданная погрешность), то S1 = S2, расчет повторяют – переход к п. 2.

5. Если |S1 – S2| < ε, т. е. заданная точность достигнута, выполняют вывод результатов: S2 – найденное значение интеграла с заданной точностью ε, m – количество интервалов.

Метод 2. Анализ формул (10.1), (10.2) и (10.3) показывает, что точное значение интеграла находится между значениями $\Phi_{\text{СР}}$ и $\Phi_{\text{ТР}}$, при этом имеет место соотношение

$$\Phi_{\text{СИ}} = (\Phi_{\text{ТР}} + 2 \cdot \Phi_{\text{СР}})/3.$$

Это соотношение часто используется для контроля погрешности вычислений. Расчет начинается с m = 2 и производится по двум методам, в результате получают $\Phi_{\text{СР}}$, $\Phi_{\text{ТР}}$. Если $|\Phi_{\text{СР}} - \Phi_{\text{ТР}}| \geq \varepsilon$, увеличивают m = 2m и расчет повторяют.

Формулы Гаусса

При построении предыдущих формул в качестве узлов интерполяционного многочлена выбирались середины и (или) концы интервала разбиения. При этом оказалось, что увеличение количества узлов не всегда приводит к уменьшению погрешности. Значительно увеличить точность можно за счет удачного расположения узлов можно.

Суть методов Гаусса порядка *n* состоит в таком расположении *n* узлов интерполяционного многочлена на отрезке $[x_i, x_{i+1}]$, при котором достигается минимум погрешности *квадратурной формулы*. Анализ показывает, что узлами, удовлетворяющими такому условию, являются нули ортогонального многочлена Лежандра степени *n* (см. подразд. 8.1).

Для *n* = 1 один узел должен быть выбран в центре отрезка, следовательно, метод средних является методом ***Гаусса первого*** порядка.

Для *n* = 2 узлы должны быть выбраны следующим образом:

$$x_i^{1,2} = x_{i+1/2} \mp \frac{h}{2} \cdot 0.5773502692\,,$$

и соответствующая формула ***Гаусса второго*** порядка имеет вид

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}[f(x_i^1)+f(x_i^2)].$$

Для $n = 3$ узлы выбираются следующим образом:

$$x_i^0 = x_{i+1/2}, \quad x_i^{1,2} = x_i^0 \mp \frac{h}{2}\cdot 0.7745966692,$$

и соответствующая формула ***Гаусса третьего*** порядка имеет вид

$$\int_a^b f(x)dx \approx \frac{h}{2}\sum_{i=1}^{n}(\frac{5}{9}f(x_i^1)+\frac{8}{9}f(x_i^0)+\frac{5}{9}f(x_i^2)).$$

10.2. Формулы численного дифференцирования

Формулы для расчета производной $d^m f / dx^m$ в точке x получаются следующим образом. Берется несколько близких к x узлов $x_1, x_2, ..., x_n (n \ge m+1)$, называемых ***шаблоном*** (точка x может быть одним из узлов). Вычисляются значения $y_i = f(x_i)$ в узлах шаблона, строится интерполяционный многочлен Ньютона (см. подразд. 8.3) и после взятия производной от этого многочлена $d^m P_{n-1} / dx^m$ получается выражение приближенного значения производной (формула численного дифференцирования) через значения функции в узлах шаблона: $d^m f / dx^m \approx \Lambda_m^n \cdot d^m f / dx^m \approx \Lambda_m^n [f] = d^m P_{n-1} / dx^m$.

При $n = m + 1$ формула численного дифференцирования не зависит от положения точки x внутри шаблона, т. к. в этом случае m-я производная от полинома m-й степени $P_m(x)$ есть константа. Такие формулы называют ***простейшими формулами*** численного дифференцирования.

Анализ оценки погрешности вычисления производной

$$\varepsilon = \max_{x_1<x<x_n}\left|\frac{d^m f}{dx^m} - \Lambda_m^n[f]\right| \le \frac{\max_x\left|f^{(m)}(x)\right|}{n-m}\max_i|x-x_i| \le Ch^{n-m}, \qquad (10.4)$$

$$h = \max|x_i - x_{i-1}|; C = \text{const}, n \ge m+1$$

показывает, что погрешность минимальна для значения x в центре шаблона и возрастает на краях. Поэтому узлы шаблона, если это возможно, выбираются симметрично относительно x. Заметим, что порядок погрешности при $h \to 0$ равен $n - m \ge 1$, и для повышения точности можно либо увеличивать n, либо уменьшать h (последнее более предпочтительно).

Приведем несколько важных формул для равномерного шаблона:

$$\frac{df}{dx} \approx \frac{dP_1}{dx} = \Lambda_1^2[f(x)] = \frac{y_2 - y_1}{h};\ x_1 \le x \le x_2. \qquad (10.5)$$

Простейшая формула (10.2) имеет второй порядок погрешности в центре интервала и первый по краям:

$$\frac{df}{dx} \approx \frac{dP_2}{dx} = \Lambda_1^3\left[f(x)\right] = \frac{y_2 - y_1}{h} + (2x - x_1 - x_2)\frac{y_1 - 2y_2 + y_3}{2h^2}. \qquad (10.6)$$

Эта формула имеет второй порядок погрешности во всем интервале $x_1 \le x \le x_3$ и часто используется для вычисления производной в крайних точках таблицы, где нет возможности выбрать симметричное расположение узлов. Заметим, что из (10.3) получаются три важные формулы второго порядка точности:

$$\frac{df(x_2)}{dx} = \Lambda_1^3\left[f(x_2)\right] = \frac{y_3 - y_1}{2h}; \qquad (10.7)$$

$$\frac{df(x_1)}{dx} = \Lambda_1^3\left[f(x_1)\right] = -\frac{3y_1 - 4y_2 + y_3}{2h}; \qquad (10.8)$$

$$\frac{df(x_3)}{dx} = \Lambda_1^3\left[f(x_3)\right] = \frac{y_1 - 4y_2 + 3y_3}{2h}; \qquad (10.9)$$

Для вычисления второй производной часто используют следующую простейшую формулу:

$$\frac{d^2 f}{dx^2} \approx \frac{d^2 P_2}{dx^2} = \Lambda_2\left[f(x)\right] = \frac{y_1 - 2y_2 + y_3}{h^2};\ x_1 \le x \le x_3, \qquad (10.10)$$

которая имеет второй порядок погрешности в центральной точке x_2.

10.3. Пример выполнения задания

Написать и отладить программу вычисления интеграла от функции $f(x) = 4x - 7\sin x$ на интервале [–2, 3] методом Симпсона с выбором: по заданному количеству разбиений n и заданной точности ε. Панель диалога будет иметь вид, представленный на рис. 10.4.

Как и в предыдущих примерах, приведем только тексты функций-обработчиков соответствующих кнопок:

```
typedef double (*type_f)(double);
double fun(double);
double Simps(type_f, double, double, int);
//---------------------- Текст функции-обработчика кнопки РАСЧЕТ ----------
--------
double a, b, x, eps, h, Int1, Int2, pogr;
int n, n1;
a = StrToFloat(Edit1->Text);        b = StrToFloat(Edit2->Text);
eps = StrToFloat(Edit3->Text);     n = StrToInt(Edit4->Text);
h = (b – a)/100;                          // Шаг вывода исходной функции
Chart1->Series[0]->Clear();
for(x = a – h; x < b + h; x + = h)
            Chart1->Series[0]->AddXY(x, fun(x));
switch(RadioGroup2->ItemIndex)   {
   case 0:
```

```
            Memo1->Lines->Add("Расчет по разбиению на n = " + IntTo-
Str(n));
            Int1 = Simps(fun,a,b,n);
        break;
        case 1:
            n1 = 2;
            Memo1->Lines->Add("Расчет по точности eps");
            Int1 = Simps(fun,a,b,n1);
            do {
                n1 *= 2;
                Int2 = Simps(fun,a,b,n1);
                pogr = fabs(Int2-Int1);
                Int1 = Int2;
            } while(pogr > eps);
            Memo1->Lines->Add("При n = " +IntToStr(n1));
      break;
      }
      Memo1->Lines->Add("Значение интеграла = " + FloatTo-
StrF(Int1,ffFixed,8,6));
    //------------------------ Метод Симпсона ------------------------------
    double Simps(type_f f, double a, double b, int n) {
      double s = 0,h,x;
      h = (b – a) / n;
      x = a;
      for(int i = 1; i <= n; i++)  {
          s += f(x) + 4 * f(x + h/2) + f(x + h);
          x += h;
      }
      return s * h / 6;
    }
    //----------------- Подынтегральная функция f(x) --------------------
    double fun(double x) {
      return 4*x – 7*sin(x);          // На интервале [–2, 3] значение 5.983
    }
```

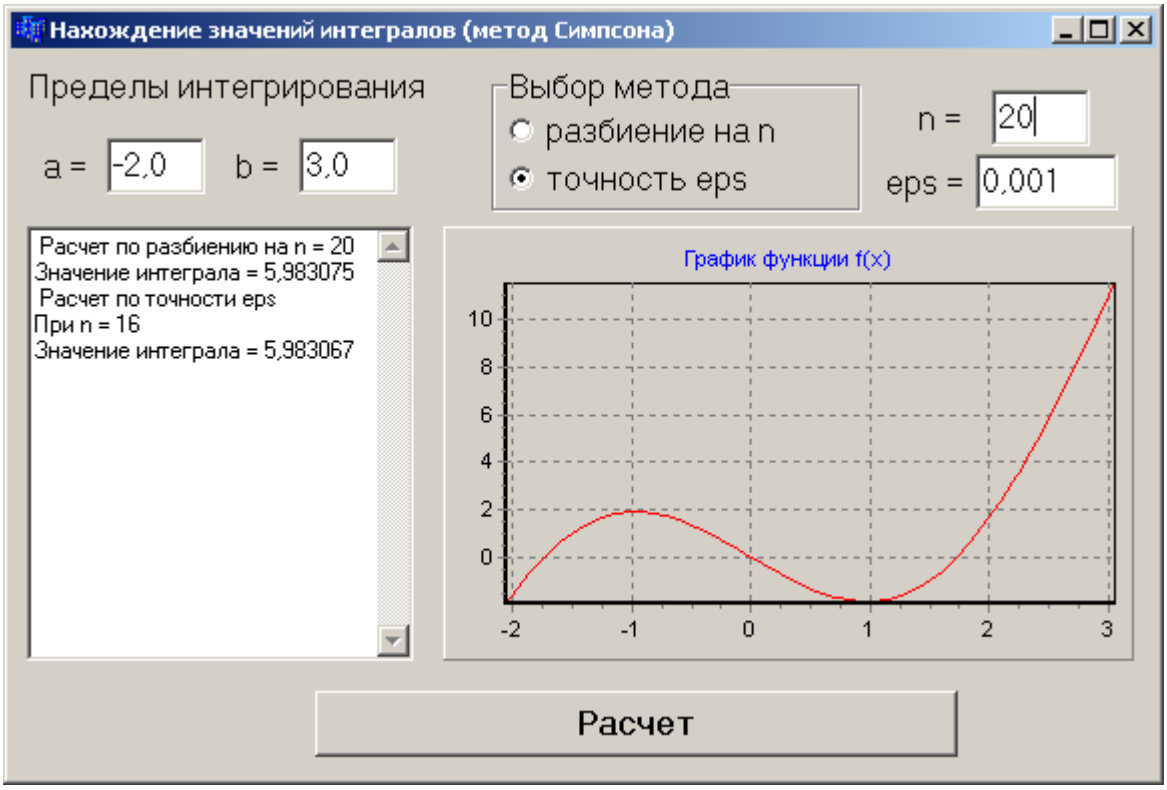


Рис. 9.4

10.4. Индивидуальные задания

Написать и отладить программу вычисления интеграла указанным методом двумя способами – по заданному количеству разбиений *n* и заданной точности ε (метод 1). Реализацию указанного метода оформить отдельной функцией, алгоритм которой описать в виде блок-схемы.

Таблица 10.1

Номер варианта	*Функция f(x)*	*a*	*b*	*Метод интегрирования*	*Значение интеграла*
1	$4x - 7\sin(x)$	–2	3	Средних	5.983
2	$x^2 - 10\sin^2(x)$	0	3	Трапеций	– 6.699
3	$\ln(x) - 5\cos(x)$	1	8	Симпсона	8.896
4	$e^x / x^3 - \sin^3(x)$	4	7	Автомат-метод 2	6.118
5	$\sqrt{x} - \cos^2(x)$	5	8	Гаусса 2	6.067
6	$\ln(x) - 5\sin^2(x)$	3	6	Гаусса 3	–3.367

Окончание таблицы 10.1

Номер варианта	*Функция f(x)*	*a*	*b*	*Метод интегрирования*	*Значение интеграла*
7	$x-5\sin^2(x)$	1	4	Средних	0.100
8	$\sin^2(x)-x/5$	0	4	Трапеций	0.153
9	x^3+10x^2	–8	2	Симпсона	713.3
10	x^3-5x^2	–2	5	Автомат-метод 2	– 69.42
11	$x^3+6x^2-0.02e^x$	–5	3	Гаусса 2	167.6
12	$x^2+5\cos(x)$	–1	4	Гаусса 3	22.09
13	$\sin^2(x)-3\cos(x)$	1	7	Средних	3.533
14	$x^3-50\cos(x)$	–2	5	Автомат-метод 2	154.73
15	$0.1x^3+x^2-10\sin(x)$	–4	2	Симпсона	20.375
16	$\sin^2(x)-x/5$	0	4	Трапеций	0.153

10.5. Контрольные вопросы

1. Что такое квадратурная формула?

2. Какой из описанных в данной лабораторной работе методов дает максимальную точность?

Лабораторная работа № 11. Методы нахождения минимума функции одного аргумента

Цель работы: изучить численные алгоритмы определения минимума функций одного аргумента.

11.1. Постановка задачи

Задача нахождения минимума функции одной переменной min $f(x)$ нередко возникает в практических приложениях. Кроме того, многие методы решения задачи минимизации функции многих переменных сводятся к многократному поиску одномерного минимума. Поэтому разработка все новых, более эффективных одномерных методов оптимизации продолжается и сейчас, несмотря на кажущуюся простоту задачи.

Наиболее часто используемые методы можно разбить на два класса:

1) *методы уточнения минимума на заданном интервале* $[a, b]$ (метод деления пополам, метод золотого сечения);

2) *методы спуска к минимуму* из некоторой начальной точки x_0 (метод последовательного перебора, метод квадратичной параболы, метод кубической параболы).

Методы из класса 1 предназначены для нахождения *условного минимума*. Задача ставится следующим образом: требуется найти такое значение x_m из отрезка $[a, b]$, при котором достигается минимум функции $y_m = f(x_m)$, т. е. для любого $x \in [a, b]$ выполняется условие $y_m \le f(x)$.

Методы из класса 2 предназначены для поиска и уточнения *безусловного локального минимума*. Задача ставится следующим образом: требуется найти такое значение x_m, $|x_m| < \infty$, при котором достигается локальный минимум $y_m = f(x_m)$, т. е. для любого x из некоторой ε окрестности $E = \{x, |x - x_m| < \varepsilon\}$ выполняется $y_m \le f(x)$. В этом случае при нахождении точки x_m обычно нет достаточно точной информации о ее положении, более того, локальных минимумов может быть несколько. Поэтому из соображений физического характера задают некоторое начальное приближение x_0, с которого начинают спуск к точке минимума.

Нахождение требуемого минимума функции осуществляется в два этапа.

1. Приближенное определение местоположения минимума из анализа таблицы значений функции.

2. Вычисление точки минимума x_m с заданной точностью одним из следующих методов: метод деления отрезка пополам, метод золотого сечения, метод последовательного перебора, метод квадратичной параболы, метод кубической параболы.

11.2. Методы оптимизации

Метод деления отрезка пополам (MDP)

Задается интервал [α, β] и погрешность ε. Вычисляются значения функции в двух точках вблизи середины интервала и отбрасывается та часть интервала, которая содержит точку с большим значением функции. Расчет происходит до тех пор, пока длина интервала не станет меньше заданной погрешности ε. Схема алгоритма представлена на рис. 11.1.

В среднем за одно вычисление функции отрезок, на котором находится *x*, уменьшается примерно в 1.33 раза. Этот метод прост в реализации, позволяет находить минимум разрывной функции, однако требует большого числа вычислений функции для обеспечения заданной точности.

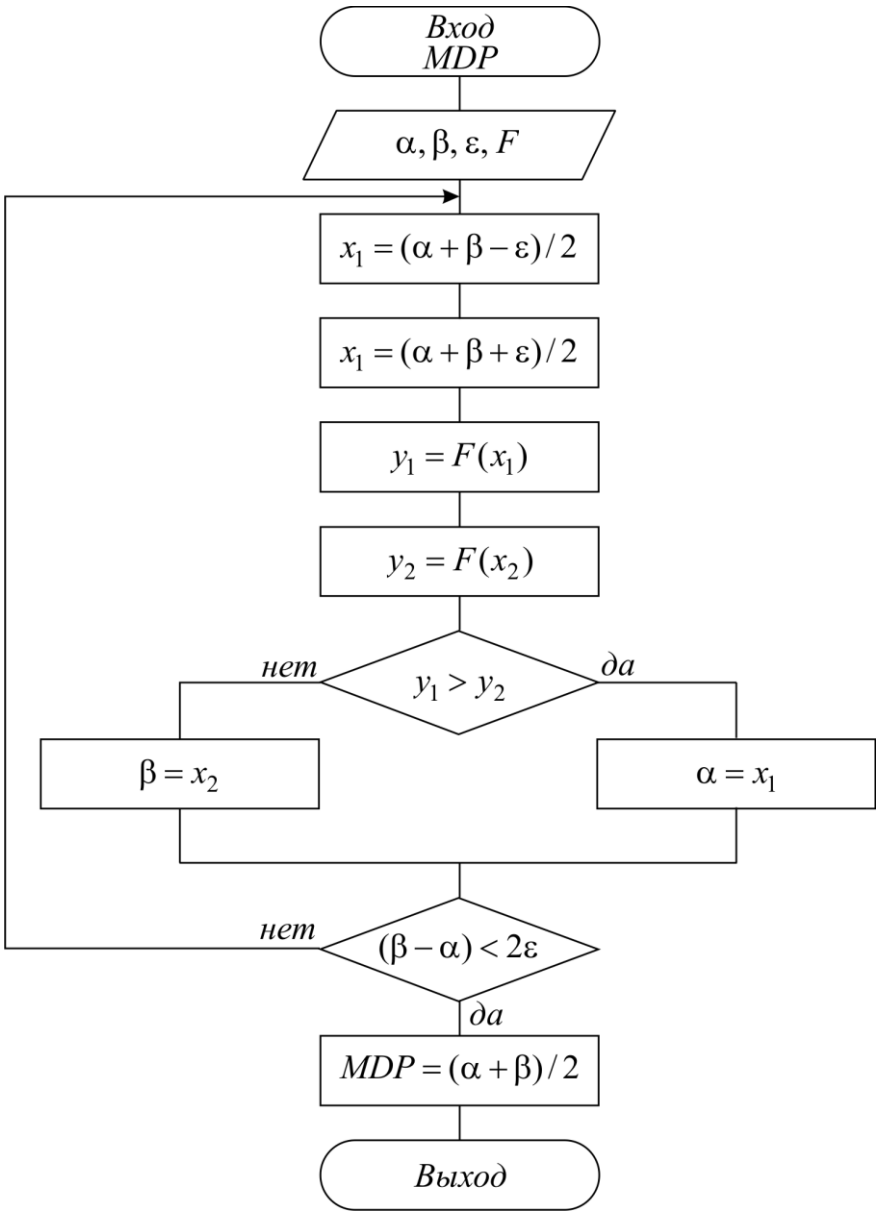


Рис. 11.1

Метод золотого сечения (MZS)

Золотое сечение – это такое деление отрезка $[\alpha, \beta]$ на две неравные части $[\alpha, x]$ и $[x, \beta]$, при котором имеет место следующее соотношение:

$$x\beta / \alpha\beta = \alpha x / x\beta = 1 - \xi, \quad \xi = 2/(3+\sqrt{5}) \cong 0.381966011.$$

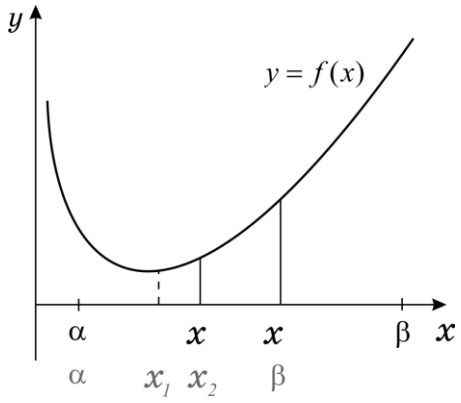


Рис. 11.2

Алгоритм поиска минимума аналогичен вышеописанному MDP и отличается тем, что вначале точки x_1 и x_2 выбираются так, чтобы выполнялось золотое отношение, и вычисляются значения функции в этих точках.

Затем, после очередного сокращения интервала путем отбрасывания неблагоприятной крайней точки на оставшемся *отрезке уже имеется точка, делящая его в золотом отношении* (точка x_1 на рис. 11.2), известно и значение функции в этой точке. Остается лишь выбрать симметричную функцию и вычислить ее значение в этой точке для того, чтобы решить, какую из крайних точек отбросить. Схема алгоритма представлена на рис. 11.3.

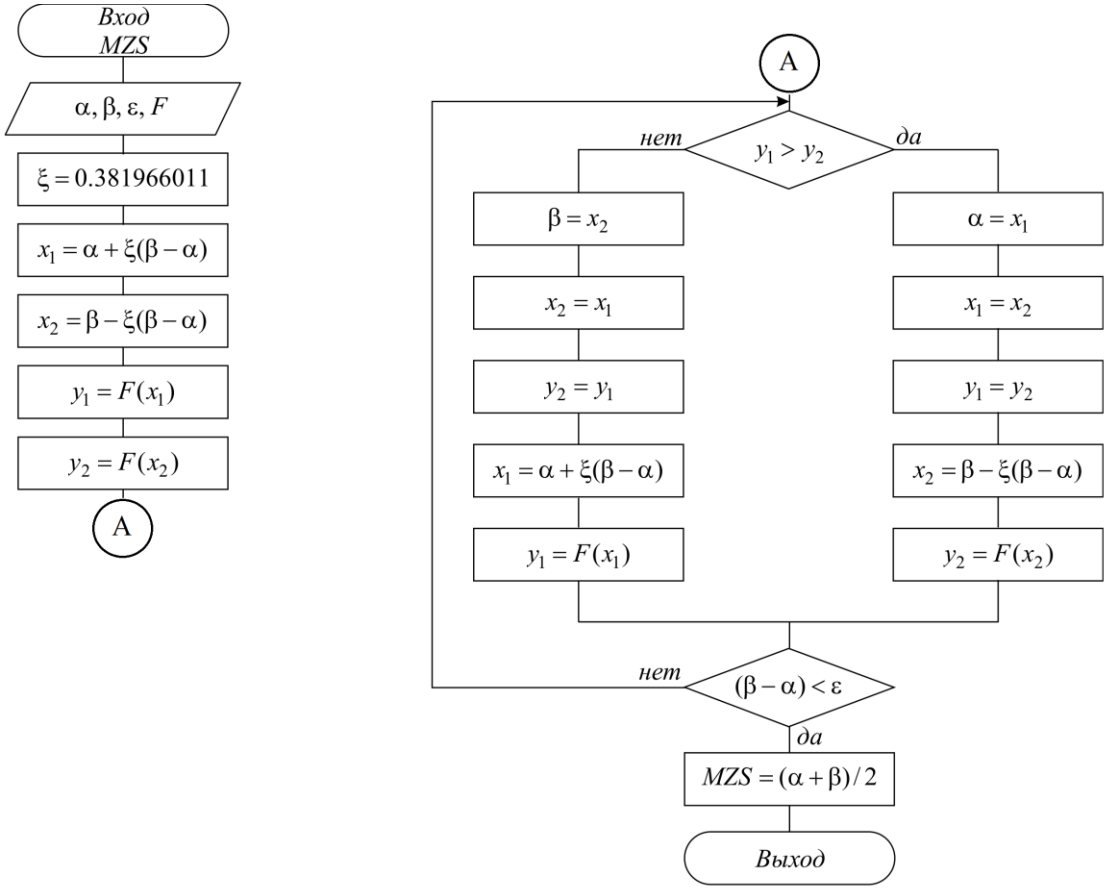


Рис. 11.3

За одно вычисление функции отрезок, на котором находится x_m, уменьшается в $1 - \xi \cong 1.61$ раза, т. е. быстрее чем MDP. Данный метод является наилучшим среди методов класса 1.

Метод последовательного перебора (MPP)

Идея этого метода состоит в том, что, спускаясь из точки x_0 с заданным шагом h в направлении уменьшения функции, устанавливают интервал длиной h, на котором находится минимум, и затем его уточняют. Уточнение можно осуществить либо методом золотого сечения, либо повторяя спуск из последней точки, уменьшив шаг и изменив его знак. Алгоритм последнего варианта приведен на рис. 11.4.

Скорость сходимости данного метода существенно зависит от удачного выбора начального приближения x_0 и шага h. Шаг h следует выбирать как половину оценки расстояния от x_0 до предполагаемого минимума x_m.

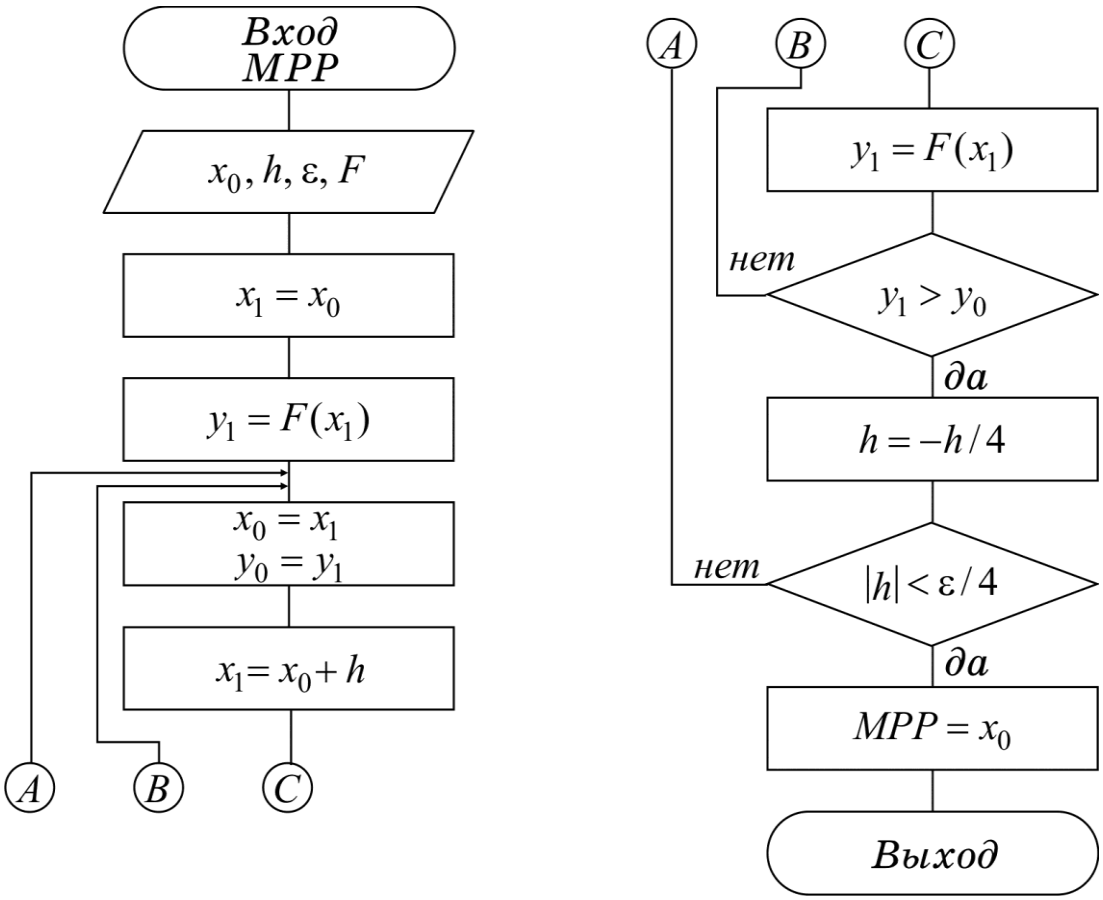


Рис. 11.4

Метод квадратичной параболы (MP2)

Для ускорения спуска к минимуму из некоторой точки x_0 используют локальные свойства функции вблизи этой точки. Так, скорость и направление убывания можно определить по величине и знаку первой производной. Вторая производная характеризует направление выпуклости: если $f'' > 0$, то функция имеет выпуклость вниз, иначе – вверх. Вблизи локального безусловного минимума дважды дифференцируемая функция всегда имеет выпуклость вниз. Поэтому, если вблизи точки минимума функцию аппроксимировать квадратичной параболой, то она будет иметь минимум. Это свойство и используется в методе квадратичной параболы, суть которого в следующем.

Вблизи точки x_0 выбираются три точки x_1, x_2, x_3. Вычисляются значения y_1, y_2, y_3. Через эти точки проводится квадратичная парабола:

$$\begin{aligned} &p(x-x_3)^2 + q(x-x_3) + r = pz^2 + qz + r; \\ &z = x - x_3;\ z_1 = x_1 - x_3;\ z_2 = x_2 - x_3;\ r = y_3; \\ &p = \frac{(y_1 - y_3)z_2 - (y_2 - y_3)z_1}{z_1 z_2 (z_1 - z_2)};\ q = \frac{(y_1 - y_3)z_2^2 - (y_2 - y_3)z_1^2}{z_1 z_2 (z_2 - z_1)}. \end{aligned} \qquad (11.1)$$

Если $p>0$, то парабола имеет минимум в точке $z_m = -q/(2p)$. Следовательно, можно аппроксимировать положение минимума функции значением $x_{m1} = x_3 + z_m$ и, если точность не достигнута, следующий спуск производить, используя эту новую точку и две предыдущие. Получается последовательность x_{m1}, x_{m2}, x_{m3}, ... , сходящаяся к точке x_m.

Данный метод сходится очень быстро и является одним из наилучших методов спуска. Следует отметить, однако, что вблизи минимума расчет по приведенным здесь формулам для p и q приводит к накоплению погрешности из-за потери значащих цифр при вычитании близких чисел. Поэтому разные авторы предлагают свои эквивалентные формулы, счет по которым более устойчив. Кроме того, в алгоритм вносятся некоторые поправки, позволяющие предусмотреть различные неприятные ситуации – переполнение, деление на нуль, уход от корня. Схема алгоритма представлена на рис. 11.5.

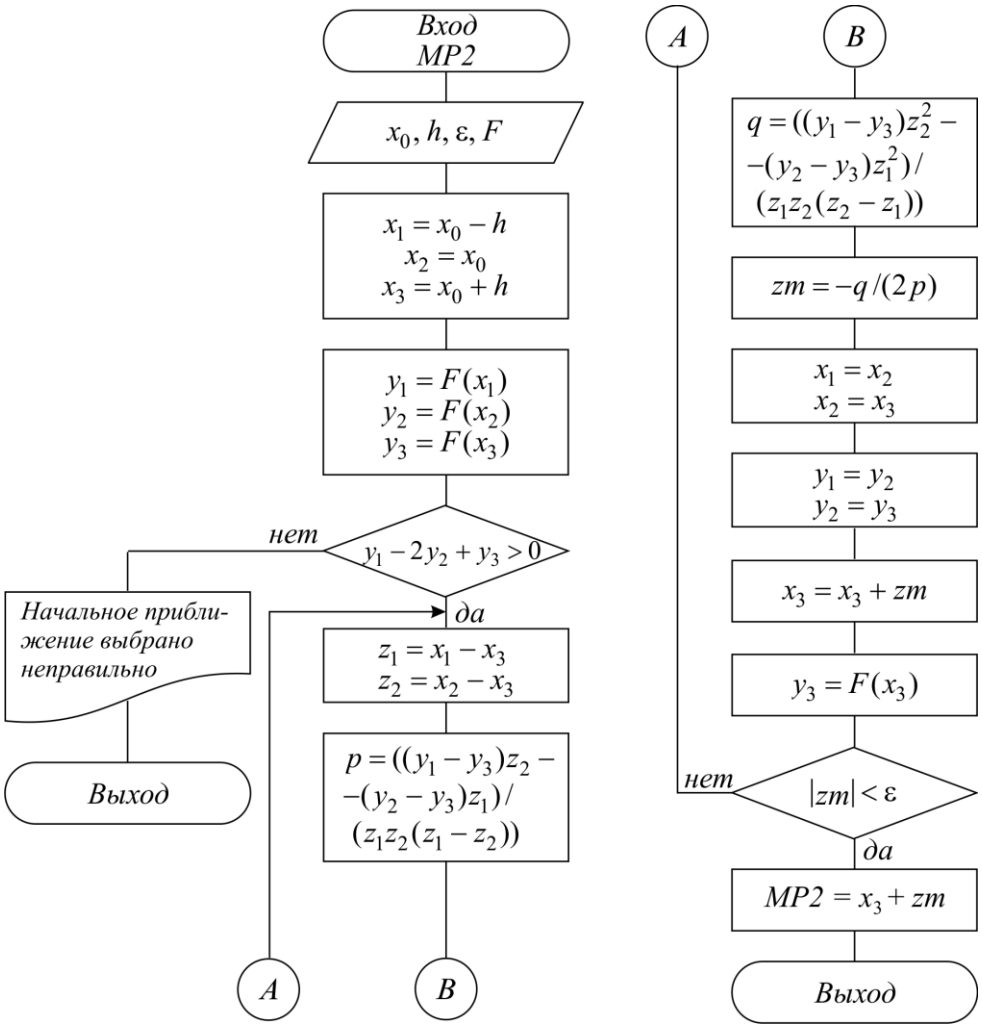


Рис. 11.5

Метод кубической параболы (MP3)

Данный метод аналогичен предыдущему, но за счет использования аппроксимации кубической параболой имеет более высокую сходимость, *если функция допускает простое вычисление производной*. При его использовании вблизи точки x_0 выбираются две точки x_1 и x_2 (обычно $x_1 = x_2$), вычисляются значения функции y_1, y_2 и ее производной $D_1 = f'(x_1)$, $D_2 = f'(x_2)$. Затем через эти точки проводится кубическая парабола, коэффициенты которой определяются таким образом, чтобы совпадали значения производных параболы и функции:

$$p(x - x_2)^3 + q(x - x_2)^2 + r(x - x_2) = s = pz^3 + qz^2 + rz + s = P(z);$$

$$z = x - x_2;\ z_1 = x_1 - x_2;$$

$$P(x_2) = y_2;\ P'(x_2) = D_2;\ P(z_1) = y_1;\ P'(z_1) = D_1.$$

Как нетрудно убедиться, коэффициенты параболы вычисляются по следующим формулам:

$$s = y_1;\ r = D_1;$$

$$p = (D_1 - D_2 - 2(y_1 - y_2 - D_2 \cdot z_1)/z_1)z_1^2;$$
$$q = (D_2 - D_1 + 3(y_1 - y_2 - D_2 \cdot z_1)/z_1)/z_1.$$

Известно, что кубическая парабола имеет минимум в точке

$$z_m = (-q + \sqrt{q^2 - 3pr})/3p.$$

Поэтому приближенное положение минимума можно получить по формуле $x_{m1} = x_2 + z_m$ и, если точность не достигнута, следующий спуск следует производить уже из точек x_2, x_{m1} (точка x_1 отбрасывается). Если подкоренное выражение окажется отрицательным, то спуск следует производить до точки перегиба параболы $z_{m1} = -q/3p$. Следует также убедиться, что в начальной точке функция вогнута вниз $\frac{D_2 - D_1}{x_2 - x_1} > 0$. Схема алгоритма представлена на рис. 11.6.

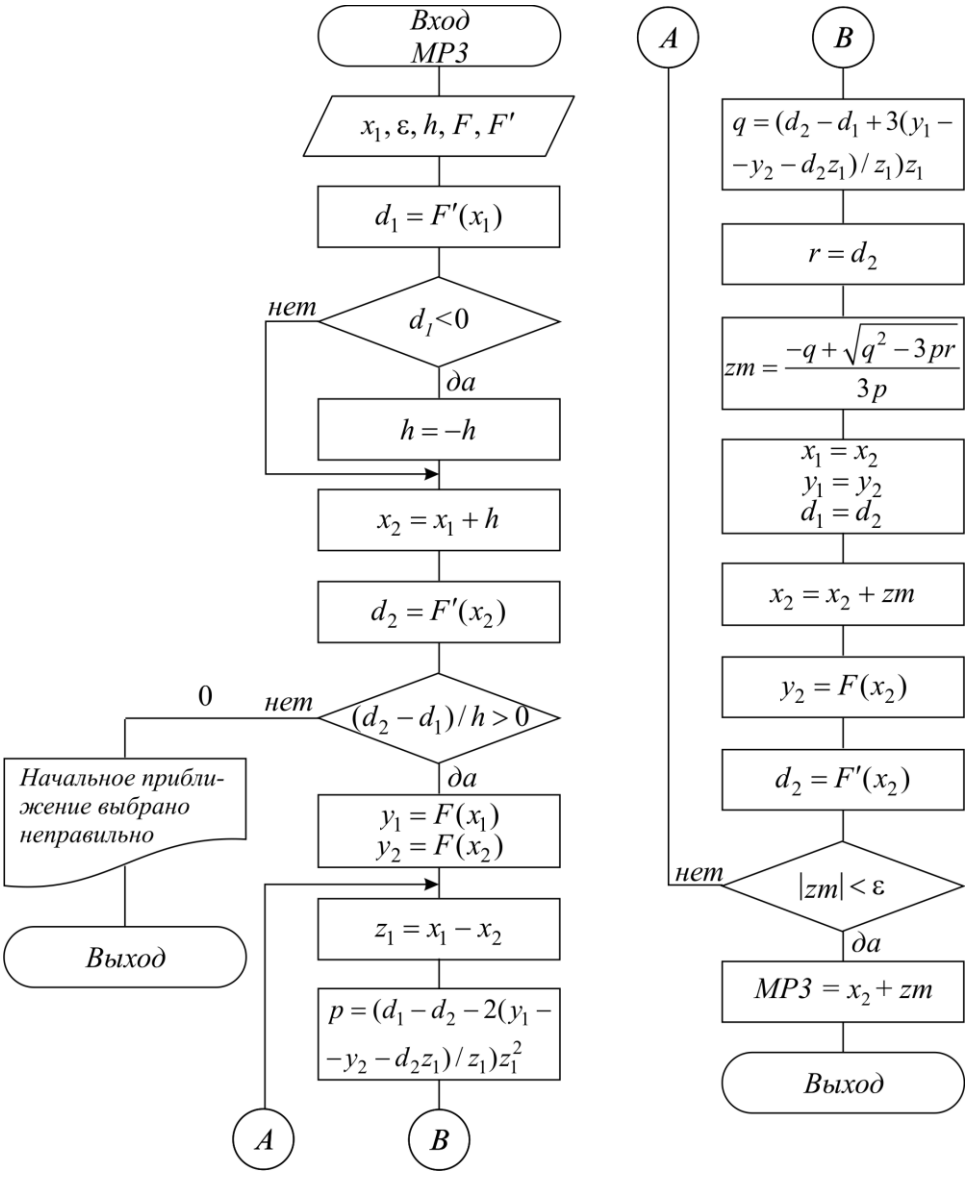


Рис. 11.6

Следует отметить, что вблизи точки минимума расчет по приведенным здесь простейшим формулам для *p*, *q* не всегда устойчив из-за ошибок округления, поэтому различные авторы рекомендуют использовать несколько преобразованные формулы.

11.3. Индивидуальные задания

В соответствии с изображенной на рис. 11.7 схемой требуется отладить программу определения минимума указанной в таблице функции заданным методом.

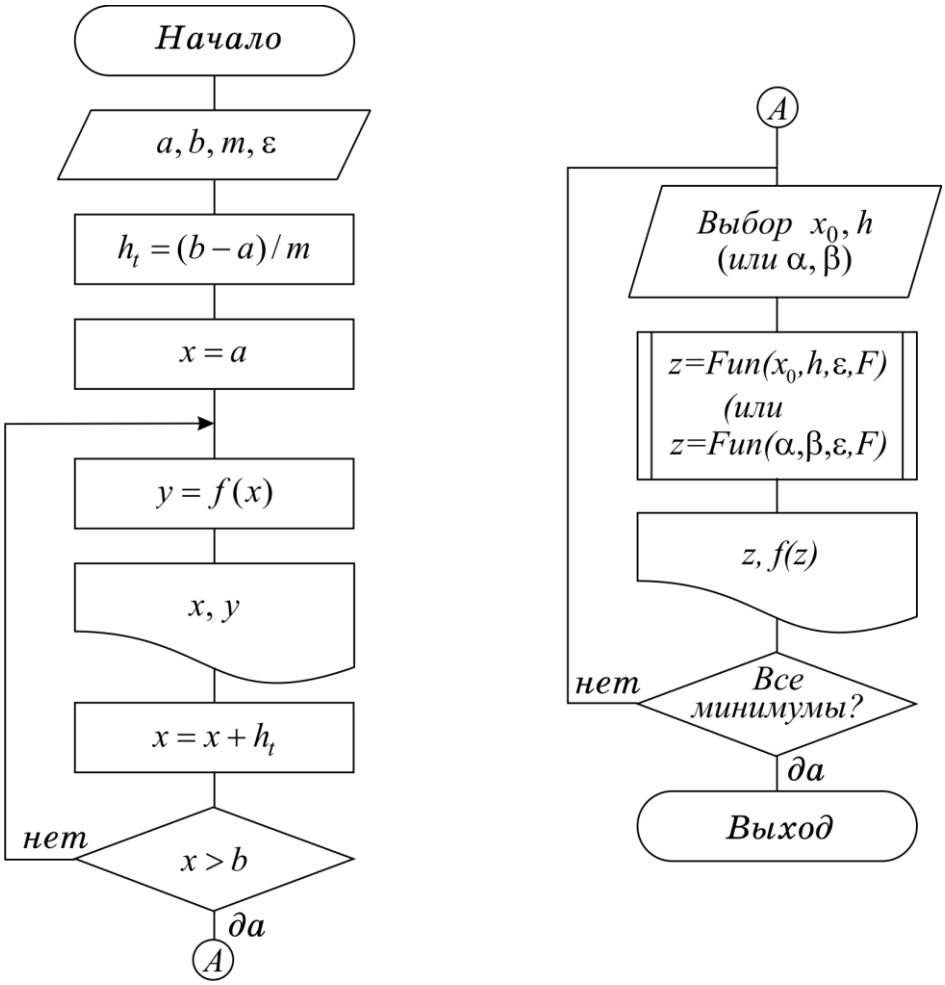


Рис. 11.7

Сначала на экране вывести таблицу значений функции и сделать запрос на ввод начального приближения (α, β или x_0, *h*) для вычисления требуемого локального минимума. В качестве функции *Fun* использовать метод в соответствии с заданным вариантом. Расчет функции, а также метод нахождения минимума оформить в виде отдельных подпрограмм. Выбрать *m* и ε по усмотрению. Все функции из табл. 11.1 на указанном интервале имеют три локальных минимума.

После выполнения расчетов построить график исследуемой функции и проанализировать зависимость количества итераций от ε ($\varepsilon=10^{-2}$, $\varepsilon=10^{-3}$, $\varepsilon=10^{-4}$, $\varepsilon=10^{-5}$), для чего встроить в алгоритм счетчик количества вычислений функции.

Таблица 11.1

Номер варианта.	*Минимизируемая функция f(x)*	*Интервал*		*Метод*
		a	*b*	
1	$x-7\sin^2(x)$	–3	6	MDP
2	$x\sin(x)-10\sin^2(x)$	–6	3	MZS
3	$\ln(x)-5\cos^2(x)$	2	11	MPP
4	$e^x/x^3-30\cos(x)$	0.2	12	MP2
5	$\sqrt{x}-\cos(x)$	4	20	MP3
6	$\ln^2(x)-10\cos^2(x)$	2	10	MDP
7	$x-5\sin^2(x)$	1	9	MZS
8	$20x\cdot\sin^2(x)-x^2$	–9	–1	MPP
9	$x^2-100\sin(x)$	–6	10	MP2
10	$20x^3\cdot\sin(x)$	–6	6	MP3
11	$\sin^4(x)-\ln(x)$	2	11	MDP
12	$\sin^2(x)+\cos(x)$	–4	4	MZS
13	$x^2-4x\cdot\sin(x)+\cos(x)$	–4	9	MPP
14	$x^2\cdot\cos(x)+2\sin(x)$	8	24	MP2
15	$x\cdot\cos(x)-x$	2	18	MP3

11.4. Контрольные вопросы

1. Что такое условный и локальный минимумы, в чем их отличие?
2. В чем суть метода последовательного перебора?
3. Объясните графически, почему метод золотого сечения эффективнее метода деления пополам?
4. Дайте геометрическую интерпретацию методов квадратичной и кубической парабол.

Лабораторная работа №12. Решение задачи Коши для обыкновенных дифференциальных уравнений

Цель работы: изучить численные алгоритмы решения задачи Коши.

12.1. Типы задач для обыкновенных дифференциальных уравнений

Обыкновенными дифференциальными уравнениями можно описать поведение системы взаимодействующих частиц во внешних полях, процессы в электрических цепях, закономерности химической кинетики и многие другие явления. Поэтому решение обыкновенных дифференциальных уравнений занимает одно из важнейших мест среди прикладных задач физики, электроники, химии и техники.

Конкретная прикладная задача может приводить к дифференциальному уравнению любого порядка или к системе таких уравнений. Известно, что произвольную систему дифференциальных уравнений любого порядка можно привести к некоторой эквивалентной системе уравнений первого порядка. Среди таких систем выделим класс систем, разрешенных относительно производной неизвестных функций:

$$\begin{cases} \dfrac{du_1(x)}{dx} = f_1(x,\ u_1,\ ...,\ u_m), \\ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \ . \\ \dfrac{du_m(x)}{dx} = f_m(x,\ u_1,\ ...,\ u_m). \end{cases} \qquad (12.1)$$

Обычно требуется найти решение системы $\vec{u}(x) = \{u_1(x),\ ...,\ u_m(x)\}$ для значений x из заданного интервала $a \le x \le b$.

Известно, что система (12.1) имеет бесконечное множество решений, семейство которых в общем случае зависит от m произвольных параметров $\vec{n} = \{\tilde{n}_1,\ ...,\ c_m\}$ и может быть записано в виде $\vec{u} = \vec{u}(x,\ \vec{c})$. Для определения значений этих параметров, т. е. для выделения одного нужного решения, надо наложить дополнительно m условий на функции $\vec{u} = \{u_1,\ ...,\ u_m\}$. В зависимости от способа постановки дополнительных условий можно выделить два основных типа задач, наиболее часто встречающихся на практике:

1) ***краевая (граничная) задача***, когда часть условий задается на границе a (при $x = a$), остальные условия – на границе b (при $x = b$); обычно это значения искомых функций и их производных на границах;
2) ***задача Коши (задача с начальными условиями)***, когда все условия заданы в начале отрезка в виде

$$u_1(a) = u_1^0;\ ...;\ u_m(a) = u_m^0. \qquad (12.2)$$

При изложении методов решения задачи Коши воспользуемся компактной записью задач (12.1), (12.2) в векторной форме.

$$\frac{d\vec{u}}{dx} = f(x,\vec{u}); \quad \vec{u}(a) = \vec{u}^0. \tag{12.3}$$

Требуется найти $\vec{u}(x)$ для $a \le x \le b$.

12.2. Основные положения метода сеток для решения задачи Коши

Чаще всего задача (12.3) решается методом сеток.

Суть метода сеток *состоит в следующем*:

1. В области интегрирования выбирается упорядоченная система точек $a = x_0 < x_1 < x_2 < ... < x_n < x_{n+1} = b,$ называемая *сеткой*. Точки x_i называют *узлами*, а $h_k = x_k - x_{k-1}$ – *шагом сетки*. Если $h_k = h = (b-a)/n$, сетка называется ***равномерной***. Для упрощения в дальнейшем будим считать сетку равномерной.

2. Решение $\vec{u}(x)$ ищется в виде таблицы значений в узлах выбранной сетки $\vec{u}^k = \vec{u}(x_k),$ для чего дифференциальное уравнение заменяется системой алгебраических уравнений, связывающих между собой значения искомой функции в соседних узлах. Такая система называется ***конечно-разностной схемой***.

Имеется несколько распространенных способов получения конечно-разностных схем. Приведем здесь один из самых универсальных – ***интегро-интерполяционный метод***.

Согласно этому способу для получения конечно-разностной схемы проинтегрируем уравнение (12.3) на каждом интервале $\left[x_k,\ x_{k+1}\right]$ для k=0, ..., n – 1 и разделим на длину этого интервала:

$$\frac{1}{h}\int_{x_k}^{x_{k+1}} \frac{d\vec{u}}{dx}dx = \frac{\vec{u}^{k+1} - \vec{u}^k}{h} = \frac{1}{h}\int_{x_k}^{x_{k+1}} \vec{f}(x,\ \vec{u}(x))dx. \tag{12.4}$$

Интеграл в правой части (12.4) аппроксимируем одной из квадратурных формул (см. подразд. 10.3), после чего получаем систему уравнений относительно ***приближенных*** неизвестных значений искомой функции, которые в отличие от точных обозначим $\vec{y}^k \approx \vec{u}^k$.

$$\frac{\vec{y}^{k+1} - \vec{y}^k}{h} = \frac{1}{h}\sum_j \alpha_j \vec{F}^j;\ \vec{F}^j = \vec{f}(x_j, \vec{y}^j);\ x_k \le x_j \le x_{k+1}. \tag{12.5}$$

Здесь x_j – точки внутри интервала, используемые для получения квадратурной формулы (см. подразд. 10.3).

Структура конечно-разностной схемы для задачи Коши (12.5) такова, что она устанавливает закон рекуррентной последовательности $\vec{y}^{k+1} = \varphi\left(\vec{y}^k\right)$ для искомого решения $\vec{y}^0,\ \vec{y}^1,\ \vec{y}^2,\ ...,\ \vec{y}^n$. Поэтому, используя начальное условие задачи (12.2) и задавая $\vec{y}^0 = \vec{u}^0$, затем по рекуррентным формулам последовательно находят все $\vec{y}^k,\ k = 1,\ ...,\ n.$

При замене интеграла приближенной квадратурной формулой вносится ***погрешность аппроксимации*** дифференциального уравнения разностным, ко-

торая получается, как невязка, если в конечно-разностном уравнении (12.5) подставить вместо $\vec{y}^k$ значение точного решения $\vec{u}^k$:

$$\psi_k = \left| \frac{\vec{u}^{k+1} - \vec{u}^k}{h} - \frac{1}{h} \sum_j \alpha_j f(x_j, \vec{u}^j) \right| .$$

Воспользовавшись соотношением (12.4), получаем простое выражение для вычисления $\psi_k(h)$:

$$\psi_k(h) = \frac{1}{h} \left| \int_{x_k}^{x_{k+1}} \vec{f}(x, \vec{u}(x)) dx - \sum_j \alpha_j \vec{F}^j \right| , \qquad (12.6)$$

которая зависит от шага сетки.

Говорят, что разностная схема (12.5) ***аппроксимирует*** исходную дифференциальную задачу с порядком p, если при $h \to 0, \psi_k(h) \le Ch^p, C - \text{const}$. Из (12.6) следует, что порядок аппроксимации на единицу меньше, чем порядок погрешности используемой квадратурной формулы на интервале $[x_k, x_{k+1}]$.

Чем больший ***порядок аппроксимации p,*** тем выше ***точность решения***:

$$\varepsilon(h) = \left\| \vec{y} - \vec{u} \right\| = \max_k \left| \vec{y}^k - \vec{u}^k \right| .$$

Основная теорема теории метода сеток утверждает, что если схема устойчива, то при $h \to 0$ погрешность решения $\varepsilon(h)$ стремится к нулю с тем же порядком, что и погрешность аппроксимации:

$$\varepsilon(h) \le C_0 \cdot \max_k \left| \psi_k(h) \right| \le C_0 \cdot C \cdot h^p , \qquad (12.7)$$

где C_0 – константа устойчивости.

Неустойчивость обычно проявляется в том, что с уменьшением h решение $\vec{y}^k \to \infty$ при возрастании k, что легко устанавливается экспериментально с помощью просчета на последовательности сеток с уменьшающимся шагом h, $h/2$, $h/4$, … . Если при этом $\vec{y}^k \to \infty$, то метод неустойчив. Таким образом, если имеется аппроксимация и схема устойчива, то, выбрав достаточно малый шаг h, можно получить решение с заданной точностью. При этом затраты на вычисления резко уменьшаются с увеличением порядка аппроксимации p, т. е. при большем p можно достичь той же точности, используя более крупный шаг h.

12.3. Виды конечно-разностных схем

Большое разнообразие методов обусловлено возможностью по-разному выбирать узлы и квадратурные формулы для аппроксимации интеграла в (12.4) при получении схемы (12.5).

M1. Явная схема первого порядка (Эйлера)

Заменим интеграл в (12.4) по формуле левых прямоугольников:

$$\frac{1}{h}\int_{x_k}^{x_{k+1}} \vec{f}(x,\vec{u}(x))dx \approx \frac{\vec{F}^k h}{h} = \vec{F}^k .$$

Получим

$$\frac{\vec{y}^{k+1}-\vec{y}^k}{h} = \vec{f}(x_k,\vec{y}^k), \quad k=0,\ 1,\ 2,\ ...,\ n . \tag{12.8}$$

Задавая $\vec{y}^0 = \vec{u}^0$, с помощью (12.8) легко получить все последующие значения $\vec{y}^k$, $k=1,\ 2,\ ...,\ n_1$, т. к. формула явно разрешается относительно $\vec{y}^{k+1}$. Погрешность аппроксимации $\psi(h)$ и соответственно точность $\varepsilon(h)$ имеют первый порядок в силу того, что формула левых прямоугольников на интервале $[x_x,\ x_{k+1}]$ имеет погрешность второго порядка, а схема устойчива.

Алгоритм представлен на рис. 12.1.

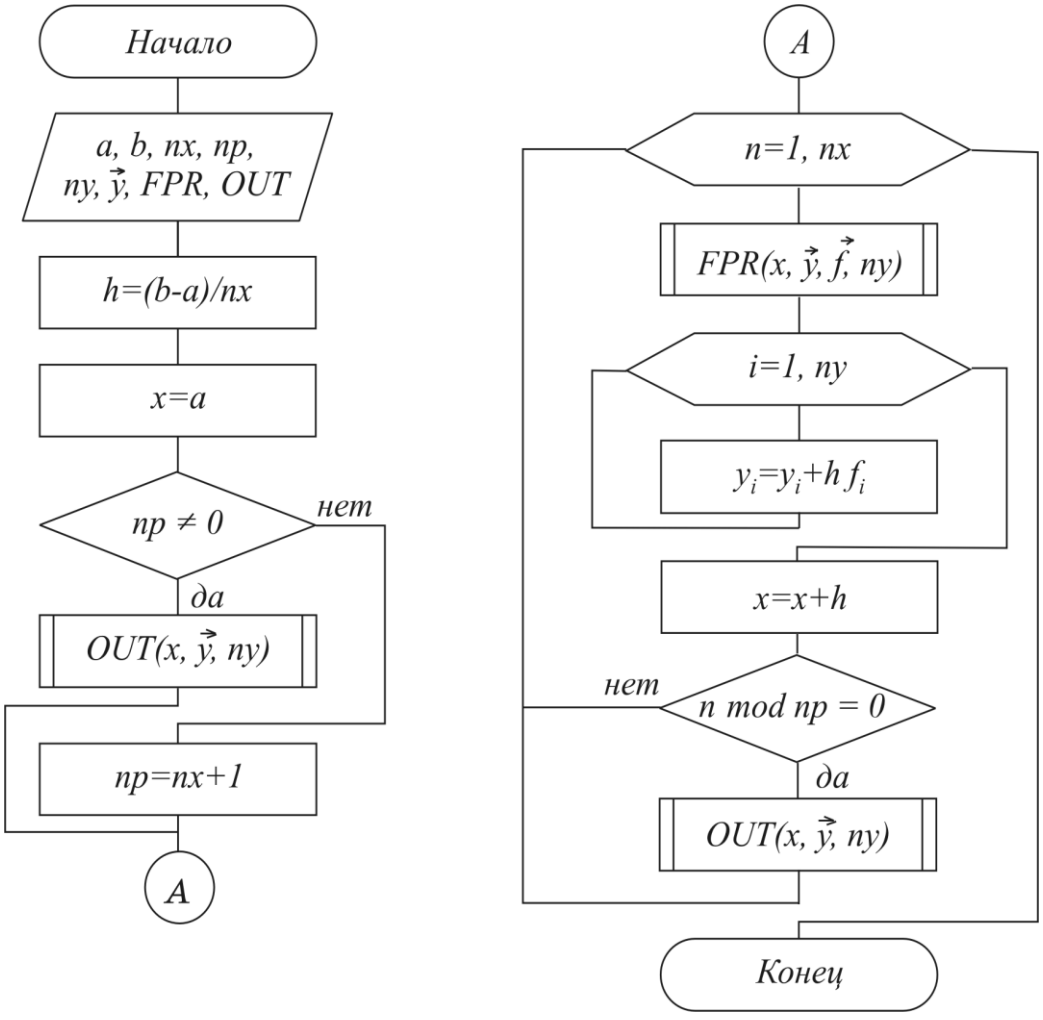


Рис. 12.1

Рассмотрим пример решения задачи Коши явным методом Эйлера. Задача Коши имеет вид:

$$y' = y^2 - x,\ \ y(1) = 0$$

на отрезке $x \in [1,3]$ с шагом по сетке $h = 0.2$.

Из постановки задачи видно, что $f(x,y) = y' = y^2 - x$, а из формулы (12.8) следует, что

$$\vec{y}^{k+1} = \vec{y}^k + h \cdot f(x,y).$$

Зная, что $y(1) = 0,\ x_0 = 1$, получаем первое значение:

$$y^1 = y^0 + h \cdot f(x_0, y^0) = 0 + 0.2 \cdot ((y^0)^2 - x_0) = -0.2.$$

Далее продолжаем процесс рассчета по формулам

$$x_1 = 1.2, y^1 = -0.2 \Rightarrow y^2 = y^1 + h \cdot f(x_1, y^1) = -0.2 + 0.2 \cdot ((y^1)^2 - x_1) = -0.432;$$

$$x_2 = 1.4, y^2 = -0.432 \Rightarrow y^3 = y^2 + h \cdot f(x_2, y^2) =$$

$$= -0.432 + 0.2 \cdot ((y^2)^2 - x_2) = -0.6746752;$$

...

$$x_{10} = 2.8, y^{10} = -1.56185 \Rightarrow y^{11} = y^{10} + h \cdot f(x_{10}, y^{10}) =$$

$$= -1.56185 + 0.2 \cdot ((y^{10})^2 - x_{10}) = -1.634;$$

$$x_{11} = 3, y^{11} = -1.634.$$

M2. Неявная схема 1-го порядка

Используя в (12.5) формулу правых прямоугольников, получим

$$\frac{\vec{y}^{k+1} - \vec{y}^k}{h} = \vec{f}(x_{k+1}, \vec{y}^{k+1}). \qquad (12.9)$$

Эта схема неразрешима явно относительно $\vec{y}^{k+1}$, поэтому для получения $\vec{y}^{k+1}$ требуется использовать итерационную процедуру решения уравнения (12.9) (см. метод простой итерации в подразд. 9.2):

$$\vec{y}^{k+1,s} = \vec{y}^k + \vec{f}(x_{k+1}, \vec{y}^{k+1,s-1}),$$

где $s = 1, 2, 3, \ldots$ – номер итерации.

За начальное приближение берется значение $\vec{y}^{k+1,0} = \vec{y}^k$ с предыдущего шага. Обычно, если значение *h* выбрано удачно, достаточно сделать 2–3 итерации для достижения заданной погрешности $\left\| y^{k+1,s} - y^{k+1,s-1} \right\| < \varepsilon$. Эффективность неявной схемы заключается в том, что у нее константа устойчивости C_0 значительно меньше, чем у явной схемы.

M3. Неявная схема второго порядка

Используя в (12.5) формулу трапеций, получим

$$\frac{\vec{y}^{k+1} - \vec{y}^k}{h} = \frac{\vec{f}(x,\ \vec{y}^k) + \vec{f}(x_{k+1},\ \vec{y}^{k+1})}{2}. \qquad (12.10)$$

Т. к. формула трапеций имеет третий порядок точности на интервале $[x_x,\ x_{k+1}]$, то погрешность аппроксимации $\psi(h)$ – второй.

Схема (12.10) не разрешена относительно $\vec{y}^{k+1}$, поэтому требуется итерационная процедура (см. схему М2):

$$\vec{y}^{k+1,s} = \vec{y}^{k} + \frac{h}{2}(\vec{f}(x_k,\ \vec{y}^{k}) + \vec{f}(x_{k+1}, \vec{y}^{k+1,s-1}),\quad s = 1,\ 2,\ ...,\ \vec{y}^{k+1,0} = \vec{y}^{k}.$$

Алгоритм представлен на рис. 12.2.

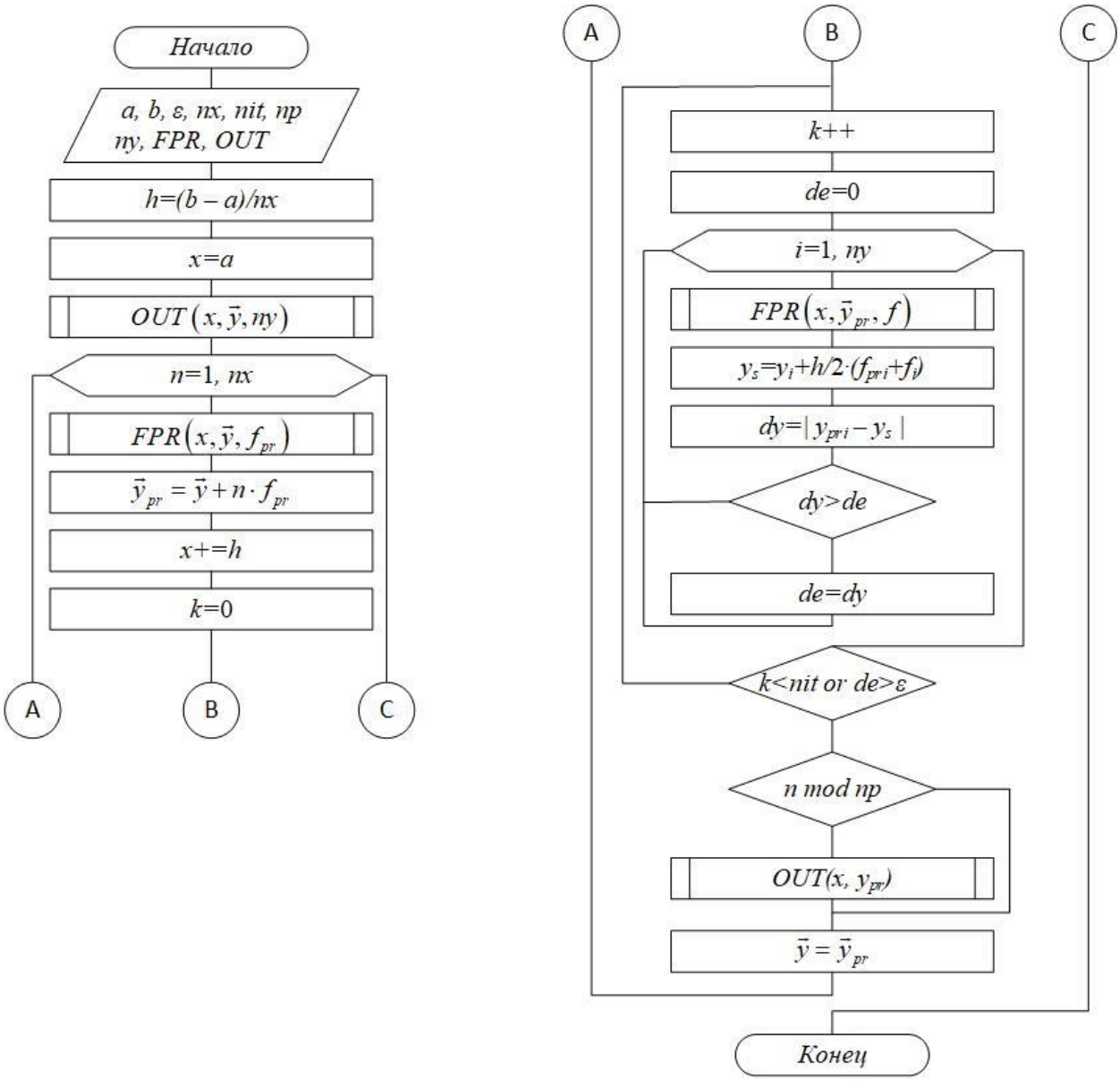


Рис. 12.2

Рассмотрим пример решения задачи Коши неявным методом Эйлера второго порядка (с пересчетом).

Задача Коши имеет вид:

$$y' = y^2 - x,\ \ y(1) = 0$$

на отрезке $x \in [1, 3]$ с шагом по сетке $h = 0.2$.

Из постановки задачи видно, что $f(x, y) = y' = y^2 - x$, а из формулы (12.10) следует, что для начала необходимо рассчитать прогноз

$$\tilde{y}_{i+1} = y_i + h \cdot f(x_i, y_i)$$

а затем произвести коррекцию:

$$y_{i+1} = y_i + h \cdot \frac{f(x_i, y_i) + f(x_{i+1}, \tilde{y}_{i+1})}{2}.$$

Рассчитаем значение по схеме для первой точки отрезка по x:

$$x_0 = 1, y_0 = 0;\ \tilde{y}_1 = y_0 + h \cdot f(x_0, y_0) = 0 + 0.2 \cdot (0 \cdot 0 - 1) = -0.2;$$

$$y_1 = y_0 + h \cdot \frac{f(x_0, y_0) + f(x_1, \tilde{y}_1)}{2} = 0 + 0.2 \cdot \frac{(0 \cdot 0 - 1) + ((-0.2)^2 - 1.2)}{2} = -0.216;$$

$$x_1 = 1.2, y_1 = -0.216;$$

$$\tilde{y}_2 = y_1 + h \cdot f(x_1, y_1) = -0.216 + 0.2 \cdot ((-0.216)^2 - 1.2) = -0.44667;$$

$$y_2 = y_1 + h \cdot \frac{f(x_1, y_1) + f(x_2, \tilde{y}_2)}{2} =$$

$$= -0.216 + 0.2 \cdot \frac{((-0.216)^2 - 1.2) + ((-0.44667)^2 - 1.4)}{2} = -0.452;$$

Продолжая, мы приходим к значению в последней точке

$$x_{10} = 2.8, y_{10} = -1,54075;$$

$$\tilde{y}_{11} = y_{10} + h \cdot f(x_{10}, y_{10}) = -1,54075 + 0.2 \cdot ((-1,54075)^2 - 2.8) = -1,62597;$$

$$y_{11} = y_{10} + h \cdot \frac{f(x_{10}, y_{10}) + f(x_{11}, \tilde{y}_{11})}{2} =$$

$$= -1,54075 + 0.2 \cdot \frac{((-1,54075)^2 - 2.8) + ((--1,62597)^2 - 3)}{2} = -1.61898.$$

Легко заметить, что данные результаты схожи с результатами, полученными с помощью явного метода Эйлера по схеме М1.

М4. Схема предиктор – корректор (Рунге – Кутта) второго порядка

Используя в (12.5) формулу средних, получим

$$\frac{\vec{y}^{k+1} - \vec{y}^k}{h} = \vec{f}(x_{k+1/2},\ \vec{y}^{k+1/2}). \qquad (12.11)$$

Уравнение разрешено явно относительно $\vec{y}^{k+1}$, однако в правой части присутствует неизвестное значение $\vec{y}^{k+1/2}$ в середине отрезка $[x_k,\ x_{k+1}]$. Для решения этого уравнения существует следующий способ. Вначале по явной схеме (12.8) рассчитывают $\vec{y}^{k+1/2}$ (предиктор):

$$\vec{y}^{k+1/2} = \vec{y}^{k} + \frac{h}{2}\vec{f}(x_k,\ \vec{y}^{k}).$$

После этого рассчитывают $\vec{y}^{k+1}$ по (12.11) (корректор). В результате схема оказывается явной и имеет второй порядок. Заметим, что схема получается из схемы М3, если в ней выполнять только две итерации (s = 1, 2).

Алгоритм представлен на рис. 12.3.

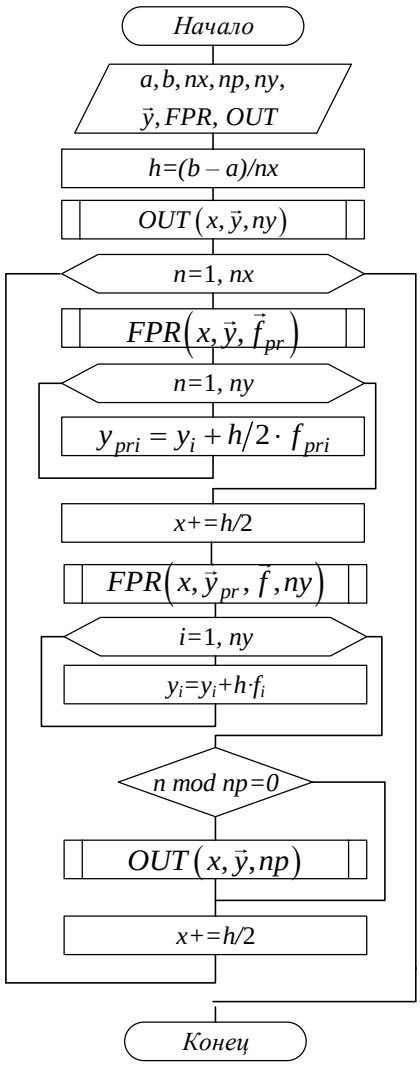


Рис. 12.3

M5. Схема Рунге – Кутта четвертого порядка

Используя в (12.5) формулу Симпсона, получим

$$\frac{\vec{y}^{k+1} - \vec{y}^{k}}{h} = \frac{1}{6}\,[\,\vec{f}(x_k,\ \vec{y}^{k}) + 4\vec{f}(x_{k+1/2},\ \vec{y}^{k+1/2}) + \vec{f}(x_{k+1},\ \vec{y}^{k+1})]. \qquad (12.12)$$

Т. к. формула Симпсона имеет пятый порядок точности, то погрешность аппроксимации данной схемы имеет четвертый порядок.

Можно по-разному реализовать расчет неявного по $\vec{y}^{k+1}$ уравнения (12.12), однако наибольшее распространение получил следующий способ. Делают предиктор вида

$$\vec{k}_0 = f(x_i, \vec{y});$$
$$\vec{k}_1 = \vec{y}^{k+1/2,1} = \vec{y}^k + h/2 \cdot \vec{f}(x_i, \vec{y});$$
$$\vec{k}_2 = \vec{y}^{k+1/2,2} = \vec{y}^k + h/2 \cdot \vec{f}(x_{i+1/2}, \vec{y}^{k+1/2,1});$$
$$\vec{k}_3 = \vec{y}^{k+1,1} = \vec{y}^k + h \cdot \vec{f}(x_{i+1/2}, \vec{y}^{k+1/2,2}),$$

затем корректор по формуле

$$\vec{y}^{k+1} = \vec{y}^k + \frac{h}{6}[\, f(x_k,\ \vec{y}^k) + 2\vec{f}(x_{k+1/2,}\ \ \vec{y}^{k+1/2,1}) +$$
$$+2\vec{f}(x_{k+1/2,}\ \ \vec{y}^{k+1/2,2}) + \vec{f}(x_{k+1}, \vec{y}^{k+1,1})].$$

Алгоритм метода представлен на рис. 12.4.

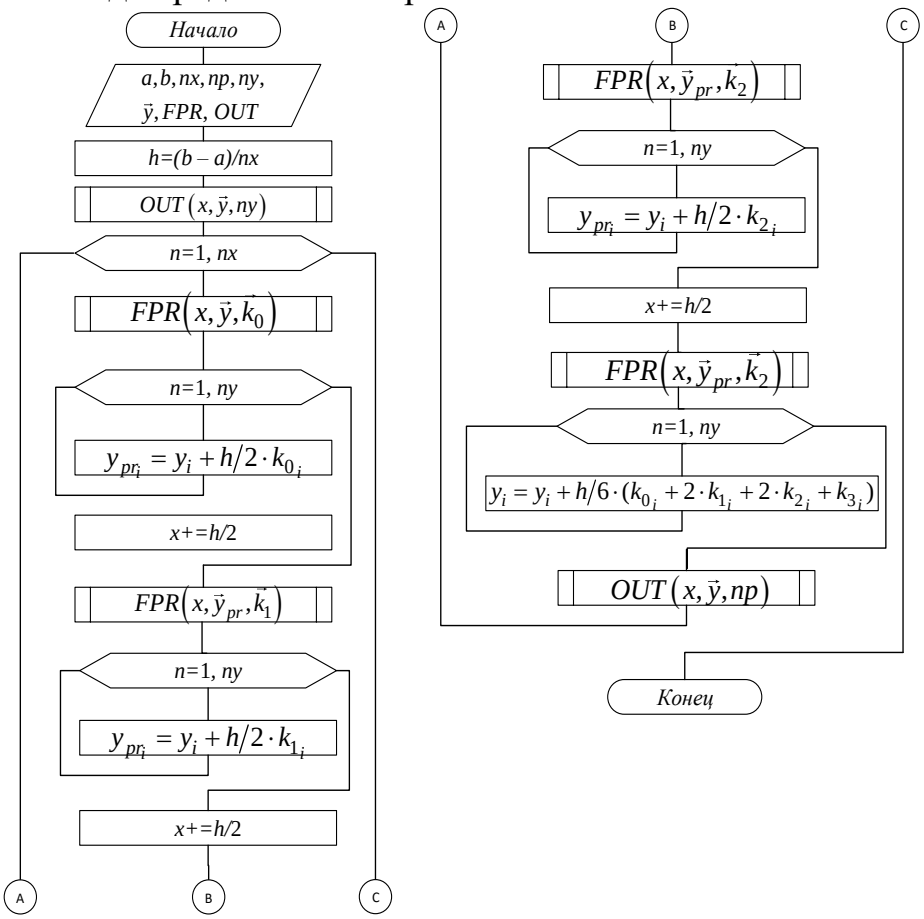


Рис. 12.4

12.4. Многошаговые схемы Адамса

При построении всех предыдущих схем для вычисления интеграла в правой части (12.4) использовались лишь точки в диапазоне одного шага $[x_k,\ x_{k+1}]$. Поэтому при реализации таких схем для вычисления следующего значения $\vec{y}^{k+1}$

требуется знать только одно предыдущее значение $\vec{y}^k$, т. е. рекуррентная последовательность получается первого порядка. Такие схемы называют *одношаговыми*. Мы, однако, видели, что для повышения точности при переходе от x_k к x_{k+1} приходилось использовать и значения функции F внутри интервала $[x_{k+1/2},\ \vec{y}^{k+1/2}]$. Схемы, в которых это используется (М4, М5, ...), называют *схемами с дробными шагами*. В этих схемах повышение точности достигается за счет дополнительных затрат на вычисление функции $F(x)$ в промежуточных точках интервала $[x_k,\ x_{k+1}]$.

Идея методов Адамса заключается в том, чтобы для повышения точности использовать уже вычисленные на предыдущих шагах значения $\vec{y}^k,\ \vec{y}^{k-1},\ \vec{y}^{k-2}, \ldots$.

Заменим в (12.4) $F(x)$ интерполяционным многочленом Ньютона вида

$$F(x) \approx F(x_k) + (x - x_k)\frac{F(x_k) - F(x_{k-1})}{h} +$$

$$+(x - x_k)(x - x_{k-1})\frac{F(x_k) - 2F(x_{k-1}) + F(x_{k-2})}{2h^2} + \ \ldots\ .$$

После интегрирования на интервале $[x_k,\ x_{k+1}]$ получим ***явную экстраполяционную схему*** Адамса (***экстраполяцией*** называется получение значений интерполяционного многочлена в точках x, выходящих за крайние узлы сетки). В нашем случае интегрирование производится на интервале $[x_k,\ x_{k+1}]$, а полином строится по узлам $x_k,\ x_{k-1},\ x_{k-2}$.

Порядок аппроксимации схемы в этом случае определяется количеством использованных при построении полинома узлов (например если используются $x_k,\ x_{k-1},$ то схема второго порядка).

Если в (12.4) $F(x)$ заменить многочленом Ньютона вида

$$F(x) \approx F(x_{k+1}) + (x - x_{k+1})\frac{F(x_{k+1}) - F(x_k)}{h} +$$

$$+(x - x_{k+1})(x - x_k)\frac{F(x_{k+1}) - 2F(x_k) + F(x_{k-1})}{2h^2} + \ldots\ ,$$

то после интегрирования получим ***неявную интерполяционную схему Адамса***. Заметим, что неявная интерполяционная схема второго порядка совпадает со схемой М3.

М6. Явная экстраполяционная схема Адамса второго порядка

$$\frac{\vec{y}^{k+1} - \vec{y}^k}{h} = 1.5 \cdot \vec{f}(x_k,\ \vec{y}^k) - 0.5 \cdot \vec{f}(x_{k-1},\ \vec{y}^{k-1})\ . \qquad (12.13)$$

Схема двухшаговая, поэтому для начала расчетов необходимо найти $\vec{y}^1$ по методу М4, после чего $\vec{y}^2,\ \vec{y}^3,\ \ldots$ вычислять по (12.13).

M7. Явная экстраполяционная схема Адамса третьего порядка

$$\frac{\vec{y}^{k+1}-\vec{y}^{k}}{h}=\frac{23}{12}\vec{f}(x_k,\ \vec{y}^{k})-\frac{16}{12}\vec{f}(x_{k-1},\ \vec{y}^{k-1})+\frac{5}{12}\vec{f}(x_{k-2},\vec{y}^{k-2})\,. \quad (12.14)$$

Схема трехшаговая, поэтому для начала расчетов необходимо найти $\vec{y}^1,\ \vec{y}^2$ по методу М5, после чего $\vec{y}^3,\ \vec{y}^4$, ... вычислять по (12.14).

M8. Неявная схема Адамса третьего порядка

$$\frac{\vec{y}^{k+1}-\vec{y}^{k}}{h}=\frac{5}{12}\vec{f}(x_{k+1},\ \vec{y}^{k+1})+\frac{8}{12}\vec{f}(x_k,\ \vec{y}^{k})-\frac{1}{12}\vec{f}(x_{k-1},\ \vec{y}^{k-1})\,. \quad (12.15)$$

Т. к.схема двухшаговая, то для начала расчетов необходимо найти $\vec{y}^1$ по методу М5, после чего $\vec{y}^2,\ \vec{y}^3$, ... вычислять по (12.15).

Для нахождения $\vec{y}^{k+1}$ требуется использовать метод простой итерации:

$$\vec{y}^{k+1,s}=\vec{y}^{k}+h\left[\frac{5}{12}\vec{f}(x_{k+1},\ \vec{y}^{k+1,s-1})+\frac{8}{12}\vec{f}(x_k,\ \vec{y}^{k})-\frac{1}{12}\vec{f}(x_{k-1},\ \vec{y}^{k-1})\right].$$

Значение $\vec{y}^{k+1,0}$ следует рассчитать по формуле (12.13):

$$\vec{y}^{k+1,0}=\vec{y}^{k}+h\cdot\left[1.5\cdot\vec{f}(x_k,\ \vec{y}^{k})-0.5\cdot\vec{f}(x_{k-1},\ \vec{y}^{k-1})\right].$$

Чаще всего бывает достаточно одной итерации. Если при этом разность $\left|\vec{y}^{k+1,0}-\vec{y}^{k+1,1}\right|$ оказывается большой, то следует уменьшить h. Схема алгоритма представлена на рис. 12.5.

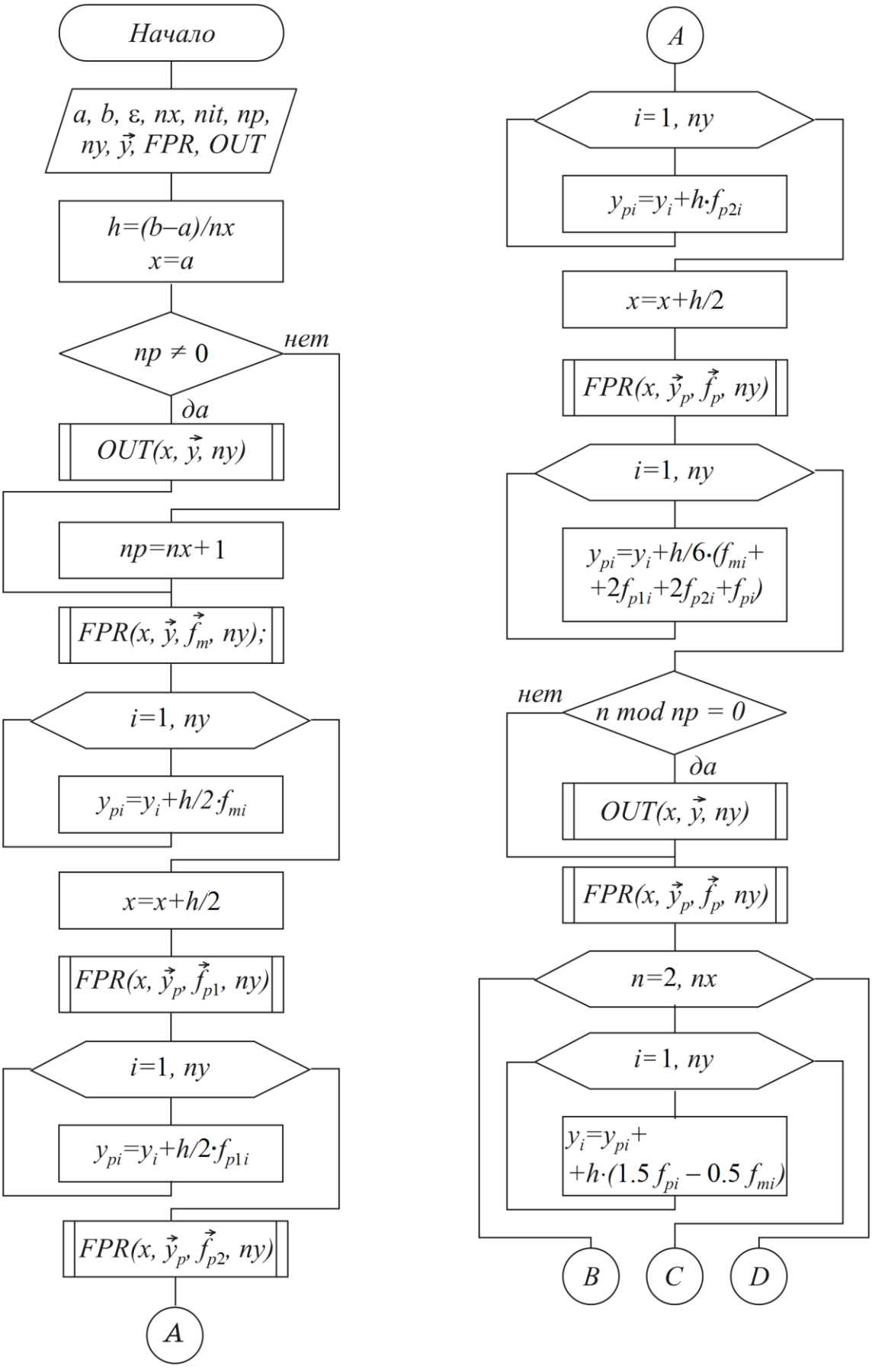


Рис. 12.5. 1-й фрагмент (окончание см. на с. 109)

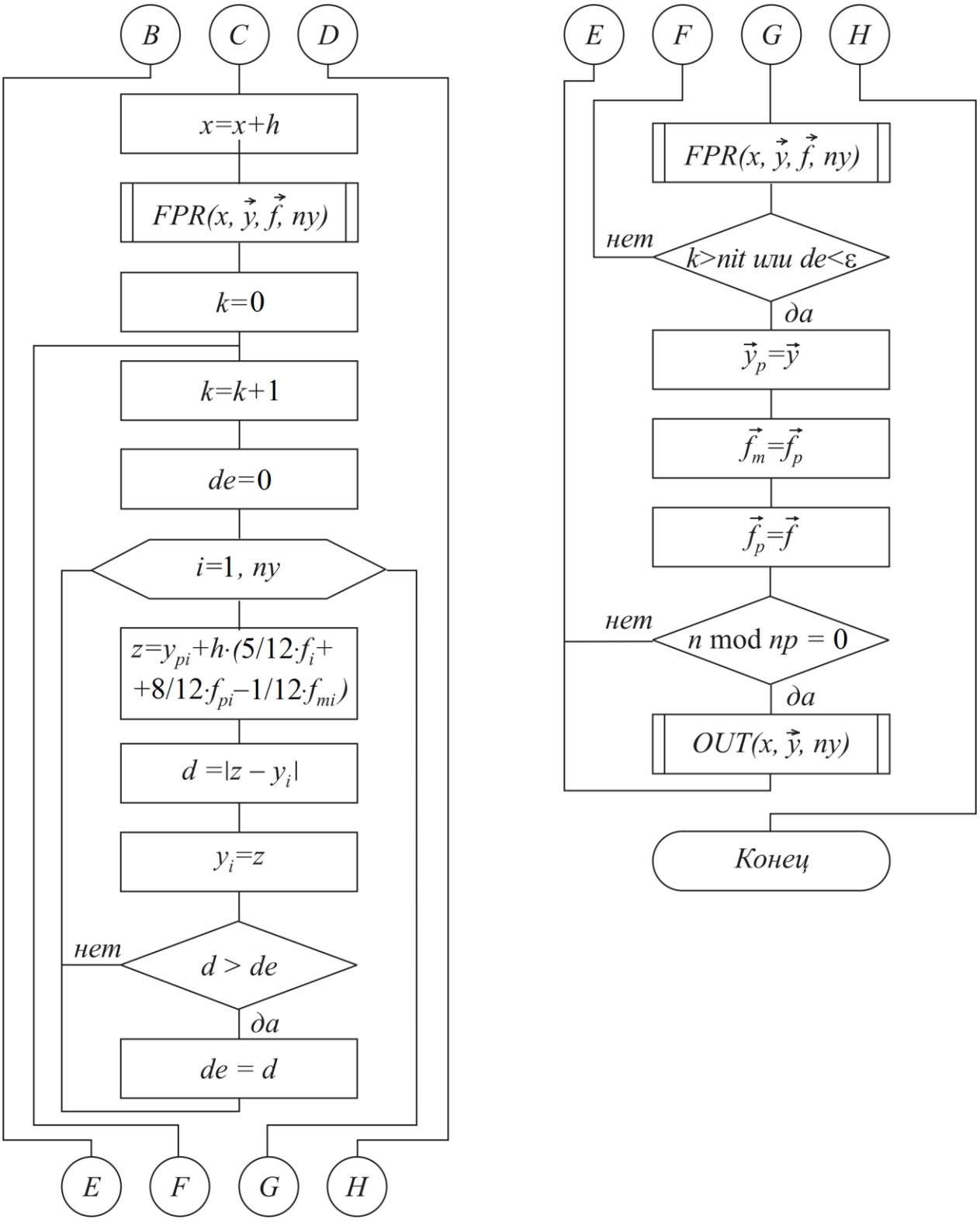


Рис. 12.5 окончание

(начало см. на с. 108)

12.5. Особенности программирования алгоритмов

Для решения задачи Коши составляется стандартная подпрограмма, которая помещается в библиотеку стандартных программ. Передача всех необходимых данных из основной программы пользователя в подпрограмму организуется через список формальных параметров, который включает:

a, b – начало и конец области интегрирования;

nx – количество шагов сетки;

np – параметр, указывающий, через какое количество шагов организовывать вывод данных;

ny – количество уравнений;

$\vec{y}$ – массив, в который сначала при обращении к подпрограмме помещается вектор начальных значений $\vec{y}(0) = \vec{u}^0$, а по окончании работы в нем помещается решение на конце интервала $\vec{y}(b)$.

Для итерационных методов дополнительно вводятся:

nit – максимально допустимое количество итераций;

ε – точность (погрешность, до которой выполняются итерации).

В подпрограмме $FPR(x, \vec{y}, \vec{F}, ny)$ для заданных x и $\vec{y}$ вычисляется вектор $\vec{f} = \vec{f}(x, \vec{y})$, в подпрограмме $OUT(x, \vec{y}, ny)$ осуществляется вывод данных.

12.6. Индивидуальные задания

Составить отдельную подпрограмму, оформленную в виде модуля, для решения задачи Коши в соответствии со схемой согласно варианту (табл. 12.1).

С помощью этой подпрограммы решить задачу для системы двух уравнений в соответствии с вариантом (см. табл. 12.1):

$$\frac{du_1}{dx} = f_1(x, u_1, u_2); \ \frac{du_2}{dx} = f_2(x, u_1, u_2);$$

$$a \le x \le b;$$

$$u_1(a) = u_1^0; u_2(a) = u_2^0.$$

Точное решение этой задачи при $u_1^0 = 2a, u_2^0 = e^a$ одинаково для всех вариантов и имеет вид $u_1 = 2x, u_2 = e^x$.

Функция FPR для варианта 15 имеет вид

```
void FPR(double x, double *y,double *f)
{
   f[0]=y[0]*exp(x)/(x*y[1]);
   f[1]=2*x/y[0]+y[1]-1;}
```

Вариант процедуры OUT, обеспечивающей вывод таблицы значений приближенного и точного решения и погрешностей (d_1, d_2), имеет вид

```
void OUT (double x, double *y)
{
    cout << "x = " << setw(5) << x << setw(10) << " u1 - y1 = " <<  2*x –
y[0] << setw(10) << " u2 – y2 = " << exp(x) – y[1] << endl;}
```

Начальные условия следующие: y[1]:=2*a; y[2]:=exp(a).

Расчеты произвести для последовательности сгущающихся сеток: $h = h_1 = (b-a)/10$, $h = h_1/2$, $h = h_2/4$, ..., – и, сравнивая полученное решение с точным решением, добиваться того, чтобы погрешность на втором конце *(x = b)* была не больше 0.0001.

Построить графики полученных решений для $h = h_1$, сравнить их с точным решением.

Таблица 12.1

Номер варианта.	$f_1(x, u_1, u_2)$	$f_2(x, u_1, u_2)$	*[a, b]*	$U_1(a)$	$u_2(a)$	*Метод*
1	$u_1/x - u_2/e^x + 1$	$u_1/(2x) + u_2 - 1$	[1, 3]	2	e^1	M1
2	$u_1 + u_2 - 2x - e^x + 2$	$u_1 + u_2 - 2x$	[1, 2]	2	e^1	M2
3	$u_1 + 2u_2/e^x - 2x$	$u_1/(2x) - e^x/u_2 + u_2$	[2, 3]	4	e^2	M3
4	$(u_1 \cdot e^x)/(x \cdot u_2)$	$2u_1 + u_2 - 4x$	[1, 4]	2	e^1	M4
5	$2u_1 + (u_2 + e^x)/e^x - 4x$	$2x \cdot u_2/u_1$	[2, 4]	4	e^2	M5
6	$u_1 \cdot u_2/(e^x \cdot x)$	$2x/u_1 + 2u_2 - e^x - 1$	[1, 3]	2	e^1	M6
7	$u_1/2x + u_2/e^x + 1$	$u_1 \cdot u_2/2x$	[2, 3]	4	e^2	M7
8	$u_1/x + u_2 - e^x$	$2x/u_1 + u_2^2/e^x - 1$	[1, 4]	2	e^1	M8
9	$u_1 + 2e^x/u_2 - 2x$	$u_1^2/x^2 + u_2 - 4$	[1, 2]	2	e^1	M7
10	$4x/u_1 - u_2 + e^x$	$u_1/2x - u_2/e^x + e^x$	[2, 4]	4	e^2	M6
11	$2x/u_1 + u_2/e^x$	$u_1 \cdot e^{2x}/(u_2 \cdot 2x)$	[3, 4]	6	e^3	M5
12	$u_1 \cdot u_2/(2e^x) - x + 2$	$u_1 + 2u_2 - 2x - e^x$	[1, 3]	2	e^1	M4
13	$u_1^2 + u_2 - 4x^2 - e^x + 2$	$u_1 \cdot e^x/u_2 + u_2 - 2x$	[1, 2]	2	e^1	M3
14	$u_1^2/2x^2 - u_2 + e^x$	$u_1 \cdot e^x/2x + u_2/e^x - 1$	[2, 4]	4	e^2	M2
15	$u_1 \cdot e^x/(x \cdot u_2)$	$2x/u_1 + u_2 - 1$	[3, 4]	6	e^3	M1

12.7. Контрольные вопросы

1. Как формулируется задача Коши для системы из n уравнений?

2. В чем суть метода сеток?

3. Что такое конечно-разностная схема, погрешность аппроксимации, устойчивость?

4. Сформулируйте содержание основной теоремы метода сеток.

5. Назовите известные вам схемы решения дифференциального уравнения.

6. В чем отличие методов Адамса от методов Рунге – Кутты?

Литература

1. Основы алгоритмизации и программирования (язык С/С++). Лабораторный практикум. В 2 ч. Ч. 1 : учеб.-метод. пособие / С. А. Беспалов [и др.] – Минск : БГУИР, 2017.

2. Основы алгоритмизации и программирования. Язык Си : учеб. пособие / М. П. Батура [и др.] – Минск : БГУИР, 2007.

3. Синицын, А. К. Программирование алгоритмов в среде Builder C++: в 2 ч. / А. К. Синицын. – Минск : БГУИР, 2004 – 2005. – 2 ч.

4. Вирт, Н. Алгоритмы и структуры данных / Н. Вирт. – СПб. : Невский диалект, 2001.

5. Архангельский, А. Я. Программирование в C++ Builder 6 / А. Я. Архангельский. – М. : БИНОМ, 2002.

6. Демидович, Е. М. Основы алгоритмизации и программирования. Язык СИ / Е. М. Демидович. – СПб : БХВ – Петербург, 2006.

7. Кнут, Д. Искусство программирования. Основные алгоритмы : в 3 т. / Д. Кнут – М. : Издательский дом «Вильямс», 2004. – 3 т.

8. Синицын, А. К. Алгоритмы вычислительной математики : учеб.-метод. пособие / А. К. Синицын, А. А. Навроцкий. – Минск : БГУИР, 2007.

9. Бахвалов, Н. С. Численные методы в задачах и упражнениях / Н. С. Бахвалов, А. В. Лапин, Е. В. Чижонков. – М. : Высш. шк., 2000.

10. Егоров, А. А. Вычислительные алгоритмы линейной алгебры : учеб. пособие / А. А. Егоров. – Минск : БГУ, 2005.

11. Соловьев, В. П. Основы численных методов : учеб.- метод. пособие / В. П. Соловьев, Т. М. Кривоносова, В. Л. Смирнов. – Минск : БГУИР, 2011.

Св. план 2017, поз. 18

Учебное издание

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ (ЯЗЫК С/С++). ЛАБОРАТОРНЫЙ ПРАКТИКУМ

В двух частях

Часть 2

Беспалов Сергей Алексеевич
Гуревич Александр Владимирович
Кривоносова Татьяна Михайловна и др.

Редактор *М.А. Зайцева*
Корректор *Е.Н. Батурчик*
Компьютерная правка, оригинал-макет

Подписано в печать. Формат 60x84 1/16. Бумага оффсетная. Гарнитура «Times».
Отпечатано на ризографе. Усл. печ. л. . Уч.-изд. л. . Тираж 200 экз. Заказ 340.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
ЛП №02330/264 от 14.04.2014.
220013, Минск, П. Бровки, 6