А.П. Полищук, С.А. Семериков

МЕТОДЫ ВЫЧИСЛЕНИЙ

в классах языка С++

```
//Аппроксимация по методу наименьших квадратов
template <class YourOwnFloatType>
polynom<YourOwnFloatType>
Approximate<YourOwnFloatType>::MNK(int rng)
{
  matrix<YourOwnFloatType> mtr(x.getm(),rng+1),
/*Прямоугольная матрица измерений */
         f(x.getm(),1); //Столбец свободных членов

  //Формирование матрицы измерений
  for(int i=0;i<mtr.getm();i++)
  {
    f[i][0]=y[i];
    for(int j=0;j<mtr.getn();j++)
      mtr[i][j]=pow(x[i],j);
  }
  //Реализуем формулу a=((mtr*mtrT)^-1)*(mtrT*f)
  matrix<YourOwnFloatType> sol=
               SLAE_Orto((~mtr)*mtr,(~mtr)*f);
  polynom<YourOwnFloatType> a(sol.getm());
  for(long i=0;i<sol.getm();i++)
    a[i]=sol[i][0];
  return a;/* Возвращаем полином - решение задачи */
}
```

А.П. Полищук, С.А. Семериков

МЕТОДЫ ВЫЧИСЛЕНИЙ
в классах языка C++

Учебное пособие для студентов вузов

Кривой Рог
Издательский отдел КГПИ
1999

Утверждено Ученым советом
Криворожского государственного педагогического
института (протокол №6 от 10.12.98)

УДК 518.5+378.147

Полищук А.П., Семериков С.А.

Методы вычислений в классах языка С++: Учебное пособие. – Кривой Рог: Издательский отдел КГПИ, 1999. – 350 с., ил.

Учебное пособие содержит классический вузовский набор алгоритмов вычислительной математики, ориентированных на программную реализацию средствами языка С++. Использование объектно-ориентированной методологии значительно облегчает усвоение курса за счет расширения языка новыми типами данных (векторы, полиномы, матрицы и др.), приближающих программную запись к естественной математической.

Для студентов высших учебных заведений, аспирантов, научных и инженерно-технических работников.

ISBN 5-7763-2587-0

Рецензенты:

А.Н. Марюта – д-р техн. наук, проф., заведующий кафедрой экономической кибернетики Днепропетровского государственного университета
А.И. Олейников – д-р. физ.-мат. наук, проф., заведующий кафедрой математики Криворожского государственного педагогического института

© А.П. Полищук, С.А. Семериков, 1999

Оглавление

0. Введение 8
0.1. Приближенные вычисления 8
0.2. Численные методы и программирование 9
0.3. Особенности машинных вычислений 14
0.4. Структура учебного пособия 17
1. Специальные классы математических объектов и операции над ними 20
1.1. Комплексные числа 20
1.1.1. Основные понятия 20
1.1.2. Операции над комплексными числами 21
1.1.2.1. Операции сравнения 21
1.1.2.2. Алгебраические операции 21
1.1.2.2.1. Сложение и вычитание 21
1.1.2.2.2. Умножение 21
1.1.2.2.3. Деление 21
1.1.2.2.4. Возведение в степень (формула Муавра) 22
1.1.3. Определение программного класса комплексных чисел 22
1.2. Векторы 26
1.2.1. Основные понятия 26
1.2.2. Определение программного класса многомерных векторов 29
1.3. Полиномы 50
1.3.1. Общие сведения 50
1.3.2. Операции над полиномами 52
1.3.3. Вычисление значений полиномов 53
1.3.4. Вычисление корней полиномов 54
1.3.5. Определение программного класса полиномов 56
2. Матрицы и задачи линейной алгебры 88
2.1. Общие сведения о матрицах и матричных операциях 88
2.2. Методы решения основных задач линейной алгебры 92
2.2.1. Методы решения систем линейных алгебраических уравнений (СЛАУ) 93
2.2.1.1. Метод Гаусса для решения СЛАУ, вычисления определителей и обращения матриц 94

2.2.1.2. Предварительная факторизация матриц (разложение в произведение двух матриц) в задачах решения СЛАУ 96
2.2.1.2.1. Факторизация матриц по методу Холецкого .. 96
2.2.1.3. Метод ортогонализации для решения СЛАУ 98
2.2.1.4. Итерационные методы решения СЛАУ 100
2.2.1.4.1. Проблема сходимости итерационных методов 100
2.2.2. Методы вычисления собственных значений матриц 102
2.2.2.1. Метод неопределенных коэффициентов 103
2.2.2.2. Метод Данилевского 103
2.2.2.3. QR-алгоритм для несимметрических матриц 105
2.2.2.4. Метод Леверрье-Фаддеева 107
2.2.2.5. Итерационный степенной метод 108
2.2.2.6. Метод Крылова 109
2.2.3. Метод наименьших квадратов (МНК) 110
2.3. Программная реализация матричного класса 115
2.3.1. Общее описание структуры матричного класса........ 115
2.3.2. Интерфейсный файл реализации матричного класса matrix.h 137
2.4. Программная реализация методов вычисления собственных значений и собственных векторов матриц 161
2.4.1. Метод неопределенных коэффициентов........ 161
2.4.2. Программная реализация метода Крылова вычисления собственных значений 164
2.4.3. Метод Леверрье-Фаддеева вычисления коэффициентов характеристического полинома 166
2.5. Элементы линейного программирования........ 167
2.5.1. Общая постановка задачи........ 167
2.5.2. Примеры задач линейного программирования 168
2.5.2.1. Задача о пищевом рационе 168
2.5.2.2. Задача о распределении ресурсов 168
2.5.2.3. Задача планирования перевозок (транспортная задача)........ 169
2.5.3. Симплекс-метод решения задачи линейного программирования 170
2.5.3.1. Приведение системы к стандартной, удобной для преобразований форме........ 171

2.5.3.2. Алгоритм замены базисных переменных 171
2.5.3.3. Алгоритм поиска опорного решения ОЗЛП 172
2.5.3.3.1. Алгоритм выбора разрешающего элемента для приближения к опорному решению 172
2.5.3.4. Алгоритм поиска оптимального решения 173
2.5.4. Транспортная задача линейного программирования 173
2.5.4.1. Общие сведения 173
2.5.4.2. Формирование опорного плана 176
2.5.4.3. Циклические переносы перевозок для улучшения плана 176
2.5.4.4. Метод потенциалов 177
2.5.5. Транспортная задача при небалансе запасов и заявок 179
2.5.6. Транспортная задача с временным критерием 180
2.6. Программная реализация задач линейного программирования 181
2.6.1. Симплекс-метод решения ОЗЛП 181
2.6.2. Программная реализация метода решения транспортной задачи 189
3. Аналитическое приближение функций, заданных таблично 198
3.1. Общая постановка задачи 198
3.2. Общая методика решения задач аппроксимации............ 201
3.2.1. Алгебраическая интерполяция.................................. 203
3.2.1.1. Классический интерполяционный полином 203
3.2.1.2. Метод интерполяции Лагранжа 203
3.2.1.3. Интерполяционный полином Ньютона 205
3.2.1.4. Интерполяция сплайнами 208
3.2.1.5. Аппроксимация функций по методу наименьших квадратов 212
3.2.2. Программная реализация класса аппроксимирующих функций... 213
4. Численное интегрирование и дифференцирование табличных функций 220
4.1. Численное интегрирование 220
4.1.1. Вычисление определенных интегралов 220
4.1.1.1. Классификация методов...................................... 220
4.1.2. Программная реализация методов численного интегрирования 225

4.2.Численное дифференцирование 227
5. Введение в численные методы решения дифференциальных уравнений 231
5.1. Обыкновенные дифференциальные уравнения (общие сведения)...231
5.2. Процессы как объект исследования и управления 232
5.3. Операционное исчисление и его применение к исследованию динамики линейных систем............................241
5.3.1. Общие сведения...241
5.3.1.1. Правила операционного исчисления 243
5.3.2. Решение линейных уравнений с постоянными коэффициентами 245
5.3.2.1. Передаточные функции линейных динамических систем 248
5.3.2.2. Частотные характеристики динамических систем 251
5.3.2.3. Фазовые портреты динамических систем 252
5.3.2.4. Ограничения области применения символического метода 252
5.3.3. Программная реализация класса символического метода...253
5.4. Конечно-разностные методы решения задачи Коши для линейных и нелинейных ОДУ 283
5.4.1. Одношаговые методы 284
5.4.1.1. Метод Эйлера...285
5.4.1.2. Методы Рунге-Кутта 2-го порядка.....................286
5.4.1.3. Метод Рунге-Кутта 4-го порядка 288
5.4.2. Многошаговые методы (методы Адамса).................289
5.4.3. Проблема устойчивости...292
5.4.4. Программная реализация численных методов решения задачи Коши...294
5.5. Двухточечные краевые задачи...300
5.5.1. Метод конечных разностей для линейных краевых (граничных) задач 300
5.5.2. Метод стрельбы для граничных задач 303
5.5.3. Программная реализация класса граничных задач...306
6. Численные методы решения систем нелинейных уравнений (СНУ) и поиска экстремумов функций 311

6.1. Общая постановка задачи 311
6.2. Решение нелинейных уравнений 312
6.2.1. Функция одной переменной при отсутствии помех . 312
6.2.1.1. Методы последовательного сокращения интервала неопределенности 313
6.2.1.2. Рекуррентные методы уточнения текущей оценки значения корня 316
6.2.2. Уточнение корня функции одной переменной в условиях помех 317
6.2.2.1. Методы стохастической аппроксимации 317
6.3. Методы поиска экстремума функций 318
6.3.1. Унимодальные функции одного аргумента при отсутствии помех 318
6.3.1.1. Метод дихотомии 318
6.3.1.2. Метод Фибоначчи 318
6.3.1.3. Метод золотого сечения 319
6.3.2. Многомерный поиск экстремума 320
6.3.2.1. Метод координатного спуска 320
6.3.2.2. Метод градиента (наискорейшего спуска или крутого восхождения или метод Бокса-Уилсона) 321
6.3.2.3. Последовательный симплексный поиск (ПСМ) субоптимальной области в многомерном пространстве. 323
6.3.2.3.1. Вводные замечания 323
6.3.2.3.2. Алгоритм последовательного симплексного метода 325
6.3.2.3.3. Подготовительные операции 325
6.3.2.3.4. Алгоритм поиска 327
6.3.2.3.5. Преимущества метода 327
6.3.2.3.6. Недостатки метода 328
6.4. Программная реализация класса подпрограмм для поиска экстремума унимодальных в заданном интервале функций одной переменной 328
6.5. Программная реализация класса подпрограмм для многомерного поиска экстремума унимодальных функций . 335
Приложение. Как переносить данные в Excel и отображать графики зависимостей 343
Литература 347

0. Введение

0.1. Приближенные вычисления

Методы вычислений делят на *аналитические* и алгоритмические (*приближенные*). Аналитические методы вычислений используются в тех очень редких на практике случаях, когда входные и выходные данные удается связать между собой точным аналитическим выражением (формулами). В подавляющем большинстве практических задач сделать это не удается – либо из-за отсутствия математической модели, связывающей требуемые результаты с исходными данными, либо из-за чрезмерной сложности этой модели, либо сами исходные данные представлены в форме, исключающей возможность прямых аналитических вычислений.

Приведем простой пример: в распоряжении работника, подсчитывающего размер оплаты предприятием за израсходованный в производстве природный газ, имеется диаграммная лента с записью расхода газа в кубометрах в час за расчетный период. Если бы нарисованная самопишущим прибором на ленте кривая была представлена аналитической функцией времени, осталось бы проинтегрировать ее на заданном интервале и вычислить полученное аналитическое выражение. Но в силу случайного характера изменений расхода газа во времени представить имеющуюся зависимость расхода от времени в аналитической форме невозможно и остается использовать алгоритмы приближенного интегрирования.

К такому же выводу придем каждый раз при попытке решения уравнений, содержащих тригонометрические функции вперемежку с алгебраическими – даже простейший случай уравнения с одним неизвестным

$$a \cdot x + b \cdot \cos(c \cdot x) = 0$$

не позволяет получить аналитические выражения для вычисления корней и вынуждает нас использовать приближенные численные методы.

0.2. Численные методы и программирование

Вычисления сопровождают те виды интеллектуальной деятельности людей, в которых необходима точная численная оценка результата, и математика – это не столько «упражнение для ума», как считал генералиссимус Суворов, сколько наука, обеспечивающая инженеров, ученых, финансистов, военных, фермеров, метеорологов и многих других методической основой для выполнения вычислений, по возможности – с помощью компьютерных программ с удобным интерфейсом пользователя.

Современная прикладная математика неразрывно связана со спецификой машинных вычислений и представляет собой сплав математики и компьютерного программирования. Разработчик технологии вычислений должен учитывать неизбежную при компьютерных вычислениях замену непрерывных задач их дискретными аналогами, и не только представить процесс обработки данных в виде вычислительной схемы с обоснованием существования и единственности решения, но и позаботиться об эффективных способах ввода и хранения исходных данных, их внутримашинной организации, об эффективности реализующей программы (времени счета и погрешностях вычислений), о возможностях наглядного (например, графического) представления полученных результатов, проанализировать возможность достижения заданной пользователем точности вычислений с учетом представления чисел в конкретной инструментальной среде. Иными словами, прикладной математик сегодня неизбежно и программист, владеющий современной технологией программирования. Результат разработки вычислительного алгоритма должен быть представлен в виде работоспособной, надежно функционирую-

щей, хорошо документированной, переносимой программы с удобным пользовательским интерфейсом и разнообразными возможностями анализа и интерпретации результатов.

Методика преподавания численных методов и соответствующие учебные пособия медленно эволюционируют от классических курсов, ориентированных на ручные вычисления, в направлении объединения вычислительных алгоритмов с их программной реализацией на ЭВМ.

Среди книг, объединяющих изложение вычислительных алгоритмов с их реализацией в компьютерных программах, следует отметить «Справочник алгоритмов на языке АЛГОЛ. Линейная алгебра» Уилкинсона и Райнша (М.: Машиностроение, 1976), охватывающий широкий класс алгоритмов решения систем линейных алгебраических уравнений, псевдообращения матриц, вычисления собственных значений и собственных векторов с исследованием области применения, подробным описанием вычислительных процедур и их АЛГОЛ-программ с оценкой точности, результатами тестирования, внимательным отношением к проблемам экономии памяти и скорости сходимости алгоритмов.

Язык ФОРТРАН представлен наиболее обширным списком пособий по численным методам:

- Плис А.И., Сливина Н.А. Лабораторный практикум по высшей математике. – М.: Высшая школа, 1983.
- Шуп Т. Решение инженерных задач на ЭВМ. Практическое руководство. – М.: Мир, 1982.
- Мак-Кракен Д., Дорн У. Численные методы и программирование на Фортране. – М.: Мир, 1977.
- Форсайт Дж., Малькольм М., Моулер К. Машинные методы математических вычислений. – М.: Мир, 1980.

Вычислительные алгоритмы с реализацией на языке БЕЙСИК приведены в «Справочнике по алгоритмам и программам на языке Бейсик для персональных ЭВМ»

В.П. Дьяконова (М.: Наука, 1987) и в книге Я.Т. Гринчишина, В.И. Ефимова, А.Н. Ломаковича «Алгоритмы и программы на Бейсике» (М.: Просвещение, 1988).

Изложение алгоритмов вычислений с программами на языке ПАСКАЛЬ приведены в книге А.Е. Мудрова «Численные методы для ПЭВМ на языках БЕЙСИК, ФОРТРАН и ПАСКАЛЬ» (Томск: Раско, 1991).

Все известные авторам пособия по численным методам и их программной реализации базируются на процедурной методологии программирования и либо на отмирающих языках программирования (АЛГОЛ, ФОРТРАН), либо на учебных языках типа БЕЙСИК, ПАСКАЛЬ. Создание и использование в учебном процессе непрофессиональных учебных языков следует отнести к «детскому возрасту» компьютерного программирования, когда эта работа представлялась непостижимо сложной и недоступной для понимания большинством пользователей. Сегодня мы уже знаем, что для создания эффективных компьютерных программ недостаточно владения лексикой алгоритмического языка – необходим еще минимум знаний по архитектуре аппаратного обеспечения используемого компьютера, возможностях операционной системы и взаимодействию с ней языкового транслятора; набор этих знаний вполне доступен для освоения пользователю средних способностей, а использование учебных суррогатов алгоритмических языков (особенно на национальным языковом базисе типа известного русскоязычного школьного учебного языка) приносит больше вреда, чем пользы. Да собственно и такой первоначально учебный язык как ПАСКАЛЬ, пытаясь в эволюции своего развития достичь возможностей простого, лаконичного профессионального языка Си, перестал быть учебным, хотя и не стал профессиональным по удобству и возможностям, сохранив, тем не менее «учебное» многословие и громоздкость конструкций.

Численные методы – основа инженерного и научного программирования, решающего задачи создания и исследования математических моделей процессов, протекающих в объектах различной природы: финансовых потоков и ресурсов в экономике страны или отдельного предприятия, воздушных потоков в атмосфере, изменения напряжений в строительных конструкциях под действием внешних сил, траекторий движения летательных аппаратов, движения электронов на атомных орбитах под воздействием внешних электромагнитных полей, взаимодействия популяций видов в растительном и животном мире и т. п. Математические модели различных процессов принято представлять в общем случае в виде систем интегро-дифференциальных уравнений с обыкновенными или частными производными; разработка методов и программ для решения таких уравнений представляет собой основной комплекс задач инженерного и научного программирования, а, следовательно, и численного анализа в целом.

При решении дифференциальных уравнений приходится осуществлять различные операции с такими математическими объектами, как матрицы (в общем случае с комплексными элементами), числовые или функциональные векторы, полиномы (с вещественными или комплексными коэффициентами).

В стройном здании математики более сложные математические объекты строятся из более простых. Так, комплексные числа представляются в виде пары вещественных координат вектора по вещественной и мнимой осям комплексной плоскости. Многомерный числовой вектор может быть представлен совокупностью его вещественных или комплексных координат по осям многомерной координатной системы (или как частный случай матрицы – одностолбцовой или однострочной), матрица может быть представлена как вектор векторов, полином известного порядка может быть задан как вектор его коэффициентов.

Для каждого класса математических объектов определен допустимый набор математических операций и способы их реализации; например, операции умножения определены для вещественных чисел, векторов и матриц, но имеют, естественно, различный смысл и алгоритмы реализации. Операция определения нулей специфична для полиномов, транспонирования и вычисления собственных значений и собственных векторов – для матриц, определения модуля – для векторов.

Представляется целесообразным построить курс вычислительных методов по принципу определения иерархии математических классов, объекты которых конструировались бы затем в программе путем объявления, то есть синтаксически так же, как и стандартные для используемого языка типы (целые, вещественные и пр.) с определением внутри класса всех необходимых для их использования в вычислениях операций. При этом под термином «операция» можно понимать как общепринятые для простых типов операции, например, арифметические, так и любые, базирующиеся на данном математическом классе вычисления, – например, решение системы линейных алгебраических уравнений или вычисление коэффициентов регрессии для заданной матрицы или вычисление корней полинома наряду с операциями полиномиальной арифметики – сложения, умножения, деления полиномов.

В больших алгоритмических языках (ADA, PL) предусматриваются сложные математические типы вроде матриц, но широко используемые малые языки высокого уровня (Си, Паскаль) не содержат этих возможностей и с точки зрения учебного курса по численным методам это хорошо, так как дает возможность глубоко изучить вычислительные алгоритмы, доводя их до программной реализации с сопутствующим вылавливанием алгоритмических ошибок.

Наиболее удобным инструментом для создания классов математических объектов является объектно-ориентирован-

ное программирование и его поддержка в языке C++. Этот язык дает возможность варьировать методы создания математических объектов путем определения в классе необходимого количества конструкторов, осуществить переопределение стандартных операций для вновь созданных классов, использовать мощный механизм одиночного и множественного наследования свойств базовых классов в производных классах, создавать параметризованные классы и функции с подстановкой типов параметров в процессе конструирования соответствующих объектов. Последовательное наращивание иерархии математических типов на базе уже созданных позволяет существенно снизить трудоемкость программирования за счет исключения повторяющихся последовательностей действий и избежать внесения в программы новых ошибок.

Нам неизвестны пособия с систематическим изложением методов вычислений на базе объектно-ориентированного подхода в программной реализации на языке C++. Отчасти это объясняется известным консерватизмом нашей системы образования, ориентированной традиционно на использование Паскаля, а отчасти тем, что специалисты по программированию достаточно высокого профессионального уровня редко работают в наших учебных заведениях. Но уже сейчас во многих средних и высших учебных заведениях осуществляется преподавание языка C и C++ и предлагаемая работа может, по нашему мнению, оказать положительное влияние на эффективность учебного процесса в области вычислительной математики.

0.3. Особенности машинных вычислений

По-видимому, основной особенностью машинных вычислений по сравнению с ручными является ограниченная длина разрядной сетки и представление вещественных чисел в формате с плавающей десятичной точкой (экспоненциальный формат с нормализованной мантиссой). Это не

позволяет осуществлять машинные операции вещественной арифметики так, как мы привыкли в чистой математике – численное представление вещественных значений требует, как правило, бесконечного количества цифр и не может быть осуществлено с абсолютной точностью. Проблема ошибок округления существует и в ручных вычислениях, но там она не так важна, потому что объем вычислений, которые могут быть выполнены вручную, невелик по сравнению с тем, что обычно выполняется на современных ЭВМ. Кроме того, при ручном счете эффекты округления непосредственно наблюдаемы и могут быть скорректированы своевременным изменением длины числа или другими мерами предохранения от чрезмерных погрешностей результата, которая может быть достаточно легко оценена.

В машинных вычислениях получение такой оценки затруднительно – ранее разработанные теории оценок ошибок округления, как правило, непригодны, поэтому обычно прибегают к статистическим оценкам, основанным на допущении о независимости ошибок округлений; опыт показывает, что на самом деле эти ошибки часто коррелированны, и получаемые результаты сомнительны.

К сожалению, нет удовлетворительных теорий анализа ошибок округлений в вычислительных алгоритмах, поэтому в практике вычислений используют несколько эвристических программных методов диагностики наличия ошибок вычислений и программной оценки величины ожидаемой ошибки – прежде чем реализовать метод вычислений в программе, неплохо бы иметь представление об ожидаемой точности. Самым простым из них является определение допустимого диапазона для значений каждой переменной в виде максимальной и минимальной границ, внутри которых оно должно находиться. При выполнении действий над переменными новый диапазон вычисляется на основании определенного с учетом подходящих округлений и на каждой стадии вычислений определяются надежные границы,

внутри которых должен размещаться верный ответ. Этот метод требует примерно удвоения объема вычислений и расходуемой памяти и позволяет получить только оценку попадания результата в область допустимых значений, но не позволяет оценить величину ошибки.

Другим распространенным методом является просчет задачи дважды – с одинарной и с двойной точностью представления чисел – предполагается, что верными можно считать совпадающие разряды в двух ответах.

Большие потери точности, как правило, происходят при вычитании двух близких по значению чисел – например, отличающихся только в последнем разряде; в начале числового результата образуется последовательность нулей, удаляемая при нормализации в операциях с плавающей точкой. Результат вычитания будет иметь значительно меньше значащих цифр, например одну, даже при отсутствии ошибок округления и использование результата вычитания в последующих вычислениях может привести к тому, что окончательный результат будет иметь только один верный знак. Если это возможно, необходимо исключить потерю знаков изменением последовательности вычислений или использовать метод так называемого счета со значащими разрядами. Он состоит в том, что нормализующий сдвиг мантиссы блокируется с сохранением ведущих нулей для сохранения количества значащих разрядов. Можно фактически осуществить нормализующий сдвиг, но с предварительным запоминанием числа ведущих нулей.

Результирующая погрешность вычислений, помимо распространения ошибок округлений при выполнении операций, обусловлена также влиянием других источников:

- ошибки дискретизации непрерывных зависимостей:
- погрешности в задании исходных данных;
- погрешности моделирования зависимости между исходными данными и вычисляемыми переменными;

- методические погрешности, определяемые несовершенством вычислительных алгоритмов;

Совершенно очевидно, что бессмысленно выдвигать высокие требования к точности вычислений при наличии больших ошибок в задании исходных данных или грубом моделировании исследуемых зависимостей. Мы постарались рассеять возможную избыточную наивную доверчивость к результатам машинных вычислений, но не можем привести библиографию источников с описанием надежных методов оценок погрешностей как до, так и после реализации вычислений. Но отношение к результатам в сложных программах с сотнями тысяч повторяющихся арифметических операций должно быть тем более скептическим, чем больше расхождения в этих результатах при выполнении программы с различными длинами мантисс в представлении чисел с плавающей точкой.

0.4. Структура учебного пособия

Настоящее учебное пособие рассчитано на сравнительно небольшой двухсеместровый курс численных методов (2 часа в неделю лекций + 2 часа лабораторных работ в компьютерном классе). Изложение курса предполагает владение основами объектно-ориентированного программирования на языке С++.

В соответствии с уже изложенной концепцией объектно-ориентированной программной реализации многие методы вычислений инкапсулируются в классы математических объектов, с которыми они работают; например, метод наименьших квадратов и метод решения систем линейных алгебраических уравнений будут размещены в классе матриц, а полиномиальная арифметика и методы вычисления полиномиальных нулей – в классе полиномов.

Главы 1 и 2 посвящены рассмотрению специальных математических типов (и операций над ними) и определению соответствующих им классов в терминах языка С++. Вна-

чале в качестве иллюстрации рассматривается необходимый при изучении последующего материала (например, методов вычисления корней полиномов при наличии среди них комплексных) предположительно знакомый слушателю и реализованный в библиотеке С++ класс комплексных чисел; его программная реализация взята прямо из среды разработки Borland C++ и по возможности откомментирована – этот материал служит своеобразным образцом в реализации других рассмотренных в этих главах математических классов – векторов, полиномов, матриц. Изучение матричного класса сопровождается изложением методов решения основных задач линейной алгебры – систем линейных уравнений, вычисления собственных значений и векторов матриц.

Глава 3 содержит изложение методов полиномиальной и экспоненциальной аппроксимации функций и их программную реализацию, также инкапсулированную в виде функций-членов специального класса.

Глава 4 является естественным прикладным продолжением предыдущей и содержит методы численного интегрирования и дифференцирования функций с использованием рассмотренных методов приближения функций.

Глава 5 посвящена методам решения обыкновенных дифференциальных уравнений; при этом рассматриваются не только численные методы, а иллюстрируется применение приближенных численных методов при программной реализации аналитических решений. Например, при решении дифференциальных уравнений методами операционного исчисления возникает задача вычисления корней характеристических уравнений, которая может быть решена численно.

Глава 6 содержит введение в поисковые методы определения экстремумов функций при отсутствии и наличии шумов в определении значения функции и методы программирования соответствующих задач.

По-видимому, некоторый материал покажется избыточным повторением предположительно известного из других математических курсов, но авторы ориентировались на известный им контингент слушателей и уровень его математической подготовки; невозможно было игнорировать также несогласованность учебных планов по высшей математике, вычислительным методам и информатике – зачастую вместо общематематической поддержки прикладных курсов картина получается обратной, так как прикладные дисциплины стоят в плане раньше поддерживающих математических дисциплин.

1. Специальные классы математических объектов и операции над ними

1.1. Комплексные числа

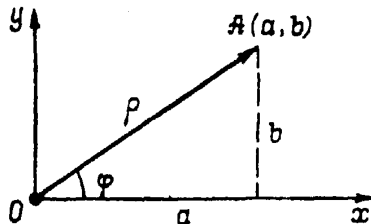
1.1.1. Основные понятия

Мнимая единица i (или j) определена как число, которое дает -1 при возведении в квадрат и приводит к надмножеству действительных чисел – комплексным числам, имеющим действительную и мнимую части и в алгебраической форме (в декартовой системе координат) имеющим вид:

$$c=Re+Im\cdot i \quad \text{или} \quad c=a+b\cdot i$$

где Re, Im – вещественные числа, причем Re – вещественная, $Im\cdot i$ (или $Im\cdot j$) – мнимая части комплексного числа.

В геометрической интерпретации в декартовой системе Re, Im – суть координаты точки на числовой плоскости с вещественной и мнимой осями или координаты конца вектора, проведенного в эту точку из начала координат.



Если использовать полярную систему координат, то получим тригонометрическую форму записи комплексного числа:

$$a+b\cdot i=\rho\cdot(\cos\varphi+i\cdot\sin\varphi)$$

где ρ или r=sqrt(Re^2+Im^2) – модуль комплексного числа, φ=arctg(Im/Re) – его аргумент, то есть угол между радиус-вектором и вещественной осью.

Используется также показательная форма записи комплексного числа

$$c = r \cdot \exp(i \cdot \varphi)$$

Два комплексных числа называют *сопряженными*, если они отличаются только знаками мнимых частей; на комплексной плоскости изображающие их точки размещены симметрично относительно вещественной оси, их модули равны, а аргументы имеют противоположные знаки.

1.1.2. Операции над комплексными числами

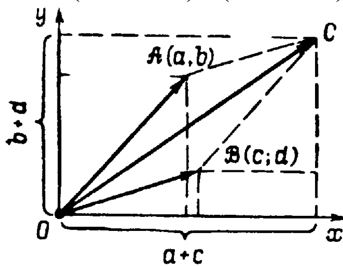
1.1.2.1. Операции сравнения

Два комплексных числа равны, если равны их вещественные и мнимые части. Понятия «больше» или «меньше» для комплексных чисел не определены.

1.1.2.2. Алгебраические операции

1.1.2.2.1. Сложение и вычитание

$$c_1 \pm c_2 = (Re_1 \pm Re_2) + (Im_1 \pm Im_2) \cdot i.$$



1.1.2.2.2. Умножение

$$c_1 \cdot c_2 = (Re_1 \cdot Re_2 - Im_1 \cdot Im_2) + (Re_1 \cdot Im_2 + Re_2 \cdot Im_1) \cdot i$$

или в тригонометрической форме

$$c_1 \cdot c_2 = r_1 \cdot r_2 \cdot (\cos(\varphi_1 + \varphi_2) + i \cdot \sin(\varphi_1 + \varphi_2)).$$

1.1.2.2.3. Деление

$$c_1 / c_2 = (Re_1 \cdot Re_2 + Im_1 \cdot Im_2)/(Re_2^2 + Im_2^2) + i \cdot [(Re_2 \cdot Im_1 - \\ - Re_1 \cdot Im_2)/(Re_2^2 + Im_2^2)]$$

или в тригонометрической форме

$$c_1 / c_2 = (r_1 / r_2) \cdot (\cos(\varphi_1 - \varphi_2) + i \cdot \sin(\varphi_1 - \varphi_2)).$$

1.1.2.2.4. Возведение в степень (формула Муавра)

$$c^n=r^n\cdot(\cos(n\cdot\varphi)+i\cdot\sin(n\cdot\varphi)).$$

1.1.3. Определение программного класса комплексных чисел

Класс комплексных чисел с операциями над ними определен в среде разработки Borland C++ в файле complex.h. Мы воспользуемся этим файлом с упрощениями, не искажающими смысла, и снабдим необходимыми, на наш взгляд, комментариями – это хороший образец для последующего самостоятельного определения класса векторов, являющегося по существу обобщением класса комплексных чисел на многомерное пространство.

Итак, файл complex.h – включаемый библиотечный файл определения комплексных чисел. Для экономии места мы опускаем директивы препроцессора условной компиляции и другие несущественные детали.

```
class complex
{
  /*Нам понадобятся рабочие переменные для хранения
Re и Im*/
  double re, im;
  /*А дальше определим общедоступные функции-члены
класса и начнем с конструкторов*/
 public:
  /*конструктор с инициализацией вещественной и
мнимой частей при объявлении комплексного числа в
прикладной программе*/
  complex(double __re_val, double __im_val=0)
  { re=__re_val; im=__im_val;}
  //конструктор по умолчанию
  complex()  {re=im=0;}
  /*текущие значения вещественной и мнимой частей
возвратят функции*/
  double  real(){ return re;}
  double  imag(){  return im;}

  //сопряженное данному комплексное число
```

```
  complex  conj(){ return complex(re,-im);}

  //Модуль и аргумент комплексного числа
  double norm () { return(re*re+im*im) ;}
  double arg(){return (!re&&!im)?0:atan2(im,re);}

  /*Операции над комплексными объектами.
Бинарные операции могут быть определены в двух ва-
риантах, различающиеся местом размещения результата
- либо с изменением текущего значения объекта, либо
без, с помещением результата во внешней по отноше-
нию к объекту переменной комплексного типа. Мы
определим простые бинарные операции как внешние
дружественные классу функции не члены класса. По-
этому сначала объявим их прототипы, а определения
приведем вне тела класса.
*/

  /*Операции проверки равенства и неравенства ком-
плексных чисел*/
  friend int operator==(complex _FAR &, complex
_FAR &);
  friend int operator!=(complex _FAR &, complex
_FAR &);
  /*Бинарные алгебраические операции над парами
комплексных чисел, комплексным и вещественным, ве-
щественным и комплексным */
  friend complex operator+(complex _FAR&, complex
_FAR &);
  friend complex operator+(double, complex _FAR &);
  friend complex  operator+(complex _FAR&, double);
  friend complex operator-(complex _FAR &, complex
_FAR &);
  friend complex operator-(double, complex _FAR &);
  friend complex operator-(complex _FAR &, double);
  friend complex operator*(complex _FAR &, complex
_FAR &);
  friend complex operator*(complex _FAR &, double);
  friend complex operator*(double, complex _FAR &);
  friend complex operator/(complex _FAR &, complex
_FAR &);
  friend complex operator/(complex _FAR &, double);
  friend complex operator/(double, complex _FAR &);
```

```
/*
Унарные +, - и * комбинированные с присвоением
арифметические операции сделаем функциями-членами,
изменяющими текущее значение комплексного объекта.
Мы объединим их объявления с определениями. */
  complex  operator+(){ return *this;}
  complex  operator-(){ return complex(-re,-im);}
  complex _FAR &  operator+=(complex _FAR &)
  {
    re+=__z2.re;
    im+=__z2.im;
    return *this;
  }
  complex _FAR & operator+=(double __re_val2)
  {
    re+=__re_val2;
    return *this;
  }
  complex _FAR & operator-=(complex _FAR & __z2)
  {
    re-=__z2.re;
    im-=__z2.im;
    return *this;
  }
  complex _FAR & operator-=(double __re_val2)
  {
    re-=__re_val2;
    return *this;
  }
  complex _FAR & operator*=(complex _FAR & __z2)
  {
    re=re*z2.real()-im*z2.imag();
    im=re*z2.imag()+z2.real()*im;
    return *this;
  }
  complex _FAR & operator*=(double __re_val2)
  {
    re*=__re_val2;
    im*=__re_val2;
    return *this;
  }
  complex _FAR & operator/=(complex _FAR & __z2)
```

```
  {
    re=(re*z2.real()+im*z2.image())/
       (z2.real()*z2.real()+z2.imag()*z2.imag());
    im=(z2.real()*im-re*z2.imag())/
       (z2.real()*z2.real()+z2.imag()*z2.imag());
    return *this;
  }
  complex _FAR & operator/=(double __re_val2)
  {
    re/=__re_val2;
    im/=__re_val2;
    return *this;
  }

  /*В заключение определим и функцию потокового вы-
вода комплексного объекта*/
  ostream _FAR& operator<<(ostream _FAR &, complex
_FAR &)
  {return os<<"("<<x.real()<<","<<x.imag()<<")";};

/* Теперь приведем реализацию дружественных опера-
торных функций, не являющихся членами класса и реа-
лизующих бинарные операции над комплексными объек-
тами. */

inline complex operator+(complex _FAR & __z1, com-
plex _FAR & __z2)
{return complex(__z1.re+__z2.re, __z1.im+__z2.im);}
inline complex operator+(double __re_val1, complex
_FAR & __z2)
{return complex(__re_val1+__z2.re, __z2.im);}
inline complex operator+(complex _FAR & __z1, dou-
ble __re_val2)
{return complex(__z1.re+__re_val2, __z1.im);}
inline complex operator-(complex _FAR & __z1, com-
plex _FAR & __z2)
{return complex(__z1.re-__z2.re, __z1.im-__z2.im);}
inline complex operator-(double __re_val1, complex
_FAR & __z2)
{return complex(__re_val1-__z2.re,-__z2.im);}
inline complex  operator-(complex _FAR & __z1, dou-
ble __re_val2)
{return complex(__z1.re-__re_val2, __z1.im);}
```

```
inline complex operator*(complex _FAR & __z1, com-
plex _FAR & __z2)
{
double r=z1.real()*z2.real()-z1.imag()*z2.imag();
double i=z1.real()*z2.imag()+z2.real()*z1.imag()};
return complex(r,i);
}
inline complex operator*(complex _FAR & __z1, dou-
ble __re_val2)
{
  return complex(__z1.re*__re_val2,
                 __z1.im*__re_val2);
}
inline complex operator*(double __re_val1, complex
_FAR & __z2)
{
  return complex(__z2.re*__re_val1,
                 __z2.im*__re_val1);
}
inline complex operator/(complex _FAR & __z1, com-
plex _FAR & __z2)
{
double
r=(z1.real()*z2.real()+z1.imag()*z2.image())/
(z2.real()*z2.real()+z2.imag()*z2.imag());
double i=(z2.real()*z1.imag()-z1.real()*z2.imag())/
(z2.real()*z2.real()+z2.imag()*z2.imag());
return complex(r,i);
}
```

1.2. Векторы

1.2.1. Основные понятия

При изучении комплексных чисел мы уже ввели понятие векторного объекта на комплексной плоскости с числовыми компонентами. Теперь расширим это понятие до векторного объекта в n-мерном пространстве, понимая под ним последовательность из n чисел (в общем случае комплексных), которые будем называть составляющими вектора.

Совокупность всех таких векторов образует n-мерное векторное пространство $\mathbf{R}^n$.

Два вектора будем считать *равными* тогда и только тогда, когда все их компоненты равны.

Умножение вектора на число определим как операцию умножения всех составляющих вектора на это число.

Сложение векторов будет сводиться к сложению их составляющих.

Нулевым будем считать вектор, у которого все составляющие равны нулю.

Линейно-зависимыми будем считать векторы

$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(n)}$$

если существуют такие не все нулевые константы C_1, C_2, …, C_n, что

$$C_1 \cdot \boldsymbol{x}^{(1)} + C_2 \cdot \boldsymbol{x}^{(2)} + \ldots + C_n \cdot \boldsymbol{x}^{(n)} = 0$$

В противном случае векторы считаются *линейно-независимыми*.

В последнем уравнении слева стоит вектор-сумма, который может по сделанному определению быть равен нулю при всех нулевых составляющих. Если количество векторов n, то это векторное уравнение равносильно системе из n уравнений с неизвестными C_1, C_2, …, C_n:

$$C_1 \cdot x_1^{(1)} + C_2 \cdot x_1^{(2)} + \ldots + C_l \cdot x_1^{(n)} = 0$$

$$C_1 \cdot x_2^{(1)} + C_2 \cdot x_2^{(2)} + \ldots + C_l \cdot x_2^{(n)} = 0$$

....

$$C_1 \cdot x_n^{(1)} + C_2 \cdot x_n^{(2)} + \ldots + C_l \cdot x_n^{(n)} = 0$$

Если число векторов больше размерности пространства $n<l$, то есть число уравнений меньше числа неизвестных, то система видимо будет иметь отличные от нулевого решения для C_j и наши векторы будут соответственно линейно-зависимыми. Другими словами – число линейно-независимых векторов не больше размерности пространства.

При $n=l$ (число уравнений равно числу неизвестных) система может иметь ненулевое решение (а векторы могут

быть линейно-зависимыми) только тогда, когда ее определитель равен нулю.

Таким образом, для линейной независимости векторов необходимо и достаточно, чтобы определитель, составленный из составляющих векторов, был отличен от нуля или чтобы ранг матрицы, составленной из векторов, был равен числу векторов.

Введем теперь новое понятие, которое в дальнейшем будет широко использоваться:

Скалярным произведением $(\boldsymbol{x}, \boldsymbol{y})$ двух векторов

$$\boldsymbol{x}(x_1, x_2,\dots, x_n) \text{ и } \boldsymbol{y}(y_1, y_2, \dots, y_n)$$

называется число, равное следующей сумме: $\sum_{s=1}^{n} x_s \cdot y_s$.

Взаимно-ортогональными или взаимно-перпендикулярными будем называть два вектора, если их скалярное произведение равно нулю.

Корень квадратный из скалярного произведения $(\boldsymbol{x}, \boldsymbol{x})$ вектора $\boldsymbol{x}(x_1, x_2,\dots, x_n)$ на этот же вектор

$$\|\boldsymbol{x}\|^2=(\boldsymbol{x}, \boldsymbol{x})=\sum_{s=1}^{n} x_s^2 ; \quad \|\boldsymbol{x}\|=\sqrt{\sum_{s=1}^{n} x_s^2} .$$

называют *нормой*, длиной или модулем вектора $\boldsymbol{x}$.

Использование скалярного произведения позволяет записать систему однородных линейных алгебраических уравнений в виде:

$$(\boldsymbol{x}, \boldsymbol{a}_j)=0 \ (i=1, 2, \dots, n; j=1, 2, \dots, n)$$

откуда очевидно, что ее решение сводится к нахождению вектора $\boldsymbol{x}$, ортогонального ко всем векторам $\boldsymbol{a}_j$. Решением этой задачи мы займемся в разделе, посвященном матрицам и линейной алгебре, а пока ограничимся тем набором понятий и операций, которые уже нами определены и разместим соответствующие данные и методы в классе С++.

1.2.2. Определение программного класса многомерных векторов

Итак, составим включаемый файл vector.h, который будет содержать все, о чем мы договорились в предыдущем теоретическом подразделе. Для этого вначале рассмотрим структуру класса для работы с векторами, алгоритмы для работы с объектами типа «вектор» и примеры методов векторного класса, реализующие эти алгоритмы:

```
template <class YourOwnFloatType> class vector
{
```

Класс для работы с векторными объектами – параметризованный, то есть мы определяем только шаблон, по которому для конкретных типов, подставляемых вместо YourOwnFloatType, компилятор автоматически генерирует класс.

```
long m;
```

Вектор будет характеризоваться размерностью (длиной), которая может быть любым натуральным числом. Скаляры можно рассматривать как вектора размерности 1, комплексные числа – как вектора размерности 2 и т.д. В общем случае размерность вектора равна числу его составляющих, т.е. размерность вектора $(a_1, a_2, \ldots, a_n) - n$.

```
YourOwnFloatType *vec;
```

Данные, характеризующие вектор, мы будем хранить по этому указателю. В данном случае это – компоненты, или, как их ещё называют, координаты вектора. Это название не случайно: геометрические вектора размерности 2 – направленные отрезки на плоскости – могут быть связаны в декартовой системе координат с арифметическими координатами своих концов, если считать их всегда исходящими из начала координат.

```
virtual void In(istream &);
virtual void Out(ostream &);
```

Виртуальные функции чтения из потока и записи в поток; их необходимо переопределить в производных классах для обеспечения возможности использования единых перегруженных операторов << и >>. Явно использовать их необходимости нет, потому они и перенесены в приватную часть.

```
public:
```

В этом разделе – общедоступные методы класса, которые можно использовать для манипуляций с векторными объектами.

```
vector(char *);
```

Конструктор класса «вектор», параметром которого является имя текстового файла. Предполагается, что первым числом в этом файле является размерность вектора, после которого следуют данные. Этот конструктор определяет, откуда мы можем взять данные о векторе – размерность и компоненты.

```
vector();
```

Это конструктор по умолчанию, создающий нулевой вектор единичной размерности. Необходим при динамическом создании массивов векторов. Геометрически такой вектор – это точка на числовой прямой в отметке «0»; такой подход связан с трудностью решения вопроса о размерности вектора по умолчанию и отнюдь не является единственно правильным. Основная причина появления этого конструктора здесь – это требование компилятора.

```
vector(long);
```

Параметром этого конструктора является размерность вектора; после создания компоненты вектора обнуляются. Если, к примеру, принимаемая размерность – 5, то мы получим вектор вида (0, 0, 0, 0, 0).

```
vector(long, YourOwnFloatType *);
```

Этот конструктор пытается создать вектор размерности, заданной первым параметром, и заполнить его данны-

ми из массива. Полезен в том случае, когда компоненты вектора заранее известны.

Пусть, к примеру, параметры этого конструктора – размерность 3 и некий массив с числами 1, 2, 3, 4, 5, 6, …. В этом случае данный конструктор создаст вектор (1, 2, 3).

```
vector(vector<YourOwnFloatType> &);
```

Конструктор копирования (инициатор копии), создающий новый вектор из уже имеющегося. Размерность нового вектора устанавливается в размерность старого, а данные переписываются без изменений. В результате его выполнения мы получаем вектор, идентичный копируемому, но находящийся в другом участке памяти.

```
~vector();
```

Деструктор. Освобождает динамически распределённую память из-под компонент вектора.

```
friend vector<YourOwnFloatType> operator+ (vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Дружественная функция сложения двух векторов. Складывает вектора только совпадающих размеров, вектор-результат конструируется и возвращается. При этом вектора одинаковых размерностей складываются по закону:

$$(a_1, a_2, ..., a_n) + (b_1, b_2, ..., b_n) = (c_1, c_2, ..., c_n),$$

$$c_i = a_i + b_i .$$

```
friend vector<YourOwnFloatType> operator+= (vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Перегруженная операция сокращённого сложения складывает первый вектор со вторым, модифицируя при этом первый вектор (записывая результат сложения в него) и возвращает результат для того, чтобы он мог участвовать в других операциях над векторами типа а+(с=у) и т.п.

```
friend vector<YourOwnFloatType> operator-(vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Дружественная функция для вычитания векторов одинаковой размерности; вычитает первый вектор из второго, конструируя и возвращая результат как новый вектор. При этом вектора одинаковых размерностей вычитаются по закону:

$$(a_1, a_2, ..., a_n) - (b_1, b_2, ..., b_n) = (c_1, c_2, ..., c_n),$$
$$c_i = a_i - b_i .$$

```
friend vector<YourOwnFloatType> operator-=(vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Сокращённое вычитание, работающее так же, как и сокращённое сложение: из первого вектора вычитается второй, результат снова записывается в первый вектор, а его копия возвращается.

```
friend YourOwnFloatType operator* (vector <YourOwn-
FloatType> &, vector<YourOwnFloatType> &);
```

Скалярное произведение двух векторов одинаковой размерности – число того же типа, что и компоненты исходных векторов. Его можно получить по следующему закону:

$$(a_1, a_2, ..., a_n) \cdot (b_1, b_2, ..., b_n) = \sum_{i=1}^{n} a_i b_i$$

```
friend vector<YourOwnFloatType> operator* (YourOwn-
FloatType, vector<YourOwnFloatType> &);
```

Ещё одна дружественная функция, перегружающая операцию умножения в несколько ином контексте, а именно – как умножение скаляра на вектор. Результатом является вектор той же размерности, что и исходный. Произведением вектора $\boldsymbol{a}(a_1, a_2, \ldots, a_n)$ на число α будет вектор $\alpha\boldsymbol{a}(\alpha a_1, \alpha a_2, \ldots, \alpha a_n)$, то есть вектор, каждая компонента которого умножена на это число.

```
friend vector<YourOwnFloatType> operator* (vector
<YourOwnFloatType> &, YourOwnFloatType);
```

Умножение скаляра на вектор операция коммутативная, что не избавляет нас от необходимости снова перегрузить операцию умножения, с теми же аргументами, но в другом порядке. Действительно, от соотношения $\alpha\boldsymbol{a}=(\alpha a_1, \alpha a_2, \ldots, \alpha a_n)$ мы легко можем перейти к соотношению $(a_1\alpha, a_2\alpha, \ldots, a_n\alpha)=\boldsymbol{a}\alpha$.

```
friend vector<YourOwnFloatType> operator*= (vector
<YourOwnFloatType> &, YourOwnFloatType);
```

Сокращённое умножение вектора на число. Первый аргумент этой функции – вектор – после выполнения данной операции модифицируется, копия результата возвращается.

```
friend ostream & operator << (ostream &, vector <YourOwn-
FloatType> &);
```

Перегруженная операция вывода вектора в поток. При этом выводятся только компоненты вектора, разделяемые пробелами; размерность не указывается. Модифицированный поток возвращается.

```
friend istream & operator >> (istream &, vector
<YourOwnFloatType> &);
```

Ввод вектора из потока осуществляется путём приёма из потока количества чисел, равного размерности вектора. Модифицированный поток ввода возвращается для участия в дальнейших операциях ввода из него.

```
vector<YourOwnFloatType> operator=(vector <YourOwn-
FloatType> &);
```

Присвоение – операция, которая не может быть перегружена с использованием механизма дружественных функций. Первым, неявным параметром этой функции является текущий объект (*this), вторым – объект, присваиваемый текущему. При этом, если размерности присваиваемого и текущего векторов совпадают, компоненты последнего просто переписываются. В противном же случае размерность текущего вектора устанавливается в размерность копируемого, а память под компоненты перераспределятся,

и только тогда происходит перепись данных. Возвращаемым значением является сам текущий вектор.

```
vector<YourOwnFloatType> operator-();
```

Унарный минус. Эта операция создаёт вектор той же размерности, что и текущий, с компонентами, имеющими противоположный знак, то есть Вектор, противоположный вектору $\boldsymbol{a}$ с координатами (a_1, a_2, …, a_n) – вектор ($-\boldsymbol{a}$) с координатами ($-a_1$, $-a_2$, …, $-a_n$).

```
vector<YourOwnFloatType> operator+();
```

Эта функция-член включена только для полноты набора. Не выполняя ничего полезного, она просто возвращает копию текущего вектора.

```
vector<YourOwnFloatType> operator~();
```

Метод, для данного ненулевого вектора конструирующий вектор, имеющий то же направление и единичную длину, т.е. выполнятся нормирование данного вектора по модулю. Так как компонентами вектора-результата фактически являются косинусы углов данного вектора к координатным осям (а также длины проекций, т.к. вектор-результат – единичный), то данную операцию называют ещё и *определением направляющих косинусов*.

Модуль вектора (a_1, a_2, …, a_n) определим как корень квадратный из скалярного произведения этого вектора на себя: $|(a_1, a_2, \dots, a_n)| = \sqrt{\sum_{i=1}^{n} a_i^2}$, а операцию нормирования по модулю – как деление каждой составляющей вектора на его модуль, или же умножение ненулевого вектора на величину, обратную модулю:

$$\frac{1}{\sqrt{\sum_{i=1}^{n} a_i^2}}\left(a_1, a_2, \dots, a_n\right) = \left(\frac{a_1}{\sqrt{\sum_{i=1}^{n} a_i^2}}, \frac{a_2}{\sqrt{\sum_{i=1}^{n} a_i^2}}, \dots, \frac{a_n}{\sqrt{\sum_{i=1}^{n} a_i^2}}\right).$$

```
YourOwnFloatType operator!();
```

Метод, для вектора любой размерности определяющий его модуль. При этом делается, заметим, далеко не всегда корректное допущение, что длина вектора выражается в единицах того же типа, что и компоненты вектора. Например, для целых векторов эта операция даст лишь приближённый результат, а для многомерных векторов (например, комплексных) данная операция имеет смысл нормы.

```
virtual long IsEqual(void *);
```

Эта виртуальная функция сравнивает текущий вектор с объектом, лежащим по адресу, передаваемому через обобщённый указатель. При этом делается неявное предположение о том, что указатель содержит адрес вектора, а не чего-либо более экзотического. Преобразуя получаемый указатель к указателю на вектор, эта функция пытается сравнить его с текущим. Заметим, что вектора будут считаться равными, если выполнятся два условия – одинаковая размерность и совпадающие компоненты:

$$(a_1, a_2, ..., a_n) = (b_1, b_2, ..., b_n) \Leftrightarrow a_1 = b_1 \wedge a_2 = b_2 \wedge ... \wedge a_n = b_n$$

```
friend long operator==(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

Используя описанную выше функцию, вводим операторное сравнение двух векторов – проверка на равенство…

```
friend long operator!=(vector<YourOwnFloatType> &, vector<YourOwnFloatType> &);
```

…и проверка на неравенство.

```
YourOwnFloatType &operator[](long a);
```

Индексирование элементов вектора – это операция, которая возвращает ссылку на компоненту вектора с заданным номером. Если этот номер выходит за границы размерности вектора, после диагностики возвращается специальный код ошибки. Так как данная функция ссылочная, то её можно использовать в операторах присваивания как слева, так и справа – ведь модифицируя ссылочный объект, мы фактически воздействуем на то, что под ним скрывается, то

есть на саму векторную компоненту. Заметим, что вектор размерности *a* можно индексировать от 0 до (*a*–1), а не от 1 до *a* (!).

```
long getm() { return m; }
```

Размерность вектора можно узнать, используя этот метод.

```
};
```

По приведенным описаниям можно составить, к примеру, такой интерфейс для данного класса:

```
#ifndef __VECTOR_H
#define __VECTOR_H
#ifndef __FSTREAM_H
#include <fstream.h>
#endif
#ifndef __IOMANIP_H
#include <iomanip.h>
#endif
#ifndef __STDLIB_H
#include <stdlib.h>
#endif
#ifndef __MATH_H
#include <math.h>
#endif
#ifndef __EXCEPT_H
#include <except.h>
#endif
#ifndef __CSTRING_H
#include <cstring.h>
#endif

/*параметризованный класс для работы с векторными
объектами*/

template <class YourOwnFloatType>/* подставьте свой
тип*/
class vector
{ //приватные данные
  long m;             //размерность (длина) вектора
```

```
  YourOwnFloatType *vec; /*указатель на элементы
вектора*/
  /*виртуальные функции чтения из потока и записи в
поток; их необходимо переопределить в производных
классах для обеспечения возможности использования
единых перегруженных операторов << и >> */
  virtual void In(istream &);
  virtual void Out(ostream &);
 public://общедоступные данные и функции
  /*загрузка вектора из файла: dimension data1
data2... */
  vector(char *);
  vector();/*создание пустого вектора единичной
размерности */
  vector(long);/*создание пустого вектора заданной
размерности */
  vector(long, YourOwnFloatType *); /*создание век-
тора заданной размерности, заполняемого данными из
массива */
  //конструктор копирования
  vector(vector<YourOwnFloatType> &);
  ~vector();//деструктор
  friend vector<YourOwnFloatType> operator+ (vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
//сложение двух векторов
  friend vector<YourOwnFloatType> operator+= (vec-
tor <YourOwnFloatType> &, vector <YourOwnFloatType>
&);//сложение с присвоением
  friend vector<YourOwnFloatType> operator-(vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
//вычитание
  friend vector<YourOwnFloatType> operator-= (vec-
tor <YourOwnFloatType> &, vector <YourOwnFloatType>
&);//вычитание с присвоением
  friend YourOwnFloatType operator* (vector
<YourOwnFloatType> &, vector<YourOwnFloatType> &);
//скалярное умножение
  friend vector<YourOwnFloatType> operator*
(YourOwnFloatType, vector<YourOwnFloatType> &);
//умножение числа на вектор
  friend vector<YourOwnFloatType> operator* (vec-
tor<YourOwnFloatType> &, YourOwnFloatType );
//умножение вектора на число
```

```
  friend vector<YourOwnFloatType> operator*= (vec-
tor<YourOwnFloatType> &, YourOwnFloatType );
//умножение вектора на число с присвоением
  friend ostream &operator<<(ostream &, vector
<YourOwnFloatType> &);//вывод вектора в поток
  friend istream &operator>>(istream &, vector
<YourOwnFloatType> &);//ввод вектора из потока
  vector<YourOwnFloatType> operator= (vector
<YourOwnFloatType> &);//присвоение
  vector<YourOwnFloatType> operator-();/*унарный
минус*/
  vector<YourOwnFloatType> operator+();/*унарный
плюс*/
  vector<YourOwnFloatType> operator~();
/*нормирование (определение направляющих косинусов)
*/
  YourOwnFloatType operator!();//модуль вектора
  virtual long IsEqual(void *);
/*эта виртуальная функция сравнивает текущий вектор
с объектом, лежащим по адресу, передаваемому через
обобщённый указатель*/
  friend long operator==(vector<YourOwnFloatType>
&, vector<YourOwnFloatType> &);/*проверка на равен-
ство */
  friend long operator!=(vector<YourOwnFloatType>
&,  vector<YourOwnFloatType> &);/*проверка на нера-
венство */
  YourOwnFloatType &operator[](long a);
     //индексирование элементов вектора
  long getm() { return m; }//размерность вектора
};

/*Теперь реализация объявленных методов класса
vector */

//индикатор ошибки - некая константа
const long double MAX_LONGDOUBLE=1.7976931348e308;

/*
  Создавать вектор можно по-разному. Например, если
он находится на внешнем устройстве в формате m d1 d2
... dm, где m - размерность вектора, а di - его ком-
поненты, то имеем следующий конструктор:
```

```
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(char *f) /*имя
файла */
{
  long i;

  ifstream fp=f;//пытаемся открыть файл
  if(!fp)//если не удалось
    throw xmsg("Не могу открыть файл "+string(f)+
"\n");
  fp>>m;//вводим размерность
  if(m<=0)//проверка на корректность
    throw xmsg("Размерность вектора некорректна
\n");        //диагностика
  try
  {
   vec=new YourOwnFloatType[m];/*попытка выделения
памяти*/
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(i=0;i<m&&fp>>vec[i];i++);/*считывание из фай-
ла */
}


/*
  В случае, когда нам известна лишь размерность
вектора, но неизвестны его составляющие, предпола-
гаем, что данный вектор является нулевым:
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long a):m(a)
//размерность вектора
{
  long i;

  if(m<=0)//проверка размерности
    throw xmsg("Размерность вектора некорректна
\n");
```

```
  //диагностика
  try
  {
   vec=new YourOwnFloatType[m];/*попытка выделения
памяти */
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(i=0;i<m;vec[i++]=0);/*обнуление компонент
вектора */
}


/*
  Наконец, нам могут быть известны как размерность,
так и компоненты вектора:
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector(long a,
YourOwnFloatType *v):m(a)
/*этот конструктор принимает размер и указатель на
данные */
{
  long i;

  if(m<=0)//проверка размерности
    throw xmsg("Размерность вектора некорректна
\n");
  //диагностика
  try
  {
   vec=new YourOwnFloatType[m];/*попытка выделения
памяти */
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(i=0;i<m;i++)
    vec[i]=v[i];/*копирование из внешнего массива в
вектор*/
```

```
}


/*
  Есть ещё один случай, когда мы ничего не можем
сказать о размерности и компонентах вектора - при
создании массива векторов, то есть матрицы, когда
для оператора new требуется конструктор без пара-
метров или когда размер вектора заранее неизвестен.
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector():m(1)
{
  try
  {
   vec=new YourOwnFloatType[m];/*попытка выделения
памяти */
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  *vec=0;//обнуляем единственный имеющийся элемент
}


/*
  В реальных расчетах могут использоваться векторы
больших размерностей, поэтому размещаются они в
свободной памяти компьютера, а когда необходимость
в них отпадает - уничтожаются.
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType>::~vector()
{
  delete []vec;//уничтожение динамического массива
}


/*
  Необходимость в индексации вектора возникает в
двух случаях:
```

- при получении составляющей вектора по её номеру и
- при изменении не всего вектора, а только одной его составляющей.

  При этом, конечно, следует учитывать возможность ошибочного задания номера составляющей: допустимый диапазон значений [0,m).
*/
```
template <class YourOwnFloatType>
YourOwnFloatType &
vector<YourOwnFloatType>::operator[](long a)
{
  static YourOwnFloatType error=MAX_LONGDOUBLE;
  if(a>=0&&a<m)//если всё ОК
    return vec[a];
  else//при выходе за пределы вектора ругаемся
  {
    cerr<<"Индекс "<<a<<" вне диапазона вектора\n";
    return error;
  }
}
```

/*
  Создавая вектор, можно попутно инициализировать его данными из уже существующего:
*/
```
template <class YourOwnFloatType>
vector<YourOwnFloatType>::vector( vec-
tor<YourOwnFloatType> &ex) : m(ex.m)
/*это конструктор копирования, принимающий ссылку
на вектор */
{
  try
  {
   vec=new YourOwnFloatType[m];/*попытка выделения
памяти*/
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(long i=0;i<m;i++)
```

```
    vec[i]=ex[i];
  /*здесь при копировании ex используется уже ин-
дексация*/
}


/*
  Сложение векторов является алгебраической опера-
цией только тогда, когда вектора одинаковой размер-
ности. Результатом сложения является вектор той же
размерности, что и исходные, компонентами которого
является сумма соответствующих компонент исходных
векторов.
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator+
(vector<YourOwnFloatType> &f, vec-
tor<YourOwnFloatType> &s)
{
  if(f.m!=s.m)//проверка на равенство размерностей
    throw xmsg("Слагаемые вектора имеют различные
длины \n");
  //диагностика
  vector<YourOwnFloatType> temp(f.m);
  //создаём временный вектор
  /*здесь работают операции индексирования для всех
трёх векторов*/
  for(long i=0;i<f.m;i++)
    temp[i]=f[i]+s[i];
  return temp;//возвращаем результирующий вектор
}


//сокращённая операция "сложение с присвоением"
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator+=
(vector<YourOwnFloatType> &f, vec-
tor<YourOwnFloatType> &s)
{
  return f=f+s;
}
```

```
/*
  Введём несколько вспомогательных унарных опера-
ций:
- "минус":
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> vector<YourOwnFloatType>::
operator-()
{
  vector<YourOwnFloatType> temp(m);
  //создаём временный вектор
  /*Если this - это указатель на текущий объект
векторного класса, то *this - это сам текущий объ-
ект класса vector, то есть тот, с которым мы сейчас
работаем. А к любому векторному объекту мы можем
применить операцию индексирования */
  for(long i=0;i<m;i++)
    temp[i]=-(*this)[i];
  //temp[i]=-vec[i];
  //temp.vec[i]=-vec[i];
  //temp.operator[](i)=-operator[](i); etc...
  return temp;//возвращаем результирующий вектор
}


//унарный плюс
template <class YourOwnFloatType>
vector<YourOwnFloatType> vector<YourOwnFloatType>::
operator+()
{
  return *this;//возвращаем самого себя
}


/*
  Операция, которую алгебраической назвать нельзя -
это, скорее, пример очень распространённого тернар-
ного отношения "скалярное произведение двух векто-
ров":
*/
template <class YourOwnFloatType>
YourOwnFloatType operator*(vector<YourOwnFloatType>
&f, vector<YourOwnFloatType> &s)
```

```
{
  if(f.m!=s.m)
    throw xmsg("Умножение векторов с несовпадающими
размерами невозможно \n");//диагностика
  YourOwnFloatType temp=0;
  for(long i=0;i<f.m;i++)
    temp+=f[i]*s[i];
    //суммируем произведения составляющих векторов
  return temp;
}


/*
  Модуль вектора как квадратный корень скалярного
произведения вектора на самого себя:
*/
template <class YourOwnFloatType> inline YourOwn-
FloatType vector<YourOwnFloatType>:: operator!()
{
  return sqrt((*this)*(*this));
}


/*
  нормирование вектора по модулю
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> vec-
tor<YourOwnFloatType>::operator~()
{
  vector<YourOwnFloatType> temp(m);
  /*скалярное произведение текущего объекта на са-
мого себя*/
  YourOwnFloatType modul=!(*this);
  for(long i=0;i<m;i++)
    temp[i]=(*this)[i]/modul;/* направляющие коси-
нусы*/
  return temp;
}


/*
  умножение числа на вектор":
```

```
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator*
(YourOwnFloatType ld, vector<YourOwnFloatType> &v)
{
  vector<YourOwnFloatType> temp=v;
  for(long i=0;i<v.getm();i++)
    temp[i]=temp[i]*ld;/* скорее, это даже "удлине-
ние" вектора */
  return temp;
}


/*
  умножение вектора на число:
*/
template <class YourOwnFloatType>
inline vector<YourOwnFloatType> operator*
(vector<YourOwnFloatType> &v, YourOwnFloatType ld)
{
  return ld*v;/*очень просто - вызвали другую функ-
цию */
}


//операция сокращённого умножения вектора на число
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator*=
(vector<YourOwnFloatType> &v, YourOwnFloatType ld)
{
  return v=v*ld;
}


/*
  Имея определённые бинарную операцию сложения век-
торов и унарную получения вектора, противоположного
к данному, можно на векторном языке, не обращаясь к
компонентам векторов, определить операцию вычита-
ния:
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator-
```

```
(vector<YourOwnFloatType> &f, vec-
tor<YourOwnFloatType> &s)
{
  return f+(-s);
  //return operator+(f,-s);
  //return operator+(f,s.operator-());
}


//операция сокращённого вычитания
template <class YourOwnFloatType>
vector<YourOwnFloatType> operator-=
(vector<YourOwnFloatType> &f, vec-
tor<YourOwnFloatType> &s)
{
  return f=f-s;
}


/*
  При переписывании одного вектора в другой возмож-
ны два случая:
  1. если размерность обоих векторов совпадает, то
просто заменяем составляющие первого вектора компо-
нентами второго;
  2. в противном случае безжалостно уничтожаем пер-
вый вектор и создаём снова, используя второй как
строительный материал.
*/
template <class YourOwnFloatType>
vector<YourOwnFloatType> vector<YourOwnFloatType>::
operator=(vector<YourOwnFloatType> &x)
{
  if(m!=x.m)//если размеры не совпадают
  {
    delete []vec;/*уничтожаем содержимое текущего
вектора */
    m=x.m;//устанавливаем новый размер
    try
    {
      vec=new YourOwnFloatType[m];/*попытка выделе-
ния памяти */
    }
```

```
    catch(xalloc)
    {
      throw xmsg("Не хватает памяти \n");
    }
  }
  for(long i=0;i<m;i++)
    vec[i]=x[i];/*копируем данные из вектора x в
текущий*/
/*присвоение - это бинарная операция, первым пара-
метром которой является объект, которому присваива-
ют, вторым - объект, который присваивают. При этом
первый объект, в отличие от всех остальных бинарных
операций, меняется, и он же возвращается в качестве
результата (это бывает необходимым для операций ви-
да a=b=c;)*/
  return *this;
}


/*Эта функция сравнивает текущий вектор с вектором,
лежащим по адресу x. Для каждого класса, производ-
ного от векторного, не имеет смысла переопределять
операторные функции проверки на равенство и нера-
венство - достаточно переопределить эту виртуальную
функцию*/
template <class YourOwnFloatType>
long vector<YourOwnFloatType>::IsEqual(void *x)
{
  if(m!=((vector<YourOwnFloatType>*)x)->m)
    //при несовпадении размерностей
    return 0; //констатируем несовпадение векторов
  for(long i=0;i<m;i++)
    if((*this)[i]!=
             (*(vector<YourOwnFloatType>*)x)[i])
      return 0;//если хоть один элемент не совпал
  return 1;
}


/*
  Сравнение векторов является тернарным отношением,
результатом которого является число нуль, если век-
торы не равны и единица в противном случае.
```

```
  Два вектора будем считать равными, если они имеют
одинаковые длины и их   соответствующие составляю-
щие совпадают:
*/
template <class YourOwnFloatType>
long operator==(vector<YourOwnFloatType> &f,
vector<YourOwnFloatType> &s)
{
  return f.IsEqual(&s);
}


/*
  Неравенство векторов определим через равенство и
операцию отрицания:
*/
template <class YourOwnFloatType> inline long oper-
ator!=(vector<YourOwnFloatType> &f,
vector<YourOwnFloatType> &s)
{
  return !(f==s);//логично
}


/*Мощный I/O-механизм C++ позволяет в естественной
форме выводить (вводить) векторы на любое устрой-
ство отображения информации. Для универсализации
считывания и записи вектора в поток снова прибегнем
к механизму виртуальных функций. С этой целью, по
аналогии с printOn, определим две функции - одну
для ввода, другую - для вывода */

template <class YourOwnFloatType>
void vector<YourOwnFloatType>::In(istream &is)
{
  for(long i=0;i<m;i++)
    is>>(*this)[i];
}

template <class YourOwnFloatType>
void vector<YourOwnFloatType>::Out(ostream &os)
{
  for(long i=0;i<m;i++)
```

```
  {
    os.precision(100);
    os<<(*this)[i]<<" ";/*компоненты разделяем про-
белами */
  }
}

//вывод в поток
template <class YourOwnFloatType>
ostream &operator<<(ostream &os,
vector<YourOwnFloatType> &x)
{
  x.Out(os);
  return os;
}

/*  - ввод из потока */
template <class YourOwnFloatType>
istream &operator>>(istream &is,
vector<YourOwnFloatType> &x)
{
  x.In(is);
  return is;/*принимаем и возвращаем ссылку на по-
ток ввода */
}

#endif
```

1.3. Полиномы

1.3.1. Общие сведения

Многочлен с целочисленными степенями переменных (полином) в общем виде записывается так:

$$f(z)=a_n\cdot z^n+a_{n-1}\cdot z^{n-1}+\ldots+a_{n-k}\cdot z^{n-k}+\ldots+a_1\cdot z+a_0,$$

где a_0, a_1, …, a_k, …, a_n – заданные числа (в общем случае – комплексные), z – переменная (в общем случае – комплексная). Старший коэффициент a_0 в соответствии со здравым смыслом мы будем считать отличным от нуля.

Значения z, при подстановке которых полином обращается в нуль, называются *корнями* (или нулями) этого полинома, то есть корни полинома есть решения уравнения

$$f(z)=a_n\cdot z^n+a_{n-1}\cdot z^{n-1}+\ldots+a_{n-k}\cdot z^{n-k}+\ldots+a_1\cdot z+a_0=0$$

Это уравнение называют *алгебраическим уравнением n-й степени*.

При делении $f(z)$ на двучлен $(z-a)$ частное $Q(z)$ будет многочленом $(n-1)$-й степени со старшим коэффициентом a_0, остаток R не будет содержать z, то есть имеет место тождество:

$$f(z)=(z-a)\cdot Q(z)+R$$

После подстановки в него $z=a$ получим $R=f(a)$ – остаток при делении полинома на $(z-a)$ равен $f(a)$ (теорема Безу). При делении без остатка (с нулевым остатком) $f(a)=0$, то есть $z=a$ должно быть корнем полинома. Зная этот корень, можно выделить из полинома множитель $(z-a)$:

$$f(z)=(z-a)\cdot f_1(z),$$

где

$$f_1(z)=b_{n-1}\cdot z^{n-1}+b_{n-2}\cdot z^{n-2}+\ldots+b_1\cdot z+b_0\ (b_{n-1}=a_0)$$

и для нахождения остальных корней надо решить уравнение на один порядок ниже:

$$b_{n-1}\cdot z^{n-1}+b_{n-2}\cdot z^{n-2}+\ldots+b_1\cdot z+b_0=0$$

В соответствии с основной теоремой алгебры, всякое алгебраическое уравнение имеет хотя бы один вещественный или комплексный корень (например, z_1) и делится на $(z-z_1)$, полином-частное тоже будет иметь корень и делится на $(z-z_2)$ и т.д. – таким образом, всякий полином степени n разлагается на $n+1$ множителей, один из которых равен старшему коэффициенту, а остальные есть двучлены вида $(z-a)$.

$$f(z)=a_n\cdot(z-z_1)\cdot(z-z_2)\cdot\ldots\cdot(z-z_n)$$

Это разложение на множители единственно.

Среди корней полинома могут быть кратные – необходимым и достаточным условием того, что значение $z=a$ яв-

ляется корнем кратности k является обращение в нуль при этом значении полинома и всех его производных до (k–1)-й включительно и необращение в нуль k-й производной. Корень кратности k некоторого полинома является корнем кратности (k–d) для d-й производной этого полинома, то есть если имеет место разложение полинома

$$f(z)=a_n\cdot(z-z_1)^{k1}\cdot(z-z_2)^{k2}\cdot\ldots\cdot(z-z_m)^{km},$$

где z_1, z_2,…z_m – различны и $k1+k2+\ldots+km=n$, то разложение производной будет

$$f'(z)=(z-z_1)^{k1\text{-}1}\cdot(z-z_2))^{k2\text{-}1}\cdot\ldots\cdot(z-z_m)^{km\text{-}1}\cdot w(z),$$

где $w(z)$ – полином, уже не имеющий общих с $f(z)$ корней.

Наибольший общий делитель двух полиномов есть произведение всех общих для них двучленных множителей с меньшими из двух вариантов показателями степени. Если полиномы не имеют общих корней, то они – взаимно-простые. Составление полинома – наибольшего общего делителя двух других полиномов можно выполнить известным в арифметике методом определения НОД двух целых чисел: полином со степенью не меньше степени второго делим на второй, затем второй делим на остаток при первом делении, этот первый остаток делим на остаток при втором делении и т.д. до получения нулевого остатка. Последний ненулевой остаток и есть НОД и если он не содержит z, то полиномы взаимно-простые. Разделив полином на его НОД и НОД его производной, получим полином, имеющий все простые корни, совпадающие с различными корнями исходного полинома – так можно освободиться от кратных корней без решения уравнения $f(z)$=0.

1.3.2. Операции над полиномами

Основные операции с полиномами хорошо известны из элементарной алгебры, поэтому здесь мы остановимся подробно только на операции деления многочленов и процедурах вычисления значений и корней полиномов. Эти операции и основанные на них основные результаты будут нами часто использоваться и поэтому заслуживают по-

дробного рассмотрения – насколько это возможно в кратком курсе численных методов. Методы определения корней многочленов, как правило (за исключением решений, получаемых в квадратурах для полиномов порядка не выше четвертого), пригодны и для вычисления корней уравнений неалгебраического типа (трансцендентных).

Деление многочленов осуществляют по правилам, принятым для целочисленного деления, при котором в качестве результата получают частное и остаток. Проще всего реализовать его по известным правилам деления «в столбик» по методу Евклида.

В качестве первого элемента частного от деления полинома $f_1(z)$ на полином $f_2(z)$ берут переменную z в степени, равной разности порядков полиномов делимого и делителя с коэффициентом, равными частному от деления коэффициента при старшей степени делимого на коэффициент делителя при старшей степени делителя; этот элемент умножают на делитель и результат вычитают из делимого.

С полученным разностным полиномом пониженного порядка все действия повторяют, получая второй и последующие элементы частного, пока его порядок не станет ниже порядка делителя – этот полином представляет собой остаток от деления.

1.3.3. Вычисление значений полиномов

Вспомнив теорему Безу о том, что остаток от деления многочлена на двучлен $(z–a)$ равен значению полинома при $z=a$, можно использовать для вычисления значения полинома операцию деления на двучлен. Схему деления для этого частного случая можно упростить, используя a вместо $(z–a)$ и суммирование вместо вычитания, а также используя запись только коэффициентов без степеней z; отсутствующие степени обозначаются нулевыми коэффициентами. Такая схема вычислений известна как *схема Горнера*.

1.3.4. Вычисление корней полиномов

Для решения этой часто возникающей на практике задачи предложено много методов и их количество говорит об отсутствии одного, обладающего убедительными преимуществами над другими. Наиболее общим естественно считать метод, позволяющий находить комплексные корни многочленов; при этом можно ограничиться подклассом многочленов с вещественными коэффициентами, так как именно они обычно возникают в практических расчетах.

Численные методы решения этой задачи строятся по одному и тому же шаблону: выбирается первое (при отсутствии априорных данных – произвольное) значение корня и вычисляется значение функции при этом значении; теперь стоит задача скорректировать текущее значение x_k так, чтобы соответствующее значение полинома оказалось ближе к нулю.

$$x_{k+1}=x_k+d.$$

Приходится определять направление и величину шага коррекции d.

Для нашего случая, когда левая часть нелинейного уравнения – полином $P(z)$, то есть легко дифференцируемая функция, наиболее подходящим будет видимо классический метод Ньютона или его модификации. Алгоритм Ньютона получается из простых геометрических соотношений – шаг коррекции можно определить как катет прямоугольного треугольника, другим катетом которого является значение полинома в текущей точке, а тангенс противолежащего ему угла есть производная полинома в той же текущей точке:

$$d=-P(z_k)/P^{(1)}(z_k),$$

$$z_{k+1}=z_k-P(z_k)/P^{(1)}(z_k),$$ где k – номер итерации.

Можно вычислить производную только в точке первого приближения и использовать ее значение на всех последующих шагах, пожертвовав некоторым снижением скорости сходимости:

$$z_{k+1}=z_k-P(z_k)/P^{(1)}(z_0).$$

Неприятности могут ожидать нас на пологих участках функции, когда первая производная близка к нулевому значению – в этом случае слишком большой шаг коррекции «выбросит» нас в далекую от корня область. Это особенно неприятно, если мы используем только однократное дифференцирование в точке первого приближения – случайное попадание на пологий участок в этом первом приближении вызовет эффект «рыскания» поисковой процедуры с большим шагом. Для устранения этого эффекта можно ограничить величину шага некоторым допустимым значением, например, делая его пропорциональным не первой производной, а той, которая достаточно далека от нулевого значения или использовать другие приемы ухода с пологого участка кривой. В этом случае алгоритм может выглядеть например так:

$$z_{k+1}=z_k+t\left(-\frac{P(z_k)}{P^{(j)}(z_k)}\right)^{\frac{1}{j}},$$

где j – порядок очередной производной с достаточно удаленным от нуля значением, t – коэффициент шага коррекции, выбираемый так, чтобы $P(z_{k+1})$ было меньше $P(z_k)$.

Перед началом поиска корней можно избавиться от кратных делением исходного полинома на его НОД с первой производной, но потом придется определять кратность корней, пока их количество не станет равным порядку полинома.

Методы определения корней для полиномов степеней ниже пятой, известные из курса элементарной алгебры, рассмотрены кратко при описании программной реализации.

1.3.5. Определение программного класса полиномов

Рассмотрим структуру класса для работы с многочленами, алгоритмы для работы с объектами типа «полином» и примеры методов полиномиального класса, реализующие эти алгоритмы:

```
template <class YourOwnFloatType> class polynom: public
vector<YourOwnFloatType>
{
```

Если мы запишем коэффициенты полинома в порядке убывания, включая нулевые, мы получим упорядоченный кортеж длиной $n+1$: $(a_n, a_{n-1}, \ldots, a_2, a_1, a_0)$, то есть не что иное, как вектор размерности $n+1$, полностью характеризующий заданный полином. Поэтому вполне естественным является то, что наш полином будет базироваться на векторе.

Как и векторный класс, он будет параметризованным. При этом тип-параметр будет у нас относиться как к коэффициентам многочлена, так и к подставляемым в него значениям. Внутренний формат для хранения нашего полинома будет $a_0+a_1x+a_2x^2+a_3x^3+a_4x^4+a_5x^5+\ldots+a_{n-1}x^{n-1}$, что обусловлено требованием удобства его индексирования, а внешнее представление будет в канонической форме – ввод и вывод будет производиться, начиная с коэффициента при наивысшей степени.

```
void optimize();
```

В процессе работы с полиномом мы можем прийти к ситуации, когда его порядок необходимо уменьшить в связи с тем, что обнулился коэффициент при старшей степени (или несколько подряд идущих коэффициентов). В связи с этим после каждой операции, которая может привести к изменению степени полинома, производится проверка, записан ли полином в канонической форме. Если нет, мы с помощью этой внутренней функции понижаем его порядок

до тех пор, пока полином не будет преобразован в каноническую форму.

```
polynom<YourOwnFloatType> reverse();
```

В связи с различием внутреннего и внешнего представления полинома иногда бывает необходимо записать полином в обратном порядке.

```
void In(istream &);
void Out(ostream &);
```

По аналогии с векторами переопределим две виртуальные функции ввода и вывода. При этом отпадает необходимость в перегрузке операций потокового ввода-вывода: однажды определённые в векторном классе, они используют именно эти виртуальные функции, а определение, из какого именно класса необходимо их вызывать, осуществляется уже на этапе выполнения, в зависимости от того, к какому типу преобразуется базовый указатель. Так, любой полиномиальный объект можно преобразовать к векторному.

```
public:
```

В этом разделе мы разместим общедоступные дружественные функции и методы полиномиального класса.

```
polynom(char *);
```

Полином из файла? Почему бы и нет! Параметром этого конструктора является имя файла, в котором находятся данные в виде [Степень Многочлена-1] [Свободный Член] [Коэффициент При Первой Степени] … [Коэффициент При Старшей Степени].

```
polynom(long,YourOwnFloatType *);
```

Конструктор, принимающий два параметра – уменьшенную на единицу степень многочлена и указатель на данные – коэффициенты многочлена, начиная со свободного члена. Почему бы не передавать в этот конструктор степень многочлена? Просто показалось удобнее пользоваться понятием не «степень многочлена», а «размерность поли-

нома», то есть количество коэффициентов, его составляющих.

```
polynom(polynom<YourOwnFloatType> &);
```

Конструктор копирования делает слепок с заданного полинома.

```
polynom();
```

Конструктор по умолчанию создаёт полином особого вида – нуль-полином, то есть многочлен размерности 1 (соответственно степени 0), свободный член которого равен нулевому элементу типа-параметра полиномиального класса.

```
polynom(long);
```

В некоторых случаях бывает полезно задать сначала степень полинома, но временно оставить неопределёнными его коэффициенты. Этот конструктор создаёт полином заданной размерности и обнуляет коэффициенты, нарушая каноническую форму записи многочлена. Это бывает необходимо редко и только в тех случаях, когда коэффициенты многочлена становятся известными после того, как задана его степень. Разумеется, и в этом случае параметром конструктора является степень многочлена-1.

```
polynom<YourOwnFloatType> operator-();
```

Полином, противоположный полиному $P(x)=a_nx^n+$ $+a_{n-1}x^{n-1}+\ldots+a_2x^2+a_1x+a_0$, определяется как $-P(x)=-a_nx^n-$ $-a_{n-1}x^{n-1}-\ldots-a_2x^2-a_1x-a_0$, то есть знак коэффициентов при степенях многочлена меняется на противоположный.

```
polynom<YourOwnFloatType> operator+();
```

Унарный плюс – операция, просто возвращающая копию текущего многочлена.

```
friend polynom<YourOwnFloatType> operator+(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Дружественная функция сложения многочленов принимает два параметра – многочлены-слагаемые, и возвращает результирующий многочлен-сумму. Суммой двух

многочленов, $f(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_2x^2+a_1x+a_0$ и $g(x)=$ $=b_mx^m+b_{m-1}x^{m-1}+\ldots+b_2x^2+b_1x+b_0$ степеней n и m соответственно ($n\geq m$) называют многочлен $c_nx^n+c_{n-1}x^{n-1}+\ldots+$ $+c_2x^2+c_1x+c_0$, где $c_i=a_i+b_i$ ($i=0, 1, \ldots, m$), $c_i=a_i$ ($i=m+1, \ldots,$ n). Разумеется, после получения многочлена-суммы его необходимо записать в канонической форме.

```
friend polynom<YourOwnFloatType> operator+=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Дружественная функция, реализующая операцию сокращённого сложения двух многочленов, складывает первый многочлен со вторым и записывает результат в первый, возвращая его копию для дальнейших преобразований.

```
friend polynom<YourOwnFloatType> operator-(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Так как операция сложения многочленов обратима, то мы можем определить вычитание многочленов, используя операции сложения и получения многочлена, противоположного к данному. Таким образом, разностью двух полином $f(x)$ и $g(x)$ является полином $P(x)$ такой, что

$$P(x)=f(x)+(-g(x)).$$

```
friend polynom<YourOwnFloatType> operator–=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

По аналогии с сокращённым сложением, мы можем определить сокращённое вычитание как дружественную функцию, первый параметр которой модифицируется.

```
friend polynom<YourOwnFloatType> operator*(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Произведением двух многочленов, $f(x)=a_nx^n+a_{n-1}x^{n-1}+$ $+\ldots+a_2x^2+a_1x+a_0$ и $g(x)=b_mx^m+b_{m-1}x^{m-1}+\ldots+b_2x^2+b_1x+b_0$, называют многочлен $c_{n+m}x^{n+m}+\ldots+c_1x+c_0$, где

$$c_i=\begin{cases} a_ib_0+a_{i-1}b_1+\ldots+a_0b_i, & i=0,1,\ldots,m; \\ a_ib_0+a_{i-1}b_1+\ldots+a_{i-m}b_m, & i=m+1,\ldots,n; \\ a_nb_{i-n}+a_{n-1}b_{i-n+1}+\ldots+a_{i-m}b_m, & i=n+1,\ldots,\mathbf{n}+m. \end{cases}$$

Кроме того, мы можем находить произведение многочленов по правилу умножения сумм:

$$f(x)\cdot g(x)=\sum_{i=0}^{n}a_i x^i\cdot\sum_{k=0}^{m}b_k x^k .$$

```
friend polynom<YourOwnFloatType> operator*=(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Сокращённое умножение полинома на полином.

```
friend polynom<YourOwnFloatType> operator*( YourOwn-
FloatType, polynom<YourOwnFloatType> &);
friend polynom<YourOwnFloatType> operator*(polynom
<YourOwnFloatType> &, YourOwnFloatType);
```

Умножение числа на полином, равно как и умножение полинома на число – это операция, воздействующая на коэффициенты полинома, и соответствует умножению числа на вектор (вектора на скаляр), то есть:

$$\alpha f(x)=\alpha a_n x^n+\alpha a_{n-1}x^{n-1}+\ldots+\alpha a_2 x^2+\alpha a_1 x+\alpha a_0.$$

```
friend polynom<YourOwnFloatType> operator*=(polynom
<YourOwnFloatType> &, YourOwnFloatType);
```

Сокращённое умножение полинома на число.

```
friend long operator<(polynom<YourOwnFloatType> &, poly-
nom<YourOwnFloatType> &);
friend long operator>(polynom<YourOwnFloatType> &, poly-
nom<YourOwnFloatType> &);
friend long operator<=(polynom<YourOwnFloatType> &, pol-
ynom<YourOwnFloatType> &);
friend long operator>=(polynom<YourOwnFloatType> &, pol-
ynom<YourOwnFloatType> &);
```

Набор операций для сравнения полиномов. Два полинома считаются равными, если они одинаковой степени и коэффициенты при соответствующих степенях x равны. В противном случае полномы не равны. При этом считается, что первый полином меньше второго, если в канонической форме его степень ниже (размерность меньше). Если же эти полиномы одинаковой размерности (а, соответственно, и

степени), то мы последовательно сравниваем их коэффициенты, начиная со старшей степени. Первое различие в коэффициентах и определяет, какой знак необходимо поставить между этими многочленами.

```
YourOwnFloatType &operator[](long);
```

Индексация полинома – операция, принимающая в качестве параметра степень одночлена, коэффициент при котором необходимо вернуть.

```
YourOwnFloatType operator()(YourOwnFloatType);
```

Получить значение полинома в заданной точке мы можем, используя эту функцию. Её задача – либо просто просуммировать произведения коэффициентов многочлена на соответствующую степень аргумента данной функции, либо вычислить это значение по теореме Безу.

```
friend long div(polynom<YourOwnFloatType> &,
polynom<YourOwnFloatType> &,
polynom<YourOwnFloatType> &,
polynom<YourOwnFloatType> &);
```

Из курса алгебры известна теорема, гласящая о том, что для любых двух многочленов $f(x)$ и $g(x)$ существует единственная пара многочленов $l(x)$ и $r(x)$, удовлетворяющих условию

$$f(x)=g(x)l(x)+r(x),$$

где степень $r(x)$ меньше степени $g(x)$ или $r(x)$ – нуль-многочлен.

Для определения вида этих многочленов рассмотрим многочлены

$$f(x)=a_nx^n+a_{n-1}x^{n-1}+\ldots+a_2x^2+a_1x+a_0 \text{ и}$$
$$g(x)=b_mx^m+b_{m-1}x^{m-1}+\ldots+b_2x^2+b_1x+b_0,\ g(x)\neq 0.$$

1. Пусть $n<m$. Тогда справедливо равенство $f(x)=g(x)0+f(x)$, удовлетворяющее искомую пару многочленов.

2. Пусть $n \geq m$. В этом случае составим вспомогательный многочлен $f_1(x) = f(x) - g(x)\frac{a_n}{b_m}x^{n-m}$ степени n_1. Если $n_1 \leq m$, то ограничиваемся одним этим многочленом, если же $n_1 \geq m$, то строим $f_2(x) = f_1(x) - g(x)\frac{a'_{n_1}}{b_m}x^{n_1-m}$ (где a'_{n_1} – старший коэффициент многочлена $f_1(x)$). Если степень n_2 многочлена $f_2(x)$ меньше m, то ограничиваемся построенными двумя многочленами, если же $n_2 \geq m$, то продолжаем (аналогично предыдущему) строить вспомогательные многочлены $f_3(x)$, $f_4(x)$, …, $f_s(x)$, …. При этом степень каждого следующего многочлена меньше степени предыдущего, поэтому на определённом k-м шаге имеем многочлен $f_k(x) = f_{k-1}(x) - g(x)\frac{a^*_{n_{k-1}}}{b_m}x^{n_{k-1}-m}$, степень которого меньше m (или $f_k(x)=0$).

Сложим почленно полученные равенства:

$$f_1(x)+f_2(x)+\ldots+f_k(x)=f(x)+f_1(x)+\ldots+f_{k-1}(x)-$$
$$-g(x)\left(\frac{a_n}{b_m}x^{n-m}+\frac{a'_{n_1}}{b_m}x^{n_1-m}+\ldots+\frac{a^*_{n_{k-1}}}{b_m}x^{n_{k-1}-m}\right).$$

Выражение в скобках представляет собой сумму многочленов, а потому (по определению суммы) также является многочленом. Обозначим его через $l(x)$. Многочлен $f_k(x)$ обозначим $r(x)$. По построению, степень $r(x)$ меньше степени $f(x)$, или $r(x)=0$.

По аналогии, назовём $f(x)$ делимым, $g(x)$ – делителем, $l(x)$ – неполным частным, а $r(x)$ – остатком от деления $f(x)$ на $g(x)$.

```
friend polynom<YourOwnFloatType> operator/(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
friend polynom<YourOwnFloatType> operator%(polynom
<YourOwnFloatType> &, polynom<YourOwnFloatType> &);
```

Используя описанную выше функцию, определим две вспомогательные операторные функции – для нахождения целой части и остатка от деления.

```
long IsEqual(void *);
```

Виртуальная функция с таким же именем, определённая в векторном классе, предназначена для сравнения текущего вектора с вектором, адрес которого передаётся через обобщённый указатель. Та же операция проделывается и для полиномов с одной единственной целью – использовать единожды, в векторном классе, определённые операторные функции == и !=, что увеличивает гибкость и универсальность данного класса за счёт использования механизма виртуальных функций.

```
polynom<YourOwnFloatType> opera-
tor=(polynom<YourOwnFloatType> &);
```

Присвоение полинома-параметра текущему.

```
friend polynom<YourOwnFloatType> derive(polynom
<YourOwnFloatType>, long);
```

Полином, как функция аналитическая, удобен тем, что для него всегда можно найти аналитическую производную. Как известно, производной степенной функции является степенная же функция, поэтому производная от полинома будет тоже полином:

$$\frac{d}{dx}\left(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0\right) = \\ n a_n x^{n-1} + (n-1)a_{n-1}x^{n-2} + \ldots + 2a_2 x + a_1 .$$

Параметром данной функции является порядок производной – количество раз, которое полином дифференцируется.

```
friend polynom<YourOwnFloatType> integral(polynom
<YourOwnFloatType>, long);
```

По аналогии с аналитической производной для полинома можно определить аналитический интеграл. Первообразная степенной функции – степенная же функция степени

на единицу выше, поэтому интеграл от полинома также будет полиномом:

$$\int\left(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_2 x^2 + a_1 x + a_0\right)dx =$$

$$\frac{a_n}{n+1}x^{n+1} + \frac{a_{n-1}}{n}x^n + \ldots + \frac{a_2}{3}x^3 + \frac{a_1}{2}x^2 + a_0 x.$$

Параметром данной функции является кратность интегрирования – количество раз, которое полином интегрируется.

```
friend polynom<YourOwnFloatType> pow(polynom
<YourOwnFloatType>, unsigned int);
polynom<YourOwnFloatType> operator^(unsigned int);
```

Используя введенную ранее операцию умножения, мы можем определить степень полинома как перегруженную операцию или как дружественную функцию.

```
polynom<YourOwnFloatType> operator^=(unsigned int);
```

Сокращённая степень.

```
};
```

От параметризованных полиномов, тип которых определяется подстановкой в классовые уголки, перейдём к комплексным полиномам вида

$$f(z)=a_n z^n+a_{n-1}z^{n-1}+\ldots+a_2 z^2+a_1 z+a_0,$$

подставив тип «комплексное число»:

```
typedef polynom<complex> cpolynom;
typedef vector<complex> cvector;
typedef matrix<complex> cmatrix;
```

Значения z, при подстановке которых многочлен обращается в нуль, называются *корнями* этого многочлена. Таким образом, корни $f(z)$ – это решения уравнения:

$$f(z)=a_n z^n+a_{n-1}z^{n-1}+\ldots+a_2 z^2+a_1 z+a_0=0,$$

называемого алгебраическим уравнением n-й степени.

По теореме Безу, остаток, получаемый при делении многочлена $f(z)$ на $(z–a)$, равен $f(a)$. В частности, для того

чтобы многочлен $f(z)$ делился на $(z–a)$ без остатка, необходимо и достаточно условие $f(a)=0$, т. е. для того чтобы многочлен делился на двучлена $(z–a)$ без остатка, необходимо и достаточно, чтобы $z=a$ было корнем этого многочлена.

Таким образом, зная корень $z=a$ многочлена $f(z)$, мы можем выделить из этого многочлена множитель $(z–a)$: $f(z)=(z–a)f_1(z)$, где $f_1(z)=b_{n-1}z^{n-1}+b_{n-2}z^{n-2}+\ldots+b_2z^2+b_1z+b_0$ $(b_{n-1}=a_{n-1})$; нахождение остальных корней приводит к решению уравнения

$$b_{n-1}z^{n-1}+b_{n-2}z^{n-2}+\ldots+b_2z^2+b_1z+b_0=0$$

$(n–1)$-й степени.

Продолжая этот процесс, мы получим окончательно следующее разложение $f(z)$ на множители: $f(z)=a_0(z–z_1)(z––z_2)\ldots(z–z_n)$, т.е. всякий многочлен n-й степени разлагается на $(n+1)$ множителей, один из которых равен старшему коэффициенту, а остальные – двучлены первой степени вида $(z–a)$. При подстановке $z=z_s$ $(s=1, 2, ..., n)$ по крайней мере один из множителей в разложении обратится в нуль, т. е. значения $z=z_s$ суть корни $f(z)$.

Рассмотрим частные методы решения алгебраических уравнений в радикалах, а также метод Ньютона для нахождения всех корней комплексного полинома.

```
cvector square(complex, complex, complex);
cvector kardano(double, double, double, double);
cvector ferrary(double, double, double, double, double);
```

Пусть задано алгебраическое уравнение степени n коэффициентами определяющего его многочлена. Количество корней этого уравнения будет равно степени многочлена, и в общем случае эти корни будут комплексными, поэтому возвращаемым значением функции, предназначенной для нахождения корней полинома, будет вектор размерности n, компоненты которого суть корни полинома.

Первая из рассматриваемых функций предназначена для решения алгебраического уравнения второй степени, или, как его ещё называют, квадратного уравнения. Пара-

метрами его являются коэффициенты при второй, первой и нулевой степенях соответственно. Алгоритм решения такого уравнения определяется следующими соотношениями:

$$a_2x^2 + a_1x + a_0 = a_2\left(x^2 + \frac{a_1}{a_2}x + \frac{a_0}{a_2}\right) =$$

$$= a_2\left(x^2 + 2\frac{a_1}{2a_2}x + \frac{a_1^2}{4a_2^2} - \frac{a_1^2}{4a_2^2} + \frac{a_0}{a_2}\right) =$$

$$= a_2\left(\left(x^2 + 2\frac{a_1}{2a_2}x + \frac{a_1^2}{4a_2^2}\right) - \frac{a_1^2}{4a_2^2} + \frac{a_0}{a_2}\right) =$$

$$= a_2\left(\left(x + \frac{a_1}{2a_2}\right)^2 - \frac{a_1^2}{4a_2^2} + \frac{4a_0a_2}{4a_2^2}\right) =$$

$$= a_2\left(x + \frac{a_1}{2a_2}\right)^2 + a_2\frac{4a_0a_2 - a_1^2}{4a_2^2} = 0 \Rightarrow$$

$$\Rightarrow \left(x + \frac{a_1}{2a_2}\right)^2 = -\frac{4a_0a_2 - a_1^2}{4a_2^2} \Rightarrow$$

$$\Rightarrow \left(x + \frac{a_1}{2a_2}\right)^2 = \frac{a_1^2 - 4a_0a_2}{4a_2^2} \Rightarrow$$

$$\Rightarrow \left|x + \frac{a_1}{2a_2}\right| = \sqrt{\frac{a_1^2 - 4a_0a_2}{4a_2^2}} \Rightarrow$$

$$\Rightarrow \left|x + \frac{a_1}{2a_2}\right| = \frac{\sqrt{a_1^2 - 4a_0a_2}}{2a_2} \Rightarrow$$

$$\Rightarrow x + \frac{a_1}{2a_2} = \pm\frac{\sqrt{a_1^2 - 4a_0a_2}}{2a_2} \Rightarrow$$

$$\Rightarrow x = -\frac{a_1}{2a_2} \pm \frac{\sqrt{a_1^2 - 4a_0a_2}}{2a_2} \Rightarrow$$

$$x = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_0 a_2}}{2a_2}$$

Для решения уравнений третьей и четвёртой степеней с комплексными коэффициентами общих методов нет. Однако существуют два частных метода для решения уравнений этих степеней с действительными коэффициентами, метод Кардано-Тартальи для решения уравнений третей степени и метод Феррари для решения уравнений четвёртой степени. Так как эти методы не охватывают все уравнения данного класса, мы на них останавливаться не будем, однако приведём ниже пример их реализации.

```
cvector newton(cpolynom);
```

Пусть необходимо найти все корни уравнения $P(z)=0$. Для численного решения можно воспользоваться итерационным методом Ньютона. Замечательной особенностью этого метода является его гарантированная сходимость на всей комплексной плоскости для выпуклых функций (а полиномы степени выше первой – функции выпуклые), причём скорость сходимости не зависит от начального приближения. Для начала итерации зададимся точностью ε, с которым нам необходимо найти решение, и начальным приближением к некоторому корню $z_0=x_0+iy_0$. Алгоритм вычислений может быть таким:

1. Находим производную полинома в точке z_k (k – номер итерации). Если она равна нулю, дифференцируем $P(z)$ дальше до тех пор, пока $P^{(j)}(z_k)$ не станет отличной от нуля.
2. Следующее приближение к корню находим по формуле $z_{k+1} = z_k + t\left(-\frac{P(z_k)}{P^{(j)}(z_k)}\right)^{\frac{1}{j}}$, где параметр t подбирается так, чтобы выполнялось условие: $|P(z_{k+1})|<|P(z_k)|$.
3. Проверяем выполнение условия $|P(z_{k+1})|<\varepsilon$; если условие не выполняется, переходим к пункту 1. Если условие

выполняется, то z_{k+1} считаем корнем уравнения, полином $P(z)$ делится на двучлен $(z–z_k)$ и получаем полином $(n–1)$-ой степени, для которого повторяем все вычисления, начиная с п. 1.

Описанные алгоритмы, к примеру, можно реализовать так:

```
#ifndef __EQUATION_H
#define __EQUATION_H

/*Этот файл включения содержит объявление двух типов - комплексного полинома и комплексного вектора, а также заголовки четырёх функций для решения уравнений 2-ой, 3-ей, 4-ой и высших степеней */


#ifndef __COMPLEX_H
#include <complex.h>
#endif
#ifndef __POLYNOM_H
#include "polynom.h" //будет определен позже
#endif
#ifndef __VECTOR_H
#include "vector.h"
#endif
#ifndef __MATRIX_H
#include "matrix.h"
#endif

typedef polynom<complex> cpolynom;
typedef vector<complex> cvector;
typedef matrix<complex> cmatrix;

cvector square(complex, complex, complex);
cvector kardano(double, double, double, double);
cvector ferrary(double, double, double, double,
double);
cvector newton(cpolynom);

#endif
```

```
#ifndef __EQUATION_H
#include "equation.h"
#endif
/*Решение квадратного уравнения с комплексными ко-
эффициентами */
cvector square(complex a2,complex a1,complex a0)
{
  complex a=a2,b=a1,c=a0;
 /*Найденные комплексные корни должны быть записаны
в двухкомпонентный комплексный вектор-результат */
  cvector res(2);
  //проверим, не нулевой ли коэффициент при x^2
  if(a!=complex(0,0))
  {
    /*если да, ищем корни через квадратичный дис-
криминант*/
    complex D=b*b-4*a*c;
    res[0]=(-b+sqrt(D))/(2*a);
    //и заносим их в соответствующие
    res[1]=(-b-sqrt(D))/(2*a);
    //компоненты вектора-результата
  }
  else
    res[0]=res[1]=-c/b;/*иначе решаем уравнение
первой степени*/
  return res;
}


/*Решение кубического уравнения с действительными
коэффициентами методом Кардано-Тартальи*/
cvector kardano(double a3, double a2, double a1,
double a0)
{
  /*Общий вид уравнения, корни которого мы ищем -
a3*x^3+a2*x^2+a1*x+a0=0. Так как уравнение имеет
третью степень, то и корней у него три, что и опре-
деляет размерность вектора-результата*/
  cvector res=3;
  if(a3==0)//если коэффициент при x^3 нулевой,
  {
    //решаем соответствующее квадратное уравнение
    cvector res2=square(a2,a1,a0);
```

```
    /*в этом случае принимаем, что два корня явля-
ются совпадающими */
    res[0]=res[1]=res2[0];
    res[2]=res2[1];
  }
  else
  {
    /*иначе - приводим кубическое уравнение к кано-
ническому виду, когда коэффициент при третьей сте-
пени равен 1 */
    double a=a2/a3,b=a1/a3,c=a0/a3;
    //находим слагаемые кубического дискриминанта
    double p=b-a*a/3,q=2*a*a*a/27-a*b/3+c;
    /*проведя переобозначение x=y-a/3, решаем в
дальнейшем уравнение y^3+p*y+q=0 */
    //находим кубический дискриминант
    double diskr=pow(q/2,2)+pow(p/3,3);
    /*если дискриминант равен 0, то имеем 3 дей-
ствительных корня, из них два совпадающих */
    if(diskr==0)
    {
      res[0]=3*q/p;
      res[1]=res[2]=-res[0]/2;
    }
    /*один действительный корень и два комплексно
сопряжённых*/
    if(diskr>0)
    {
      double what=-q/2+sqrt(diskr);
      double u0=(what>0)?pow(what,1.0/3.0):-pow(-
what, 1.0/3.0);
      double v0=-p/(3*u0);
      res[0]=u0+v0;
      res[1]=res[2]=-res[0]/2;
      complex k3(0,sqrt(3)*(u0-v0)/2);
      res[1]+=k3;
      res[2]-=k3;
    }
    //три различных действительных корня
    if(diskr<0)
    {
      cvector zkub12=square(1,q,-p*p*p/27);
      complex u0=pow(zkub12[0],1/3.);
```

```
      res[0]=2*real(u0);
      res[1]=-real(u0)-imag(u0)*sqrt(3);
      res[2]=-real(u0)+imag(u0)*sqrt(3);
    }
    //переходим обратно от y к x
    for(int i=0;i<3;i++)
      res[i]-=a/3;
  }
  return res;//возвращаем вектор-результат
}


/*Метод Феррари для решения уравнений четвёртой
степени с действительными коэффициентами*/
cvector ferrary(double a4, double a3, double a2,
double a1, double a0)
{
  cvector res=4;//всего корней будет 4
  if(!a4)/*если коэффициент при четвёртой степени x
нулевой, пытаемся решить уравнение третьей степе-
ни*/
  {
    cvector res3=kardano(a3,a2,a1,a0);
    res[0]=res[1]=res3[0];
    for(int i=1;i<3;i++)
      res[i+1]=res3[i];
  }
  else
  {
    double a=a3/a4,b=a2/a4,c=a1/a4,d=a0/a4;
    /*составляем и находим корни кубической резоль-
венты*/
    cvector cy0=kardano(1,-b,-4*d+a*c,-d*a*a+4*b*d-
c*c);
    double y0;
    /*ищем хотя бы один действительный корень, с
помощью которого сводим уравнение четвёртой степени
к совокупности квадратных*/
    for(int i=0;i<3;i++)
      if(!imag(cy0[i]))
      {
        y0=real(cy0[i]);
        break;
```

```
      }
    double A=a*a/4-b+y0,B=a*y0/2-c;
    double x12=-B/(2*A);
    //решаем квадратные уравнения
    cvector sq1=square(1,a/2-sqrt(A),y0/2+sqrt(A)
*x12);
    cvector sq2=square(1,a/2+sqrt(A),y0/2-sqrt(A)
*x12);
    for(long i=0;i<2;i++)
      res[i]=sq1[i],res[i+2]=sq2[i];
  }
  return res;//возвращаем результат
}


/*модифицированный метод Ньютона поиска комплексных
корней полинома*/
cvector newton(cpolynom p)
{
  double t=1,eps=1e-8,j;

  p*=1;/* выполняем умножение на 1 для того, чтобы
отбросить ведущие нули */
  //результирующий вектор будет иметь размерность,
  //равную порядку полинома
  cvector result=p.getm()-1;
  /*если вектор-результат одномерный, то имеем дело
с полиномом первой степени*/
  if(result==cvector())
  {
    result[0]=-p[0]/p[1];
    return result;
  }
  //если двумерный - с квадратным уравнением и т.д.
  if(result.getm()==2)
    return square(p[2],p[1],p[0]);
  if(result.getm()==3&&!imag(p[3])&&!imag(p[2])&&
!imag(p[1])&&!imag(p[0]))
    return kardano(real(p[3]), real(p[2]), re-
al(p[1]), real(p[0]));
  if(result.getm()==4&&!imag(p[4])&&!imag(p[3])&&
!imag(p[2])&&!imag(p[1])&&!imag(p[0]))
```

```
    return ferrary(real(p[4]), real(p[3]), re-
al(p[2]), real(p[1]), real(p[0]));
  complex z=0;/*начальное приближение к корню поло-
жим равным (0,0)*/
  do
  {
    complex dz=0;
    /*находим порядок и значение ненулевой произ-
водной в точке z */
    for(j=1;dz==complex(0);dz=derive(p,j++)(z));
    z+=t*pow(-p(z)/dz,1/(--j));/*модифицируем при-
ближение*/
    t*=(1-eps);
  }while(abs(p(z))>=eps);
  //повторяем до достижения заданной точности
  /*В этом цикле находится только один корень z.
Для нахождения остальных делим исходный полином на
x-z, понижая тем самым степень на единицу, и нахо-
дим ещё один корень, и т.д. до первой степени*/
  result[0]=z;
  cpolynom z1=2;
  z1[0]=-z,z1[1]=1;
  cvector more=newton(p/z1);
  for(long i=1;i<result.getm();i++)
    result[i]=more[i-1];
  return result;//возвращаем вектор результата
}
```

По приведённым алгоритмам работы с полиномиальными объектами можно составить, к примеру, такой интерфейс для данного класса:

```
#ifndef __POLYNOM_H
#define __POLYNOM_H
#ifndef __VECTOR_H
#include "vector.h"
#endif
#ifndef __IOSTREAM_H
#include <iostream.h>
#endif
#define max(a,b)    (((a) > (b)) ? (a) : (b))
```

```
#define min(a,b)    (((a) < (b)) ? (a) : (b))

/*
 Параметризованный класс для полиномов
 Внутренний формат:
 a0+a1*x+a2*x^2+a3*x^3+a4*x^4+a5*x^5+...+a(n-
1)*x^(n-1)
 Ввод и вывод производится, начиная с коэффициента
при наивысшей степени
*/
//Наш полином будет базироваться на векторе
template <class YourOwnFloatType>
class polynom: public vector<YourOwnFloatType>
{
  //эти функции являются внутренними
  void optimize();/*преобразование полинома в кано-
ническую форму*/
  polynom<YourOwnFloatType> reverse();/*запись по-
линома в обратном порядке*/
  void In(istream &);/*по аналогии с векторами -
функции ввода*/
  void Out(ostream &);/*    и вывода*/
 public:
  polynom(char *);//полином из файла
  polynom(long, YourOwnFloatType *);/*полином из
массива*/
  polynom(polynom<YourOwnFloatType> &);
/*конструктор копирования*/
  polynom();//конструктор по умолчанию
  polynom(long);//полином заданной степени
  polynom<YourOwnFloatType> operator-();/*унарный
минус*/
  polynom<YourOwnFloatType> operator+();/*унарный
плюс*/
  friend polynom<YourOwnFloatType> opera-
tor+(polynom <YourOwnFloatType> &, poly-
nom<YourOwnFloatType> &); //сложение
  friend polynom<YourOwnFloatType> operator+= (pol-
ynom <YourOwnFloatType> &, polynom <YourOwn-
FloatType> &); //сокращённое сложение
  friend polynom<YourOwnFloatType> operator-
(polynom <YourOwnFloatType> &, poly-
nom<YourOwnFloatType> &); //вычитание
```

```
  friend polynom<YourOwnFloatType> operator-= (pol-
ynom <YourOwnFloatType> &, polynom <YourOwn-
FloatType> &); //сокращённое вычитание
  friend polynom<YourOwnFloatType> operator* (poly-
nom <YourOwnFloatType> &, polynom <YourOwn-
FloatType> &); /*умножение полинома на полином*/
  friend polynom<YourOwnFloatType> operator*= (pol-
ynom <YourOwnFloatType> &, polynom <YourOwn-
FloatType> &); /*сокращённое умножение на полином*/
  friend polynom<YourOwnFloatType> operator*
(YourOwnFloatType, polynom<YourOwnFloatType> &);
//умножение числа на полином
  friend polynom<YourOwnFloatType> opera-
tor*(polynom <YourOwnFloatType> &, YourOwn-
FloatType); //умножение полинома на число
  friend polynom<YourOwnFloatType> operator*= (pol-
ynom <YourOwnFloatType> &, YourOwnFloatType);
//сокращённое умножение на число
  //набор операций для сравнения полиномов
  friend long operator<(polynom<YourOwnFloatType>
&, polynom<YourOwnFloatType> &);
  friend long operator>(polynom<YourOwnFloatType>
&, polynom<YourOwnFloatType> &);
  friend long operator<=(polynom<YourOwnFloatType>
&, polynom<YourOwnFloatType> &);
  friend long operator>=(polynom<YourOwnFloatType>
&, polynom<YourOwnFloatType> &);
  YourOwnFloatType &operator[](long);/*индексация
полинома*/
  YourOwnFloatType operator()(YourOwnFloatType);
//значение полинома в заданной точке
  friend long div(polynom<YourOwnFloatType> &, pol-
ynom<YourOwnFloatType>&, polynom<YourOwnFloatType>
&, polynom<YourOwnFloatType> &);//деление
  friend polynom<YourOwnFloatType> opera-
tor/(polynom <YourOwnFloatType> &, polynom
<YourOwnFloatType> &); //целая часть от деления
  friend polynom<YourOwnFloatType> opera-
tor%(polynom <YourOwnFloatType> &, poly-
nom<YourOwnFloatType> &); //остаток от деления
  long IsEqual(void *);//проверка на равенство
  polynom<YourOwnFloatType> operator=
(polynom<YourOwnFloatType> &);//присвоение
```

```
  friend polynom<YourOwnFloatType> derive
(polynom<YourOwnFloatType>,long);//производная
  friend polynom<YourOwnFloatType> integral
(polynom<YourOwnFloatType>,long);//интеграл
  friend polynom<YourOwnFloatType> pow
(polynom<YourOwnFloatType>,unsigned int);//степень
  polynom<YourOwnFloatType> operator^(unsigned
int); //степень как операция
  polynom<YourOwnFloatType> operator^=(unsigned
int); //сокращённая степень
};


/*конструкторы полинома будут аналогичны конструк-
торам вектора*/
//полином из файла
template <class YourOwnFloatType>
polynom<YourOwnFloatType>:: polynom(char *f): vec-
tor<YourOwnFloatType>(f)
{
}


//полином степени a-1
template <class YourOwnFloatType>
polynom<YourOwnFloatType>::polynom(long a): vec-
tor<YourOwnFloatType>(a)
{
}


//нуль-полином
template <class YourOwnFloatType>
polynom<YourOwnFloatType>::polynom(): vec-
tor<YourOwnFloatType>()
{
}


/*полином (a-1)-ой степени с коэффициентами из мас-
сива v*/
```

```
template <class YourOwnFloatType> polynom <YourOwn-
FloatType>:: polynom(long a, YourOwnFloatType *v):
vector<YourOwnFloatType>(a,v)
{
}


//конструктор копирования
template <class YourOwnFloatType> polynom <YourOwn-
FloatType>:: polynom(polynom<YourOwnFloatType>
&ex): vector<YourOwnFloatType>(ex)
{
}


/*для индексации полинома вызываем соответствующий
метод векторного класса*/
template <class YourOwnFloatType> YourOwnFloatType
&polynom<YourOwnFloatType>::operator[](long a)
{
  return (*(vector<YourOwnFloatType>*)this)[a];
}


//вычисление значения полинома в точке x
template <class YourOwnFloatType> YourOwnFloatType
polynom<YourOwnFloatType>:: opera-
tor()(YourOwnFloatType x)
{
/*
  YourOwnFloatType temp=0,px=1;
  for(long i=0;i<getm();i++,px*=x)
    temp+=(*this)[i]*px;
  return temp;
*/
  polynom<YourOwnFloatType> temp=2;
  temp[0]=-x, temp[1]=1;
  return ((*this)%temp)[0];
}


//унарный минус
template <class YourOwnFloatType>
```

```
polynom<YourOwnFloatType> poly-
nom<YourOwnFloatType>:: operator-()
{
  return *(polynom*)
         &(-(*(vector<YourOwnFloatType>*) this));
}


//унарный плюс - просто возвращаем себя
template <class YourOwnFloatType>
polynom<YourOwnFloatType> poly-
nom<YourOwnFloatType>::operator+()
{
  return *this;
}


//сложение двух полиномов
template <class YourOwnFloatType> polynom <YourOwn-
FloatType> operator+(polynom<YourOwnFloatType> &f,
polynom<YourOwnFloatType> &s)
{
  long ms=max(f.getm(),s.getm());
  polynom<YourOwnFloatType> temp(ms);
  //создаём временный полином
  /*здесь работают операции индексирования для всех
трёх полиномов*/
  for(long i=ms-1;i>=min(f.getm(),s.getm());i--)
    temp[i]=(f.getm()>s.getm())?f[i]:s[i];
  for(long i=min(f.getm(),s.getm())-1;i>=0;i--)
    temp[i]=f[i]+s[i];
  temp.optimize();
  /*пока есть, удаляем 0-коэффициент при старшей
степени */
  return temp;
}


//сокращённое сложение, определяемое через обычное
template <class YourOwnFloatType> polynom <YourOwn-
FloatType> operator+=
(polynom <YourOwnFloatType> &f,
polynom <YourOwnFloatType> &s)
```

```
{
  return f=f+s;
}


/*вычитание полиномов, выраженное через сложение и
отрицание*/
template <class YourOwnFloatType> poly-
nom<YourOwnFloatType> operator-
(polynom <YourOwnFloatType> &f, poly-
nom<YourOwnFloatType> &s)
{
  return f+(-s);
}


//сокращённое вычитание
template <class YourOwnFloatType> poly-
nom<YourOwnFloatType> operator-=
(polynom <YourOwnFloatType> &f, poly-
nom<YourOwnFloatType> &s)
{
  return f=f-s;
}


//умножение полиномов
template <class YourOwnFloatType>
polynom<class YourOwnFloatType> opera-
tor*(polynom<YourOwnFloatType> &f, poly-
nom<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType>
temp(f.getm()+s.getm());
  for(long i=0;i<f.getm();i++)
    for(long j=0;j<s.getm();j++)
      temp[i+j]+=f[i]*s[j];
  temp.optimize();
  return temp;
}


//сокращённое умножение
```

```
template <class YourOwnFloatType>
polynom<class YourOwnFloatType> opera-
tor*=(polynom<YourOwnFloatType> &f, poly-
nom<YourOwnFloatType> &s)
{
  return f=f*s;
}


//умножение числа на полином
template <class YourOwnFloatType> poly-
nom<YourOwnFloatType> operator*(YourOwnFloatType
ld, polynom<YourOwnFloatType> &v)
{
  polynom<YourOwnFloatType> temp=v;
  temp=*(polynom<YourOwnFloatType>*)
    &((*(vector<YourOwnFloatType>*)(&temp))*ld);
  temp.optimize();
return temp;
}


//умножение полинома на число
template <class YourOwnFloatType>
polynom <YourOwnFloatType> opera-
tor*(polynom<YourOwnFloatType> &v, YourOwnFloatType
ld)
{
  return ld*v;
}


//сокращённое умножение на число
template <class YourOwnFloatType>
polynom <YourOwnFloatType> operator*=
(polynom <YourOwnFloatType> &v,YourOwnFloatType ld)
{
  return v=v*ld;
}


//присвоение
```

```
template <class YourOwnFloatType> polynom <YourOwn-
FloatType> polynom<YourOwnFloatType>:: opera-
tor=(polynom<YourOwnFloatType> &x)
{
  //присваиваем как вектора
  (*(vector<YourOwnFloatType>*)this)=(*(vector
<YourOwnFloatType>*)&x);
  optimize();//приводим к нормальному виду
  return *this;//и возвращаем результат
}


//установление актуальной степени многочлена
template <class YourOwnFloatType>
void polynom<YourOwnFloatType>::optimize()
{
  if(getm()!=1)//если полином не нулевой степени
  {
    if((*this)[getm()-
1]==(YourOwnFloatType)0)/*если коэффициент при
наивысшей степени нулевой*/
    {
      polynom<YourOwnFloatType> temp(getm()-1);
//создаём временный полином степени на 1 меньше
      for(long i=0;i<getm()-1;i++)
        temp[i]=(*this)[i];
      *this=temp;//переписываем его в текущий
      optimize();//снова проверяем полином
    }
  }
}


/*в очень редких случаях может быть необходимым ре-
версировать полином*/
template <class YourOwnFloatType>
polynom<YourOwnFloatType> poly-
nom<YourOwnFloatType>::reverse()
{
  polynom temp=*this;
  for(long i=0;i<getm();i++)
    temp[i]=(*this)[getm()-i-1];
  return temp;
```

```
}


//неотрицательная степень полинома как функция
template <class YourOwnFloatType>
polynom<YourOwnFloatType>
pow(polynom<YourOwnFloatType> x, unsigned int p)
{
  polynom<YourOwnFloatType> temp;
  if(p==0)
    temp[0]=1;/* полином в нулевой степени - это
просто число 1*/
  else
  {
    temp=x;
    for(long i=0;i<p-1;i++)
      temp*=x;
  }
  return temp;//возвращаем результирующий полином
}


//производная j-го порядка
template <class YourOwnFloatType>
polynom<YourOwnFloatType> de-
rive(polynom<YourOwnFloatType> p, long j)
{
  if(j==0)/*нулевая производная полинома есть сам
полином*/
    return p;
  if(j<0)/*отрицательная производная интерпретиру-
ется как интеграл*/
    return integral(p,-j);
  if(p.getm()==1)
    return polynom<YourOwnFloatType>();
    //если более понижать некуда
  polynom<YourOwnFloatType> result=p.getm()-1;
  for(long i=0;i<result.getm();i++)
  //вычисляем первую производную
    result[i]=(i+1)*p[i+1];
  if(j==1)//если её и надо было найти -
    return result;//возвращаем результат
  else//иначе
```

```
    return derive(result,j-1);
    //вычисляем производную от данной
}


//интеграл от полинома
template <class YourOwnFloatType>
polynom<YourOwnFloatType> inte-
gral(polynom<YourOwnFloatType> p, long j)
{
  if(j<=0)
  /*отрицательная кратность интегрирования тракту-
ется как производная*/
    return derive(p,-j);
  polynom<YourOwnFloatType> result=p.getm()+1;
  for(long i=0;i<p.getm();i++)
    result[i+1]=p[i]/(i+1);
  if(j==1)
    return result;
  else
    return integral(result,j-1);
}


//степень как операция
template <class YourOwnFloatType> poly-
nom<YourOwnFloatType> polynom<YourOwnFloatType>::
operator^(unsigned int p)
{
  return pow(*this,p);
}


//сокращённая степень
template <class YourOwnFloatType>
polynom<YourOwnFloatType>
polynom<YourOwnFloatType>::
operator^=(unsigned int p)
{
  return (*this)=pow(*this,p);
}
```

```
/*Перегружать операторы ввода из потока и вывода в
поток необходимости нет - достаточно перегрузить
две виртуальные функции ввод полинома из потока,
начиная с коэффициентов при старших степенях*/
template <class YourOwnFloatType>
void polynom<YourOwnFloatType>::In(istream &is)
{
  for(long i=getm()-1;i>=0;i--)
    is>>(*this)[i];
  optimize();
}


/*вывод полинома в поток, начиная с коэффициентов
при старших степенях*/
template <class YourOwnFloatType>
void polynom<YourOwnFloatType>::Out(ostream &os)
{
  for(long i=getm()-1;i>=0;i--)
  {
    os.precision(100);
    //устанавливаем огромную точность вывода
    os<<(*this)[i]<<" ";/*компоненты разделяем про-
белами*/
  }
}


//сравнение текущего полином с лежащим по адресу x
template <class YourOwnFloatType>
long polynom<YourOwnFloatType>::IsEqual(void *x)
{
  polynom<YourOwnFloatType>
test1=*(polynom<YourOwnFloatType>*)x,
test2=(*this);
  test1.optimize();
  test2.optimize();
  return ((vector<YourOwnFloatType>*)&test1)->
     vector<YourOwnFloatType>::IsEqual(&test2);
}


/*
```

```
  Из двух многочленов одинаковой длины меньше тот,
у которого коэффициент при старшей степени меньше.
При равенстве рассматриваем более низкую степень и
т.д.
*/
template <class YourOwnFloatType>
long operator<(polynom<YourOwnFloatType> &f, poly-
nom <YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> test1=f,test2=s;
  test1.optimize();
  test2.optimize();
  if(test1.getm()<test2.getm())
    return 1;
  if(test1.getm()>test2.getm()||test1==test2)
    return 0;
  //точно не равны - выясняем, кто же из них меньше
  for(long i=test1.getm()-1;i>=0;i--)
    if(test1[i]<test2[i])
      return 1;
    else
      if(test1[i]>test2[i])
        return 0;
  return 0;
}


/*все остальные операции определяем через уже из-
вестные*/
template <class YourOwnFloatType>
long operator>(polynom<YourOwnFloatType> &f, poly-
nom <YourOwnFloatType> &s)
{
  return (!(f<s))&&(f!=s);
}


template <class YourOwnFloatType>
long operator<=(polynom<YourOwnFloatType> &f, poly-
nom <YourOwnFloatType> &s)
{
  return (f<s)||(f==s);
}
```

```
template <class YourOwnFloatType>
long operator>=(polynom<YourOwnFloatType> &f, poly-
nom <YourOwnFloatType> &s)
{
  return (f>s)||(f==s);
}


/* деление полинома на полином по алгоритму Евклида
*/
template <class YourOwnFloatType> long div(polynom
<YourOwnFloatType> &f, polynom<YourOwnFloatType>
&g, polynom<YourOwnFloatType> &l,
polynom<YourOwnFloatType> &r)
{
  polynom<YourOwnFloatType> tf=f,tg=g;
  tf.optimize();
  tg.optimize();
  if(tg==polynom<YourOwnFloatType>())
  //попытка деления на ноль
    throw xmsg("Попытка деления на 0\n");
  if(tf.getm()<tg.getm())
  {
    l=polynom<YourOwnFloatType>();
    r=tf;
    return 1;
  }
  for(l=polynom<YourOwnFloatType>();tf.getm()>=
tg.getm();)
  {
    polynom<YourOwnFloatType> x(2);
    x[1]=1;
    x^=tf.getm()-tg.getm();
    x*=tf[tf.getm()-1]/tg[tg.getm()-1];
    r=tf-=tg*x;
    l+=x;
  }
  return 1;
}
```

```
//целая часть от деления
template <class YourOwnFloatType> polynom <YourOwn-
FloatType> operator/(polynom<YourOwnFloatType> &f,
polynom<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> l,r;
  div(f,s,l,r);
  return l;
}


//остаток от деления
template <class YourOwnFloatType> polynom <YourOwn-
FloatType> operator%(polynom<YourOwnFloatType> &f,
polynom<YourOwnFloatType> &s)
{
  polynom<YourOwnFloatType> l,r;
  div(f,s,l,r);
  return r;
}


#endif
```

2. Матрицы и задачи линейной алгебры

2.1. Общие сведения о матрицах и матричных операциях

Матрицы, матричные операции и вычислительные процедуры линейной алгебры составляют основу инженерного и научного программирования, поэтому мы считаем необходимым напомнить основные положения этих областей, чтобы они были «под рукой» при рассмотрении методов программной реализации матричного класса.

Мы уже определили вектор $\boldsymbol{x}(x_1, x_2, \ldots, x_n)$ в n-мерном пространстве как последовательность комплексных чисел. ***Линейным преобразованием n-мерного пространства*** называют такое преобразование, которое вызывает переход вектора $\boldsymbol{x}(x_1, x_2, \ldots, x_n)$ в вектор $\boldsymbol{y}(y_1, y_2, \ldots, y_n)$ по формулам:

$$y_i=a_{i1}x_1+a_{i}x_2+\ldots+a_{in}x_n \ (i=1, 2, \ldots, n).$$

Обозначим эту операцию буквой **А**. Очевидно, она может быть записана как система n векторов

$$a_i(a_{i1}, a_{i2}, \ldots, a_{in}),$$

осуществляющих эту операцию, и представлена в виде таблицы (матрицы) значений a_{ij}:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix},$$

а формула преобразования может быть записана в компактном виде:

$$\boldsymbol{y}=\mathbf{A}\boldsymbol{x}.$$

Если операция **А** преобразует различные векторы в различные, а это соответствует тому, что определитель матрицы **А** отличен от нуля, то операция и обозначающая ее мат-

рица называются ***неособенными***. В этом случае вектор $\boldsymbol{x}$ может быть получен обратным преобразованием $\mathbf{A}^{-1}$

$$\boldsymbol{x}=\mathbf{A}^{-1}\boldsymbol{y},$$

и обозначающая обратное преобразование матрица $\mathbf{A}^{-1}$ имеет элементы

$$\{\mathbf{A}^{-1}\}_{ij}=\frac{\mathbf{A}_{ij}}{D(\mathbf{A})},$$

где через $D(\mathbf{A})$ обозначен связанный с матрицей ***определитель***, представляющий собой число det **A**, определяемое по известным правилам, а именно:

$$\det \mathbf{A} = \sum_{(\alpha_1,\alpha_2,\ldots,\alpha_n)} (-1)^{\chi} a_{1\alpha_1} a_{2\alpha_2} \ldots a_{n\alpha_n} ,$$

где сумма распространена на всевозможные перестановки $(\alpha_1, \alpha_2, \ldots, \alpha_n)$ элементов 1, 2, …, n и, следовательно, содержит $n!$ слагаемых, причем $\chi=0$, если перестановка четная, и $\chi=1$, если перестановка нечетная.

Через A_{ij} обозначены ***алгебраические дополнения*** определителя относительно элементов a_{ij}, через i и j – индексы (номера) строк и столбцов в матрице.

Если обратная матрица $\mathbf{A}^{-1}$ существует, то матрица **A** является невырожденной; эквивалентными являются такие признаки невырожденности, как неравенство нулю определителя матрицы, линейная независимость вектор-столбцов или вектор-строк матрицы.

Последовательное применение двух операций приводит нас к понятию ***произведения***:

$$\boldsymbol{y}=\mathbf{A}\boldsymbol{x};\ \boldsymbol{z}=\mathbf{B}\boldsymbol{y};\ \boldsymbol{z}=\mathbf{BA}\boldsymbol{x}=\mathbf{C}\boldsymbol{x},$$

а матрица результирующего преобразования **BA** определяется так:

$$\mathrm{C}_{ij}=\{\mathrm{BA}\}_{ij}=\sum_{s=1}^{n}\{\boldsymbol{B}\}_{is}\{\boldsymbol{C}\}_{sj} ,$$

и его результат зависит от порядка сомножителей, т.е. $\mathbf{BA}\neq\mathbf{AB}$.

Очевидно, что *для прямоугольных матриц произведение имеет смысл только при равенстве числа столбцов левого сомножителя числу строк правого*.

Количество строк результирующей матрицы должно быть равно количеству строк левого сомножителя, а количество столбцов – количеству столбцов правого сомножителя.

Если линейное преобразование вызывает только растяжение составляющих любого вектора вдоль координатных осей

$$y_i=k_i x_i \ (i=1, 2, \ldots, n),$$

то оно выражается ***диагональной матрицей***

$$\mathbf{A} = \begin{pmatrix} a_{11} & 0 & \ldots & 0 \\ 0 & a_{22} & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & a_{mn} \end{pmatrix}$$

Если отличный от нуля вектор таков, что осуществляемая матрицей операция преобразования пространства приводит только к изменению его длины без изменения направления, то есть

$$\mathbf{A}\boldsymbol{x}=\lambda\boldsymbol{x}, \ (\lambda \text{ - число}) \text{ или}$$

$$\sum_i a_{ki} x_i - \lambda x_k = 0 \ (k=1, 2, \ldots, n),$$

то он называется ***собственным вектором***, его направление – ***собственным направлением***, а коэффициент изменения его модуля λ – *собственным значением* матрицы **A**.

Последнее уравнение представляет собой однородную систему с матрицей, имеющей равный нулю определитель. После разворачивания определителя получим полином n-го порядка (он носит название характеристического полинома) относительно λ, а приравнивание его нулю дает алгебраическое уравнение для определения всех возможных собственных значений (характеристическое уравнение):

$$\det(\mathbf{A}-\lambda\mathbf{E})=0.$$

Совокупность n собственных значений называют ***спектром матрицы***, а максимальное по модулю собственное значение – ***спектральным радиусом матрицы***. Вычисление собственных значений достаточно сложная задача, но для одного класса матриц, а именно верхних и нижних треугольных (с нулевыми элементами выше или ниже главной диагонали) вычисление вообще не требуется – собственными значениями этих матриц являются элементы главной диагонали.

Если матрица вещественна и симметрична (перемена индексов строк и столбцов не изменяет значение матричного элемента), то все ее собственные значения вещественны. Если, кроме того, матрица является положительно определенной ($\boldsymbol{x}^{\mathrm{T}}\mathbf{A}\boldsymbol{x}>0$ при всех $x\neq0$), то все ее собственные значения положительны.

Две квадратные матрицы **A** и **B** считаются ***подобными***, если существует невырожденная матрица **P** такая, что $\mathbf{B}=\mathbf{PAP}^{-1}$.

Подобные матрицы имеют одинаковые собственные значения.

Мы определили вектор как автономный математический объект, но вполне могли бы считать его частным случаем одностолбцовой или однострочной матрицы и считать для него справедливыми уже рассмотренные операции умножения.

В частности, матрица типа 1хn называется ***вектор-строкой***, а матрица типа mх1 – ***вектор-столбцом***.

Число (скаляр) можно рассматривать как матрицу типа 1х1.

Матрица, все элементы которой равны нулю, называется ***нулевой***. В отличие от единичной, нулевая матрица может быть как квадратной, так и прямоугольной.

В приведенных рассуждениях мы использовали матрицу как математический объект, обозначаемый символом, над которым мы осуществляем действия аналогично дей-

ствиям над обычными числами – по существу, мы *толкуем матрицу как некоторое гиперкомплексное число.*

Но существенны и отличия матричной алгебры от алгебры комплексных чисел – например, некоммутативность операции умножения и обусловленную этим неоднозначность операции деления (если ее рассматривать как умножение матрицы на обратную делителю матрицу, то результат зависит от порядка сомножителей). Еще одной особенностью умножения является возможность получения нулевого результата при обоих ненулевых сомножителях.

Остальные элементы матричной алгебры достаточно просты:

Две матрицы считаются ***равными*** только при равенстве всех их элементов с одинаковыми индексами.

Сложение матриц сводится к суммированию элементов с одинаковыми индексами.

Целые положительные степени матриц получают последовательным умножением матрицы на саму себя

$$\mathbf{A}^p=\mathbf{AA}\ldots\mathbf{A},$$

а ***целые отрицательные степени*** вводятся как целые положительные степени обратной матрицы:

$$\mathbf{A}^{-p}=(\mathbf{A}^{-1})^p.$$

Операция транспонирования подразумевает замену строк столбцами.

2.2. Методы решения основных задач линейной алгебры

В линейной алгебре рассматриваются 4 класса основных задач:

- решение систем линейных алгебраических уравнений (СЛАУ);
- вычисление определителей;
- обращение матриц;

- определение собственных значений и собственных векторов матриц.

Все эти задачи имеют важное прикладное значение при решении различных проблем науки и техники как самостоятельно, так и в качестве вспомогательных алгоритмов в других задачах вычислительной математики, математической физики, обработки результатов экспериментальных исследований.

2.2.1. Методы решения систем линейных алгебраических уравнений (СЛАУ)

СЛАУ $\boldsymbol{y}=\mathbf{A}\boldsymbol{x}$ задается матрицей **A** коэффициентов a_{ij} при неизвестных составляющих x_j вектора решения $\boldsymbol{x}$, где i – порядковый номер уравнения или матричной строки. Справа матрица расширена столбцом вектора свободных членов $\boldsymbol{y}$.

Если ранг матрицы **A** (количество содержащихся в ней линейно-независимых векторов) равен размерности вычисляемого вектора, то общий метод решения нами собственно уже рассмотрен в виде матричной формулы

$$\boldsymbol{x}=\mathbf{A}^{-1}\boldsymbol{y}$$

где $\boldsymbol{x}$ – вектор (матрица-столбец) решения, $\boldsymbol{y}$ – вектор-столбец свободных членов, $\mathbf{A}^{-1}$ – обратная матрица, которую можно получить по методу Крамера.

Но метод Крамера для обращения матрицы, связанный с вычислением определителя и алгебраических дополнений, неэффективен – из-за большого количества арифметических операций даже для небольших матриц (с порядком в несколько десятков) он занимает слишком много компьютерного времени, а для матриц порядка десятков и сотен тысяч становится полностью неприемлем. Поэтому в практических вычислениях обычно используют методы так называемого ***псевдообращения*** – точные или приближенные.

2.2.1.1. Метод Гаусса для решения СЛАУ, вычисления определителей и обращения матриц

Среди прямых методов наиболее простым и популярным при решении систем порядка до 200-400 (в зависимости от быстродействия используемого компьютера) является ***метод Гаусса***, известный под названием *метода последовательного исключения переменных или метода Гауссовых исключений*.

Основывается он на том, что любую вектор-строку в матрице можно заменить ее линейной комбинацией с любой другой вектор-строкой этой же матрицы (это справедливо и для вектор-столбцов). Вычитая последовательно из каждой j-й строки каждую вышележащую i-ю ($i<j$), предварительно умноженную на (a_{ji}/a_{ii}), мы удалим из неё составляющую вектора $\boldsymbol{x}$ с номерами меньше j. Таким образом, все строки, кроме первой, будут содержать ненулевые элементы только вправо начиная с главной диагонали – матрица превратится в верхнюю треугольную, а в последней строке будет только один элемент, что соответствует уравнению вида $c_{nn}x_n=d_n$, из которого можно без труда вычислить x_n.

На этом завершается так называемый *прямой ход алгоритма Гаусса*. *Обратный ход* начинается с подстановки значения $x_n a_{n-1,n}$ в строку с номером n-1, вычисления значения x_{n-1} и так далее до первой строки и вычисления x_1 и завершения решения.

В связи с наличием в алгоритме операции деления на диагональный элемент перед выполнением этой операции среди строк с номером больше i отыскивается строка с наибольшим по абсолютному значению значением коэффициента в i-м столбце и меняется местами с i-й. Этим исключается опасность деления на нуль и матрица по возможности приближается к диагонально-преобладающей, что повышает устойчивость получаемого решения. Эта операция носит название «***выбор главного элемента***».

Если найденный наибольший по абсолютному значению элемент столбца, являющийся кандидатом на замену, равен 0, то матрица вырождена и вычисления следует прекратить.

Алгоритм Гауссовых исключений с выбором главного элемента можно кратко записать так:

Для k=1 до n-1

найти $m \geq k$ такое что $|a_{ik}|=\max(|a_{ik}|: i \geq k)$

если $a_{mk}=0$, то **A** – вырождена, вычисления прекратить,

иначе поменять местами a_{kj} и a_{mj} ($j=k$, $k+1$, …, n),

поменять местами b_k и b_m

Для $i=k+1$ до n

$l_{ik}=a_{ik}/a_{kk}$;

для $j=k+1$ до n

$a_{ij}-=l_{ik}a_{kj}$; $b_i-=l_{ik}b_k$.

Существует много вариантов метода Гаусса, из них наиболее эффективен *метод Жордана-Гаусса* или метод полного исключения. Отличается он тем, что при использовании i-го уравнения для исключения переменных исключение проводят не только для нижележащих, но и для вышележащих строк. Это исключает необходимость в обратном ходе Гаусса и этот метод следует применять, если нет необходимости в «попутном» с решением СЛАУ вычислении определителя матрицы.

Определитель матрицы равен произведению диагональных элементов приведенной к треугольному виду матрицы и, по-видимому, прямой ход Гаусса является одним из лучших методов его вычисления. Но при перемножении диагональных элементов необходимо отслеживать возможную потерю точности при малых множителях и менее вероятную возможность переполнения при больших.

Процесс Гауссова исключения является также эффективным методом вычисления обратной матрицы. Из определяющего обратную матрицу соотношения $\mathbf{AA}^{-1}=\mathbf{E}$ выте-

кает, что i-й столбец обратной матрицы $\boldsymbol{a}_i^{-1}$ может быть получен решением системы $\mathbf{A}\boldsymbol{a}_i^{-1}=\boldsymbol{e}_i$, где $\boldsymbol{e}_i$ – i-й столбец единичной матрицы того же порядка, что и **A**. Для вычисления всей обратной матрицы необходимо решить n систем линейных уравнений, при этом матрица **A** расширяется справа единичной матрицей nхn. Прямой ход Гаусса при решении полученных систем можно осуществлять для всех систем одновременно.

2.2.1.2. Предварительная факторизация матриц (разложение в произведение двух матриц) в задачах решения СЛАУ

Алгоритм Гауссовых исключений можно использовать для факторизации матриц. Если сконструировать нижнюю треугольную матрицу **L** с единицами по главной диагонали так, что элементы l_{ij} будут равны множителям, использованным при исключении j-й переменной из i-го уравнения, и верхнюю треугольную матрицу **U**, которая получается после прямого хода Гаусса, то исходную матрицу **A** можно представить в виде произведения $\mathbf{A}=\mathbf{LU}$, что и является треугольной факторизацией матрицы **A**. Если теперь, используя прямую подстановку, решить треугольную систему $\mathbf{L}\boldsymbol{y}=\boldsymbol{b}$ относительно $\boldsymbol{y}$, то получим вектор правой части для системы $\mathbf{U}\boldsymbol{x}=\boldsymbol{y}$ и решение исходной системы $\mathbf{A}\boldsymbol{x}=\boldsymbol{b}$ можно выполнить в три этапа: факторизации $\mathbf{A}=\mathbf{LU}$, решения $\mathbf{L}\boldsymbol{y}=\boldsymbol{b}$ и решения $\mathbf{U}\boldsymbol{x}=\boldsymbol{y}$. Этот подход используется в некоторых вычислительных вариантах процесса исключения.

2.2.1.2.1. Факторизация матриц по методу Холецкого

Для произвольных невырожденных матриц выбор главного элемента является необходимой процедурой, но некоторые типы матриц не требуют никаких перестановок – в частности, диагонально-доминирующие матрицы и положительно определенные симметричные матрицы. В случае симметричной положительно определенной матрицы

можно использовать вариант Гауссова исключения под названием *метода Холецкого*. Он основан на разложении

$$\mathbf{A}=\mathbf{L}\mathbf{L}^{\mathrm{T}},$$

где $\mathbf{L}$ – нижняя треугольная матрица, у которой на главной диагонали не обязательно стоят единицы, как было в **LU**-разложении. При положительности диагональных элементов $\mathbf{L}$ разложение будет единственным. Правило получения элементов матрицы $\mathbf{L}$ через элементы $\mathbf{A}$ получим, приравнивая элементы в левой и правой частях $\mathbf{A}=\mathbf{L}\mathbf{L}^{\mathrm{T}}$. Принимая во внимание, что $l_{ij}=0$ при $j>i$, получаем:

$$\begin{bmatrix} a_{11} & \dots & \dots & \dots & a_{1n} \\ \dots & \dots & \dots & \dots & \dots \\ a_{i1} & \dots & a_{ii} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ a_{n1} & \dots & \dots & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} l_{11} & \dots & \dots & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ l_{i1} & \dots & l_{ii} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ l_{n1} & \dots & \dots & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} l_{11} & \dots & l_{i1} & \dots & l_{n1} \\ \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & l_{ii} & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \dots & l_{nn} \end{bmatrix}.$$

Приравнивая элементы первого столбца слева и справа от знака равенства видим, что $a_{i1}=l_{i1}l_{i1}$, так что первый столбец матрицы $\mathbf{L}$ находится по формулам $l_{11}=(a_{11})^{1/2}$, $l_{i1}=a_{i1}/l_{11}$, $i=2, \dots, n$.

Аналогично получаем $a_{ii}=\sum_{k=1}^{i} l_{ik}^2$, $a_{ij}=\sum_{k=1}^{j} l_{ik}l_{jk}$, $j<i$ для последовательного определения столбцов матрицы $\mathbf{L}$ по следующему алгоритму:

Разложение Холецкого

Для $j=1$ до n

$$l_{jj}=\left(a_{jj}-\sum_{k=1}^{j-1} l_{jk}^2\right)^{1/2}.$$

Для $i=j+1$ до n

$$l_{ij}=\left(a_{ij}-\sum_{k=1}^{j-1} l_{ik}l_{ik}\right)\Big/ l_{jj}.$$

После вычисления матрицы **L** решение линейной системы может быть получено точно так, как в случае **LU**-разложения: решаем $\mathbf{L}\boldsymbol{y}=\boldsymbol{b}$, затем решаем $\mathbf{L}^{\mathrm{T}}\boldsymbol{x}=\boldsymbol{y}$.

Чтобы метод Холецкого работал, необходимо, чтобы

$$a_{jj}-\sum l_{jk}^2>0,$$

что соблюдается при положительной определенности матрицы **A**. При этом метод будет и численно устойчивым.

Метод легко адаптируется для ленточных матриц. Если p – число ненулевых диагоналей ниже и выше главной, то алгоритм Холецкого принимает вид:

Разложение Холецкого для ленточных матриц

Для j=1 до n

$$q=\max(1, j\text{-}p),\ l_{jj}=(a_{jj}-\sum_{k=q}^{j-1} l_{jk}^2)^{1/2}.$$

Для $i=j+1$ до $\min(j+p, n)$

$$r=\max(1, i\text{-}p),\ l_{ij}=(a_{ij}-\sum_{k=r}^{j-1} l_{ik}l_{jk})/l_{jj}.$$

2.2.1.3. Метод ортогонализации для решения СЛАУ

Метод позволяет осуществить его реализацию при помощи чрезвычайно компактного алгоритма и компьютерной программы, не требует никаких проверок сходимости и сколько-нибудь существенных преобразований исходной системы, операция деления на коэффициенты матрицы в нем отсутствует, а имеющаяся операция деления на норму вектор-строки является намного безопасней, так как вектор нулевой длины не может присутствовать в невырожденной матрице. Все сказанное обусловило его широкое использование в прикладных задачах.

Сущность метода ортогонализации в следующем.

Перенесем свободные члены всех уравнений системы в левые части, будем считать их (n+1)-ми составляющими векторов a_i и положим $x_{n+1}=1$.

Получим систему в виде

$$\sum_{j=1}^{n+1} a_{ij} x_j = 0 \;\; (i=1, 2, \ldots, n)$$

Суммы в левых частях уравнений можно интерпретировать как скалярные произведения векторов ($\boldsymbol{a}$, $\boldsymbol{x}$); в этом случае искомым решением системы будет некоторый вектор $\boldsymbol{x}$ в (n+1)-мерном пространстве, ортогональный базису, образованному системными векторами $\boldsymbol{a}_i$.

Так как сам базис в общем случае не ортонормирован, то необходима дополнительная процедура построения системы взаимно ортогональных векторов, выражающихся линейно через исходные векторы $\boldsymbol{a}_i$, чтобы не изменить решение системы. Выполним это так:

Первый вектор-строку образуем просто делением исходного на его длину:

$$\boldsymbol{b}_1 = \frac{\boldsymbol{a}_1}{|\boldsymbol{a}_1|}.$$

Из второго вычтем вектор, равный по длине проекции второго на направление первого:

$$\boldsymbol{b}_2 = \boldsymbol{a}_2 - (\boldsymbol{a}_2, \boldsymbol{b}_2)\boldsymbol{b}_1$$

и отнормируем

$$\boldsymbol{b}_2 = \frac{\boldsymbol{b}_2}{|\boldsymbol{b}_2|}.$$

Из третьего вычтем уже 2 составляющие

$$\boldsymbol{b}_2 = \boldsymbol{a}_3 (\boldsymbol{a}_3, \boldsymbol{b}_1)\boldsymbol{b}_1 - (\boldsymbol{a}_3, \boldsymbol{b}_2)\boldsymbol{b}_2$$

и отнормируем по модулю и т.д.

Добавим к векторам $\boldsymbol{a}_i$ линейно независимый от них произвольный вектор $\boldsymbol{a}_{n+1}$ и, когда до него дойдет очередь, проведем с ним такую же процедуру ортогонализации всем векторам $\boldsymbol{a}_i$ (i=1, 2,…, n). Останется смасштабировать его делением на $b_{n+1,n+1}$, так как последний по договоренности равен 1. Его первые n составляющих и образуют искомый вектор решения.

Всю описанную процедуру рекомендуют повторить от 3 до 5 раз для повышения точности результата.

2.2.1.4. Итерационные методы решения СЛАУ

Для решения задачи методом итераций система уравнений должна быть преобразована к виду:

$$x_1^{k+1}=f(x_2^k, x_3^k, ..., x_n^k),$$
$$x_2^{k+1}=f(x_1^k, x_3^k, ..., x_n^k),$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$x_n^{k+1}=f(x_1^k, x_2^k, ..., x_{n-1}^k, x_n^k).$$

Итерационный процесс начинается при k=0 заданием начальных значений компонент вектора решения $\boldsymbol{x}$ (в общем случае произвольных); эти значения подставляются в правые части приведенных уравнений для вычисления следующего приближения, это следующее значение становится предыдущим, подставляется в систему для получения следующего и т.д. либо до достижения заданного приращения значений корней на очередной итерации, либо до выполнения заданного числа итераций. Это метод *простых итераций*. Его можно видоизменить, если в каждое следующее уравнение подставлять значения уже вычисленных компонент вектора решения:

$$x_1^{k+1}=f(x_2^k, x_3^k, ..., x_n^k),$$
$$x_2^{k+1}=f(x_1^{k+1}, x_3^k, ..., x_n^k),$$
$$\ldots\ldots\ldots\ldots\ldots\ldots$$
$$x_n^{k+1}=f(x_1^{k+1}, x_2^{k+1}, ..., x_{n-1}^{k+1}, x_n^k).$$

Эта модификация носит название *метода Зейделя*.

2.2.1.4.1. Проблема сходимости итерационных методов

Под сходимостью итерационного процесса понимается наличие предела у последовательности получаемых решений и равенство этого предела при бесконечном числе итераций точному решению системы. Для анализа сходимости приведем систему к виду:

$$\boldsymbol{x}=\mathbf{A}\boldsymbol{x}+\boldsymbol{b} \quad (*)$$

Теорема сходимости:

Процесс итераций для линейной системы (*) сходится к единственному решению, если какая-нибудь каноническая

норма матрицы **A** меньше 1, т.е. для итерационного процесса

$$\boldsymbol{x}_k=\boldsymbol{b}+\mathbf{A}\boldsymbol{x}_{k-1}\ (k=1,\ 2,\ ...)$$

достаточное условие сходимости при произвольном начальном приближении $\boldsymbol{x}_0$ есть:

$$||\mathbf{A}||<1.$$

Доказательство:

Последовательность приближений

$$\boldsymbol{x}_1=\boldsymbol{b}+\mathbf{A}\boldsymbol{x}_0$$
$$\boldsymbol{x}_2=\boldsymbol{b}+\mathbf{A}\boldsymbol{x}_1$$
$$..............$$
$$\boldsymbol{x}_k=\boldsymbol{b}+\mathbf{A}\boldsymbol{x}_{k-1}$$

после подстановки последовательно сверху вниз приводит к:

$$\boldsymbol{x}_k=(\mathbf{E}+\mathbf{A}+\mathbf{A}^2+\mathbf{A}^3+...+\mathbf{A}^{k-1})\boldsymbol{b}+(\mathbf{A}^k\boldsymbol{x}_0)$$

Так как $||\mathbf{A}^k||$ стремится к 0 при k стремящемся к бесконечности и $||\mathbf{A}||<1$, то $\lim \mathbf{A}^k=0$ и $\lim (\mathbf{E}+\mathbf{A}+\mathbf{A}^2+\mathbf{A}^3+...+\mathbf{A}^{k-1})=$ $=(\mathbf{E}-\mathbf{A})^{-1}$ – по теореме сходимости матричного степенного ряда.

Отсюда получаем $\boldsymbol{x}=\lim \boldsymbol{x}^k=(\mathbf{E}-\mathbf{A})^{-1}\boldsymbol{b}$ или $(\mathbf{E}-\mathbf{A})\boldsymbol{x}=\boldsymbol{b}$ и $\boldsymbol{x}=\mathbf{A}\boldsymbol{x}+\boldsymbol{b}$, т.е. предельный вектор $\boldsymbol{x}$ есть решение системы.

В соответствии с приведенной теоремой перед началом выполнения итераций необходимо привести матрицу к диагонально преобладающей форме перестановками строк и равносильными преобразованиями, нормировать строки делением на диагональные элементы, вычислить какую-либо норму матрицы и проверить условие непревышения ее значением 1.

Примечание: ***норма матрицы*** – это вещественное число, вычисленное по ее элементам и характеризующее свойства матрицы. В качестве нормы матрицы могут использоваться:

Максимальная из сумм модулей элементов строк

$$||\mathbf{A}||=\max \sum_{j=1}^{n}\left|a_{ij}\right| \quad i=1,\ 2,\ \ldots,\ n$$

Максимальная из сумм модулей элементов столбцов

$$\|\mathbf{A}\|=\max \sum_{j=1}^{n}\left|a_{ij}\right| \ j=1, 2, \ldots, n$$

Евклидова норма *– квадратный корень из суммы квадратов всех элементов*

$$\|\mathbf{A}\|=\sqrt{\sum_{i=1}^{n}\sum_{j=1}^{n}a_{ij}^{2}}\ .$$

2.2.2. Методы вычисления собственных значений матриц

Проблема нахождения собственных значений возникает во многих вычислительных и исследовательских задачах, например, при исследовании динамики процессов в различных областях – в технике, биологии, экономике и т.д. Но решение этой проблемы связано с существенными трудностями – до настоящего времени не разработаны удовлетворительные по точности и эффективности общие методы, пригодные для матриц общего вида и учитывающие часто встречающуюся на практике плохую обусловленность этих матриц, приводящую к неустойчивости результатов вычислений к малым изменениям значений матричных элементов. Существует много специальных методов, предназначенных для матриц специальной структуры – симметричных, ленточных, квазидиагональных и пр.

Во всех случаях, когда это оказывается возможным, стараются с применением *преобразований подобия* (не изменяющих собственных значений матрицы) привести матрицу либо к треугольной форме (и избежать процедур получения и решения характеристического уравнения), либо к форме, позволяющей получить коэффициенты характеристического полинома непосредственно из преобразованной подобной матрицы. Но известные методы таких приведений достаточно сложны, их обоснование и подробное изложение требует специального курса по проблемам соб-

ственных значений, поэтому мы ограничимся кратким обзором наиболее характерных подходов к решению этой задачи.

2.2.2.1. Метод неопределенных коэффициентов

Общий вид характеристического полинома известен, если известен порядок матрицы, – он представляет собой скалярное произведение двух векторов; составляющими одного являются коэффициенты при степенях λ, а составляющие второго – степени λ, которые можно трактовать как «коэффициенты при коэффициентах»:

$$P(\lambda)=c_n\lambda^n+c_{n-1}\lambda^{n-1}+\ldots+c_j\lambda+\ldots+c_1\lambda+c_0$$

Если задаться последовательностью значений λ_i и их степени подставить в вышеприведенную его запись, вычислить соответствующую последовательность значений определителя D_i (например, приведением матрицы к треугольной форме по методу прямого хода Гауссовых исключений и перемножением диагональных элементов), то можем составить систему линейных уравнений относительно c:

$$P(\lambda_i)=D_i$$

Решение этой системы предположительно даст нам коэффициенты характеристического полинома и останется только задача вычисления его корней. Реализация этого метода приведена в нашем матричном классе. Недостаток метода – необходимость многократного вычисления значений определителя.

2.2.2.2. Метод Данилевского

Основан на использовании преобразования подобия $\mathbf{F}^{-1}\mathbf{AF}$ матрицы **A**, где **F** – произвольная матрица, к такой форме, из которой можно непосредственно получить коэффициенты характеристического полинома. По этому методу исходная матрица **A** приводится к так называемой канонической форме Фробениуса, верхняя строка которой со-

держит значения коэффициентов характеристического полинома:

$$\mathbf{F}=\begin{pmatrix} p_1 & p_2 & p_3 & \cdots & p_{n-1} & p_n \\ 1 & 0 & 0 & ... & 0 & 0 \\ 0 & 1 & 0 & ... & 0 & 0 \\ ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & 1 & 0 \end{pmatrix},$$

характеристический полином которой имеет вид:

$$|\mathbf{F}-\lambda\mathbf{E}|=(-1)^n(\lambda^n-p_1\lambda^{n-1}-\ldots-p_n)$$

Преобразование подобия матрицы **A** осуществляется $(n-1)$ раз с помощью матриц

$$\mathbf{M}_i,\ \mathbf{M}_i^{-1},$$

где $i=1, 2, \ldots, n-1$ – порядковый номер преобразования, n – порядок матрицы **A**.

При этом матрица $\mathbf{M}_i^{-1}$ имеет элементы

$$\mathbf{M}_i^{-1}[n\text{-}i, j]=\mathbf{A}_{i\text{-}1}[n\text{-}i+1, j], j=1, 2, \ldots, n;$$

остальные ее элементы таковы: на главной диагонали все единицы, кроме уже заполненного элемента $\mathbf{M}_i^{-1}[n\text{-}i,\ n\text{-}i]$, остальные – нули. Элементы матрицы

$$\mathbf{M}_i[n\text{-}i, j]=-\frac{\mathbf{A}_{i-1}[n-i+1, j]}{\mathbf{A}_{i-1}[n-i+1, n-i]},$$

кроме диагонального, который равен

$$\mathbf{M}_i[n\text{-}i,\ n\text{-}i]=\frac{1}{\mathbf{A}_{i-1}[n-i+1, n-i]}.$$

Нижний индекс у обозначения матрицы **A** обозначает исходную матрицу после i-го преобразования подобия. Остальные элементы формируются так же, как и у матрицы $\mathbf{M}_i^{-1}$ – еще не заполненные диагональные равны 1, остальные – нули. Чтобы было яснее, развернем эти матрицы для первого и второго преобразования подобия:

$$\mathbf{M}_1=\begin{vmatrix} 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{-\mathbf{A}[n,1]}{\mathbf{A}[n,n-1]} & \frac{-\mathbf{A}[n,2]}{\mathbf{A}[n,n-1]} & \dots & \frac{-\mathbf{A}[n,n-2]}{\mathbf{A}[n,n-1]} & \frac{1}{\mathbf{A}[n,n-1]} & \frac{-\mathbf{A}[n,n]}{\mathbf{A}[n,n-1]} \\ 0 & 0 & \dots & 0 & 0 & 1 \end{vmatrix}$$

$$\mathbf{M}_1^{-1}=\begin{vmatrix} 1 & 0 & \dots & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ \mathbf{A}[n,1] & \mathbf{A}[n,2] & \dots & \mathbf{A}[n,n-1] & \mathbf{A}[n,n] \\ 0 & 0 & \dots & 0 & 1 \end{vmatrix}$$

$$\mathbf{A}_1=\mathbf{M}_1^{-1}\mathbf{A}\mathbf{M}_1$$

$$\mathbf{M}_2=\begin{vmatrix} 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ \frac{-\mathbf{A}_1[n-1,1]}{\mathbf{A}[n-1,n-2]} & \frac{-\mathbf{A}_1[n-1,2]}{\mathbf{A}[n-1,n-2]} & \dots & \frac{1}{\mathbf{A}[n-1,n-2]} & \frac{-\mathbf{A}_1[n-1,n-1]}{\mathbf{A}[n-1,n-2]} & \frac{-\mathbf{A}_1[n-1,n]}{\mathbf{A}[n-1,n-2]} \\ 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \end{vmatrix}$$

$$\mathbf{M}_2^{-1}=\begin{vmatrix} 1 & 0 & \dots & 0 & 0 & 0 \\ 0 & 1 & \dots & 0 & 0 & 0 \\ \dots & \dots & . & \dots & \dots & \dots \\ \mathbf{A}_1[n-1,1] & \mathbf{A}_1[n-1,2] & \dots & \mathbf{A}_1[n-1,n-2] & \mathbf{A}_1[n-1,n-1] & \mathbf{A}_1[n-1,n] \\ 0 & 0 & \dots & 0 & 1 & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \end{vmatrix}$$

$$\mathbf{A}_2=\mathbf{M}_2^{-1}\mathbf{A}_1\mathbf{M}_2 \text{ и т.д.}$$

2.2.2.3. QR-алгоритм для несимметрических матриц

Эта задача существенно более трудная, чем для симметрических матриц и QR-алгоритм, по-видимому, один из наиболее общих методов ее решения. Основная идея этого метода состоит в разложении исходной матрицы $\mathbf{A}_0=\mathbf{A}$ в произведение ортогональной матрицы и верхней треугольной. Последовательность преобразований дает нам очередные модификации матрицы **A**: $\mathbf{A}_k=\mathbf{Q}_k\mathbf{R}_k$, $\mathbf{A}_{k+1}=\mathbf{R}_k\mathbf{Q}_k$, k=0, 1,

2, …, где каждая из матриц $\mathbf{Q}_k$ является ортогональной (если матрица **A** вещественна) или унитарной (если A невещественна). $\mathbf{R}_k$ – верхние треугольные матрицы.

Примечание: *унитарной* называется матрица **U**, удовлетворяющая соотношению $\mathbf{U}^*\mathbf{U}=\mathbf{E}$, где $\mathbf{U}^*$ – матрица, сопряженная к **U**, т.е. получающаяся из **U** заменой элементов на комплексно-сопряженные с последующим транспонированием, **E** – единичная матрица.

Для преобразований будем использовать известный *алгоритм Хаусхолдера*, состоящий в следующем:

пусть некоторый вектор $\boldsymbol{w}_1$ выбран так, что

$$\boldsymbol{w}_1^T = \mu(\mathbf{A}_0[1,1]-s, \mathbf{A}_0[2,1],..., \mathbf{A}_0[n,1]),$$

$$s = \sqrt{\sum_{j=1}^{n} \mathbf{A}_0[j,1]^2}, \mu = \left[2s\left(s-\mathbf{A}_0[1,1]\right)\right]^{-1/2}$$

Тогда $\mathbf{A}_2=(\mathbf{E}-2\boldsymbol{w}_1\boldsymbol{w}_1^T)\mathbf{A}_0=\begin{vmatrix} s & * & * & * \\ 0 & * & * & * \\ .... & * & * & * \\ 0 & * & * & * \end{vmatrix}$, где * обозначает в общем случае ненулевой элемент. Выберем теперь вектор $\boldsymbol{w}_2$ так, что

$$\boldsymbol{w}_2^T = \mu_2(\mathbf{A}_2[2,2]-s_2,..., \mathbf{A}_2[n,2]),$$

$$\boldsymbol{s}_2 = \pm\sqrt{\sum_{j=2}^{n} \mathbf{A}_2[j,2]},$$

$$\mu_2 = \{2s_2(s_2-\mathbf{A}_2[2,2])\}^{-1/2}$$

Лежащие ниже главной диагонали элементы двух первых столбцов матрицы $(\mathbf{E}-2\boldsymbol{w}_2\boldsymbol{w}_2^T)\mathbf{A}_2$ обратятся в нуль. Продолжая этот процесс с помощью векторов $\boldsymbol{w}_i$ с нулями в первых i–1 позициях, получим

$$(\mathbf{E}-2\boldsymbol{w}_{n-1}\boldsymbol{w}_{n-1}^T)\ldots(\mathbf{E}-2\boldsymbol{w}_2\boldsymbol{w}_2^T)(\mathbf{E}-2\boldsymbol{w}_1\boldsymbol{w}_1^T)\mathbf{A}=\mathbf{R}=\mathbf{QA}$$

где **R** – верхняя треугольная матрица, а **Q** – ортогональная в силу того, что составляющие ее сомножители в круглых скобках есть ортогональные матрицы. Так как в этом случае $\mathbf{Q}^{-1}=\mathbf{Q}^{T}$, то можем записать **A**=**QR**, представляющее собой QR-разложение матрицы **A**. ***Теорема о QR-алгоритме*** звучит так:

Если все собственные значения матрицы различны по абсолютной величине и $\mathbf{A}=\mathbf{PDP}^{-1}$, где **D** – диагональная матрица из собственных значений матрицы **A**, то генерируемые QR-алгоритмом матрицы $\mathbf{A}_k$ сходятся к верхней треугольной матрице с собственными значениями в главной диагонали, а элементы ниже диагонали сходятся к нулю с линейной скоростью, пропорциональной отношению собственных значений.

Для повышения эффективности приведенного алгоритма перед его применением матрицу рекомендуют привести к так называемой *форме Хессенберга* с нулями ниже первой за главной поддиагональю; это приведение осуществляют с применением преобразований Хаусхолдера. После этого разложение матрицы выполняется значительно проще или уже рассмотренными преобразованиями Хаусхолдера, либо (что еще лучше) воспользоваться *преобразованиями Гивенса* (плоскими вращениями). Но мы эти усовершенствования алгоритма рассматривать не будем. Скажем только, что QR-алгоритм практически не налагает на исходную матрицу существенных ограничений и является численно устойчивым.

2.2.2.4. Метод Леверрье-Фаддеева

Метод использует для нахождения собственных значений характеристический полином, который по созданному Леверрье и усовершенствованному Фаддеевым алгоритму получают следующим образом:

строят последовательность матриц

$\mathbf{A}_1=\mathbf{A}$; Sp $\mathbf{A}_1=p_1$; $\mathbf{B}_1=\mathbf{A}_1-p_1\mathbf{E}$;

$\mathbf{A}_2=\mathbf{A}\mathbf{B}_1$; Sp $\mathbf{A}_2=p_2$; $\mathbf{B}_2=\mathbf{A}_2-p_2\mathbf{E}$;

……………………………………………..

$\mathbf{A}_{n-1}=\mathbf{A}\mathbf{B}_{n-2}$; $(1/(n-1))$Sp $\mathbf{A}_{n-1}=p_{n-1}$; $\mathbf{B}_{n-1}=\mathbf{A}_{n-1}-p_{n-1}\mathbf{E}$;

$\mathbf{A}_n=\mathbf{A}\mathbf{B}_{n-1}$; $(1/n)$Sp $\mathbf{A}_n=p_n$; $\mathbf{B}_n=\mathbf{A}_n-p_n\mathbf{E}$;

В результате получают уравнение

$$\lambda^n-p_1\lambda^{n-1}-p_2\lambda^{n-2}-\ldots-p_n=0.$$

Попутно можно получить обратную матрицу

$$\mathbf{A}^{-1}=\mathbf{B}_{n-1}/p_n$$

Собственные векторы находим по формулам

$$\boldsymbol{x}_0=\boldsymbol{e};\ \boldsymbol{x}_i^k=\lambda_k\boldsymbol{x}_{i-1}^k+\boldsymbol{b}_i^k\ \ i=1,\ 2,\ \ldots,\ n\text{-}1$$

где $\boldsymbol{e}$ – столбец единичной матрицы, $\boldsymbol{b}_i^k$ – одноименный столбец матрицы $\mathbf{B}_k$, а собственный вектор $\boldsymbol{x}_{n-1}^k$ соответствует λ_k.

2.2.2.5. Итерационный степенной метод

Среди классических методов, помимо редко используемого на практике метода непосредственного разворачивания векового определителя, для вычисления наибольших по абсолютному значению собственных чисел больших разреженных матриц иногда оказывается полезным так называемый степенной метод, основанный на следующем. Пусть собственные числа матрицы **A** вещественны и наибольшее по модулю не является кратным. При заданном векторе $\boldsymbol{x}_0$ формируется последовательность

$$\boldsymbol{x}_{k+1}=\mathbf{A}\boldsymbol{x}_k.$$

Вектор $\boldsymbol{x}_0$ может быть разложен по направлениям собственных векторов $\boldsymbol{v}_1$, $\boldsymbol{v}_2$, …, $\boldsymbol{v}_n$ матрицы **A**:

$$\boldsymbol{x}_0=c_1\boldsymbol{v}_1+\ldots+c_n\boldsymbol{v}_n.$$

Если, например, $c\neq0$, то, учитывая, что $\mathbf{A}^k\boldsymbol{v}_i=\lambda^k\boldsymbol{v}_i$, получим

$$\boldsymbol{x}_k=c_1\lambda_1^k\boldsymbol{v}_1+c_2\lambda_2^k\boldsymbol{v}_2+\ldots+c_n\lambda_n^k\boldsymbol{v}_n=$$

$$=\lambda_1^k\left[c_1\boldsymbol{v}_1+c_2\left(\frac{\lambda_2}{\lambda_1}\right)^k\boldsymbol{v}_2+\ldots+c_n\left(\frac{\lambda_n}{\lambda_1}\right)^k\boldsymbol{v}_n\right]$$

При $k \to \infty$ множители с дробными отношениями в круглых скобках стремятся к нулю, а векторы $\boldsymbol{x}_k$ стремятся по направлению к собственному вектору $\boldsymbol{v}_1$; их модули будут либо стремится к нулю (если $|\lambda_1|<1$), либо к бесконечности ($|\lambda_1|>1$). Если текущие значения векторов $\boldsymbol{x}_k$ нормировать по его наибольшей по абсолютному значению – координате σ_k, то $\sigma_k \to \lambda_1$ и $\boldsymbol{x}_k \to c_1\boldsymbol{v}_1$ при $k \to \infty$.

Другими словами, последовательность $\{\boldsymbol{x}_k\}$ сходится к вектору, пропорциональному $\boldsymbol{v}_1$.

Достоинство метода – отсутствие преобразований матрицы **A**, а главный недостаток – медленная сходимость, особенно при близких значениях первого и второго по величине собственных чисел. При кратном наибольшем собственном числе метод вообще не сходится. Другая проблема возникает при необходимости вычисления следующих после наибольшего собственных чисел – если не блокировать уже найденное значение, то итерации снова будут сходиться к λ_1. Обычно применяемый прием сводится к процедуре сдвига диагональных элементов матрицы **A** на величину p, тогда

$$(\mathbf{A}-p\mathbf{E})\boldsymbol{x}_i=(\lambda-p)\boldsymbol{x}_i$$

и соответствующим подбором p можно привести итерации к другому собственному значению. Например, если взять p равным уже найденному наибольшему с обратным знаком, то можно повторной итерацией найти наименьшее по модулю собственное число. Но каждое последующее будет вычисляться все с большей погрешностью.

2.2.2.6. Метод Крылова

Рекуррентная процедура формирования векторов, приведенная для степенного метода, используется в методе Крылова при формировании матрицы системы линейных уравнений для вычисления коэффициентов характеристического полинома. Порядок этого полинома известен – он

равен рангу матрицы и характеристическое уравнение может быть приведено к виду:

$$P(\lambda)=(-1)^n(\lambda^n-p_{n-1}\lambda^{n-1}-\ldots-p_1\lambda-p_0)=0.$$

Если вместо λ^i подставить векторы $\boldsymbol{y}_i=\mathbf{A}^i\boldsymbol{y}_0$, где $\boldsymbol{y}_0$ – произвольный начальный вектор, то получим систему линейных уравнений для вычисления коэффициентов p_i. Ее решение одним из уже рассмотренных методов (например, методом ортогонализации или Гаусса) определит коэффициенты характеристического уравнения, а решение характеристического уравнения даст собственные значения матрицы.

Собственные векторы $\boldsymbol{x}_i$ матрицы **A**, соответствующие собственным числам λ_i, определяют как линейную комбинацию векторов $\boldsymbol{y}_i$ из соотношения:

$$\boldsymbol{x}_i=\sum_{j=0}^{n-1} b_{ij}\boldsymbol{y}_{nj}\ ,$$

где b_{ij} – коэффициенты полинома, полученного при делении $D(\lambda)$ на $(\lambda-\lambda_i)$.

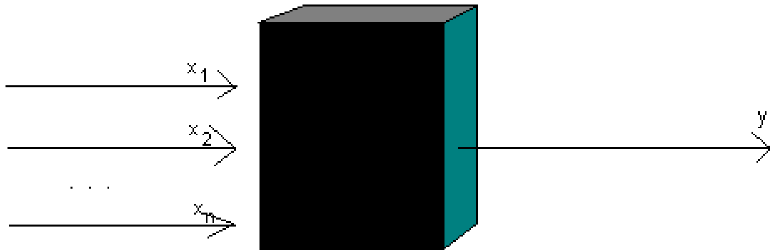
2.2.3. Метод наименьших квадратов (МНК)

Мы размещаем этот метод, относящийся к методам аналитического приближения функций, заданных в табличной форме, в матричном классе просто потому, что он базируется на конструировании матриц и на матричных операциях – такое размещение приводит к уменьшению накладных расходов на вызов метода по сравнению с размещением его в отдельном классе.

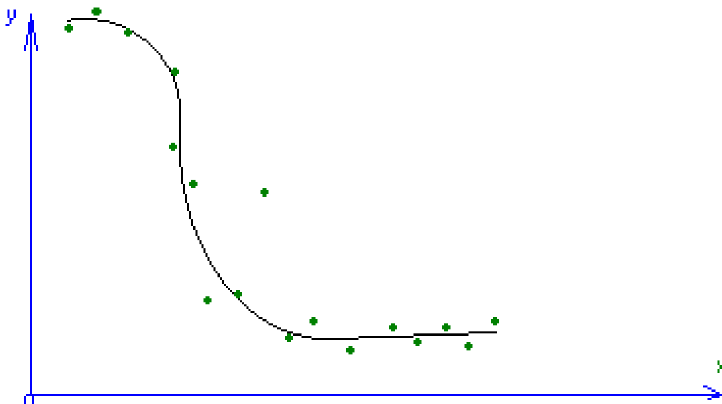
Описание метода.

В задачах моделирования при определении статических характеристик объектов (зависимости скалярной выходной переменной y от векторной входной переменной $\boldsymbol{x}$) может стоять обратная рассмотренной выше задача – определить вектор коэффициентов линейной алгебраической модели $\boldsymbol{a}$; в этом случае решению подлежит система $\boldsymbol{y}=\mathbf{X}\boldsymbol{a}$, где $\mathbf{X}$ – матрица полученных экспериментально значений (или зна-

чений некоторых функций от экспериментальных данных) составляющих вектора входа, $\boldsymbol{y}$ – соответствующие строкам **X** значения составляющих вектора, $\boldsymbol{a}$ – подлежащий определению вектор коэффициентов модели, составляющие которого надо вычислить так, чтобы обеспечить наилучшее приближение вычисленных по модели значений $y_i(\boldsymbol{x}_i)$ (здесь y_i – скаляр, $\boldsymbol{x}_i$ – вектор).



Термин «наилучшее приближение» в МНК трактуется как критерий Гаусса – минимум суммы Q квадратов отклонений вычисленных (модельных) значений $y_{mt}=ax_t$ от измеренных $\boldsymbol{y}$, где t=1, 2, …, N – номер или момент времени отсчета значений $\boldsymbol{x}$ и $\boldsymbol{y}$.



Очевидно, что Q есть функция вектора $\boldsymbol{a}$:

$$Q(\boldsymbol{a})=\sum_{t=1}^{N}\left(y_t-\sum_i a_i x_{ti}\right)^2=(\boldsymbol{y}-\mathbf{X}\boldsymbol{a})^{\mathrm{T}}(\boldsymbol{y}-\mathbf{X}\boldsymbol{a})=\boldsymbol{y}^{\mathrm{T}}\boldsymbol{y}-$$

$$-2\boldsymbol{y}^{\mathrm{T}}\mathbf{X}\boldsymbol{a}+\boldsymbol{a}^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{a}\to\min.$$

Определение значений аргумента, обеспечивающих минимум функции $Q(\boldsymbol{a})$, осуществим как в обычном анализе:

$$\partial Q/\partial \boldsymbol{a}=-2\mathbf{X}^{\mathrm{T}}\boldsymbol{y}+2\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{a}=0,$$

или

$$\mathbf{X}^{\mathrm{T}}\boldsymbol{y}=\mathbf{X}^{\mathrm{T}}\mathbf{X}\boldsymbol{a}.$$

Это обычная система линейных уравнений с симметричной матрицей коэффициентов $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ (носит название матрицы Грамма), столбцом свободных членов $\mathbf{X}^{\mathrm{T}}\boldsymbol{y}$ и искомым вектором $\boldsymbol{a}$.

Решение этой системы линейных уравнений дает значение вектора $\boldsymbol{a}$, обеспечивающее минимум суммы квадратов отклонений (в силу положительности второй производной), если ранг матрицы наблюдений $\mathbf{X}$ равен размерности вектора $\boldsymbol{a}$:

$$\boldsymbol{a}=(\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\boldsymbol{y}.$$

Порядок N матрицы $\mathbf{X}$ обычно значительно больше ее ранга

$N>(\mathrm{rank}(\mathbf{X})=m)$ (m – размерность вектора $\boldsymbol{a}$),

чтобы была возможность выполнения усреднения решения по значительному числу измерений и соответственно эффективного «сглаживания» случайных выбросов.

Вычисление непосредственно по последней формуле, связанное с необходимостью обращения матрицы, как правило, не выполняют. Для решения таких систем разработан ряд эффективных специальных методов – разложения по сингулярным числам, разложение матрицы с применением метода Хаусхолдера и др. Мы не будем рассматривать эти методы, а в нашем матричном классе решение выполнено одним из описанных выше методов – методом Гаусса.

Решение задачи может быть значительно упрощено, если вектор-столбцы матрицы измерений $\mathbf{X}$ будут представлять собой взаимно-ортогональные векторы, удовлетворяя условию $(\boldsymbol{x}_j, \boldsymbol{x}_k)=0$ при $j\neq k$. Так как элементами матрицы линейных уравнений $\mathbf{X}^{\mathrm{T}}\mathbf{X}$ являются скалярные произведе-

ния этих векторов, то все ее элементы, кроме диагональных, обратились бы в нуль и система уравнений распалась бы на отдельные уравнения. Составляющие вектора $\boldsymbol{a}$ можно было бы вычислить простым делением столбца свободных членов на диагональные элементы соответствующих строк:

$$a_i = \frac{\sum_{t=1}^{N} x_{it} y_t}{\sum_{t=1}^{N} x^2_{it}}.$$

Если измерения проводятся в режиме активного эксперимента, когда есть возможность формировать матрицу измерения по усмотрению исследователя, последовательности значений составляющих векторов $\boldsymbol{x}$ задают из условия обеспечения ортогональности, а значения y просто фиксируют как отклик объекта на заданные значения $\boldsymbol{x}$. Соответствующие методы рассматриваются в теории ортогонального планирования эксперимента и не входят в наш курс.

Мы же рассмотрим другой подход к проблеме ортогонализации. Пусть для простоты входная переменная x – скаляр, а выходную мы хотели бы представить как линейную комбинацию некоторых функций этой скалярной переменной:

$$y=(\boldsymbol{a}, \boldsymbol{f}(x)),$$

где $\boldsymbol{a}$ – вектор коэффициентов, $\boldsymbol{f}(x)$ – вектор функций заданного типа.

Матрица измерений $\mathbf{F}$ будет содержать в этом случае вектор-столбцы значений выбранных функций и для получения диагональной матрицы $\mathbf{F}^{\mathrm{T}}\mathbf{F}$ системы линейных уравнений нам понадобятся функции, удовлетворяющие условию

$$(f_j(x), f_k(x))=0 \text{ при } j \neq k.$$

Системы таких функций называются *ортогональными* и среди них есть класс ортогональных полиномов, а в этом

классе наиболее простой системой являются ортогональные полиномы Чебышева с весовой функцией 1. Строятся они так:

полином нулевой степени выбирается единичным $t_0(x)=1$,

полином первой степени $t_1(x)=x–a_1$, где a_1 определяется из условия ортогональности $(t_1, t_0)=0$, что дает

$\sum(x_t + a_1)=0$ или $Na_1+\sum x_t=0$ и $a_1=–(1/N)\sum x_t$ (среднее арифметическое).

Все последующие полиномы строятся по рекуррентной формуле по двум предыдущим:

$$t_{r+1}(x)=(x+b_{r+1})t_r(x)+g_{r+1}t_{r-1}(x),$$

где

$$b_{r+1}=-\frac{\sum x_t[t_r(x_t)]^2}{\sum[t_r(x_t)]^2},\ g_{r+1}=\frac{\sum x_t t_{r-1}(x_t)t_t(x_t)}{\sum[t_{r-1}(x_t)]^2}.$$

Теперь коэффициенты модели будут определяться без решения системы уравнений:

$$a_r=\frac{\sum y_t t_r(x_t)}{\sum[t_r(x_t)]^2},\ r=1, 2, ..., m.$$

Удобство использования ортогональных функций еще и в том, что при необходимости повысить степень полинома нет необходимости пересчитывать уже найденные коэффициенты модели – мы просто строим полином t_{m+1}, определяем его коэффициенты, достраиваем к матрице **F** один столбец значений этого полинома и вычисляем коэффициент a_{m+1}.

Результирующая сумма квадратов отклонений может быть вычислена по формуле:

$$\sum_{t=1}^{N}[y_t-\sum_{r=0}^{m}a_r t_r(x_t)]^2=\sum_{t=1}^{N}y_t^2-2\sum_{r=0}^{m}a_r\sum_{t=1}^{N}y_t t_r(x_t)+\sum_{r=0}^{m}a_r^2\sum_{t=1}^{N}[t_r(x_t)]^2.$$

Рассчитанные коэффициенты обычно подвергают проверке на значимость по критерию Стьюдента, незначимые коэффициенты (соответствующие им члены модели прак-

тически не влияют на значение выходной переменной) желательно удалить из модели. При использовании ортогональных базовых функций пересчет оставшихся коэффициентов не нужен, а в остальных вариантах – обязателен. Полученные в результате вычисления коэффициентов математические модели (они называются уравнениями регрессии y на x) требуют проверки на адекватность реальным данным. Эта проверка выполняется на отдельной, не участвовавшей в расчетах коэффициентов контрольной выборке данных, по отношению остаточной дисперсии y к общей дисперсии (остаточные отклонения опытных данных вычисляются по отношению к модельным, а общие – по отношению к оценке математического ожидания); полученное отношение сравнивается с критериальным, определяемым по распределению вероятностей Фишера. Но это предмет математической статистики, а не численных методов и мы не будем на этом подробнее останавливаться.

2.3. Программная реализация матричного класса

2.3.1. Общее описание структуры матричного класса

Рассмотрим структуру класса для работы с матрицами, алгоритмы для работы с объектами типа «матрица» и примеры методов матричного класса, реализующие эти алгоритмы:

```
template <class YourOwnFloatType> class matrix
{
```

Как и векторный, класс для работы с матричными объектами – параметризованный; это всего лишь шаблон, по которому для конкретных типов, подставляемых вместо YourOwnFloatType, компилятор автоматически генерирует класс. Для матричного типа подставляемыми могут быть типы, для которых определены стандартные арифметиче-

ские операции и перегружены основные операции типа возведения в степень и т.п.

```
long m, n;
```

В этих двух приватных переменных мы будем хранить размерность матрицы – число строк (в первой переменной) и столбцов (во второй). Вполне понятным требованием к этим числам является то, что они должны быть натуральными.

```
vector<YourOwnFloatType> *mtr;
```

Рассмотрим матрицу

$$\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}.$$

Каждая строка (также как и столбец) этой матрицы представляет собой упорядоченную последовательность числовых объектов, то есть вектор. Традиционно принято, что векторами являются именно строки матрицы:

$$\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}, \begin{array}{l} \boldsymbol{A}_1 = (a_{11}, a_{12}, \dots, a_{1n}), \\ \boldsymbol{A}_2 = (a_{21}, a_{22}, \dots, a_{2n}), \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots, \\ \boldsymbol{A}_m = (a_{m1}, a_{m2}, \dots, a_{mn}). \end{array}, \rightarrow \mathbf{A}=\begin{pmatrix} \boldsymbol{A}_1 \\ \boldsymbol{A}_2 \\ \dots \\ \boldsymbol{A}_m \end{pmatrix}.$$

Возможность такого представления матрицы и удобство хранения векторов-строк даёт нам основание использовать в качестве внутреннего типа для хранения матрицы указатель на вектор.

```
public:
```

Общедоступные методы (функции-члены) и дружественные функции, используемые для управления матричными объектами. Большинство из них являются реализациями соответствующих операций матричной алгебры, а потому и реализованы в виде перегруженных операций.

```
matrix(char *);
```

В качестве параметра этот конструктор принимает имя текстового файла, в котором хранится матрица. При этом предполагается, что первые два числа в файле – это размерность матрицы, то есть количество её строк и столбцов. Затем идут числа, составляющие матрицу, количество которых должно быть всего mxn. При считывании числа размещаются построчно: сначала заполняется первая строка, затем – вторая и т.д. Избыток данных в файле игнорируется, при недостатке матрица заполняется нулями, начиная с позиции, следующей за последним числом.

```
matrix();
```

Для создания массивов матриц нам необходим конструктор по умолчанию. Он создаёт пустую матрицу размером 1x1, вызывая конструктор векторного класса по умолчанию. Эта матрица фактически моделирует один элемент подставляемого типа.

```
matrix(long, long);
```

Этот конструктор принимает два параметра – натуральных числа, первое из которых задаёт количество строк в матрице, а второе – количество столбцов. Так как никаких сведений об элементах матрицы не даётся, считается, что данный конструктор предназначен для создания нулевых матриц заданного размера. Следовательно, при использовании этого конструктора отпадает необходимость очищать матрицу «перед употреблением».

```
matrix(long, long, YourOwnFloatType *);
```

Этот конструктор принимает три параметра – натуральных числа, первое из которых задаёт количество строк в матрице, а второе – количество столбцов, и указатель на элементы того типа, который используется для хранения данных в векторах. После создания матрицы заданных размеров, происходит попытка заполнения её size1xsize2 элементами из массива, на который указывает третий пара-

метр. Если этот указатель не инициализирован, или указывает на область памяти, не содержащую объектов подставляемого типа, или количество таких объектов меньше, чем произведение количества строк на количество столбцов, часть или вся матрица, в зависимости от ситуации, будет содержать мусор, над которым и будут производиться дальнейшие операции в предположении о корректности хранящихся данных.

```
matrix(matrix &);
```

Конструктор копирования матричного класса используется тогда, когда необходимо слепить матрицу по образу и подобию уже существующей. Это означает, что конструируемая матрица будет иметь ту же размерность, что и копируемая матрица-параметр, а все элементы их будут совпадать. Разумеется, данные этих матриц хранятся в разных областях памяти, и действия над одной матрицей не отражаются на другой, как это происходило бы в синтаксически похожей ситуации объявления ссылочного объекта:

matrix a=e;/*матрицы а и е одинаковые, но в разных областях памяти*/

matrix &c=a;/*матрицы с и а одинаковые и в одной области памяти*/

```
~matrix();
```

Деструктор. Если матрица была размещена в оперативной памяти как динамический объект, то он будет вызываться при выполнении оператора delete, во всех остальных – при выходе матричного объекта из области видимости. Так как единственное динамически распределяемое поле нашего класса – это указатель на вектор, то из-под него память и будет освобождена. При этом вызовутся деструкторы каждой вектор-строки, хранящейся в нашей матрице, уничтожая сами данные из строк.

```
friend matrix<YourOwnFloatType> operator+(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType> &);
```

Дружественная функция сложения матриц принимает два параметра – матрицы одинакового типа (одинаковой размерности), а матрицу-сумму возвращает как результат. Суммой двух матриц $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$ и $\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \dots & \dots & \dots & \dots \\ b_{m1} & b_{m2} & \dots & b_{mn} \end{pmatrix}$ одинакового типа называется матрица $\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{pmatrix}$ того же типа, элементы которой c_{ij} равны суммам соответствующих элементов a_{ij} и b_{ij} матриц **A** и **B**, т. е. $c_i = a_{ij} + b_{ij}$. Таким образом,

$$\mathbf{A} + \mathbf{B} = \begin{pmatrix} a_{11} + b_{11} & a_{12} + b_{12} & \dots & a_{1n} + b_{1n} \\ a_{21} + b_{21} & a_{22} + b_{22} & \dots & a_{2n} + b_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} + b_{m1} & a_{m2} + b_{m2} & \dots & a_{mn} + b_{mn} \end{pmatrix}.$$

С другой стороны, так как в качестве одного из представлений матрицы мы имеем векторное, то сумму матриц достаточно легко представить через сумму векторов, составляющих строки матриц слагаемых:

$$\begin{pmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \dots \\ \boldsymbol{A_m} \end{pmatrix} + \begin{pmatrix} \boldsymbol{B_1} \\ \boldsymbol{B_2} \\ \dots \\ \boldsymbol{B_m} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A_1} + \boldsymbol{B_1} \\ \boldsymbol{A_2} + \boldsymbol{B_2} \\ \dots \\ \boldsymbol{A_m} + \boldsymbol{B_m} \end{pmatrix} \to \begin{pmatrix} \boldsymbol{C_1} \\ \boldsymbol{C_2} \\ \dots \\ \boldsymbol{C_m} \end{pmatrix}.$$

```
friend matrix<YourOwnFloatType> operator-(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType> &);
```

Дружественная функция вычитания матриц принимает два параметра – матрицы одинакового типа (одинаковой размерности), а матрицу-разность возвращает как результат. Разность матриц определяется аналогично сумме:

$$\mathbf{A}-\mathbf{B}=\begin{pmatrix} a_{11}-b_{11} & a_{12}-b_{12} & \dots & a_{1n}-b_{1n} \\ a_{21}-b_{21} & a_{22}-b_{22} & \dots & a_{2n}-b_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1}-b_{m1} & a_{m2}-b_{m2} & \dots & a_{mn}-b_{mn} \end{pmatrix} \text{ или}$$

$$\begin{pmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \dots \\ \boldsymbol{A_m} \end{pmatrix}-\begin{pmatrix} \boldsymbol{B_1} \\ \boldsymbol{B_2} \\ \dots \\ \boldsymbol{B_m} \end{pmatrix}=\begin{pmatrix} \boldsymbol{A_1}-\boldsymbol{B_1} \\ \boldsymbol{A_2}-\boldsymbol{B_2} \\ \dots \\ \boldsymbol{A_m}-\boldsymbol{B_m} \end{pmatrix}\rightarrow\begin{pmatrix} \boldsymbol{C_1} \\ \boldsymbol{C_2} \\ \dots \\ \boldsymbol{C_m} \end{pmatrix}.$$

```
friend matrix<YourOwnFloatType> operator*(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType> &);
```

Дружественная функция некоммутативного умножения матриц, как и любая матричная бинарная операция, принимает два параметра – перемножаемые матрицы, и возвращает матрицу-результат, если типы матриц таковы, что последний определён.

Пусть $\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$ и $\mathbf{B}=\begin{pmatrix} b_{11} & b_{12} & \dots & b_{1q} \\ b_{21} & b_{22} & \dots & b_{2q} \\ \dots & \dots & \dots & \dots \\ b_{p1} & b_{p2} & \dots & b_{pq} \end{pmatrix}$ – матрицы типов соответственно mхn и pхq. Если число столбцов матрицы **A** равно числу строк матрицы **B**, т. е. $n=p$, то для этих матриц определена матрица **C** типа mхq, называемая их произведением:

$$\mathbf{C} = \begin{pmatrix} c_{11} & c_{12} & \dots & c_{1n} \\ c_{21} & c_{22} & \dots & c_{2n} \\ \dots & \dots & \dots & \dots \\ c_{m1} & c_{m2} & \dots & c_{mn} \end{pmatrix},$$

где $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \ldots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$ (i=1, 2, …, m; j=1, 2, …, q).

Из определения вытекает следующее правило умножения матриц: чтобы получить элемент, стоящий в i-й строке и j-м столбце произведения двух матриц, нужно элементы i-й строки первой матрицы умножить на соответствующие элементы j-го столбца второй и полученные произведения сложить.

Произведение **AB** имеет смысл тогда и только тогда, когда матрица **A** содержит в строках столько элементов, сколько элементов имеется в столбцах матрицы **B**. В частности, можно перемножать квадратные матрицы лишь одинакового порядка. В тех частных случаях, когда **AB**=**BA**, матрицы **A** и **B** называются *коммутативными*. Так, например, единичная матрица **E** коммутативна с любой квадратной матрицей **A** того же порядка, причем

AE=EA=A.

Таким образом, единичная матрица **E** играет роль единицы при умножении.

```
friend matrix<YourOwnFloatType> operator* (YourOwn-
FloatType, matrix<YourOwnFloatType> &);
friend matrix<YourOwnFloatType> operator*(matrix
<YourOwnFloatType> &, YourOwnFloatType );
```

В отличие от умножения матрицы на матрицу, умножение матрицы на скаляр – операция, результат которой – произведение – однозначен и определён всегда: это матрица той же размерности, что и исходная.

Произведением матрицы **A** на число α (или произведением числа α на матрицу **A**) называется матрица, элементы которая получены умножением всех элементов матрицы **A** на число α, т. е.

$$\alpha\mathbf{A} = \mathbf{A}\alpha = \begin{pmatrix} \alpha a_{11} & \alpha a_{12} & \dots & \alpha a_{1n} \\ \alpha a_{21} & \alpha a_{22} & \dots & \alpha a_{2n} \\ \dots & \dots & \dots & \dots \\ \alpha a_{m1} & \alpha a_{m2} & \dots & \alpha a_{mn} \end{pmatrix} \text{ или } \alpha\mathbf{A} = \mathbf{A}\alpha = \begin{pmatrix} \alpha \boldsymbol{A_1} \\ \alpha \boldsymbol{A_2} \\ \dots \\ \alpha \boldsymbol{A_m} \end{pmatrix}.$$

```
friend ostream &operator<<(ostream &, ma-
trix<YourOwnFloatType> &);
friend istream &operator>>(istream &, ma-
trix<YourOwnFloatType> &);
```

Вывод матрицы в поток и ввод матрицы из потока – две операции, классически перегружаемые для каждого типа. Первым параметром каждой из функций есть поток вывода (ввода), вторым – матричный объект, сконструированный ранее. Особый интерес представляет операция ввода из потока: при считывании данных содержимое текущей матрицы замещается объектами, считываемыми из потока – именно с этой целью передача матрицы в функцию организована по ссылке. При записи (вставке) матрицы в поток используется соответствующая перегруженная операция векторного класса для каждого из векторов-строк матрицы.

```
friend matrix<YourOwnFloatType> SLAE_Orto(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType> &);
```

Первый параметр этой функции – квадратная матрица *N*x*N*, второй – *N*x1, возвращаемое значение – матрица *N*x1. Рассмотрим детальнее, что это за функция, для чего она предназначена и почему её аргументы именно такие.

Пусть нам необходимо решить систему линейных уравнений вида:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ .... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases}$$

Из элементов этой системы составим три матрицы:

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ ... & ... & ... & ... \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}, \ \mathbf{X} = \begin{pmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{pmatrix} \text{ и } \mathbf{B} = \begin{pmatrix} b_1 \\ b_2 \\ ... \\ b_n \end{pmatrix}.$$

Умножив матрицу **A** на матрицу **X**, мы получим матрицу-столбец, элементы которой представляют собой левую часть записанной системы. Так как левая часть системы равна правой, то полученная матрица столбец не может быть ничем иным, как матрицей-столбцом **B**. Таким образом, мы имеем право записать систему линейных алгебраических уравнений в матричной форме как

AX=B.

В записанной системе матрица-столбец **X** представляет собой вектор неизвестных, которые необходимо будет определить, **A** – матрица коэффициентов при неизвестных и **B** – правая часть СЛАУ (матрица-столбец свободных членов, взятых с обратным знаком). Итак, задачу решения системы уравнений можно свести к задаче нахождения такой матрицы **X**, которая при умножении справа на матрицу **A** даёт матрицу **B**. Этим и объясняется требование к параметрам рассматриваемой функции: первая матрица содержит коэффициенты системы уравнений, вторая – столбец свободных членов, взятый с обратным знаком, а возвращаемым значением данной функции будет столбец неизвестных.

Будем исходить из предположения, что такая матрица существует. Тогда домножим это уравнение с обеих сторон

слева на некую матрицу $\mathbf{A}^{-1}$ такую, которая при умножении на матрицу **A** слева даёт единичную матрицу:

$$\mathbf{A}^{-1}\mathbf{AX}=\mathbf{A}^{-1}\mathbf{B},$$
$$\mathbf{EX}=\mathbf{A}^{-1}\mathbf{B},$$
$$\mathbf{X}=\mathbf{A}^{-1}\mathbf{B}.$$

Зная способ получения матрицы $\mathbf{A}^{-1}$ из матрицы **A**, мы можем решить поставленную задачу. Этот путь мы, однако, рассматривать сейчас не будем, а подойдём к проблеме с другой стороны, для чего рассмотрим *метод ортогонализации*. Пусть исходная система переписана в виде:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n - b_1 = 0 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n - b_2 = 0 \\ .... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n - b_n = 0 \end{cases}.$$

Введя обозначение $a_{i(n+1)} = -b_i$, получим

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n + a_{1(n+1)} = 0 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n + a_{2(n+1)} = 0 \\ .... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n + a_{n(n+1)} = 0 \end{cases}.$$

Левую часть каждого уравнения системы можно рассматривать как скалярное произведение двух векторов: $\boldsymbol{A_i} = \left(a_{i1}, a_{i2},..., a_{in}, a_{i(n+1)}\right)$ и $\boldsymbol{X} = \left(x_1, x_2,..., x_n, 1\right)$. При такой постановке решение системы сводится к построению вектора, ортогонального к каждому вектору $\boldsymbol{A}_i$. Добавим к системе векторов $\boldsymbol{A}_i$ линейно независимый от них вектор $\boldsymbol{A}_{n+1}$=(0, 0, …, 0, 1). В векторном пространстве размерности n+1 будем строить такой его ортонормированный базис $\boldsymbol{B}_1$, $\boldsymbol{B}_2$, …, $\boldsymbol{B}_{n+1}$, чтобы при любом k=1, 2, …, n+1 векторы $\boldsymbol{B}_1$, $\boldsymbol{B}_2$, …, $\boldsymbol{B}_k$ образовывали ортонормированный базис подпространства P_k, порожденного рассматриваемыми векторами $\boldsymbol{A}_1$, $\boldsymbol{A}_2$, …, $\boldsymbol{A}_k$. Для этого достаточно строить в подпространстве P_k не-

который ортогональный базис $\boldsymbol{U}_1$, $\boldsymbol{U}_2$, …, $\boldsymbol{U}_k$, а затем нормировать его.

Вычисления будем вести в соответствии со следующим алгоритмом. Положим:

$$\boldsymbol{U}_1 = \boldsymbol{A}_1;\ \boldsymbol{B}_1 = \frac{\boldsymbol{U}_1}{\sqrt{\sum_{j=1}^{n+1} \boldsymbol{U}_{1j}^2}}.$$

Если для некоторого $k \geq 1$ уже построены векторы $\boldsymbol{U}_1$, $\boldsymbol{U}_2$, …, $\boldsymbol{U}_k$ и векторы $\boldsymbol{B}_1$, $\boldsymbol{B}_2$, …, $\boldsymbol{B}_k$, то вектор $\boldsymbol{U}_{k+1}$ вычисляется с помощью нескольких итераций по формулам

$$\boldsymbol{U}_{k+1}^{(0)} = \boldsymbol{A}_{k+1};\ \boldsymbol{U}_{k+1}^{(t)} = \boldsymbol{U}_{k+1}^{(t-1)} - \sum_{j=1}^{k} \boldsymbol{U}_{k+1}^{(t-1)} \cdot \boldsymbol{B}_j^2.$$

Здесь $\boldsymbol{U}_{k+1}^{(0)}$ – начальное, а $\boldsymbol{U}_{k+1}^{(t)}$ – очередное приближение вектора $\boldsymbol{U}_{k+1}$. Вектор $\boldsymbol{B}_{k+1}$ по вектору $\boldsymbol{U}_{k+1}$ определяется в соответствии с формулой

$$\boldsymbol{B}_{k+1} = \frac{\boldsymbol{U}_{k+1}}{\sqrt{\sum_{j=1}^{n+1} \boldsymbol{U}_{(k+1)j}^2}}.$$

Значения корней системы связаны с координатами вектора $\boldsymbol{B}_{n+1}$ так:

$$x_i = \frac{\boldsymbol{B}_{(n+1)i}}{\boldsymbol{B}_{(n+1)(n+1)}}.$$

Обычно этот процесс рекомендуется повторить 3-4 раза с целью как можно более точной ортогонализации и, соответственно, более точного решения рассматриваемой системы. Более просто рассмотренный алгоритм можно сформулировать так:

1. Нормируем первую вектор-строку системы.
2. Ортогонализируем второй вектор к первому, а затем нормируем его.

3. Ортогонализируем третий вектор к первому и ко второму, а затем нормируем, и т.д., со всеми остальными векторами.

При этом самый последний вектор, которым мы дополнили систему, будет ортогонален ко всем предыдущим, а, значит, будет являться её решением.

```
friend matrix<YourOwnFloatType> SLAE_Gauss(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType> &);
```

Для разнообразия рассмотрим ещё один метод решения систем линейных уравнений – метод Гаусса, или метод последовательного исключения неизвестных. Суть его состоит в преобразовании системы

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ .... \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases}$$

к системе с треугольной матрицей, из которой затем последовательно (обратным ходом) получаются значения всех неизвестных.

Подвергнем эту систему следующему преобразованию. Считая, что $a_{11} \neq 0$ (ведущий элемент), разделим на a_{11} коэффициенты первого уравнения (выполнения условия $a_{11} \neq 0$ можно добиться всегда путем перестановки уравнений системы):

$$x_1 + \alpha_{12}x_2 + ... + \alpha_{1n}x_n = \beta_1$$

Пользуясь этим уравнением, легко исключить неизвестное x_1 из остальных уравнений системы (для этого достаточно из каждого уравнения вычесть его, предварительно умноженное на соответствующий коэффициент при x_1). Затем над остальными уравнениями системы совершим аналогичное преобразование: выберем из их числа уравнение с ведущим элементом и исключим с его помощью из остальных уравнений неизвестное x_2. Повторяя этот про-

цесс, вместо исходной системы получим равносильную ей систему с треугольной матрицей коэффициентов при неизвестных:

$$\begin{cases} x_1 + \alpha_{12}x_2 + \alpha_{13}x_3 + \ldots + \alpha_{1n}x_n = \beta_1 \\ \qquad x_2 + \alpha_{23}x_3 + \ldots + \alpha_{2n}x_n = \beta_2 \\ \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ \qquad\qquad\qquad\qquad\qquad x_n = \beta_n \end{cases},$$

из которой последовательно находятся значения всех неизвестных x_n, x_{n-1}, …, x_1.

Таким образом, процесс решения по методу Гаусса распадается на два этапа. Первый этап, состоящий в последовательном исключении неизвестных, называют прямым ходом. Второй – нахождение значений неизвестных – принято называть обратным ходом.

```
matrix<YourOwnFloatType> minor(long, long);
```

Эта функция в программах обычно не применяется и в основном служит для внутреннего употребления. Имеет два параметра – номер строки и столбца данной матрицы, которые не переписываются в матрицу-результат (создание матрицы из имеющейся без заданных строки и столбца). При этом размерность матрицы (количество строк и количество столбцов) уменьшается на единицу.

```
friend YourOwnFloatType det2(matrix<YourOwnFloatType>
&);
```

Вычисление значения определителя квадратной матрицы является важной задачей линейной алгебры. Так, численное решение системы уравнений имеет смысл лишь в том случае, когда матрица, составленная из коэффициентов при неизвестных этой системы, невырожденная, т. е. когда ее определитель отличен от нуля. Однако и в том случае, когда определитель системы отличен от нуля, но очень мал по абсолютной величине, к полученным в ходе решения значениям корней нужно относиться с осторожностью, по-

скольку они могут значительно отличаться от истинного значения неизвестных. Поэтому решение системы линейных уравнений полезно сопровождать вычислением определителя этой системы. Вычисление определителя может представлять и самостоятельный интерес.

Определитель (детерминант) порядка n, задаваемый выражением

$$\det \mathbf{A} = \begin{vmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{vmatrix},$$

представляет собой число, определяемое по изложенному ранее правилу. Из курса линейной алгебры известно правило, по которому определитель порядка n можно выразить через n определителей порядка на единицу ниже:

$$\Delta = a_{i1}A_{i1} + a_{i2}A_{i2} + \dots + a_{in}A_{in}.$$

Это – операция разложения определителя по элементами i-ой строки; $A_{ij} = (-1)^{i+j} M_{ij}$ – алгебраическое дополнение элемента a_{ij}, M_{ij} – минор элемента a_{ij}, т.е. определитель (n-1)-го порядка, получаемый из определителя Δ вычёркиванием i-ой строки и j-го столбца.

Описанная процедура позволяет реализовать очень простую рекурсивную функцию для вычисления детерминанта разложением по одной из строк; при этом, однако, число умножений и делений, необходимых для вычисления определителя n-го порядка, равно

$$\frac{n-1}{3}\left(n^2 + n + 3\right),$$

что сдерживает применение такого метода для вычисления детерминантов высоких порядков.

```
friend YourOwnFloatType det(matrix<YourOwnFloatType>
&);
```

Для системы уравнений из матрицы коэффициентов мы можем составить детерминант и вычислить его значение. При этом любые изменения матрицы коэффициентов ведут к изменению её детерминанта. Рассмотрим, как в методе Гаусса меняется определителем исходной системы. Обозначим определитель системы уравнений через D. После того, как мы разделим левую и правую части первого уравнения на ведущий элемент a_{11}, определитель преобразованной системы будет равен D/a_{11}, последующие преобразования, связанные с исключением x_1 из остальных уравнений системы, величину определителя не изменяют.

На втором шаге, когда мы разделим обе части преобразованного второго уравнения на второй ведущий элемент a_{22}, определитель полученной системы будет равен $D/(a_{11}\cdot a_{22})$. Операции по исключению x_2 из уравнений системы вновь не изменяют величины определителя.

Осуществляя аналогичные действия, мы на n-м шаге придем к системе, определитель которой будет равен $D/(a_{11}\cdot a_{22}\cdot\ldots\cdot a_{nn})$. Но матрица коэффициентов при неизвестных системы – треугольная, с единицами на главной диагонали, поэтому ее определитель равен 1. Получаем, что $D/(a_{11}\cdot a_{22}\cdot\ldots\cdot a_{nn})=1$, то есть $D=a_{11}\cdot a_{22}\cdot\ldots\cdot a_{nn}$.

Таким образом, для вычисления определителя системы уравнений нужно получить произведение ведущих элементов, используемых на каждом шаге метода Гаусса.

```
matrix<YourOwnFloatType> opera-
tor=(matrix<YourOwnFloatType> &);
```

Присвоение матриц – бинарная операция, которую реализуют как метод (функцию-член) класса. Если тип (размерность) текущей матрицы не совпадает с размерностью присваиваемой, мы вначале перераспределяем память под её вектора таким образом, чтобы количество векторов и их размерности в обеих матрицах совпали. Затем происходит повекторное копирование строк копируемой матрицы в

строки текущей. Возвращаемое значение – копия текущей матрицы.

```
matrix<YourOwnFloatType>
operator*=(matrix<YourOwnFloatType> &x);
```

При работе со встроенными типами данных удобной возможностью является использование набора сокращённых операций, в которых действие комбинируется со знаком присвоения. Например, эта функция перегружает операцию сокращённого умножения. Будучи реализована как метод класса, она умножает текущую матрицу на переданную как параметр, а результат умножения не только возвращается, но и перезаписывается в текущую матрицу. Действие этой операции, разумеется, имеет те же ограничения, что накладываются на умножение матриц.

```
matrix<YourOwnFloatType> operator+= (ma-
trix<YourOwnFloatType> &x);
matrix<YourOwnFloatType> operator-= (ma-
trix<YourOwnFloatType> &x);
```

Как и сокращённое умножение, сокращённые операции сложения и вычитания реализованы как методы матричного класса. Обе эти функции работают, используя уже определённые операции сложения, вычитания и присвоения.

```
matrix<YourOwnFloatType> operator^(long);
```

Для квадратных матриц мы можем определить такую операцию, как возведение матрицы в целую степень (отрицательные степени при этом не рассматриваются) по следующему закону:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}^k = \begin{cases} \begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{pmatrix}, k = 0 \\ \underbrace{\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix} x \dots x \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{pmatrix}}_{k\ раз}, k > 0 \end{cases}$$

```
matrix<YourOwnFloatType> operator^=(long);
friend matrix<YourOwnFloatType>
pow(matrix<YourOwnFloatType> &, long);
```

Для удобства работы определим также функцию-член для сокращённой операции возведения в неотрицательную степень, а также дружественную функцию для возведения в степень.

```
matrix<YourOwnFloatType> operator~();
```

Заменив в матрице $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$ типа mхn строки соответственно столбцами, получим так называемую транспонированную матрицу $\mathbf{A}^{\mathbf{T}} = \begin{pmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \dots & \dots & \dots & \dots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{pmatrix}$ типа nхm.

Транспонированная матрица обладает следующими свойствами:

1) дважды транспонированная матрица совпадает с исходной;

2) транспонированная матрица суммы равна сумме транспонированных матриц слагаемых;

3) транспонированная матрица произведения равна произведению транспонированных матриц сомножителей, взятому в обратном порядке.

Если матрица **A** квадратная, то $\det \mathbf{A}^T = \det \mathbf{A}$. Матрица **A** называется симметричной, если она совпадает со своей транспонированной, т. е. если $\mathbf{A}^T = \mathbf{A}$. Отсюда вытекает, что симметричная матрица – квадратная и элементы ее, симметричные относительно главной диагонали, равны между собой. Произведение $\mathbf{C} = \mathbf{A}\mathbf{A}^T$, очевидно, представляет собой симметричную матрицу, так как $\mathbf{C}^T = (\mathbf{A}\mathbf{A}^T)^T = (\mathbf{A}^T)^T\mathbf{A}^T = \mathbf{A}\mathbf{A}^T = \mathbf{C}$.

Рассмотрим систему линейных алгебраических уравнений, записанных в матричной форме, и предположим, что количество уравнений в этой системе превышает количество неизвестных. Разумеется, при использовании, к примеру, метода последовательного исключения неизвестных мы после приведения системы к треугольному виду получим «лишние» уравнения. Если эта система уравнений определяет какую-то зависимость, то точное восстановление этой зависимости становится проблематичным. Однако, используя операцию транспонирования, мы можем получить приблизительные решения этой системы, причём такие, что вычисленная по ним правая часть каждого уравнения будет достаточно мало отличаться от исходной. Запишем эту процедуру в форме матричного уравнения:

$$\mathbf{A}\mathbf{X} = \mathbf{B}$$

$$\mathbf{A}^T\mathbf{A}\mathbf{X} = \mathbf{A}^T\mathbf{B}$$

$$\mathbf{A}^T\mathbf{A} = \mathbf{A}', \ \mathbf{A}^T\mathbf{B} = \mathbf{B}'$$

$$\mathbf{A}'\mathbf{X} = \mathbf{B}'$$

Если решение исходной системы найти было невозможно, то решение преобразованной вполне возможно. Рассмотрим, каковы размерности матриц в последнем уравнении.

Размерность матрицы **A** – mxn, размерность матрицы $\mathbf{A}^{\mathrm{T}}$ – nxm, размерность матрицы **X** – nx1, размерность матрицы **B** – mx1. Размерность произведения $\mathbf{A}^{\mathrm{T}}\mathbf{A}$, согласно правилу умножения матриц, будет nxn, размерность произведения $\mathbf{A}^{\mathrm{T}}\mathbf{B}$ – nx1, что удовлетворяет параметрам функции для решения системы уравнений. Система, определённая таким способом, называется нормальной; матрица этой системы называется матрицей плана, а решение этой системы будет оптимальным по критерию минимизации суммы квадратов отклонений от истинных значений, или, как его называют, критерию МНК – метода наименьших квадратов.

```
matrix<YourOwnFloatType> operator!();
```

Пусть у нас есть набор значений некоторой неизвестной функции, связывающей набор переменных со значением-результатом. Это может быть, к примеру, функция зависимости скорости роста стебля от влажности, температуры, давления и каких-то ещё параметров. Выдвинем гипотезу о виде связи этих параметров с результатом, причём будем считать, что эта зависимость задаётся в виде линейной комбинации известных функций, связывающих эти параметры.

Подставляя конкретные значения в функцию-гипотезу, мы получим систему уравнений, в правой части которой будет линейная комбинация неизвестных коэффициентов функции-гипотезы с числовыми значениями, полученными при подстановке, а в левой – результаты измерений. Если количество уравнений в этой системе у нас будет меньше, чем число коэффициентов в функции-гипотезе, то у нас просто недостаточно данных, чтобы вычислить эти коэффициенты. Если эти значения совпадают или уравнений больше, чем неизвестных, такую систему приводят к нормальной и определяют коэффициенты функции-гипотезы – модели данного процесса. Обозначая через **X** матрицу системы, составленной по модельной функции, через **Y** –

матрицу-столбец измеренных значений, а через **A** – матрицу-столбец неизвестных коэффициентов модели, имеем:

$\mathbf{XA}=\mathbf{Y}$ – фиктивное матричное уравнение

$\mathbf{X}^{T}\mathbf{XA}=\mathbf{X}^{T}\mathbf{Y}$ – реальное матричное уравнение

$(\mathbf{X}^{T}\mathbf{X})\mathbf{A}=\mathbf{X}^{T}\mathbf{Y}$

$\mathbf{A}=(\mathbf{X}^{T}\mathbf{Y})\cdot(\mathbf{X}^{T}\mathbf{X})^{-1}$

И снова мы встречаемся с особой матрицей, которую обозначили как матрицу в минус первой степени, причём она весьма явно связана с нахождением решения системы уравнений. Это свойство мы используем в следующей функции, а пока рассмотрим понятие обратной матрицы.

Обратной матрицей по отношению к данной называется матрица, которая, будучи умноженной как справа, так и слева на данную матрицу, дает единичную матрицу. Для матрицы **A** обозначим обратную ей матрицу через $\mathbf{A}^{-1}$. Тогда по определению имеем:

$$\mathbf{AA}^{-1}=\mathbf{A}^{-1}\mathbf{A}=\mathbf{E},$$

где **E** – единичная матрица. Нахождение обратной матрицы для данной называется обращением данной матрицы.

Квадратная матрица называется неособенной, если определитель ее отличен от нуля. В противном случае матрица называется особенной, или сингулярной. Всякая неособенная матрица имеет обратную матрицу, которая записывается как:

$$\mathbf{A}^{-1}=\begin{pmatrix} \frac{A_{11}}{\Delta} & \frac{A_{21}}{\Delta} & \dots & \frac{A_{n1}}{\Delta} \\ \frac{A_{12}}{\Delta} & \frac{A_{22}}{\Delta} & \dots & \frac{A_{n2}}{\Delta} \\ \dots & \dots & \dots & \dots \\ \frac{A_{1n}}{\Delta} & \frac{A_{2n}}{\Delta} & \dots & \frac{A_{nn}}{\Delta} \end{pmatrix},$$

где A_{ij} – алгебраические дополнения соответствующих элементов a_{ij}.

Особенная квадратная матрица обратной не имеет, так как её определитель равен нулю.

Укажем некоторые основные свойства обратной матрицы:

1. Определитель обратной матрицы равен обратной величине определителя исходной матрицы.

2. Обратная матрица произведения квадратных матриц равна произведению обратных матриц сомножителей, взятому в обратном порядке.

3. Транспонированная обратная матрица равна обратной от транспонированной данной матрицы.

Определив таким образом обратную для данной матрицу, мы можем решить систему уравнений двумя операциями – обращением и умножением.

```
matrix<YourOwnFloatType> operator*();
```

Обращение матрицы описанным выше способом – операция достаточно трудоёмкая. Вспомним, что в процессе решения системы уравнений мы фактически неявно находим обратную матрицу. Для того, чтобы найти её в явном виде, используем следующий приём: запишем вместе квадратную матрицу и единичную. Будем осуществлять такие элементарные преобразования над ними, чтобы привести первую матрицу к диагональной форме. А вторая матрица – будет содержать матрицу, обратную первой.

```
YourOwnFloatType operator&();
operator YourOwnFloatType();
```

Для удобства использования составим две операторные функции, определяющие численный детерминант как унарную операцию. Так как детерминант можно рассматривать как оператор преобразования матрицы к числовому значению, запишем его ещё как функцию преобразования к типу.

```
matrix<YourOwnFloatType> operator-();
```

Унарный минус – операция, из матрицы $\mathbf{A}=\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{pmatrix}$ составляющей противоположную матрицу того же типа:

$$-\mathbf{A}=(-1)\mathbf{A}=\begin{pmatrix} -a_{11} & -a_{12} & \dots & -a_{1n} \\ -a_{21} & -a_{22} & \dots & -a_{2n} \\ \dots & \dots & \dots & \dots \\ -a_{m1} & -a_{m2} & \dots & -a_{mn} \end{pmatrix}.$$

```
matrix<YourOwnFloatType> operator+();
```

Унарный плюс – операция, приведённая только для полноты набора операций; возвращаемое значение – копия текущей матрицы.

```
friend long operator==(matrix<YourOwnFloatType> &, matrix
<YourOwnFloatType> &);
```

Дружественная функция проверки на равенство сравнивает две матрицы и возвращает ненулевое значение, если они равны.

Две матрицы считаются равными, если они одного и того же типа, т.е. имеют одинаковые размерности, и соответствующие их элементы (вектора-строки) равны.

```
friend long operator!=(matrix<YourOwnFloatType> &, matrix
<YourOwnFloatType> &);
```

Дружественная функция проверки на неравенство сравнивает две матрицы и возвращает ненулевое значение, если они не равны. Операция заключается в проверке этих матриц на равенство и логическое отрицание полученного результата.

```
vector<YourOwnFloatType> &operator[](long a);
```

Метод матричного класса для индексирования элементов матрицы принимает в качестве параметра номер век-

тор-строки матрицы, которую необходимо получить. Если этот номер корректен, возвращается ссылка на соответствующий вектор, который, в свою очередь, тоже можно проиндексировать. Таким образом, однократное применение операции индексирования позволяет получить вектор, а двукратное – соответствующий элемент этого вектора. Так как данная операция возвращает ссылочное значение, его можно использовать в операциях присваивания как слева, так и справа. При выходе индекса за допустимый диапазон значений в стандартный поток вывода вставляется специальное диагностическое сообщение и возвращается специальный вектор-признак ошибки.

```
long getm() { return m; }
long getn() { return n; }
```

Два служебных метода класса для определения числа строк и числа столбцов.

```
};
```

2.3.2. Интерфейсный файл реализации матричного класса matrix.h

По приведенным описаниям можно составить, к примеру, такой интерфейс для данного класса:

```
#ifndef __MATRIX_H
#define __MATRIX_H
#ifndef __VECTOR_H
#include "vector.h"
#endif
#ifndef __FSTREAM_H
#include <fstream.h>
#endif
#ifndef __MATH_H
#include <math.h>
#endif
#ifndef __STDIO_H
#include <stdlib.h>
#endif
#ifndef __EXCEPT_H
```

```
#include <except.h>
#endif
#ifndef __CSTRING_H
#include <cstring.h>
#endif
#ifndef __COMPLEX_H
#include <complex.h
#endif

/*параметризованный класс для работы с матричными
объектами*/
template <class YourOwnFloatType> class matrix
{ //приватные данные
  long m,n;/*row, columns - размерность матрицы в
строках и столбцах */
  vector<YourOwnFloatType> *mtr; /*указатель на
вектор данных*/
 public://общедоступные члены
  matrix(char *);/*загрузка матрицы из файла в фор-
мате m n d11 d12 ... dmn*/
  matrix();//пустая матрица размером 1x1
  matrix(long, long);/*пустая матрица заданного
размера*/
  matrix(long, long, YourOwnFloatType *);/*матрица
size1xsize2 из массива */
  matrix(matrix &);//конструктор копирования
  ~matrix();//деструктор
  friend matrix<YourOwnFloatType> operator+(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType>
&);//сложение
  friend matrix<YourOwnFloatType> operator-(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType>
&);//вычитание
  friend matrix<YourOwnFloatType> operator*(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType>
&);//умножение матриц
  friend matrix<YourOwnFloatType> operator*(
YourOwnFloatType, matrix<YourOwnFloatType>
&);//умножение числа на матрицу
  friend matrix<YourOwnFloatType> operator*(matrix
<YourOwnFloatType> &, YourOwnFloatType
);/*умножение матрицы на число */
```

```
  friend ostream &operator<<(ostream &,matrix
<YourOwnFloatType> &);//вывод матрицы в поток
  friend istream &operator>>(istream &,matrix
<YourOwnFloatType> &);//ввод матрицы из потока
/*первый параметр - квадратная матрица NxN, второй
- Nx1 (матричное уравнение) */
  friend matrix<YourOwnFloatType> SLAE_Orto(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType>
&);//метод ортогонализации
  friend matrix<YourOwnFloatType> SLAE_Gauss(matrix
<YourOwnFloatType> &, matrix<YourOwnFloatType>
&);//метод Гаусса с выбором главного элемента
  friend YourOwnFloatType det2(matrix
<YourOwnFloatType> &); //аналитический детерминант
  friend YourOwnFloatType det(matrix
<YourOwnFloatType> &); //численный детерминант
  matrix<YourOwnFloatType> minor(long, long);
/*минор - создание матрицы из имеющейся без задан-
ных строки и столбца */
  matrix<YourOwnFloatType> operator=(matrix
<YourOwnFloatType> &);//присвоение
  matrix<YourOwnFloatType> operator*=(matrix
<YourOwnFloatType> &x);/*набор сокращённых операций
умножения,*/
  matrix<YourOwnFloatType> operator+=(matrix
<YourOwnFloatType> &x);//сложения
  matrix<YourOwnFloatType> operator-=(matrix
<YourOwnFloatType> &x); //и вычитания
  matrix<YourOwnFloatType>
operator^(long);/*степень матрицы как операция */
  matrix<YourOwnFloatType> operator^=(long);
/*сокращённая степень */
  friend matrix<YourOwnFloatType>pow(matrix
<YourOwnFloatType> &,long);
//степень матрицы как дружественная функция
  matrix<YourOwnFloatType> operator~();
//транспонирование
  matrix<YourOwnFloatType> operator!();
/*аналитическое обращение матрицы */
  matrix<YourOwnFloatType> operator*(); /*численное
обращение матрицы */
  YourOwnFloatType operator&();/*численный детерми-
нант унарная операция */
```

```
  operator YourOwnFloatType();
/*быстрый детерминант как оператор преобразования к
типу-параметру*/
  matrix<YourOwnFloatType> operator-();/*унарный
минус*/
  matrix<YourOwnFloatType> operator+();/*унарный
плюс*/
  friend long operator==(matrix<YourOwnFloatType>
&, matrix<YourOwnFloatType> &);/*проверка на равен-
ство*/
  friend long operator!=(matrix<YourOwnFloatType>
&, matrix<YourOwnFloatType> &);/*проверка на нера-
венство*/
  vector<YourOwnFloatType> &operator[](long a);
//индексирование матрицы
  long getm() { return m; }//число строк
  long getn() { return n; }//число столбцов
};

/*Вспомогательные функции - функция, определяющая
знак числа */

inline long sign(long double x)
{
  return (x<0) ? -1 : 1;
}

double fabs(complex x) //
{
  return abs(x);
}

extern const long double MAX_LONGDOUBLE;
/*
Напомним, что матрица - это тоже вектор, элементами
которого являются не числовые объекты, а арифмети-
ческие вектора. В связи с этим между векторным и
матричным классами есть много общего, однако иногда
можно наблюдать и существенные отличия. Возьмём,
скажем, классический файловый метод хранения дан-
ных. Если для вектора достаточно было указать одно
служебное число - его длину (размерность), то для
матрицы их требуется уже два, причём первое из этих
```

```
чисел будет определять количество векторов в матри-
це, а второе - размерность каждого из этих векто-
ров. В файловых операциях с векторами нам поможет
определённая в векторном классе операция ввода век-
тора с некоторого устройства */

template <class YourOwnFloatType> matrix
<YourOwnFloatType>::matrix(char *f) /*имя файла с
матрицей*/
{
  long i;
  ifstream fp=f;//пытаемся открыть файл
  if(!fp)
    throw xmsg("Не могу открыть файл "+string(f)+
"\n");
  fp>>m>>n;//размеры матрицы считываем из файла
  if(m<=0||n<=0)
    throw xmsg("Некорректный размер матрицы \n");
  try
  {
    //здесь работает конструктор без параметров
    mtr=new vector<YourOwnFloatType>[m];/*попытка
выделения памяти */
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(i=0;i<m;i++)/*и только здесь вектора расширя-
ется до нужной размерности */
    mtr[i]=vector<YourOwnFloatType>(n);
  for(i=0;i<m&&fp>>mtr[i];i++);/*при вводе исполь-
зуем перегруженную операцию класса vector*/
}


/*Зная только размеры матрицы, мы можем считать её
нулевым элементом данного размера и сконструировать
соответствующим образом: */
template <class YourOwnFloatType> matrix
<YourOwnFloatType>::matrix(long a, long b):
m(a),n(b) //число строк и число столбцов
{
```

```
  long i;
  if(m<=0||n<=0)
    throw xmsg("Некорректный размер матрицы \n");
  try
  {
    //здесь работает конструктор без параметров
    mtr=new vector<YourOwnFloatType>[m];/*попытка
выделения памяти */
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(i=0;i<m;i++)/*расширяем вектора до нужного
размера*/
    mtr[i]=vector<YourOwnFloatType>(n);
}


/*Получив о матрице все возможные сведения, имеет
смысл, сконструировав её, инициализировать соответ-
ствующими элементами: */
template <class YourOwnFloatType>matrix <YourOwn-
FloatType>::matrix(long a, long b, YourOwnFloatType
*mt):m(a),n(b)
{
  if(m<=0||n<=0)
    throw xmsg("Некорректный размер матрицы \n");
  try
  {
    //здесь работает конструктор без параметров
    mtr=new vector<YourOwnFloatType>[m];/*попытка
выделения памяти*/
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(long i=0;i<m;i++)
    mtr[i]=vector<YourOwnFloatType>(n);//расширение
  for(long i=0,l=0;i<m;i++)
    for(long j=0;j<n;j++)
```

```
      mtr[i][j]=mt[l++];/*сконструировав, копируем
данные из массива в матрицу*/
}


/*Обратный случай (когда мы не знаем ни размеров,
ни элементов матричных векторов) решается достаточ-
но просто - созданием матрицы из одного единствен-
ного вектора длиною в один элемент: */
template <class YourOwnFloatType>
matrix<YourOwnFloatType>::matrix():m(1),n(1)
{
  try
  {
  //здесь работает конструктор без параметров
    mtr=new vector<YourOwnFloatType>[m];/*попытка
выделения памяти, а обнулится она вектором*/
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
}


/*Деструктор: уничтожение матрицы автоматически
уничтожает все её векторы: */
template <class YourOwnFloatType>
matrix<YourOwnFloatType>::~matrix()
{
  delete []mtr;/*вызов деструкторов для всех векто-
ров*/
}


/* Как и векторы, матрицы тоже бывает необходимым
проиндексировать; результатом этого действия, есте-
ственно, будет соответствующий вектор: */
template <class YourOwnFloatType> vector <YourOwn-
FloatType> &matrix<YourOwnFloatType>:: opera-
tor[](long a)
{
  static vector<YourOwnFloatType> error;
```

```
  error[0]=MAX_LONGDOUBLE;
  if(a>=0&&a<m)
    return mtr[a];/*а дальше можно индексировать
вектор*/
  else
  {
    cerr<<"Индекс "<<a<<" вне матрицы \n";
    return error;
  }
}


/*Конструктор копирования. Если в памяти ЭВМ уже
есть какая-то матрица, с неё можно снять копию-
слепок: */
template <class YourOwnFloatType> matrix
<YourOwnFloatType>::matrix(matrix<YourOwnFloatType>
&ex): m(ex.m), n(ex.n)
{
  try
  {
    //здесь работает конструктор без параметров
    mtr=new vector<YourOwnFloatType>[m];/*попытка
выделения памяти*/
  }
  catch(xalloc)
  {
    throw xmsg("Не хватает памяти \n");
  }
  for(long i=0;i<m;i++)
    mtr[i]=ex[i];/*очень громоздкое индексирование
и присваивание векторов*/
}


/* Сложение матриц одинаковой размерности сводится
к сложению соответствующих векторов: */
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> operator+(matrix <YourOwnFloatType> &f,
matrix<YourOwnFloatType> &s)
{
  if(f.m!=s.m||f.n!=s.n)
```

```
    throw xmsg("Размерности слагаемых матриц не 
совпадают \n");
  matrix<YourOwnFloatType> temp(f.m, f.n);
/*временная матрица*/
  for(long i=0;i<f.m;i++)
    temp[i]=f[i]+s[i];//слагаем вектора матриц
  return temp;//возвращаем результат
}


/*
Как и для векторов, определим унарный "минус":
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: operator-()
{
  matrix<YourOwnFloatType> temp(m, n);
  for(long i=0;i<m;i++)
    temp[i]=-(*this)[i];
  return temp;//возвращаем результат
}


//унарный плюс
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: operator+()
{
  return *this; /*возвращаем себя без каких-либо 
изменений*/
}


/*
В отличие от скалярного произведения векторов, 
умножение матриц является бинарной алгебраической 
операцией, однако только для отдельных видов мат-
риц, к тому же эта операция не является коммутатив-
ной!
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> operator*(matrix <YourOwnFloatType> &f, 
matrix<YourOwnFloatType> &s)
{
```

```
  if(f.n!=s.m)
  throw xmsg("Умножение матриц с данными размерами
невозможно \n");
  matrix<YourOwnFloatType> temp(f.m, s.n);
  for(long j=0;j<s.n;j++)
for(long i=0;i<f.m;i++)
    for(long k=0;k<f.n;k++)
      temp[i][j]+=f[i][k]*s[k][j];
  return temp;
}


/*
Операция увеличения матрицы в некоторое число раз:
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> operator*(matrix <YourOwnFloatType> &f,
YourOwnFloatType s)
{
  matrix<YourOwnFloatType> temp=f;
  for(long i=0;i<f.m;i++)
    temp[i]=temp[i]*s;/*здесь используем умножение
вектора на число*/
  return temp;
}


/*
Оператор возведения матрицы в целую степень можно
определить так: любая матрица в нулевой степени яв-
ляется единичной, положительная степень определяет-
ся через произведение, отрицательная - через обра-
щение матрицы и произведение. Последний случай мож-
но красиво реализовать с использованием рекурсии,
как, впрочем, и предыдущий.
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: opera-
tor^(long l)
{
  matrix<YourOwnFloatType> temp(getm(),getn());
  if(getm()!=getn())
```

```
    throw xmsg("Для неквадратных матриц степень не
определена \n");
  if(l==0)
  {
    for(long i=0;i<getm();i++)
      temp[i][i]=1;
    return temp;
  }
  if(l>0)
  {
    temp=(*this);
    for(long i=1;i<l;i++)
      temp*=(*this);
    return temp;
  }
  else
    return (*(*this))^(-l);
}


//сокращённая операция возведения матрицы в степень
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: opera-
tor^=(long l)
{
  return (*this)=(*this)^l;
}


/*
Степень матрицы в виде функции
*/
template <class YourOwnFloatType> inline matrix
<YourOwnFloatType> pow(matrix<YourOwnFloatType> &x,
long l)
{
  return x^l;
}


/*
```

```
Умножение числа на матрицу реализуем через уже име-
ющуюся операцию умножения матрицы на число для со-
хранения коммутативности
*/
template <class YourOwnFloatType> inline matrix
<YourOwnFloatType> operator*(YourOwnFloatType f,
matrix<YourOwnFloatType> &s)
{
  return s*f;
}


/*
Вычитание традиционно реализуем через сложение и
унарный минус:
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> operator-(matrix <YourOwnFloatType> &f,
matrix<YourOwnFloatType> &s)
{
  return f+(-s);
}


/*Операция присвоения
При присвоении содержимое матрицы x переписывается
в текущую сразу только тогда, когда их размеры сов-
падают. В противном случае приходится уничтожать
текущую матрицу, заново её создавать с новыми раз-
мерами и лишь тогда производить повекторное копиро-
вание
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>::operator= (ma-
trix <YourOwnFloatType> &x)
{
  if(m!=x.m||n!=x.n)
  {
    delete []mtr;//уничтожаем содержимое матрицы
    m=x.m, n=x.n;//устанавливаем новые размеры
    try
    {
```

```
      mtr=new vector<YourOwnFloatType>[m];/*попытка
выделения памяти */
    }
    catch(xalloc)
    {
      throw xmsg("Не хватает памяти \n");
    }
  }
  for(long i=0;i<m;i++)
    mtr[i]=x[i];
  return *this;//возвращение себя
}


/*
  Набор сокращённых операций:
  умножение
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: opera-
tor*=(matrix<YourOwnFloatType> &x)
{
  return (*this)=(*this)*x;
}


/*
  сложение
*/
template <class YourOwnFloatType> inline matrix
<YourOwnFloatType> matrix<YourOwnFloatType>:: oper-
ator+=(matrix<YourOwnFloatType> &x)
{
  return (*this)=(*this)+x;
}


/*
  вычитание
*/
template <class YourOwnFloatType> inline matrix
<YourOwnFloatType> ma-
```

```
trix<YourOwnFloatType>::operator-
=(matrix<YourOwnFloatType> &x)
{
  return (*this)+=-x;/* используем только что сде-
ланное сокращённое сложение*/
}


/*
Очень часто бывает нужна унарная операция транспо-
нирования:
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: operator~()
{
  matrix<YourOwnFloatType> temp(n,m);
  for(long j=0;j<n;j++)
    for(long i=0;i<m;i++)
      temp[j][i]=mtr[i][j];
  return temp;//переставлены столбцы и строки
}


/*Сравнение матриц
  Сравнивая матрицы, мы сначала учитываем, одинако-
вой ли они размерности, а потом в случае необходи-
мости сравниваем векторы:
*/
template <class YourOwnFloatType> long operator==
(matrix<YourOwnFloatType> &f, ma-
trix<YourOwnFloatType> &s)
{
  if(f.m!=s.m||f.n!=s.n)
    return 0;
  for(long i=0;i<f.m;i++)
    if(f[i]!=s[i])
      return 0;
  return 1;
}


/*
  Неравенство
```

```
*/
template <class YourOwnFloatType> inline long oper-
ator!=(matrix<YourOwnFloatType> &f, ma-
trix<YourOwnFloatType> &s)
{
  return !(f==s);//!operator==(f,s);
}


/*Вывод матрицы в поток
Результат иногда хочется посмотреть. Например, так:
*/
template <class YourOwnFloatType> ostream & opera-
tor<<(ostream &os,matrix<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)/*построчный вывод векто-
ров матрицы*/
    os<<x[i]<<"\n";
  return os;
}

/*Или так... (Запись матрицы в файл в градациях се-
рого)*/
template <class YourOwnFloatType>
void writebmp(matrix<YourOwnFloatType> &x, char
*name)
{
  ofstream f(name, ios::out|ios::binary);/*Битовое
изображение */
  BITMAPFILEHEADER fh;/*файловый заголовок */
  BITMAPINFOHEADER ih;//информационный заголовок
  RGBQUAD bmiColors[256];//256 цветов
  fh.bfType=0x4d42;//буквы "BM"
  fh.bfSize=sizeof(fh)+sizeof(ih)+
  sizeof(bmiColors)+x.getm()*((x.getn()%4==0)
  ?x.getn():(x.getn()+4-x.getn()%4));//размер файла
  fh.bfReserved1=fh.bfReserved2=0;
  fh.bfOffBits=sizeof(fh)+sizeof(ih)+
       sizeof(bmiColors);//смещение до данных
  f.write((unsigned char*)&fh,sizeof(fh));/*пишем
файловый заголовок */
  ih.biSize=40;//размер информационного заголовка
  ih.biWidth=x.getn();//ширина изображения
```

```
  ih.biHeight=x.getm();//высота
  ih.biPlanes=1;//количество планов изображения
  ih.biBitCount=8;//бит на цвет
  ih.biCompression=BI_RGB;//без сжатия
  ih.biSizeImage=0;//не актуален
  ih.biXPelsPerMeter=2963;/*рекомендуемое разреше-
ние по X*/
  ih.biYPelsPerMeter=3158;//                  по У
  ih.biClrUsed=256;//количество используемых цветов
  ih.biClrImportant=0;//не актуально
  f.write((unsigned char*)&ih,sizeof(ih));/*пишем
информационный заголовок*/
  for(int i=0;i< (1 << ih.biBitCount);i++)/*цикл по
количеству цветов*/
  {
    /*приравнивая красную, зелёную и голубую со-
ставляющие цвета, получаем оттенок серого*/
    bmiColors[i].rgbBlue=bmiColors[i].rgbGreen=
                         bmiColors[i].rgbRed=i;
    bmiColors[i].rgbReserved=0;
    f.write((unsigned char*)(bmiColors+i),
      sizeof(RGBQUAD));//пишем элемент палитры
  }
  YourOwnFloatType __max=x[0][0],
__min=x[0][0];//поиск диапазона значений матрицы
  for(i=0;i<x.getm();i++)
    for(long j=0;j<x.getn();j++)
    {
      if(x[i][j]>__max)
        __max=x[i][j];
      if(x[i][j]<__min)
        __min=x[i][j];
    }
  for(i=x.getm()-1;i>=0;i--)/*строки в BMP-файле
лежат снизу вверх*/
  {
    for(long j=0;j<x.getn();j++)//записываем строку
    { //формируем номер в палитре оттенков серого
      BYTE b=0xff*(x[i][j]-__min)/(__max-__min);
      f.write(&b,1);
    }
    if(x.getn()%4)/*дополняем строку до границы
двойного слова*/
```

```
      for(j=0;j<4-x.getn()%4;j++)
        f.write(&j,1);
  }
}


/*
Ввод матрицы из потока целиком перекладываем на
векторный класс, используя его метод для ввода
*/
template <class YourOwnFloatType> istream & opera-
tor>>(istream &is,matrix<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)
    is>>x[i];
  return is;
}


/*Решение СЛАУ
В матричной форме систему линейных алгебраических
уравнений (СЛАУ) можно представить в виде
                       A*X=B,
где A - матрица коэффициентов при неизвестных,
X - вектор-столбец этих самих неизвестных, а
B - вектор-столбец свободных членов, взятых с об-
ратными знаками.
Если система имеет решение, то оно будет таким:
                A^(-1)*A*X=A^(-1)*B
               (A^(-1)*A)*X=A^(-1)*B
                   E*X=A^(-1)*B
                    X=A^(-1)*B,
Существенным моментом является выбор метода реше-
ния. Для небольших систем (<200) можно использовать
прямые методы (например, метод Гаусса); для реаль-
ных расчётов мы рекомендуем метод ортогонализации,
в основе которого лежат ортогональные преобразова-
ния матриц
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> SLAE_Orto(matrix <YourOwnFloatType> &f,
matrix<YourOwnFloatType> &s)
{
```

```
   matrix<YourOwnFloatType> mtr2(f.m+1,f.m+1),
res(f.m,1);
  //формируем матрицу из двух для решения СЛАУ
  for(long i=0;i<f.m;i++)
    for(long j=0;j<f.n;j++)
      mtr2[i][j]=f[i][j];
  for(long i=0;i<f.m;i++)/*заносим в последнюю
строку 0 0 ... 0 1*/
    mtr2[i][f.m]=-s[i][0];
  mtr2[f.m][f.m]=1;
  mtr2[0]=~mtr2[0];//нормируем нулевую строку
  /*для улучшения сходимости повторяем этот процесс
3 раза*/
  for(long k=0;k<3;k++)
  {
    for(long l=1;l<f.m+1;l++)
    {
      for(long i=0;i<l;i++)
      {
        YourOwnFloatType p=mtr2[l]*mtr2[i];
                /*скалярное произведение */
        for(long j=0;j<(f.m+1);j++)
          mtr2[l][j]-=p*mtr2[i][j];
      }
      mtr2[l]=~mtr2[l];
    }
  }
  for(long i=0;i<f.m;i++)//переписываем результат
    res[i][0]=mtr2[f.m][i]/mtr2[f.m][f.m];
  return res;
}


/*
  Метод Гаусса с выбором главного элемента
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> SLAE_Gauss (matrix <YourOwnFloatType>
&f,matrix<YourOwnFloatType> &s)
{
  long i, j, k, num;
  matrix<YourOwnFloatType>
mtr2(f.m,f.m+1),res(f.m,1);
```

```
//формируем матрицу из двух для решения СЛАУ
for(i=0;i<f.m;i++)
  for(j=0;j<f.n;j++)
    mtr2[i][j]=f[i][j];
for(i=0;i<f.m;i++)
  mtr2[i][f.m]=s[i][0];
//выбор главного элемента
for(i=0;i<f.m;i++)
{
  YourOwnFloatType max=mtr2[0][i];
  for(j=i,num=i;j<f.m;j++)
    if(fabs(mtr2[j][i])>max)
      max=fabs(mtr2[j][i]),num=j;
  if(num!=i)
    for(k=0;k<f.m+1;k++)
    {
      YourOwnFloatType temp=mtr2[num][k];
      mtr2[num][k]=mtr2[i][k];
      mtr2[i][k]=temp;
    }
}
for(i=0;i<f.m;i++)
if(!mtr2[i][i])
  throw xmsg("Возможно, матрица вырождена \n");
//Прямой ход Гаусса
for(i=0;i<f.m;i++)
{
  double sw=mtr2[i][i];
  for(j=0;j<f.m+1;j++)
    mtr2[i][j]/=sw;
  for(k=i+1;k<f.m;k++)
  {
    double c=mtr2[k][i];
    for(j=0;j<f.m+1;j++)
      mtr2[k][j]-=mtr2[i][j]*c;
  }
}
//Обратный ход Гаусса
for(i=f.m-2;i>=0;i--)
{
  double s=0;
  for(j=i+1;j<f.m;j++)
    s+=mtr2[i][j]*mtr2[j][f.m];
```

```
    mtr2[i][f.m]-=s;
  }
  //переписываем результат
  for(i=0;i<f.m;i++)
    res[i][0]=mtr2[i][f.m];
  return res;
}


/*
Для обращения матрицы и нахождения детерминанта
классическими методами нужна функция получения ми-
нора матрицы для данного её элемента
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>::minor (long a,
long b)
{
  matrix<YourOwnFloatType> temp(getm()-1,getn()-1);
  for(long i1=0,i2=0;i1<getm();i1++)
  if(i1!=a)
  {
    for(long j1=0,j2=0;j1<getn();j1++)
      if(j1!=b)
        temp[i2][j2]=(*this)[i1][j1],
    j2++;
    i2++;
  }
  return temp;
}


/*
Классический детерминант - вычисление разложением
по одной из строк
*/
template <class YourOwnFloatType> YourOwnFloatType
det2(matrix<YourOwnFloatType> &x)
{
  if(x.getm()!=x.getn())
    throw xmsg("Детерминант определён только для
квадратных матриц \n");
```

```
  if(x.getm()==1)/*особые случаи - детерминанты 1-
го и 2-го порядка*/
    return x[0][0];
  if(x.getm()==2)
    return x[0][0]*x[1][1]-x[0][1]*x[1][0];
  YourOwnFloatType result=0;
  for(long i=0;i<x.getm();i++)
    result+=pow(-1,1+i)*det(x.minor(1,i))*x[1][i];
  return result;
}


/*
Медленная (аналитическая) операция обращения квад-
ратной матрицы
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: operator!()
{
  if(getm()!=getn())
    throw xmsg("Для неквадратных матриц обращение
не определено \n");
  matrix<YourOwnFloatType> temp=*this;
  YourOwnFloatType _det=det(*this);
  if(_det==(YourOwnFloatType)0)
    throw xmsg("Обращение особенной матрицы невоз-
можно \n");
  for(long i=0;i<getm();i++)
    for(long j=0;j<getn();j++)
      temp[i][j]=pow(-1,2+i+j)*
                  det(minor(j,i))/_det;
  return temp;
}


/*
Численное нахождение детерминанта методом Гаусса
*/
template <class YourOwnFloatType>
YourOwnFloatType det(matrix<YourOwnFloatType> &y)
//быстрый детерминант
{
  YourOwnFloatType sw,c,det,max;
```

```
  long i,j,k,how,num;

  matrix<YourOwnFloatType> x=y;
  if(x.getm()!=x.getn())
    throw xmsg("Детерминант определён только для
квадратных матриц \n");
  if(x.getm()==1)
    return x[0][0];
  if(x.getm()==2)
    return x[0][0]*x[1][1]-x[0][1]*x[1][0];
  for(how=i=0;i<x.getm();i++)
  {
    max=x[0][i];
    for(j=i,num=i;j<x.getm();j++)
      if(fabs(x[j][i])>fabs(max))
      {
        max=fabs(x[j][i]);
        num=j;
      }
    if(num!=i)
    {
      for(k=0;k<x.getm();k++)
      {
        YourOwnFloatType temp=x[num][k];
        x[num][k]=x[i][k];
        x[i][k]=temp;
      }
      how++;
    }
  }
  for(i=0;i<x.getm();i++)
  {
    for(sw=x[i][i],j=i+1;j<x.getm();j++)
      x[i][j]/=sw;
    for(k=i+1;k<x.getm();k++)
      for(c=x[k][i],j=0;j<x.getm();j++)
        x[k][j]-=x[i][j]*c;
  }
  for(det=1,i=0;i<x.getm();i++)
    det*=x[i][i];
  det*=pow(-1,how);
  return det;
}
```

```
/*
Быстрое (численное) обращение матрицы
*/
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: operator*()
{
  long i, j, k, l;
  YourOwnFloatType v,maxabs,s,tsr;
  matrix<YourOwnFloatType> temp(getm(),2*getn());


  if(getm()!=getn())
    throw xmsg("Для неквадратных матриц обращение
не определено \n");
  if(det(*this)==(YourOwnFloatType)0)
    throw xmsg("Обращение особенной матрицы невоз-
можно \n");
  for(i=0;i<getm();i++)
  {
    for(j=0;j<getn();j++)
      temp[i][j]=(*this)[i][j];
    for(j=getm();j<2*getm();j++)
      temp[i][j]=(j==i+getm()) ? 1 : 0;
  }
  for(i=0;i<getm();i++)
  {
   for(maxabs=fabs(temp[i][i]),k=i,l=i+1;
       l<getm();l++)
    if(fabs(temp[l][i])>fabs(maxabs))
      maxabs=fabs(temp[l][i]), k=l;
    if(k!=i)
      for(j=i;j<2*getm();j++)
        v=temp[i][j],
        temp[i][j]=temp[k][j], temp[k][j]=v;
  }
  for(i=0;i<getm();i++)
  {
    for(s=temp[i][i],j=i+1;j<2*getm();j++)
      temp[i][j]/=s;
    for(j=i+1;j<getm();j++)
    {
```

```
      for(tsr=temp[j][i],k=i+1;k<2*getm();k++)
        temp[j][k]-=temp[i][k]*tsr;
    }
  }
  for(k=getm();k<2*getm();k++)
    for(i=getm()-1;i>=0;i--)
    {
      for(tsr=temp[i][k],j=i+1;j<getm();j++)
        tsr-=temp[j][k]*temp[i][j];
      temp[i][k]=tsr;
    }
  matrix<YourOwnFloatType> result=(*this);
  for(i=0;i<getm();i++)
    for(j=getm();j<2*getm();j++)
      result[i][j-getm()]=temp[i][j];
  return result;
}


/*
Быстрый детерминант как операция
*/
template <class YourOwnFloatType> inline YourOwn-
FloatType matrix<YourOwnFloatType>::operator&()
{
  return det(*this);
}


/*
Детерминант рассматривается как оператор преобразо-
вания матрицы в число
*/
template <class YourOwnFloatType> inline ma-
trix<YourOwnFloatType>::operator YourOwnFloatType()
{
  return det(*this);/*используем быстрый детерми-
нант*/
}

#endif
```

2.4. Программная реализация методов вычисления собственных значений и собственных векторов матриц

2.4.1. Метод неопределенных коэффициентов

Под стандартной матричной проблемой собственных значений мы понимаем задачу отыскания комплексных чисел λ_1, λ_2, …, λ_n и соответствующих ненулевых векторов $\boldsymbol{X}_1$, …, $\boldsymbol{X}_n$ удовлетворяющих уравнению $\mathbf{A}\boldsymbol{X}=\lambda\boldsymbol{X}$, где $\mathbf{A}$ – заданная комплексная матрица размера nxn. Решения этого уравнения являются корнями характеристического уравнения $\det(\mathbf{A}-\lambda\mathbf{E})=0$. Левая часть этого уравнения есть полином степени n по λ, и, следовательно, характеристическое уравнение имеет ровно n корней. Если собственное значение λ_i найдено, то соответствующий собственный вектор можно определить как решение однородной системы уравнений $(\mathbf{A}-\lambda_i\mathbf{E})\boldsymbol{X}=0$.

Пусть

$$D(\lambda)=\lambda^n+p_1\lambda^{n-1}+\ldots+p_n$$

есть вековой определитель матрицы $\mathbf{A}$, т.е. $\det(\mathbf{A}-\lambda\mathbf{E})=0$. Если в этом равенстве последовательно положить i=0, 1, 2, …, n-1, то для коэффициентов p_i (i=1, 2, …, n) получим систему линейных уравнений, решая которую, можно определить коэффициенты характеристического полинома. Найдя коэффициенты, с помощью метода Ньютона определяем собственные значения, а затем – собственные векторы.

```
/*Выполняем переименование используемых типов с за-
меной шаблона типом complex - нам вероятно придется
иметь дело с комплексными корнями полиномов
*/
typedef polynom<complex> cpolynom;
typedef vector<complex> cvector;
typedef matrix<complex> cmatrix;
```

```
//Прототипы функций
cvector eigenval(cmatrix&);
cmatrix eigenvec(cmatrix&, cvector&);

/*Подпрограмма вычисления собственных значений про-
извольной матрицы по методу неопределенных коэффи-
циентов */
cvector eigenval(cmatrix &x)
{
  if(x.getn()!=x.getm())
    throw xmsg("Для неквадратных матриц собственные
значения не определены \n");
/*Вид характеристического полинома хорошо известен
- это обычный полином n степени с (n+1)-им коэффи-
циентом. Для того, чтобы найти эти коэффициенты,
используем следующий способ: характеристический по-
лином прямыми методами получается развёрткой детер-
минанта, и значение полинома в некоторой точке x
соответствует значению детерминанта матрицы, из
диагональных элементов которой вычтен x. Тогда,
взяв набор из (n+1) несовпадающих точек, мы можем
составить систему из n уравнений, линейных относи-
тельно коэффициентов характеристического полинома.
Результатом решения этой системы и будут искомые
коэффициенты, зная которые, мы можем найти корни
полинома, и являющиеся искомыми собственными значе-
ниями */
  cmatrix a(x.getm()+1,x.getm()+1),b(x.getm()+1,1);
  for(long i=0;i<a.getm();i++)//набор точек
  {
    for(long j=0;j<a.getm();j++)/*вычисляем степени
при коэффициентах*/
      a[i][j]=pow(i,a.getm()-j-1);
    cmatrix temp=x;
    for(long j=0;j<temp.getm();j++)
      temp[j][j]-=i;/* вычитаем из элементов глав-
ной диагонали текущую точку и находим детерминант*/
    b[i][0]=det(temp);
  }
  cmatrix result=SLAE_Orto(a,b);//решаем СЛАУ
  cpolynom lambdaeq=result.getm();
  for(long i=0;i<result.getm();i++)/*формируем ха-
рактеристический полином и находим*/
```

```
    lambdaeq[i]=result[result.getm()-i-1][0];
  return newton(lambdaeq);/*его корни модифициро-
ванным методом Ньютона*/
}


/*Подпрограмма вычисления собственных векторов при
известных матрице и её собственных значениях*/
cmatrix eigenvec(cmatrix &x, cvector &e)
{
  if(x.getn()!=x.getm())
    throw xmsg("Для неквадратных матриц собственные
вектора не определены \n");
  if(x.getm()!=e.getm())/*при несовпадении размер-
ностей*/
    throw xmsg("Собственные значения не соответ-
ствуют матрице \n");
  cmatrix ev(x.getm(),x.getm());/*в строках этой
матрицы будут находиться собственные вектора*/
  for(long i=0;i<e.getm();i++)/*цикл по собственным
значениям*/
  {
    cmatrix temp=x;
    for(long j=0;j<temp.getm();j++)/*вычитаем еди-
ничную матрицу, умноженную */
      temp[j][j]-=e[i]; //на собственное значение
/*Для нахождения ненулевого решения однородной си-
стемы уравнений проделаем следующее: примем одну из
компонент вектора равной 1. Затем перенесём в си-
стеме уравнений в правую часть с противоположным
знаком столбец, соответствующий этой компоненте.
Решив такую СЛАУ, мы найдём все остальные компонен-
ты вектора. Для того, чтобы вектора не совпадали,
для каждого из них "объединичим" свою компоненту -
сначала первую, затем - вторую и т.д. */
    cmatrix a(temp.getm()-1,temp.getm()-1),
            b(temp.getm()-1,1);/*матрицы для СЛАУ*/
    for(long k=0;k<a.getm();k++)
      for(long j=0,l=0;j<temp.getm();j++)
        if(j==i)/*компоненту вектора, номер которой
соответствует номеру собственного значения, сделаем
равным 1, а соответствующий столбец переносим в
```

```
правую часть; все остальное остаётся в левой ча-
сти*/
          b[k][0]=-temp[k][j];
        else
          a[k][l++]=temp[k][j];
    cmatrix res=SLAE_Orto(a,b);/*решаем СЛАУ отно-
сительно неизвестных составляющих*/
    for(long j=0,l=0;j<ev.getm();j++)/*компонуем
собственный вектор*/
      ev[i][j]=((i==j)?complex(1,0):res[l++][0]);
      ev[i]=~ev[i];/*для удобства нормируем его по
модулю*/
  }
  return ev;//возвращаем массив векторов
}
```

2.4.2. Программная реализация метода Крылова вычисления собственных значений

```
#include <conio.h>
#include "matrix.h"
#include "equation.h"

cvector Krylov_eigenval(cmatrix &x)
{
  if(x.getn()!=x.getm())
    throw xmsg("Для неквадратных матриц собственные
значения не определены");
/*Согласно данному методу, нам необходимо составить
последовательность линейно независимых векторов по
специальному закону. Для удобства использования мы
будем хранить их как вектор-столбцы, то есть как
последовательность одностолбцовых матриц:*/
  cmatrix *B; /*указатель под последовательность
векторов*/
  //пытаемся выделить память
  try
  {
    B=new cmatrix[x.getm()+1];
  }
  /*перехват этого исключения происходит при не-
хватке памяти*/
  catch(xalloc)
```

```
  {
/*не пытаясь исправить эту ситуацию, выбрасываем, в
свою очередь, стандартное исключение с диагностиче-
ским текстом */
    throw xmsg("Не хватает памяти");
  }
/*заполняем массив B векторами (одностолбцовыми
матрицами) заданного размера */
  for(long i=0;i<x.getm()+1;i++)
    B[i]=cmatrix(x.getm(),1);
  /*создаём пока пустую матрицу, хранящую в своих
столбцах строящуюся последовательность векторов*/
  cmatrix test(x.getm(),x.getm());
  do
  {
    //формируем случайный вектор
    for(long i=0;i<B[0].getm();i++)
      B[0][i][0]=random(x.getm()+1);
    /*действуя на вектор линейным оператором, за-
данным матрицей, находим следующий вектор и т.д.*/
    for(long i=0;i<x.getm();i++)
      B[i+1]=x*B[i];
    /*переносим вектора из массива в соответствую-
щие столбцы матрицы*/
    for(long i=0;i<test.getm();i++)
      for(long j=0;j<test.getm();j++)
        test[j][i]=B[i][j][0];
    randomize();/*переустанавливаем генератор слу-
чайных чисел*/
/*процесс построения повторяем до тех пор, пока не
выполнится условие линейной независимости векторов
- неравенство нулю детерминанта матрицы, составлен-
ной из построенных векторов*/
  }while(det(test)==(complex)0);
  /*разлагаем последний вектор по предыдущим (в си-
лу их линейной независимости) как по базису и нахо-
дим коэффициенты разложения*/
  cmatrix coeffs=SLAE_Orto(test,B[x.getm()]);
  delete[] B;//освобождаем память из-под массива
  //формируем полином из вычисленных коэффициентов
  cpolynom p=x.getm()+1;
  p[x.getm()]=1;
  for(long i=0;i<coeffs.getm();i++)
```

```
    p[i]=-coeffs[i][0];
/*находим корни полинома (собственные значения) мо-
дифицированным методом Ньютона*/
  return newton(p);
}
```

2.4.3. Метод Леверрье-Фаддеева вычисления коэффициентов характеристического полинома

```
/*функция, предназначенная она для нахождения следа
матрицы - это всего-навсего сумма элементов главной
диагонали*/
template <class YourOwnFloatType>
YourOwnFloatType Sp(matrix<YourOwnFloatType> &x)
{
  YourOwnFloatType res=(YourOwnFloatType)0;
  /*предполагается, что ищется след квадратной мат-
рицы, однако соответствующая проверка не выполняет-
ся*/
  for(long i=0;i<x.getm();i++)
    res+=x[i][i];
  return res;//возвращаем след
}


cvector Leverrie_Faddeev_eigenval(cmatrix &x)
{
  if(x.getn()!=x.getm())
    throw xmsg("Для неквадратных матриц собственные
значения не определены");
  //создаём пока пустой полином
  cpolynom p=x.getm()+1;
  /*устанавливаем коэффициент при самой старшей
степени в единицу*/
  p[x.getm()]=1;
  cmatrix a=x,b=x;//рабочие матрицы
  cmatrix e(x.getm(),x.getm());//нулевая матрица,
  for(long i=0;i<e.getm();i++) //плавно переходящая
    e[i][i]=1; //в единичную
  /*вычисляем коэффициенты характеристического по-
линома*/
  for(long i=0;i<x.getm();i++)
  {
```

```
    p[x.getm()-(i+1)]=-Sp(a)/(i+1);
    b=a+p[x.getm()-(i+1)]*e;
    a=x*b;
  }
  return newton(p);/*находим корни модифицированным
методом Ньютона*/
}

//Вариант главной функции для тестирования

void main()
{
  cmatrix a="test.z";
  cout<<a<<endl;
  getch();
  cvector l=Leverrie_Faddeev_eigenval(a);
  cout<<a<<endl<<l<<endl<<endl;
  getch();
  cmatrix v=eigenvec(a,l);
  cout<<v<<endl;
  getch();

  for(long i=0;i<v.getm();i++)
  {
    cmatrix x(v.getm(),1);
    for(long j=0;j<x.getm();j++)
      x[j][0]=v[i][j];
    cout<<a*x<<"="<<endl<<l[i]*x<<endl<<endl;
    getch();
  }
}
```

2.5. Элементы линейного программирования

2.5.1. Общая постановка задачи

Линейное программирование является одной из многочисленных областей применения линейной алгебры в прикладных задачах, не поддающихся решению методами математического анализа.

Основная задача линейного программирования (ОЗЛП) возникает при числе уравнений, меньшем числа независи-

мых переменных. В этом случае не остается ничего иного, как выбрать N переменных по числу уравнений в качестве *базисных*, а остальные оставить *свободными*; решений системы относительно базиса – бесконечное множество, если на них не наложить ограничений и не предъявить дополнительных требований.

В ОЗЛП такими требованиями, обеспечивающими возможность получения единственного решения, является требование неотрицательности корней $x_{ij}\geq 0$ системы линейных уравнений

$$a_{11}x_1+a_{12}x_2+...+a_{1n}x_n=b_1$$
$$a_{21}x_1+a_{22}x_2+...+a_{2n}x_n=b_2$$
$$..$$
$$a_{m1}x_1+a_{m2}x_2+...+a_{mn}x_n=b_m$$

и необходимость обеспечения минимума некоторой линейной целевой функции

$$\min L=c_1x_1+c_2x_2+...+c_nx_n$$

2.5.2. Примеры задач линейного программирования

2.5.2.1. Задача о пищевом рационе

Имеется 4 вида продуктов П$_1$, П$_2$, П$_3$, П$_4$ стоимостью c_1, c_2, c_3, c_4 каждый и содержащих соответственно

П$_1$ – a_{11} единиц белков, a_{12} углеводов, a_{13} жиров

П$_2$ – a_{21} единиц белков, a_{22} углеводов, a_{23} жиров

П$_3$ – a_{31} единиц белков, a_{32} углеводов, a_{33} жиров

П$_4$ – a_{41} единиц белков, a_{42} углеводов, a_{43} жиров

Общая стоимость $L=c_1x_1+c_2x_2+c_3x_3+c_4x_4$ должна быть минимальной при условии обеспечения в рационе белков – b_1 единиц, углеводов – b_2 единиц, жиров – b_3 единиц и неотрицательности значений $x_1\geq 0$; $x_2\geq 0$; $x_3\geq 0$; $x_4\geq 0$.

2.5.2.2. Задача о распределении ресурсов

Имеются ресурсы R_1, R_2, R_3, ..., R_m в количествах b_1, b_2, b_3, ..., b_m соответственно. С помощью этих ресурсов могут

производиться товары T_1, T_2, T_3, ..., T_n. Для производства одной единицы товара T_{ij} необходимо a_{ij} единиц ресурса R_i (i=1, 2, ..., m; j=1, 2, ..., n), и каждая единица ресурса R_i стоит D_i рублей. Каждая единица товара может быть реализована по цене C_i (i=1, 2, ..., n). Спрос составляет K_j единиц товара T_j (j=1, 2, ..., n). Необходимо определить, какое количество какого товара необходимо произвести, чтобы прибыль от реализации была максимальной.

Если $x_1, x_2, \ldots, x_n$ - количества товаров T_1, T_2, ..., T_n и положить $x_1=K_1$, $x_2=K_2$, ..., $x_n=K_n$, а также учесть достаточность ресурсов, получаем систему:

$$a_{11}x_1+a_{12}x_2+\ldots+a_{1n}x_n=b_1$$
$$a_{21}x_1+a_{22}x_2+\ldots+a_{2n}x_n=b_2$$
$$\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots$$
$$a_{m1}x_1+a_{m2}x_2+\ldots+a_{mn}x_n=b_m$$

Себестоимость единицы товара T_j равна $S_j=a_{1j}D_1+a_{2j}D_2+\ldots+a_{mj}D_m$.

Прибыль Q_j от реализации единицы товара $T_jQ_j=C_j–S_j$.

Общая прибыль $L=Q_1x_1+Q_2x_2+\ldots+Q_nx_n$.

Задача: выбрать такие неотрицательные x_1, x_2, ..., x_n, которые удовлетворяют ограничениям и обращают в максимум функцию L этих переменных. Аналогично формулируются другие практически важные задачи – об оптимальной загрузке оборудования, об оптимальном планировании перевозок и т. д.

2.5.2.3. Задача планирования перевозок (транспортная задача)

В p пунктах-отправителях имеется по k_j единиц однотипных грузов, предназначенных для поставки в n пунктов – адресатов по g_i штук в каждый, а стоимости перевозок единицы груза в каждый пункт различны для различных комбинаций отправитель–поставщик и равны c_{ij}, где j – номер пункта-отправителя (j=1, 2, …, p), i – номер пунктаполучателя (i=1, 2, …, n). Обозначим количество грузов,

перевозимых от отправителя j получателю i через $x_{ij} \geq 0$. Составим n уравнений (по одному для каждого получателя):

$$\sum_{j=1}^{p} x_{ij} = g_i,\ i=1,\ \ldots,\ n.$$

плюс p ограничений по возможностям отправителей:

$$\sum_{j=1}^{p} x_{ij} \leq k_j,\ j=1,\ \ldots,\ p.$$

и линейный критерий оптимальности решения – минимум общей стоимости перевозок:

$$L = \sum_{i=1}^{n} \left(\sum_{j=1}^{p} c_{ij} \right) x_{ij} \ \to \min$$

Эта задача носит название *транспортной задачи линейного программирования* – она значительно проще общей задачи, так как все коэффициенты основной системы уравнений равны 1.

2.5.3. Симплекс-метод решения задачи линейного программирования

Допустимым решением ОЗЛП называют любую совокупность неотрицательных x_1, x_2, ..., x_n, удовлетворяющую исходной системе линейных уравнений.

Оптимальным называют *допустимое* решение, при котором заданная линейная целевая функция обращается в минимум.

Наиболее универсальным методом решения ОЗЛП является *симплекс-метод*, основанный на том, что вначале отыскивается произвольное допустимое решение, а затем оно последовательно улучшается до *оптимального*.

Схема решения ОЗЛП симплекс-методом:

1) составляется *симплекс-таблица* из коэффициентов системы, представляющая собой расширенную матрицу, дополненную строкой коэффициентов целевой функции, например (N+1)-й, а также нулевыми строкой и столбцом для учета производимых перестановок.

2) выбирается произвольный набор *базисных* переменных по числу имеющихся в наличии линейных алгебраических уравнений;
3) система разрешается относительно базисных переменных, выражая их через остальные (*свободные*);
4) свободные переменные приравнивают нулю, получая некоторое решение в виде компонент вектора свободных членов;
5) проверяют решение на допустимость – то есть на отсутствие отрицательных компонент;
6) при наличии отрицательных свободных членов производят последовательную замену базисных переменных на свободные до получения допустимого решения, которое называют *опорным*.
7) опорное решение является *оптимальным*, если в строке коэффициентов целевой функции отсутствуют отрицательные элементы, то есть целевая функция не может быть уменьшена за счет увеличения каких-нибудь свободных переменных сверх нуля.

2.5.3.1. Приведение системы к стандартной, удобной для преобразований форме

Это приведение состоит в разрешении системы относительно выбранных базисных переменных, что достигается, например, решением по методу Гаусса за один «ход»; если выбрать первые n элементов базисными, то задача состоит в преобразовании левой части nxn матрицы в единичную с одновременной обработкой всех элементов.

После решения системы относительно базисных переменных и подстановки их в линейную форму матрица приобретает вид с единичной матрицей в правой части.

2.5.3.2. Алгоритм замены базисных переменных

Процедура переразрешения системы ОЗЛП относительно новых базисных переменных может быть формализована и сведена к алгоритму (последовательности допу-

стимых однообразных действий над элементами системной матрицы). Формулы преобразования могут быть легко выведены и должны выполняться в следующем порядке:

1) Разрешающий элемент матрицы (имеющий один индекс – номер базисного элемента, а второй – свободного для замены) заменяется обратной ему величиной;

2) Все элементы разрешающего столбца умножаются на обновленный разрешающий элемент и меняют знак;

3) Все элементы, кроме разрешающих столбца и строки, вычисляются так

$$m[i,j]{:=}m[i,j]+m[i,jr]*m[ir,j];$$

где jr – разрешающий столбец, ir – разрешающая строка;

4) Все оставшиеся элементы разрешающей строки умножаются на обновленный разрешающий элемент.

2.5.3.3. Алгоритм поиска опорного решения ОЗЛП

если все свободные члены в матрице, разрешенной относительно базиса, положительны, то они и представляют собой допустимое опорное решение и можно переходить к этапу его оптимизации;

если среди свободных членов есть отрицательные, то их вектор не годится в качестве опорного решения, так как все составляющие, по условию, неотрицательны, и необходимо производить последовательный обмен между базисными и свободными переменными, пока не избавимся от отрицательных свободных членов или не придем к выводу об отсутствии допустимого опорного решения.

2.5.3.3.1. Алгоритм выбора разрешающего элемента для приближения к опорному решению

В первой найденной строке матрицы с отрицательным свободным членом ищем отрицательный элемент; если такого нет, то система несовместима с требованием неотрицательности решений (вся правая часть уравнения может быть только отрицательной).

Если отрицательный элемент найден, то этот столбец является разрешающим и осталось выбрать разрешающую строку.

Двигаясь по разрешающему столбцу, исследуем все его элементы, имеющие одинаковый (совпадающий) знак со свободным членом и выбираем тот, отношение к которому свободного члена минимально – это разрешающая строка.

2.5.3.4. Алгоритм поиска оптимального решения

Оптимизация найденного опорного решения состоит в перемещении по допустимым опорным решениям таким образом, чтобы линейная форма приняла минимально возможное значение; для этого проверим наличие положительных элементов в строке линейной формы ((N+1)-й в матрице), которое свидетельствует о возможности уменьшения значения формы увеличением соответствующих переменных.

Если положительный элемент найден, то в соответствующем столбце ищем разрешающий элемент – положительный и с минимальным отношением к нему свободного члена.

Это все повторяется до тех пор, пока в строке линейной формы не останется положительных элементов (без учета свободного члена).

Если в столбце, содержащем положительный элемент формы, нет положительного системного элемента, то оптимизируемая функция не ограничена снизу и ОЗЛП не имеет оптимального решения.

2.5.4. Транспортная задача линейного программирования

2.5.4.1. Общие сведения

Классическая формулировка транспортной задачи линейного программирования выглядит так:

- имеется m пунктов отправления $A_1, A_2, \ldots, A_m$, в которых есть запасы однородного товара в количествах соответственно $a_1, a_2, \ldots, a_m$ единиц;
- имеется также n пунктов назначения $B_1, B_2, \ldots, B_n$, желающих получить соответственно $b_1, b_2, \ldots, b_n$ единиц товара;
- предполагается, что сумма всех заявок потребителей равна сумме всех запасов у поставщиков:

$$\sum_{i=1}^{m} a_i = \sum_{j=1}^{n} b_j \ ;$$

- задана матрица стоимостей перевозок единицы товара от каждого отправителя до каждого получателя.

Необходимо составить такой план перевозок, при котором все заявки были бы удовлетворены, а стоимость всех перевозок была минимально возможной - это *транспортная задача по критерию стоимости.*

Если x_{ij} – количества груза, отправляемого i-м отправителем j-му получателю (их значения называют перевозками), то эти неотрицательные переменные должны удовлетворять следующим условиям:

- каждый отправитель отправит получателям весь свой запас, то есть

$$\sum_{i=1}^{n} x_{1i} = a_1; \ \sum_{i=1}^{n} x_{2i} = a_2; \ \ldots, \ \sum_{i=1}^{n} x_{mi} = a_m;$$

- каждый получатель получит в сумме то, что заявил:

$$\sum_{j=1}^{m} x_{j1} = b_1; \ \sum_{j=1}^{m} x_{j2} = b_2; \ \ldots. \ \sum_{j=1}^{m} x_{jn} = b_n;$$

- суммарная стоимость перевозок должна быть минимальной:

$$\sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} = \min.$$

Это типичная задача линейного программирования с ограничениями типа равенств и ее можно решить сим-

плекс-методом, но ряд ее особенностей позволяют упростить решение. Во-первых, все коэффициенты при x в уравнениях равны 1; во-вторых, не все $m+n$ уравнений являются независимыми из-за равенства запасов и заявок, а только $m+n-1$. Поэтому можно разрешить эти уравнения относительно $m+n-1$ базисных переменных, выразив их через остальные свободные. Количество свободных переменных равно:

$$k=mn-(m+n-1)=(m-1)(n-1).$$

Любую совокупность (x_{ij}) называют планом перевозок; план считается *допустимым*, если он удовлетворяет условиям баланса между запасами и заявками – все заявки удовлетворены, а все запасы исчерпаны. План может считаться *опорным*, если в нем отличны от нуля не более $m+n-1$ базисных перевозок x_{ij}, а остальные равны нулю. План является *оптимальным*, если он приводит к наименьшей среди всех допустимых планов суммарной стоимости перевозок.

Методы решения транспортной задачи не требуют манипуляций с симплекс-таблицей – решение получают с помощью простых операций с так называемой *транспортной таблицей.* Номера строк этой таблицы – это номера отправителей, номера столбцов – номера получателей, всего m строк и n столбцов, в ячейках таблицы записываются соответствующие перевозки. В каждом опорном плане, включая оптимальный, не более $m+n-1$ ненулевых базисных перевозок, остальные свободные равны нулю. Решение ТЗ сводится к следующему: найти значения положительных перевозок, суммы которых в каждой строке равны запасу соответствующего отправителя, а суммы по столбцам равны заявкам соответствующих получателей. Общая стоимость перевозок – минимальна.

2.5.4.2. Формирование опорного плана

Решение транспортной задачи в отличие от общей задачи линейного программирования всегда существует. Среди допустимых планов всегда есть оптимальный в силу заведомой неотрицательности стоимости перевозок в минимизируемой линейной форме. Для построения опорного плана существует много разработанных методов, из них простейшим является так называемый «метод северо-западного угла».

По этому методу таблицу перевозок заполняют, начиная с левой верхней ячейки, удовлетворяя последовательно заявки потребителей за счет ближайших в списке поставщиков, двигаясь слева направо и сверху вниз – или с запада на восток и с севера на юг. Движение по клеткам закончится, когда все запасы будут исчерпаны и все заявки удовлетворены, то есть сразу получается план, удовлетворяющий балансу запасов и потребностей. Полученный план является допустимым и опорным, но не оптимальным, так как стоимости перевозок при его составлении игнорировались.

2.5.4.3. Циклические переносы перевозок для улучшения плана

Для улучшения плана некоторые перевозки можно переносить из одной клетки в другую без нарушения баланса – в одних клетках перевозки увеличиваются за счет уменьшения в других.

Циклом в транспортной таблице будем называть совокупность клеток, соединенных замкнутой ломаной линией, поворачивающейся в каждой клетке на угол в 90 градусов. Перенос какого-то количества единиц груза по циклу равновесие между заявками и запасами не нарушается и допустимый план остается допустимым.

Ценой цикла называют изменение стоимости перевозок при перемещении по циклу единицы груза – она равна сумме стоимостей перевозок по клеткам цикла, причем

стоимость перевозки берется со знаком «+» для клеток с увеличивающейся перевозкой и со знаком «–» для клеток с уменьшающейся перевозкой. Очевидно, что план будет улучшаться только при перемещении перевозок по циклам с отрицательной ценой. Перевозки не могут быть отрицательными, поэтому со знаком минус могут выступать только базисные клетки с ненулевыми положительными перевозками – брать будем только там, где есть что взять. Если циклов с отрицательной ценой в таблице больше нет, значит улучшить план больше нельзя и он является оптимальным.

Для облегчения поиска улучшающих циклов обычно отыскивают в таблице свободную клетку с наименьшей стоимостью перевозок и заполняют ее, освобождая одну из базисных клеток, оставляя общее число базисных клеток неизменным.

Для любой свободной клетки всегда существует единственный цикл, одна из вершин которого лежит в этой свободной клетке, а все остальные – в базисных клетках. Если цена этого цикла при заполнении свободной клетки неотрицательной перевозкой отрицательна, то перемещение перевозок по этому циклу приведет к улучшению плана. Допустимое для перемещения по циклу количество груза определяется минимальным значением удаляемых перевозок - иначе возникнут недопустимые отрицательные перевозки.

2.5.4.4. Метод потенциалов

Этот специальный метод решения ТЗ позволяет автоматически выделять циклы с отрицательной ценой и определять их цены.

Идея метода состоит в следующем. Для каждого поставщика и потребителя вводятся псевдоплатежи за перевозки единицы груза поставщиком и получателем α_i и β_j соответственно, не зависящие от направления перевозок.

Псевдостоимость перевозки единицы груза от A_i к B_j равна $\widetilde{c}_{ij}=\alpha_i+\beta_j$; $i=1, 2, \ldots, m$; $j=1, 2, \ldots, n$. При этом α_i и β_j не обязательно положительны.

Известна так называемая *«теорема о платежах»*, которую мы приведем без доказательства, отсылая интересующихся к литературным источникам:

для каждой совокупности псевдоплатежей (α_i, β_j) суммарная псевдостоимость перевозок при любом допустимом плане перевозок (x_{ij}) сохраняет одно и то же значение

$$\sum_{i=1}^{n}\sum_{j=1}^{m}\widetilde{c}_{ij}x_{ij}=\text{const}$$

Для невырожденного плана перевозок (с числом базисных клеток в таблице перевозок $m+n-1$) псевдоплатежи (α_i, β_j) определяют так, чтобы во всех базисных клетках псевдостоимости перевозок были равны стоимостям: $\widetilde{c}_{ij}=\alpha_i+\beta_j=c_{ij}$ при $x_{ij}>0$. Оказывается, что соотношение между псевдостоимостями и стоимостями в свободных клетках является индикатором оптимальности плана. Это положение фиксируется следующей теоремой:

Если для всех базисных клеток плана ($x_{ij}>0$) $\widetilde{c}_{ij}=\alpha_i+\beta_j=c_{ij}$, а для всех свободных клеток ($x_{ij}=0$) $\widetilde{c}_{ij}=\alpha_i+\beta_j\leq c_{ij}$, то план оптимален и не может быть улучшен.

Эта теорема справедлива также и для вырожденного плана с наличием нулевых базисных клеток.

План с указанными в теореме свойствами называют *потенциальным*, а соответствующие ему псевдоплатежи (α_i, β_j) – *потенциалами* пунктов A_i, B_j, то есть всякий потенциальный план оптимален.

Таким образом, задача сводится к поиску потенциального плана и решается методом последовательных приближений – сначала задаются произвольной системой платежей, удовлетворяющей условию $\widetilde{c}_{ij}=c_{ij}$ для всех базисных

клеток, затем одновременно меняют систему платежей так, чтобы они приближались к потенциалам, используя следующее свойство псевдоплатежей и псевдостоимостей:

При любой системе платежей, удовлетворяющей условию $\tilde{c}_{ij}=c_{ij}$ *для всех базисных клеток, для каждой свободной клетки цена цикла переноса равна разности между стоимостью и псевдостоимостью в данной клетке* $c_{ij}-\tilde{c}_{ij}$.

Алгоритм решения задачи по методу потенциалов может выглядеть так:

- Берется любой опорный план с $m+n-1$ базисных клеток.
- Для этого плана определяются псевдоплатежи по условию $\alpha_i+\beta_j=c_{ij}$; один из платежей можно определить произвольным, например нулевым.
- Вычисляются псевдостоимости для всех свободных клеток $\tilde{c}_{ij}=\alpha_i+\beta_j$;
- Если хотя бы в одной свободной клетке псевдостоимость превышает стоимость, осуществляют переброски перевозок по циклу для любой свободной клетки с отрицательной ценой.
- Заново подсчитываются платежи и псевдостоимости и так до тех пор, пока во всех свободных клетках не окажутся псевдостоимости не более стоимостей.

2.5.5. Транспортная задача при небалансе запасов и заявок

При избытке запасов задача легко сводится к сбалансированному варианту введением фиктивного потребителя с заявкой на весь избыток запасов и нулевыми стоимостями перевозки единицы товара от всех поставщиков – после этого задача решается одним из рассмотренных выше методов.

При избытке заявок по сравнению с запасами задача тоже может быть сведена к сбалансированному варианту

либо умножением всех заявок на понижающий коэффициент, равный отношению суммы заявок к сумме запасов, либо корректировкой заявок коэффициентами с учетом важности потребителей.

2.5.6. Транспортная задача с временным критерием

Иногда на первый план выдвигается не экономический критерий минимизации общей стоимости перевозок, а минимизация общей длительности перевозок – при перевозках скоропортящихся продуктов или боеприпасов в условиях войны. В этом случае рассматривается транспортная задача со сбалансированными запасами и заявками, но вместо таблицы стоимостей перевозок задается таблица длительностей доставки товаров от всех поставщиков ко всем потребителям в предположении неограниченности транспортных средств для выполнения перевозок – в этом случае длительности перевозок не зависят от количества доставляемого товара. Очевидно, что весь план будет выполнен, когда завершится самая длительная перевозка. В общем случае это не задача линейного программирования, так как критерий оптимальности плана – время T – не является линейной функцией элементов таблицы длительностей перевозок. Можно свести эту задачу к нескольким задачам линейного программирования или использовать различные расчетные методы непосредственного определения оптимального решения, например, метод запрещенных клеток. По этому методу начальный план перевозок составляется не методом северо-западного угла, а предварительным запретом ставить ненулевые перевозки в клетки с самыми длительными перевозками – при этом в первую очередь заполняются клетки с малыми временами. Получив план с некоторым максимальным временем перевозок, запрещаем использование клеток с длительностью доставки большим этого времени и отыскиваем цикл улучшения плана без ис-

пользования запрещенных клеток – если это удается, попытку повторяем, а если нет – значит, план улучшить уже нельзя. Такой эвристический подход может оказаться эффективнее других методов решения этой оптимизационной задачи.

2.6. Программная реализация задач линейного программирования

2.6.1. Симплекс-метод решения ОЗЛП

```
//файл включения для векторных и матричных объектов
#include "matrix.h"

/*определяем типы "действительная матрица" и "дей-
ствительный вектор"*/
typedef matrix<double> dmatrix;
typedef vector<double> dvector;


/*процедура перестановки векторов матрицы для выбо-
ра главного элемента*/
void rotate(dmatrix &x, long index)
{
 /*нам необходимо, чтобы в процедуре прямого хода
на главной диагонали не было нулевых составляющих;
если же таковые есть, находим "ниже" по матрице
вектор-уравнение с ненулевой соответствующей компо-
нентой и ставим его на место текущего*/
  for(long i=index;i<x.getm();i++)
  if(x[i][index])
  {
    dvector temp=x[i];
    x[i]=x[index];
    x[index]=temp;
  }
}
```

```
/*функция, выполняющая прямой ход Гаусса с попутным
подсчётом количества линейно-независимых векторов
данной матрицы (ранга матрицы)*/
long rank(dmatrix &x)
{
  long i;
  //перебираем вектора матрицы
  for(i=0;i<x.getm();i++)
  {
    if(!x[i][i])/*если диагональный элемент равен
нулю*/
      rotate(x,i);//меняем строки матрицы
    /*делаем нулевым столбец под текущим диагональ-
ным элементом*/
    for(long j=i+1;j<x.getm();j++)
      if(x[i][i])
        x[j]+=(-x[j][i]/x[i][i])*x[i];
  }
  //создаём пустой вектор заданной размерности
  dvector empty=x.getn();
  /*количество нулевых векторов в матрице после
прямого хода и определяет ранг данной матрицы*/
  for(i=0;i<x.getm();i++)
    if(x[i]==empty)/*сравниваем текущий вектор с
нулевым*/
      break;
  return i;//возвращаем ранг
}


/*процедура обратного хода Гаусса без нормирования
элементов главной диагонали*/
void diagonalize(dmatrix &x)
{
  for(long i=x.getm()-1;i>=0;i--)
    for(long j=i-1;j>=0;j--)
      if(x[i][i])
        x[j]+=(-x[j][i]/x[i][i])*x[i];
}

/*Процедура обмена двух столбцов матрицы - переста-
новки местами переменных с попутным переразрешением
полученной системы. Параметрами её являются матрица
```

```
ОЗЛП, вектор линейной формы, номера обмениваемых
переменных и вектор, в котором учитывается произво-
димая перестановка*/
void swap(dmatrix &x, dvector &l, long sw1, long
sw2, dvector &swapvalues)
{
  //временный вектор для обмена значениями
  dvector save(x.getm()+2);
  //сохраняем первый столбец
  for(long i=0;i<x.getm();i++)
    save[i]=x[i][sw1];
  /*сохраняем элемент линейной формы, соответствую-
щий первому столбцу*/
  save[x.getm()]=l[sw1];
  //сохраняем номер переставляемого столбца и т.д.
  save[x.getm()+1]=swapvalues[sw1];
  for(long i=0;i<x.getm();i++)
    x[i][sw1]=x[i][sw2];
  l[sw1]=l[sw2];
  swapvalues[sw1]=swapvalues[sw2];
  for(long i=0;i<x.getm();i++)
    x[i][sw2]=save[i];
  l[sw2]=save[x.getm()];
  swapvalues[sw2]=save[x.getm()+1];
  rank(x);//осуществляем прямой и
  diagonalize(x);//обратный ход Гаусса
  for(long i=0;i<x.getm();i++)
    x[i]*=(1/x[i][i]);
/*подставляем значений переменных в линейную форму,
определяя её коэффициенты в данной перестановке*/
  for(long i=0;i<x.getm();i++)
    l+=-l[i]*x[i];
}


/*процедура определения опорного решения принимает
три параметра - матрицу ОЗЛП, вектор линейной формы
и вектор перестановок*/
void getoporsol(dmatrix &x, dvector &l, dvector
&swapvalues)
{
  long i,j,k,s;
  double otn;
```

```
  /*ищем уравнение с отрицательным свободным членом
(заметим, что в нашей записи все переменные нахо-
дятся в левой части уравнения, поэтому коэффициенты
при них следует брать с обратным знаком)*/
  for(i=0;i<x.getm();i++)
    if(-x[i][x.getn()-1]<0)
      break;
  if(i==x.getm())
    return;/*опорное решение найдено - все свобод-
ные члены неотрицательны */
  /*в строке, где мы нашли отрицательный свободный
член, среди коэффициентов пытаемся найти положи-
тельный*/
  for(j=x.getm();j<x.getn()-1;j++)
    if(x[i][j]<0)
      break;
  /*если положительный коэффициент мы найти не
смогли, то нарушается принцип неотрицательности пе-
ременных, что недопустимо*/
  if(j==x.getn()-1)
    throw xmsg("Допустимых решений нет");
  /*среди строк, в которых в определённом (разреша-
ющем) столбце отрицательному свободному члену соот-
ветствует положительный коэффициент, выбираем ту, в
которой отношение -x[k][n-1]/x[k][j] принимает
наименьшее значение*/
  for(k=0;k<x.getm();k++)
   if(sign(x[k][j])==sign(-x[k][x.getn()-1])
                                        &&x[k][j])
    {
      s=k;
      otn=-x[k][x.getn()-1]/x[k][j];
      break;
    }
  for(;k<x.getm();k++)
   if(sign(x[k][j])==sign(-x[k][x.getn()-1])
                                        &&x[k][j])
      if(otn>-x[k][x.getn()-1]/x[k][j])
        s=k,
        otn=-x[k][x.getn()-1]/x[k][j];
  /*j - разрешающий столбец, s - разрешающая стро-
ка. Меняем в нашей матрице столбцы с соответствую-
щими индексами */
```

```
  swap(x, l, s, j, swapvalues);
  //снова пытаемся определить опорное решение
  getoporsol(x,l,swapvalues);
}


/*если вектор оптимального решения существует, то
данная функция его определит, используя матрицу
ОЗЛП, линейную форму и вектор перестановок*/
dvector getoptsol(dmatrix &x, dvector &l, dvector
&swapvalues)
{
  long i,j,k,s;
  double otn;
  //ищем в линейной форме отрицательный коэффициент
  for(i=0;i<l.getm()-1;i++)
    if(l[i]<0)
      break;
  /*если таковых нет, полученное решение является
оптимальным*/
  if(i==l.getm()-1)
  {
    //формируем вектор результата
    dvector res=l.getm()-1;
    /*используя массив перестановок, записываем
значения компонент вектора решения на их прежние
позиции*/
    for(long i=0;i<x.getm();i++)
      res[swapvalues[i]]=-x[i][x.getn()-1];
    return res;//возвращаем оптимальное решение
  }
  /*иначе - определяем номер "опасного" уравнения,
в котором увеличение переменной, соответствующей
отрицательному коэффициенту линейной формы, может
привести к нарушению условия неотрицательности пе-
ременных */
  for(j=0;j<x.getm();j++)
    if(-x[j][i]<0)
      break;
  /*если такого уравнения нет, то данную переменную
можно увеличивать безгранично, а, значит, оптималь-
ного решения ОЗЛП не существует*/
  if(j==x.getm())
```

```
    throw xmsg("Линейная форма не ограничена сни-
зу");
  /*если же такие уравнения нашлись, определяем
среди них ту переменную, которая при увеличении
данной наиболее быстро обратится в нуль*/
  for(k=0;k<x.getm();k++)
    if(-x[k][i]<0)//опасное уравнение
    {
      s=k;
      otn=-x[k][x.getn()-1]/x[k][i];
      break;
    }
  for(;k<x.getm();k++)
    if(-x[k][i]<0)//опасное уравнение
      if(otn>-x[k][x.getn()-1]/x[k][i])
        s=k,//наиболее угрожаемая переменная
        otn=-x[k][x.getn()-1]/x[k][i];
  /*меняем местами переменную при отрицательном ко-
эффициенте линейной формы с наиболее угрожаемой */
  swap(x,l,s,i,swapvalues);
  /*после замены вновь пытаемся найти оптимальное
решение*/
  return getoptsol(x,l,swapvalues);
}


/*функция для решения ОЗЛП, заданной основной мат-
рицей (левой частью), столбцом свободных членов
правой части и линейной формой без св. члена */
dvector simplex(dmatrix a, dvector b, dvector c)
{
  if(a.getm()!=b.getm())
    throw xmsg("Количество уравнений должно совпа-
дать");
  if(a.getn()!=c.getm())
    throw xmsg("Количество неизвестных должно сов-
падать");
/*матрица для хранения системы уравнений, заданной
левой и правой частями*/
  dmatrix ab(a.getm(),a.getn()+1);
  dvector swapvalues=c.getm();/*вектор учёта обмена
переменных*/
  //вначале порядок переменных не нарушен
```

```
  for(long i=0;i<swapvalues.getm();i++)
    swapvalues[i]=i;
  //формируем матрицу СЛАУ из левой и правой частей
  for(long i=0,j;i<a.getm();i++)
  {
    for(j=0;j<a.getn();j++)
      ab[i][j]=a[i][j];
    ab[i][j]=-b[i];
  }
  long r;
  //определяем ранги основной и расширенной матрицы
  if((r=rank(a))!=rank(ab))
    throw xmsg("Система несовместна - ранги основ-
ной и расширенной матриц не совпадают");
  /*в случае совместной системы ранги этих матриц
совпадают, поэтому мы можем отбросить уравнения-
следствия*/
  dmatrix x(r,ab.getn());
  /*в новую матрицу ОЗЛП переписываем только линей-
но-независимые уравнения*/
  for(long i=0;i<r;i++)
    x[i]=ab[i];
  /*добавляем в вектор линейной формы пока нулевой
свободный член*/
  dvector l(c.getm()+1);
  for(long i=0;i<c.getm();i++)
    l[i]=c[i];
  //выполняем обратный ход Гаусса
  diagonalize(x);
  for(long i=0;i<x.getm();i++)
    x[i]*=(1/x[i][i]);
  /*подстановка значений базисных переменных в ли-
нейную форму*/
  for(long i=0;i<x.getm();i++)
    l+=-l[i]*x[i];
  //поиск опорного решения
  getoporsol(x,l,swapvalues);
  //возвращаем оптимальное решение
  return getoptsol(x,l,swapvalues);
}

//тестовая программа работает с задачей вида:
//-5*x1-x2+2*x3<=2
```

```
//-x1+x3+x4<=5
//-3*x1+5*x4<=7 L=5*x1-2*x3
//x1>=0
//x2>=0
//x3>=0
//x4>=0
//==>
//5*x1+x2-2*x3>=-2
//x1-x3-x4>=-5
//3*x1-5*x4>=-7 L=5*x1-2*x3
//x1>=0
//x2>=0
//x3>=0
//x4>=0
//==>
//5*x1+x2-2*x3+2>=0
//x1-x3-x4+5>=0
//3*x1-5*x4+7>=0 L=5*x1-2*x3
//x1>=0
//x2>=0
//x3>=0
//x4>=0
//==>
//y1=5*x1+x2-2*x3+2
//y2=x1-x3-x4+5
//y3=3*x1-5*x4+7 L=5*x1-2*x3
//y1>=0
//y2>=0
//y3>=0
//x1>=0
//x2>=0
//x3>=0
//x4>=0
//==>
//-y1+5*x1+x2-2*x3+2=0
//-y2+x1-x3-x4+5=0
//-y3+3*x1-5*x4+7=0 L=5*x1-2*x3
//y1>=0
//y2>=0
//y3>=0
//x1>=0
//x2>=0
//x3>=0
```

```
//x4>=0
//
void main()
{
  double mtr[]=
  {
    -1,0,0, 5,1,-2,0,
    0,-1,0, 1,0,-1,-1,
    0,0,-1, 3,0,0,-5
  };
  double vec1[]={-2, -5, -7};
  double vec2[]={0,0,0,5,0,-2,0};
  dmatrix a(3,7,mtr), b(3,vec1), c(7,vec2);
  dvector res=simplex(a,b,c);
  cout<<res<<endl;
/*значение линейной формы - скалярное произведение
вектора переменных на вектор коэффициентов, т.е.
сумма соответствующих произведений*/
  double L=c*res;
  cout<<"L="<<L<<endl;
}
```

2.6.2. Программная реализация метода решения транспортной задачи

```
//файл включения для векторных и матричных объектов
#include <winsys\geometry.h>
#include <classlib\arrays.h>
#include <values.h>
#pragma hdrstop
#include "matrix.h"

/*определяем типы "действительная матрица" и "дей-
ствительный вектор"*/
typedef matrix<double> dmatrix;
typedef vector<double> dvector;
//определяем тип "массив точек"
typedef TArray<TPoint> Array;


/*набор функций, проверяющих, возможно ли движение
в заданном направлении, то есть имеется ли в задан-
ном направлении базисная ячейка*/
```

```
TPoint up(dmatrix x,TPoint cur)
{
  for(long i=cur.X()-1;i>=0;i--)
    if(x[i][cur.Y()]>0)/*если есть пункт, в который
можно передвинуться*/
      return TPoint(i,cur.Y());/*возвращаем его ко-
ординаты*/
  return TPoint(-1,-1);/*если не нашли - возвращаем
признак ошибки*/
}


TPoint down(dmatrix x,TPoint cur)
{
  for(long i=cur.X()+1;i<x.getm();i++)
    if(x[i][cur.Y()]>0)
      return TPoint(i,cur.Y());
  return TPoint(-1,-1);
}


TPoint left(dmatrix x,TPoint cur)
{
  for(long i=cur.Y()-1;i>=0;i--)
    if(x[cur.X()][i]>0)
      return TPoint(cur.X(),i);
  return TPoint(-1,-1);
}


TPoint right(dmatrix x,TPoint cur)
{
  for(long i=cur.Y()+1;i<x.getn();i++)
    if(x[cur.X()][i]>0)
      return TPoint(cur.X(),i);
  return TPoint(-1,-1);
}


/*эта функция проверяет, есть ли пункт в заданном
массиве – цикле*/
long exist(Array way,TPoint newpoint)
{
```

```
  /*дополнительно проверяем координаты пункта на 
корректность*/
  if(newpoint==TPoint(-1,-1))
    return 1;
  for(long i=0;i<way.GetItemsInContainer();i++)
    if(way[i]==newpoint)
      return 1;
  return 0;
}


/*функция, определяющая, лежат ли три точки в одной 
строке (столбце)*/
inline long OnOneLine(TPoint p1, TPoint p2, TPoint 
p3)
{
  return (p1.X()==p2.X()&&p1.X()==p3.X())||
         (p1.Y()== p2.Y()&&p1.Y()==p3.Y());
}


/*часто повторяющиеся действия можно реализовать не 
только в виде функций, но и, к примеру, в виде мак-
ро. Например:
- в этом фрагменте проверяется, превышает ли число 
неудачных (тупиковых) попыток продвинуться в одном 
из четырёх возможных направлений количество этих 
направлений. Если это так, то пункт, из которого мы 
пытаемся пройти дальше, объявляется тупиковым (по-
мечается нулём), затем изымается из массива  пунк-
тов цикла - пути перевозки груза, общее количество 
пунктов перевозки уменьшается на единицу и количе-
ство неудачных попыток объявляется нулевым*/
#define otkat if(tupik>4)\
{\
  x[way[i].X()][way[i].Y()]=0;\
  way.Detach(i);\
  i--;\
  tupik=0;\
}
/*- этот фрагмент выполняются после того, как опре-
делен следующий пункт, куда можно перевезти груз. 
При этом вначале количество неудачных попыток уве-
```

```
личивается на единицу - а вдруг идти в этот пункт
нельзя? Затем выполняется проверка координат пункта
на допустимость и наличие пункта в цикле (не прово-
зили ли мы уже здесь груз в текущем цикле перево-
зок). Если всё Ок, пункт добавляется в цикл перево-
зок, количество неудачных попыток устанавливается в
ноль, а количество пунктов перевозки в цикле увели-
чивается. Если же из текущей точки можно перейти в
начальную, то цикл перевозок считается замкнутым и
поиск новых пунктов можно прервать*/
#define addpoint tupik++;\
  if(!exist(way,newpoint))\
  {\
    way.Add(newpoint);\
    tupik=0;\
    i++;\
    if(i!=1&&(newpoint.X()==pq.X()\
       ||newpoint.Y()==pq.Y()))\
      break;\
  }


/*эта функция в заданной матрице перевозок ищет
цикл, соответствующий перевозке из заданной точки*/
Array findway(dmatrix x, TPoint pq)
{
  /*Вычёркивание тех пунктов, которые могут завести
в тупик. Для этой цели вычёркиваем из матрицы все
ряды, кроме строки p и столбца q, имеющие не более
одной базисной клетки. Этот процесс повторяем до
тех пор, пока есть что вычёркивать*/
  for(long count=0,flag=1;flag;)
  {
    flag=0;/*вначале считаем, что вычёркивать нече-
го*/
    for(long i=0;i<x.getm();i++)
      if(i!=pq.X())
      {
        for(long j=count=0;j<x.getn();j++)
          if(x[i][j]>0)
            count++;
        //если в строке только одна базисная клетка
        if(count<2&&count)
```

```
        {
          flag=1;//взводим флаг вычёркивания
          for(long j=0;j<x.getn();j++)
            x[i][j]=0;//обнуляем текущую строку
        }
      }
    //ту же операцию проводим со столбцами
    for(long j=0;j<x.getn();j++)
      if(j!=pq.Y())
      {
        for(long i=count=0;i<x.getm();i++)
          if(x[i][j]>0)
            count++;
        if(count<2&&count)
        {
          flag=1;
          for(long i=0;i<x.getm();i++)
            x[i][j]=0;
        }
      }
  }

  long tupik=0;//количество тупиков обнуляем
  /*объявляем массив для хранения координат пунк-
тов, участвующих в цикле перевозок*/
  Array way(0,0,20);
  /*добавляем в массив первый (несуществующий
пункт)*/
  way.Add(pq);
  for(long i=0;;)
  {
    /*пытаемся двигаться в различных направлениях
так, чтобы составить цикл*/
    otkat
    TPoint newpoint=down(x,way[i]);
    addpoint
    else
    {
      otkat
      newpoint=up(x,way[i]);
      addpoint
    }
    otkat
```

```
    newpoint=left(x,way[i]);
    addpoint
    else
    {
      otkat
      newpoint=right(x,way[i]);
      addpoint
    }
  }
  /*если в цикле три точки лежат в одном ряду, вы-
чёркиваем среднюю*/
  for(long i=0;i<way.GetItemsInContainer()-2;i++)
    if(OnOneLine(way[i],way[i+1],way[i+2]))
      way.Detach(i+1);
  return way;/*возвращаем цикл перевозок для данной
точки*/
}


/*составляем матрицу методом "северо-западного уг-
ла", распределяя запасы a в соответствии с заявками
b*/
dmatrix transp_getopor(dvector a, dvector b)
{
  dmatrix x(a.getm(),b.getm());
  for(long j=0,i=0;j<b.getm();)
  {
    if(a[i]>=b[j])//если запасы превышают заявки
    {
      x[i][j]=b[j];//удовлетворяем заявки
      a[i]-=b[j]; //уменьшаем запасы
      j++;/*переходим к обслуживанию следующей за-
явки*/
    }
    else//иначе
    {
      x[i][j]=a[i];/*удовлетворяем, сколько есть
запасов*/
      b[j]-=a[i]; //уменьшаем количество заявок
      i++; //переходим к следующему запасу
    }
  }
  return x;//возвращаем опорное решение
```

```
}


//передвижение груза по циклу в матрице перевозок
void movegruz(dmatrix &x,Array way)
{
  /*определяем пункт с минимальной ценой в отрица-
тельных вершинах цикла*/
  double _min=x[way[1].X()][way[1].Y()];
  for(long i=3;i<way.GetItemsInContainer();i++)
    if(x[way[i].X()][way[i].Y()]<_min)
      _min=x[way[i].X()][way[i].Y()];
  /*в положительных вершинах количество груза уве-
личиваем, в отрицательных – уменьшаем*/
  for(long i=0;i<way.GetItemsInContainer();i+=2)
    x[way[i].X()][way[i].Y()]+=_min;
  for(long i=1;i<way.GetItemsInContainer();i+=2)
    x[way[i].X()][way[i].Y()]-=_min;
}


//определяем цену цикла
double getprice(dmatrix c,Array way)
{
  double price=0;
  /*положительные вершины прибавляем, отрицательные
– вычитаем*/
  for(long i=0;i<way.GetItemsInContainer();i+=2)
    price+=c[way[i].X()][way[i].Y()];
  for(long i=1;i<way.GetItemsInContainer();i+=2)
    price-=c[way[i].X()][way[i].Y()];
  return price;//возвращаем цену цикла
}


/*поиск оптимальной (минимизирующей стоимость) пе-
ревозки; возвращаемое значение - матрица, первые
два столбца в которой указывают номер склада (от 0)
и номер пункта назначения (тоже от 0), а последний
-  количество перевозимого груза ("из первого
столбца во второй")*/
dmatrix transp_optim(dmatrix &x, dmatrix c)
{
```

```
  long i,j,count;
  for(i=0;i<x.getm();i++)
    for(j=0;j<x.getn();j++)
      if(!x[i][j])//если текущая ячейка пустая
      {
        Array way=findway(x,TPoint(i,j));/*ищем для
неё цикл перевозки*/
        if(getprice(c,way)<0)/*если найденный цикл
имеет отрицательную цену, то мы можем улучшить по-
ложение дел,*/
        {
          movegruz(x,way);//перевезя по нему груз
          j=x.getn();/*перезапускаем после перевоз-
ки цикл просмотра пустых ячеек*/
          i=-1;
        }
      }
  //подсчитываем количество перевозок
  for(i=count=0;i<x.getm();i++)
    for(j=0;j<x.getn();j++)
      if(x[i][j])
        count++;
   dmatrix res(count,3);/*создаём матрицу, в кото-
рую заносим направление перевозки и количество гру-
за*/
  for(i=count=0;i<x.getm();i++)
    for(j=0;j<x.getn();j++)
      if(x[i][j])
        res[count][0]=i,       res[count][1]=j,
        res[count][2]=x[i][j], count++;
  return res;//возвращаем план перевозок
}


/*решение транспортной задачи. Эта функция принима-
ет матрицу стоимости и вектора запасов и заявок (a
и b)*/
dmatrix transp(dmatrix c, dvector a, dvector b)
{
  //проверка входных параметров на корректность
  if(c.getm()!=a.getm())
    throw xmsg("Некорректные входные параметры");
  if(c.getn()!=b.getm())
```

```
    throw xmsg("Некорректные входные параметры");
  double sum1=0,sum2=0;
  for(long i=0;i<a.getm();sum1+=a[i++]);
  for(long i=0;i<b.getm();sum2+=b[i++]);
  if(sum1!=sum2)
    throw xmsg("Некорректные входные параметры");
  dmatrix x=transp_getopor(a,b);/*поиск опорного
решения*/
  return transp_optim(x,c); //оптимизация решения
}


//Тест
void main()
{
  double mtr[]=//матрица стоимости
  {
    10,7,6,8,
     5,6,5,4,
     8,7,6,7
  };
  double vec1[]={31,48,38};//запасы
  double vec2[]={22,34,41,20};//заявки
  dmatrix c(3,4,mtr), a(3,vec1), b(4,vec2);
  dmatrix res=transp(c,a,b);
  cout<<res<<endl;
  double L=0;
  //подсчитываем цену перевозки
  for(long i=0,count=0;i<c.getm()&&
                       count<res.getm();i++)
    for(long j=0;j<c.getn()&&count<res.getm();j++)
      if(i==res[count][0]&&j==res[count][1])
        L+=c[i][j]*res[count][2],
        count++;
  cout<<"L="<<L<<endl;
}
```

3. Аналитическое приближение функций, заданных таблично

3.1. Общая постановка задачи

Представление функции в виде дискретной последовательности значений аргумента (так называемых узлов) и соответствующих значений функции в узловых точках – обычное и естественное представление при обработке данных на цифровой вычислительной машине.

Например, подключенные к машине цифровые преобразователи измерительных устройств поставляют последовательность значений параметра (температуры, давления, расхода, расстояния, скорости и т.п.) в дискретные моменты времени. Мы можем использовать также и дискретные представления аналитических функций достаточно сложной структуры. Если при этом возникает потребность в выполнении таких операций, как интегрирование, дифференцирование функции или просто необходимость оценить ее значения в неузловых точках, то желание осуществить аналитическую замену дискретной последовательности тоже выглядит достаточно естественным. Для упрощения будем считать вначале, что значения функции заданы при равноотстоящих друг от друга значениях аргумента – т.е. если не оговорено альтернативное представление, будем рассматривать задачу приближения при равноотстоящих узлах; это дает возможность представить аргумент функции в виде, например, последовательности целых чисел с выбранным постоянным шагом между ними.

Постановка задачи аналитического приближения (***аппроксимации***) функций и методы ее решения определяются следующими основными факторами, которые необходимо проанализировать в самом начале:

1) В нашем распоряжении числовой ряд значений функции, полученный на некотором интервале значений

аргумента – назовем его интервалом наблюдения. Если аналитическая замена этого ряда необходима для последующей работы внутри интервала наблюдения, то соответствующая задача называется ***задачей интерполяции***. Может оказаться, что нас интересуют оценки значений функции правее интервала наблюдения, т.е. в той области значений аргумента, где значения функции не заданы – в этом случае говорят о ***задаче экстраполяции или прогнозирования***. Наконец, может стоять задача получения достоверной оценки значения функции непосредственно на правой границе интервала наблюдения или в любом его узле – эта задача возникает при наличии существенных ошибок в определении значений функции (так называемых шумов в наблюдениях) и называется ***задачей фильтрации или сглаживания***. Таким образом, задача аналитического приближения табличных функций – это комплексная задача интерполяции, экстраполяции и сглаживания. Её «направленность» в значительной мере определяет методы ее решения. Когда аргументом функции является время, можно говорить о направленности в прошлое, будущее или оценках настоящего.

2) Второй существенный аспект проблемы состоит в определении, что называть «хорошим» или «наилучшим» приближением, т.е. в выборе ***критерия оптимальности приближения***. Классические методы приближения функций используют в качестве критерия оптимальности приближения требование точного совпадения значений приближающей функции с табличными значениями в узлах (приближения по критерию Лагранжа). Преимущество такого подхода – в простоте теории приближения и вычислительных процедур; недостаток – в игнорировании неизбежного в реальных условиях наличия шумов в наблюдениях. Общая задача интерполяции, сглаживания и экстраполяции была впервые математически сформулирована А.Н. Колмогоровым, в начале второй мировой войны ею занимался

Норберт Винер в приложении к управлению зенитным огнём, связанным с необходимостью прогнозировать координаты цели на время полета снаряда. Задача слежения за целью всегда связана с ее непредвиденными маневрами и ошибками наблюдения, поэтому сглаживание и упреждение траектории цели – актуальная задача, не вписывающаяся в классическую постановку с точным отслеживанием значений в узлах. В дальнейшем необходимость в сглаживании и упреждении при различных условиях и различных требованиях к качеству и области допустимых прогнозов возникла в различных задачах экономики, метеорологии, автоматическом регулировании и пр.

При необходимости учета ошибок наблюдений чаще всего используют классический критерий минимума суммы квадратов разностей между расчетными по приближающей функции и наблюдаемыми значениями в узлах (***критерий Гаусса***) и его современные модификации.

Другой критерий связывают с именем Чебышева; он состоит в требовании минимизировать максимальную разность между наблюдаемыми и расчетными значениями функции в узлах – так называемый ***минимаксный критерий***. Существуют и другие подходы к оценке качества приближения.

3) Третий ключевой момент состоит в выборе класса приближающих аналитических функций. Основное требование, которое у нас возникает к этим функциям – это независимость результатов аппроксимации от начала отсчета, т.е. от сдвига по последовательности значений аргумента. Другими словами, необходимо, чтобы конечное множество функций выбранного для аппроксимации класса переходило само в себя при замене x на $x+k$. Таким свойством обладают:

- линейные комбинации степенных функций $1, x, x^2, \ldots, x^n$;
- тригонометрические функции $\cos(a_i x)$, $\sin(a_i x)$;

– экспоненциальные функции вида $e^{-a_i z}$, часто встречающиеся в реальных задачах.

Использование любого другого конечного множества аппроксимирующих функций (кроме трех перечисленных) подразумевает наличие естественного начала отсчета, так как выбор начала повлияет на результат - но в большинстве задач нет естественного начала и мы вынуждены выбирать из перечисленных классов функций.

В соответствии с перечисленными классами различают ***полиномиальную*** аппроксимацию, Фурье или ***тригонометрическую*** аппроксимацию и ***экспоненциальную*** аппроксимацию.

Еще одно важное свойство, которое хотелось бы придать множеству аппроксимирующих функций – это инвариантность к изменению масштаба, т.е. чтобы множество не изменялось при замене x на kx. Из рассмотренных множеств таким свойством обладает только множество

$$1, x, x^2, \ldots, x^n,$$

поэтому при отсутствии в задаче естественного масштаба мы вынуждены либо выбирать полиномы, либо воздействовать на результат волевым выбором масштаба. Правда, большинство практических задач, в частности, связанных с функциями времени, имеют естественный масштаб и преобладание полиномиальной аппроксимации над другими видами объясняется большей продвинутостью полиномиальной алгебры и простотой вычислительных процедур с полиномами.

3.2. Общая методика решения задач аппроксимации

Обычно в качестве аппроксимирующей функции выбирают линейную комбинацию функций, выбранных из рассмотренных классов, вида:

$$t(x) = \sum_{i=0}^{n} c_i \varphi_i(x)$$

с действительными коэффициентами c_i. Если такой обобщенный многочлен сформирован и выбран критерий оптимальности приближения, то можно составить и решить систему алгебраических уравнений, линейных относительно коэффициентов c_i:

$$\sum_{i=0}^{n} c_i \varphi_i(x_k) = f(x_k),$$

где k=0, 1, 2, … – порядковый номер узла, $f(x_k)$ – заданное табличное значение функции в k-м узле.

Очевидно, что количество таких уравнений должно быть не меньше n+1, а способ ее решения зависит от критерия приближения.

Если требуется точное проведение аппроксимирующей функции через узловые точки, то есть решается задача интерполяции в постановке Лагранжа, то $k=n+1$, а к системе функций $\varphi_i(x)$ предъявляются следующее требование:

Чтобы для заданной функции $f(x)$, определенной на отрезке $[a, b]$, и набора n+1 узлов x_0, x_1, ..., x_n ($x \subset [a, b]$) существовал и был единственным обобщенный интерполяционный многочлен $t(x)=\sum_{i=0}^{n} c_i \varphi_i(x)$, необходимо и достаточно, чтобы любой обобщенный многочлен (с хотя бы одним ненулевым коэффициентом) по выбранной системе функций $\varphi_i(x)$ имел на отрезке $[a, b]$ не более n корней.

В настоящем пособии рассмотрены только основные методы алгебраической аппроксимации в разрезе задач интерполяции функций и сглаживания по методу наименьших квадратов. Специфичные вопросы прогнозирования и фильтрации требуют рассмотрения стохастических подходов и не укладываются в короткий вводный курс.

3.2.1. Алгебраическая интерполяция

3.2.1.1. Классический интерполяционный полином

Использование в качестве $\varphi_i(x)$ последовательности степенных функций приводит к аппроксимирующей функции в виде классического полинома $\sum_{i=0}^{n} c_i x^i$ и к системе $n+1$ линейных уравнений вида: $\sum_{i=0}^{n} c_i x_k^i = f(x_k)$ относительно c_i. Определитель этой системы есть определитель Вандермонда, который не равен нулю, если мы не используем повторяющихся узлов. Решение СЛАУ любым из известных методов относительно c_i полностью определяет интерполяционный полином и оценка значения $f(x)$ в любой неузловой точке может быть получена вычислением значения полинома в этой точке. Этот полином можно также использовать в дальнейших аналитических процедурах дифференцирования, интегрирования и пр.

В следующих разделах мы рассмотрим другие формы записи интерполяционных полиномов, но необходимо понимать, что независимо от формы записи два интерполяционных полинома степени n, проведенных через одни и те же $n+1$ узловые точки, тождественно равны, так как их разность есть полином степени не выше n и его значение обращается в нуль в этих узловых точках, т.е. тождественно равна нулю.

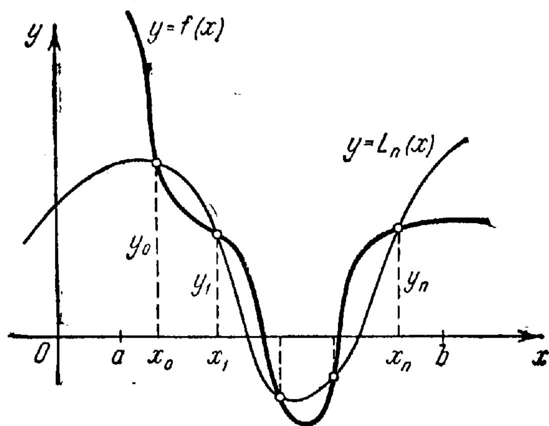
3.2.1.2. Метод интерполяции Лагранжа

При использовании классического полинома мы сначала получали полином (его коэффициенты), а затем использовали его для интерполяции. Метод Лагранжа предполагает строить интерполяционный полином для каждого вычисляемого значения, объединяя его построение с вычислением. Основная идея этого метода состоит в том, что

сначала ищется многочлен, значение которого равно 1 в одной узловой точке и 0 – в остальных. Функция

$$L_j(x)=\frac{\prod_{i=0}^{n}(x-x_i)}{\prod_{i=0}^{n}(x_j-x_i)}$$

при $i\neq j$ есть многочлен степени n, значение которого 1 при $x=x_j$ и 0 если $x=x_i$, $i\neq j$.



Тогда многочлен $f(x_i)L_j(x)$ будет принимать значение $f(x_i)$ в i-й узловой точке и 0 в остальных узлах. Тогда многочлен $f(x)=\sum_{j=0}^{n} f(x_j)L_j(x)$ степени n проходит через n+1 точку (x_i, y_i).

Эта запись неудобна для вычислений и ее приводят к так называемой барицентрической форме. Пусть все $f(x_i)=1$, тогда $\sum_{j=1}^{n} L_j(x)=1$. Положим $A_j=\frac{1}{\prod_i (x_j-x_i)}$ и разделим числитель и знаменатель на $\prod_j (x-x_j)$. Получим расчетную формулу:

$$f(x)=\frac{\sum_{j=0}^{n}[A_j f(x_j)/(x-x_j)]}{\sum_{j=0}^{n}[A_j/(x-x_j)]}.$$

Так как метод Лагранжа требует вычисления интерполяционного полинома при каждом определении значения в межузловых промежутках, используют его только при интерполяции на небольшом количестве точек.

3.2.1.3. Интерполяционный полином Ньютона

Полином Ньютона для интерполяции имеет вид:

$$N_n(x)=A_0+A_1(x–x_0)+A_2(x–x_0)(x–x_1)+...+A_n(x–x_0)(x–x_1)...(x–x_{n-1})=$$

$$=A_0+\sum_{i=1}^{n}A_i\prod_{j=1}^{i}(x-x_{j-1}). \qquad (*)$$

Равносильный вариант полинома можно получить при симметричной перенумерации узлов и значений функции в исходной таблице от n до 0:

$$N_n(x)=B_n+B_{n-1}(x–x_n)+B_{n-2}(x–x_n)(x–x_{n-1})+...+$$

$$+B_0(x–x_n)(x–x_{n-1})...(x–x_1)=B_n+\sum_{i=1}^{n}B_{n-i}\prod_{j=1}^{i}(x-x_{n-j+1}).$$

Коэффициенты полиномов определяются из условий Лагранжа

$$N_n(x_j)=f(x_j),\ 0\le j\le n$$

Полагаем $x=x_0$, тогда в формуле (*) все слагаемые, кроме A_0, обращаются в нуль и $A_0=f(x_0)$.

Затем полагаем $x=x_1$, тогда $f_0+A_1(x_1-x_0)=f_1$, откуда находим коэффициент

$$A_1=(f(x_0)–f(x_1))/(x_0–x_1)=f_{01},$$

который называется разделенной разностью первого порядка. Величина этой разности близка к первой производной функции $f(x)$ при малом расстоянии между узлами x_0 и x_1.

При $x=x_2$ полином принимает значение

$$N_n(x_2)=f(x_0)+A_1(x_2-x_0)+A_2(x_2-x_0)(x_2-x_1),$$

из условия Лагранжа определим искомый коэффициент

$$A_2=(f_{01}-f_{02})/(x_1-x_2)=f_{012},\ \text{где}\ f_{02}=(f(x_0)-f(x_2))/(x_0-x_2).$$

Величина f_{012} называется разделенной разностью второго порядка, которая при близком расположении x_0, x_1, x_2 будет пропорциональна второй производной функции $f(x)$.

Аналогично при $x=x_3$ находим коэффициент A_3:

$$A_3=(f_{012}-f_{013})/(x_2-x_3)=f_{0123},$$

$$\text{где}\ f_{013}=(f_{01}-f_{03})/(x_1-x_3),\ f_{03}=(f(x_0)-f(x_3))/(x_0-x_3).$$

Для коэффициента A_k методом математической индукции запишем следующее выражение:

$$A_k=(f_{01\ldots k\text{-}1}-f_{01\ldots k})/(x_{k\text{-}1}-x_k).$$

Полученные результаты сведем в таблицу:

x	$f(x)$	1	2	3	4
x_0	$f(x_0)$				
x_1	$f(x_1)$	$f_{01}=\dfrac{f(x_0)-f(x_1)}{x_0-x_1}$			
x_2	$f(x_2)$	$f_{02}=\dfrac{f(x_0)-f(x_2)}{x_0-x_2}$	$f_{012}=\dfrac{f_{01}-f_{02}}{x_1-x_2}$		
x_3	$f(x_3)$	$f_{03}=\dfrac{f(x_0)-f(x_3)}{x_0-x_3}$	$f_{013}=\dfrac{f_{01}-f_{03}}{x_1-x_3}$	$f_{0123}=\dfrac{f_{012}-f_{013}}{x_2-x_3}$	
x_4	$f(x_4)$	$f_{04}=\dfrac{f(x_0)-f(x_3)}{x_0-x_3}$	$f_{014}=\dfrac{f_{01}-f_{04}}{x_1-x_4}$	$f_{0124}=\dfrac{f_{012}-f_{014}}{x_2-x_4}$	$f_{01234}=\dfrac{f_{0123}-f_{0124}}{x_3-x_4}$

Для построения интерполяционного полинома Ньютона используются только диагональные элементы приведенной таблицы, остальные являются промежуточными данными.

Добавление новых узлов в таблицу не изменит уже вычисленных коэффициентов, таблица будет дополнена новыми строками и столбцами разделенных разностей.

Погрешность полиномиальной аппроксимации функции $f(x)$ определяется соотношением:

$$|f(x)-N_n(x)|<\frac{M_{n+1}}{(n+1)!}\prod_{i=1}^{n}|x-x_i|,\ где\, M_{n+1}=\max|f^{(n+1)}(x)|.$$

Так как нам обычно неизвестны заранее производные функции $f(x)$, то по приведенной формуле можно провести апостериорную оценку погрешности, основанную на том, что при достаточно близком расположении узлов разделенные разности являются приближенными оценками производных соответствующего порядка k, деленными на $k!$.

Поэтому правая часть приведенного неравенства приближенно совпадает по модулю с новым членом полинома Ньютона, появляющимся при добавлении (n+1)-го узла. Таким образом, вычисление модуля каждого из членов суммы (*) позволяет установить, сколько узлов следует использовать для аппроксимации исходной функции с заданной точностью. Если узлы расположены равномерно с шагом h, то наименьшая погрешность будет в интервалах, примыкающих к центральному узлу, за счет минимальной величины произведения в правой части оценки погрешности. Особенно резко возрастает погрешность при попытках экстраполяции.

В центральном интервале оценка погрешности может быть получена по:

$$|f(x)-N_n(x)|<M_{n+1}(h/2)^{n+1}\sqrt{2/\pi n}$$

После определения коэффициентов полинома Ньютона вычисление его значений при конкретных аргументах x наиболее экономично проводить по схеме Горнера, получаемой последовательным вынесением за скобки множителей $(x–x_j)$ в формуле (*):

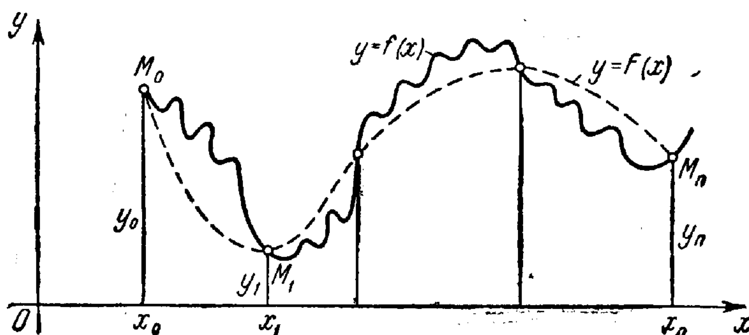
$$N_n(x)=f_0+(x–x_0)(f_{01}+(x–x_1)(f_{012}+(x–x_2)(f_{0123}+\ldots)\ldots).$$

В отличие от алгоритма вычисления полинома Лагранжа, при интерполяции полиномом Ньютона удается разде-

лить задачи определения коэффициентов и вычисления значений полинома при различных значениях x.

3.2.1.4. Интерполяция сплайнами

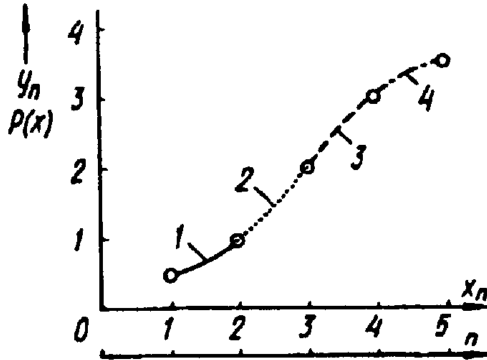
Может показаться, что алгебраической интерполяции всегда сопутствует порядок полинома, равный количеству заданных узлов. На самом деле это не так; представим, что необходимо построить график заданной таблично функции с числом узлов в несколько десятков – нецелесообразность использования столь высоких степеней интерполяционного полинома становится очевидной. Но есть и другая причина для снижения степени интерполяционного полинома: несмотря на выполнение условий Лагранжа в узлах интерполяции, интерполяционная функция может иметь значительное отклонение от аппроксимируемой кривой между узлами – возникает так называемый ***эффект волнистости***.



Поэтому осуществляют, как правило, кусочную аппроксимацию заданной функции, разбивая весь набор узлов на группы по 2-4 узла и аппроксимируя функцию на отрезках полиномами степеней не выше третьей. Для задач типа численного интегрирования функций этого вполне достаточно, но, скажем, для упомянутой задачи построения графиков надо позаботиться о «сшивании» соседних полиномов в точках соприкосновения, иначе кривые на отдельных участках в этих точках будут иметь различные наклоны

(вплоть до разных знаков первых производных) и результирующая кривая не будет гладкой.

Для проведения гладких кривых через узловые точки, помимо условий Лагранжа, накладывают дополнительные требования к интерполяционной кривой.



Метод интерполяции, получивший свое название от упругой металлической линейки, накладываемой на узловые точки аппроксимируемой кривой, называется ***методом сплайновой интерполяции***. По законам упругости металлическая линейка, опирающаяся на ряд точек, проходит между ними по линии с нулевой четвертой производной $\varphi^{(4)}(x)=0$. Если в качестве функции $\varphi(x)$ выбрать полином, то его степень должна быть не выше третьей - этот полином и носит название кубического сплайна, имеющего на интервале $[x_{j-1}, x_j]$ вид:

$$\varphi_i=a_i+b_i(x-x_{i-1})+c_i(x-x_{i-1})^2+d_i(x-x_{i-1})^3,$$

где a_i, b_i, c_i, d_i – коэффициенты сплайна, определяемые из дополнительных условий, i=1, 2, ..., n – номера сплайнов. При сплайн-интерполяции на каждом интервале $[x_{j-1}, x_j]$ строится отдельный полином 3-й степени со своими коэффициентами, которые определяются из условия сшивания соседних сплайнов в узловых точках:

1) выполнение условия Лагранжа $\varphi_i(x_{i-1})=f(x_{i-1})$, $\varphi_i(x_i)=f(x_i)$;

2) непрерывность первой и второй производной в узлах $\varphi_i^{(1)}(x_i)=\varphi_{i+1}^{(1)}(x_i), \varphi_i^{(2)}(x_i)=\varphi_{i+1}^{(2)}(x_i)$.
3) условия на концах могут быть различными – в том числе со свободными концами, т.е. описываться уравнениями прямых и в этом случае иметь нулевые вторые производные:

$$\varphi_1^{(2)}(x_0)=0, \varphi_n^{(2)}(x_n)=0 .$$

Алгоритм определения коэффициентов кубических сплайнов с учетом оговоренных условий выглядит так:

Условия Лагранжа в узлах $x_{i\text{-}1}$ после подстановки i-го сплайна принимают вид:

$$a_i=f(x_{i\text{-}1}),\ a_i+b_ih_i+c_ih_i^2+d_ih_i^3=f(x_i),\ \text{где } h_i=x_i-x_{i\text{-}1},\ 1\le i\le n\ .\ (*)$$

Продифференцировав дважды сплайн по x, получим:

$$\varphi_i^{(1)}=b_i+2c_i(x-x_{i\text{-}1})+3d_i(x-x_{i\text{-}1})^2,$$

$$\varphi_i^{(2)}=2c_i+6d_i(x-x_{i\text{-}1}).$$

Из условий непрерывности производных при переходе в точке x_i от i-го к (i+1)-му сплайну с учетом предыдущих выражений получим:

$$b_i+2c_ih_i+3d_ih_i^2=b_{i+1},\ c_i+3d_ih_i=c_{i+1}.\ (**)$$

Из граничных условий на основании последнего выражения для второй производной получим

$$c_1=0,\ c_n+3d_nh_n=0.\ (***)$$

Вышеприведенные соотношения (*), (**), (***) представляют собой СЛАУ относительно коэффициентов сплайнов a_i, b_i, c_i, d_i. Преобразуем ее так, чтобы неизвестными была только одна группа коэффициентов c_i:

$$d_i=(c_{i+1}-c_i)/3h_j,\ b_i=(f(x_i)-f(x_{i\text{-}1})/h_i-(c_{i+1}+2c_i)h_j/3.$$

После подстановки этих выражений в (**) получим уравнение относительно c_i. Для симметрии записи в полученном уравнении уменьшим значение индекса i на единицу

$$h_{i\text{-}1}c_{i\text{-}1}+2(h_{i\text{-}1}+h_i)c_i+h_ic_{i+1}=3[(f(x_i)-f(x_{i\text{-}1}))/h_i-(f(x_{i\text{-}1})-f(x_{i\text{-}2}))/h_{i\text{-}1}],$$

где $2\le i\le n$.

При $i=n$ для свободных концов $c_{n+1}=0$. Таким образом, $n-1$ уравнение для c_i вместе с $c_1=0$ и $c_{n+1}=0$ образуют СЛАУ для вычисления c_i, а b_i и d_i мы уже выразили через c_i. Коэффициенты a_j равны значениям аппроксимируемой функции в узлах $a_i=f(x_{i-1})$.

В каждое из уравнений системы входят только 3 неизвестных – c_{i-1}, c_i, c_{i+1}, поэтому матрица СЛАУ является трёхдиагональной. Для решения таких систем наиболее эффективен метод прогонки (частный случай метода Гаусса). Рассмотрим один из его вариантов применительно к нашей задаче. Чтобы сократить запись, представим последнее уравнение в виде:

$$h_{i-1}c_{i-1}+s_ic_i+h_ic_{i+1}=r_i,$$

где $s_i=2(h_{i-1}+h_i)$, $r_i=3[(f(x_i)-f(x_{i-1}))/h_i-(f(x_{i-1})-f(x_{i-2}))/h_{i-1}]$.
Как и в методе Гаусса, работаем в 2 этапа: в прямом ходе вычисляем значения коэффициентов линейной связи каждого предыдущего неизвестного c_i с последующим c_{i+1}.

Из последнего уравнения при $i=2$ с учетом граничного условия (***) установим связь коэффициента c_2 с коэффициентом c_3:

$$c_2=k_2-l_2c_3,$$

где $k_2=r_2/s_2$, $l_2=h_2/s_2$ – прогоночные коэффициенты.
Затем, подставляя c_2 в последнее уравнение при $i=3$, получим линейную связь c_3 с c_4:

$$c_3=k_3-l_3c_4.$$

Аналогично для любых соседних коэффициентов с номерами i, $i+1$ можно установить линейную связь в виде

$$c_i=k_i-l_ic_{i+1}.$$

В процессе выполнения прямого хода метода прогонки необходимо вычислить значения всех прогоночных коэффициентов k_i, l_i, для которых получены рекуррентные соотношения. Подставим формулу связи ($i-1$)-го и i-го коэффициентов $c_{i-1}=k_{i-1}-l_{i-1}c_i$ в уравнение и в результате получим:

$$c_i=\frac{r_i-h_{i-1}k_{i-1}}{s_i-h_{i-1}l_{i-1}}-\frac{h_i}{s_i-h_{i-1}l_{i-1}}c_{i+1}.$$

Сравнение этого соотношения с формулой для c_i позволяет записать рекуррентные формулы для определения прогоночных коэффициентов l_i, k_i:

$$k_i=\frac{r_i - h_{i-1}k_{i-1}}{s_i - h_{i-1}h_{i-1}},\ l_i = \frac{h_i}{s_i - h_{i-1}l_{i-1}}.$$

Учитывая граничное условие (***) и соотношение $c_1=k_1-l_1c_2$, а также полагая $c_2\neq0$, задаем начальные коэффициенты $k_1=0$, $l_1=0$. Затем по формуле для k_i, l_i вычислим все n пар прогоночных коэффициентов k_i, l_i. На основании соотношения $c_n=k_n-l_nc_{n+1}$ и граничного условия $c_{n+1}=0$ получим $c_n=k_n$. Далее, последовательно применяя формулу $c_i=k_i-l_ic_{i+1}$ при $i=n-1$, $n-2$, …, 2 вычислим значения искомых c_{n-1}, c_{n-2}, …, c_2. Эта процедура называется обратным ходом метода прогонки.

После определения всех c_i другие коэффициенты сплайнов вычисляются с помощью уже приведенных их представлений через c_i.

3.2.1.5. Аппроксимация функций по методу наименьших квадратов

Если мы снимаем требование обязательного прохождения аппроксимирующей функции через узловые точки и заменяем его требованием минимума суммы квадратов разностей между значениями аппроксимирующей и аппроксимируемой функций в узлах, то приходим к ***методу наименьших квадратов***, который не игнорирует наличие ошибок в значения аппроксимируемой функции, а пытается усреднить их влияние на результат аппроксимации. Теория метода и его программная реализация в матричном классе нами уже рассмотрена, а его включение в класс аппроксимирующих функций – дело техники объектно-ориентированного программирования. Поэтому мы не будем повторять уже изложенные материал, отметим только, что МНК используется обычно не для кусочной аппроксимации на отрезках общего интервала (хотя и такой подход возмо-

жен), а для аппроксимации на всем интервале задания исходной функции. При использовании ортогональных функций в качестве базисных значительно упрощается вычисление коэффициентов разложения исходной функции по базисным функциям, что позволяет последовательным наращиванием количества членов разложения определить минимальную степень полинома для достижения приемлемой точности аппроксимации.

3.2.2. Программная реализация класса аппроксимирующих функций

Класс аппроксимирующих функций должен, по-видимому, содержать в качестве членов-данных два вектора одинаковой размерности для хранения последовательности узлов и значений заданной функции в этих узлах. Набор функций-членов, помимо обязательного набора конструкторов, должен включать методы реализации рассмотренных алгоритмов и, возможно, подпрограммы для интерфейса пользователя. Ниже приведено содержание включаемого файла с определением класса аппроксимаций, содержащего указанные компоненты с сопровождающими комментариями.

```
#include "polynom.h"
#include "matrix.h"

/*Класс полиномиальных аппроксимаций */

template <class YourOwnFloatType>
class Approximate
{
/*Для решения задачи аппроксимации нам понадобится
массивы узлов и значений функции в узлах, т.е. объ-
екты класса vector */
  vector<YourOwnFloatType> x, y;/* Узлы и значения
в узлах*/
 public:/* Общедоступные методы */
```

```
/*Основной конструктор в качестве параметра прини-
мает два вектора - вектор узлов и вектор значений в
узлах */
  Approximate(vector<YourOwnFloatType> _x, vector
<YourOwnFloatType> _y): x(_x), y(_y)
  {
    /*Проверка размерностей на корректность */
    if(x.getm()!=y.getm())
      throw xmsg("Количество узлов не совпадает с
количеством значений в них");
  /*Наверное, в этом случае задача аппроксимации
теряет смысл: */
    if(x.getm()<2)
      throw xmsg("Слишком мало точек");
  }
  /*Конструктор копирования */
  Approximate(Approximate &_): x(_.x), y(_.y) {}
/*Результатом решения должен быть, естественно,
объект класса polynom */
  //Прототипы методов класса аппроксимаций
  //Интерполяция каноническим полиномом
  polynom<YourOwnFloatType> classic(),
  //Интерполяция полиномом Ньютона
          newton();
/*Интерполяция кубическими сплайнами. Возвращаемое
значение - массив полиномов, организуемый динамиче-
ски. По окончании работы память из под него в обя-
зательном порядке необходимо освободить */
  polynom<YourOwnFloatType>* spline();
/*Аппроксимация функции одной переменной по методу
наименьших квадратов при степенном базисе. Аргумент
- порядок полинома */
  polynom<YourOwnFloatType> MNK(int);
};


/*Теперь определения методов класса полиномиальных
аппроксимаций */
//Интерполяция каноническим полиномом
template <class YourOwnFloatType>
polynom<YourOwnFloatType> Approxi-
mate<YourOwnFloatType>::classic()
{
```

```
  /*Матрицы для хранения левой и правой частей си-
стемы уравнений */
  matrix<YourOwnFloatType> mtr(x.getm(),x.getm()),
f(x.getm(),1);

  //Формирование матрицы СЛАУ
  for(int i=0;i<x.getm();i++)
  {
    /*Правая часть системы - значения функции в уз-
лах */
    f[i][0]=y[i];
    /*Левая часть системы - вектор степеней узлов
*/
    for(int j=0;j<x.getm();j++)
      mtr[i][j]=pow(x[i],j);
  }
  matrix<YourOwnFloatType> sol=SLAE_Gauss(mtr,f);
/*Вызываем метод Гаусса из матричного класса для
решения СЛАУ. /*Результат решения построенной си-
стемы - коэффициенты полинома */
  polynom<YourOwnFloatType> res=x.getm();
  /*формируем полином из коэффициентов */
  for(int i=0;i<x.getm();i++)
    res[i]=sol[i][0];
  return res;/*Возвращаем результат */
}


//Интерполяция полиномом Ньютона
template <class YourOwnFloatType>
polynom<YourOwnFloatType> Approxi-
mate<YourOwnFloatType>::newton()
{
  //Формирование коэффициентов полинома
  matrix<YourOwnFloatType> ta-
ble(x.getm(),x.getm());
/*Первый столбец таблицы - значения в узлах, во
всех последующих - разделённые разности соответ-
ствующего порядка */
  for(int i=0;i<x.getm();i++)
    table[i][0]=y[i];
  for(int j=1;j<x.getm();j++)
    for(int i=j;i<x.getm();i++)
```

```
      table[i][j]=
   (table[j-1][j-1]-table[i][j-1])/(x[j-1]-x[i]);
/*В диагонали сформированной матрицы находятся ко-
эффициенты полинома в Ньютоновой форме. Для того,
чтобы перейти к канонической записи полинома, про-
делаем следующее: */
/*Объявим и инициализируем два вспомогательных по-
линома - полиномиальный X и полиномиальную единицу
*/
  polynom<YourOwnFloatType> Odin,X=2,p;
  Odin[0]=1,X[1]=1; /*инициализируем их */
/*сформируем искомый полином как сумму произведений
одночленов соответствующих степеней */
  for(int i=0;i<x.getm();i++)
  {
  polynom<YourOwnFloatType> temp=table[i][i]*Odin;
    for(int j=0;j<i;j++)
      temp*=(X-x[j]*Odin);
    p+=temp;
  }
  return p;/* возвращаем результат */
}

//Интерполяция кубическими сплайнами.
/*Общее количество сплайнов равно количеству меж-
узловых промежутков, т.е. x.getm()-1. В связи с
этим результатом выполнения данной функции будет не
один полином, а x.getm()-1*/
template <class YourOwnFloatType>
polynom<YourOwnFloatType>* Approxi-
mate<YourOwnFloatType>::spline()
{
/*создаём две матрицы для левой и правой частей
СЛАУ, результатом решения которой будут производные
второго порядка каждого сплайна */
  matrix<YourOwnFloatType> mtr(x.getm(),x.getm()),
right(x.getm(),1);
  /*0,n-1 - в этих точках вторая производная равна
нулю*/
  mtr[0][0]=1,right[0][0]=0;
  mtr[x.getm()-1][x.getm()-1]=1,
  right[x.getm()-1][0]=0;
  for(long i=1;i<mtr.getm()-1;i++)
```

```
  {
    YourOwnFloatType hi=x[i+1]-x[i],
                     him1=x[i]-x[i-1];
    mtr[i][i-1]=him1;
    mtr[i][i]=2*(him1+hi);
    mtr[i][i+1]=hi;
    right[i][0]=
         6*(-(y[i]-y[i-1])/him1+(y[i+1]-y[i])/hi);
  }
/*Находим вторые производные и выражаем через них и
исходные данные коэффициенты сплайна в форме Тейло-
ра */
  matrix<YourOwnFloatType>
d2y=SLAE_Orto(mtr,right);
  vector<YourOwnFloatType> a=x.getm()-1;
  vector<YourOwnFloatType> b=a,c=a,d=a;
  for(long i=0;i<a.getm();i++)
  {
    YourOwnFloatType hi=x[i+1]-x[i];
    a[i]=y[i];
    b[i]=(y[i+1]-y[i])/hi-hi*(d2y[i+1][0]+2*
                                 d2y[i][0])/6;
    c[i]=d2y[i][0]/2;
    d[i]=(d2y[i+1][0]-d2y[i][0])/(6*hi);
  }
  /*Выделяем память под массив полиномов */
  polynom<YourOwnFloatType> *arr=
        new polynom <YourOwnFloatType>[a.getm()];
/*Составляем два служебных полинома - полиномиаль-
ный X и полиномиальную единицу */
  polynom<YourOwnFloatType> Odin,X=2;
  Odin[0]=1,X[1]=1;
  /*Формируем полиномы, описывающие элементарные
сплайны */
  for(int i=0;i<a.getm();i++)
    arr[i]=a[i]*Odin+b[i]*(X-x[i]*Odin)+c[i]*
         ((X-x[i]*Odin)^2)+d[i]*((X-x[i]*Odin)^3);
  return arr;/*Возвращаем указатель на заполненный
массив - не забудьте его освободить! */
}


//Аппроксимация по методу наименьших квадратов
```

```
/*Аргумент - степень полинома, которым будем ап-
проксимировать экспериментально полученные данные
*/
template <class YourOwnFloatType>
polynom<YourOwnFloatType> Approxi-
mate<YourOwnFloatType>::MNK(int rng)
{
  matrix<YourOwnFloatType> mtr(x.getm(),rng+1),
/*Прямоугольная матрица измерений */
         f(x.getm(),1); //Столбец свободных членов

  //Формирование матрицы измерений
  for(int i=0;i<mtr.getm();i++)
  {
    f[i][0]=y[i];
    for(int j=0;j<mtr.getn();j++)
      mtr[i][j]=pow(x[i],j);
  }
  //Реализуем формулу a=((mtr*mtrT)^-1)*(mtrT*f)
  matrix<YourOwnFloatType> sol=
                 SLAE_Orto((~mtr)*mtr,(~mtr)*f);
  polynom<YourOwnFloatType> a(sol.getm());
//Полином - решение
  for(long i=0;i<sol.getm();i++)
    a[i]=sol[i][0];
  return a;/* Возвращаем полином - решение задачи
аппроксимации*/
}

/*Определяем типы данных "действительный полином" и
"действительный вектор" */
typedef polynom<double> dpolynom;
typedef vector<double> dvector;

#include <conio.h>

/*Тестирующая программа */
void main()
{
  /*Узлы и значения в узлах */
  double xdata[5]={1,2,3,4,5},
ydata[5]={0.5,1,2,3,3.5};
  dvector x(5,xdata),y(5,ydata);
```

```
  Approximate<double> test(x,y);/*Объявляем тесто-
вый объект аппроксимационного класса */
  dpolynom what=test.classic();/*Канонический поли-
ном */
  cout<<what<<endl<<what(2)<<endl;
  what=test.newton();     /*Ньютоновский полином */
  cout<<what<<endl<<what(2)<<endl;
  what=test.MNK(1);             /*МНК, прямая */
  cout<<what<<endl<<what(2)<<endl;
  what=test.MNK(2);             /*МНК, парабола */
  cout<<what<<endl<<what(2)<<endl;
  what=test.MNK(3);   /*МНК, кубическая парабола */
  cout<<what<<endl<<what(2)<<endl;
  dpolynom *arr=test.spline();  /*получаем указа-
тель на массив сплайнов */
  cout<<endl<<endl;
  for(long i=0;i<x.getm()-1;i++) /*выводим сплайны
с попутным тестом их в узлах */
    cout<<arr[i]<<"     "<<arr[i](x[i])
        <<"     "<<arr[i](x[i+1])<<endl;
  delete[] arr;/*освобождаем память из под массива
сплайнов */
  getch();
}
```

4. Численное интегрирование и дифференцирование табличных функций

4.1. Численное интегрирование

Методы вычисления определенных интегралов $\int_a^b f(x)dx$ отличаются от методов вычисления неопределенных интегралов $\int_a^x f(x)dx$. В первом случае результатом вычислений будет число, во втором – функция, заданная последовательностью значений для заданной последовательности значений аргумента x.

4.1.1. Вычисление определенных интегралов

4.1.1.1. Классификация методов

Ставится задача вычислить интеграл вида $\int_a^b f(x)dx$, где a и b – нижний и верхний пределы интегрирования; $f(x)$ – функция, заданная на отрезке $[a, b]$ последовательностью значений в узлах, в простейшем случае – равноотстоящих.

К численным методам обращаются также при невозможности записать первообразную функцию аналитически через элементарные функции или когда такая запись имеет слишком сложный вид. В любом случае материалом для вычисления интеграла служит конечная последовательность дискретных значений, но при интегрировании вычисляемых аналитических функций есть возможность изменять шаг между узлами для повышения точности вычислений.

Простейший метод вычисления определенных интегралов состоит в замене подынтегральной функции $f(x)$ ап-

проксимирующей функцией $\upsilon(x)$, для которой первообразная легко выражается в элементарных функциях.

Используемые на практике методы численного интегрирования можно классифицировать по способу аппроксимации подынтегральной функции.

Методы Ньютона-Котеса основаны на полиномиальной аппроксимации и отличаются друг от друга степенью аппроксимирующего полинома. Эти алгоритмы просты и легко программируются.

Сплайновые методы базируются на сплайновой аппроксимации подынтегральной функции, различаются по типу выбранных сплайнов и применяются в задачах обработки данных с использованием сплайнов.

Методы наивысшей алгебраической точности (Гаусса-Кристофеля и др.) применимы в основном к аналитическим функциям, используют неравноотстоящие узлы, расположенные по алгоритму, обеспечивающему минимальную погрешность интегрирования для наиболее сложных функций при заданном количестве узлов.

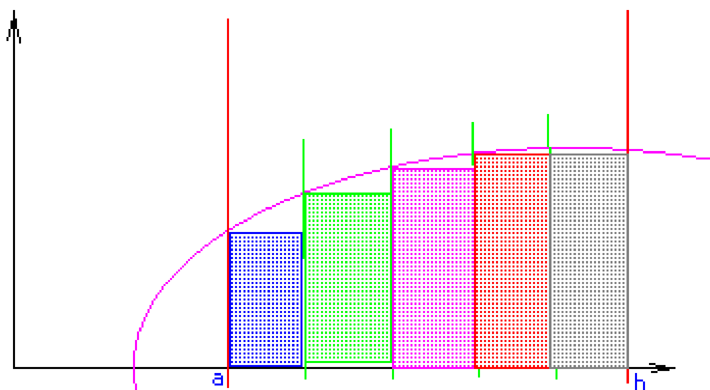
Методы Монте-Карло используют случайное расположение узлов и дает результат вероятностного смысла.

Независимо от метода, в процессе численного интегрирования необходимо вычислить приближенное значение интеграла и оценить погрешность, которая зависит от числа участков разбиения интервала интегрирования – уменьшается с ростом числа участков за счет более точной аппроксимации и одновременно растет за счет погрешности суммирования частичных интегралов. Эта составляющая с некоторого значения N начинает преобладать, что препятствует чрезмерному дроблению интервала интегрирования.

Мы рассмотрим только методы вычисления определенных интегралов из семейства методов Ньютона-Котеса.

Методы прямоугольников. Это простейшие из класса Ньютона-Котеса методы основаны на аппроксимации подынтегральной функции полиномом нулевой степени –

константой на заданном отрезке интервала. Для такой аппроксимации достаточно одной точки – любого значения подынтегральной функции в любом узле; это значение считается постоянным на всем промежутке между соседними узлами. Осмысленный выбор левой или правой границ шага разбиения в качестве значения на всем частичном интервале сделать трудно даже при наличии априорной информации о поведении функции в интервале интегрирования – метод дает только грубую оценку значения определенного интеграла – априорная нижняя оценка погрешности интегрирования $R_{APRIOR}=\frac{h}{2}\int_{x_0}^{x_n} f^{(1)}(x)dx$.



Для главного члена оценки априорных погрешностей использую конструкцию вида $R_{APRIOR}=Ah^p$, где A – коэффициент, зависящий от метода интегрирования, h – шаг интегрирования, p – порядок метода. Эту зависимость можно использовать для большинства методов численного интегрирования.

Для оценки апостериорной погрешности используют так называемую первую формулу Рунге

$$R_{APOSTER}=(w_h-w_{kh})/(k^p-1),$$

где w_h, w_{kh} – значения некоторой переменной w, вычисленной с шагом h и kh, k – коэффициент увеличения шага. Эта

формула позволяет получить главную составляющую апостериорной оценки погрешности двойным просчетом с различными шагами.

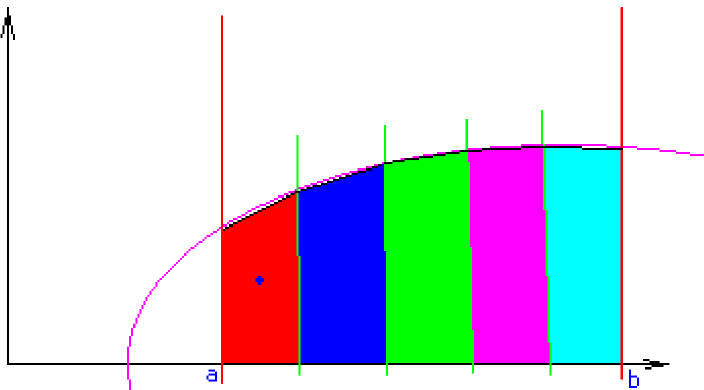
Для случая, когда порядок метода неизвестен, используют формулу Эйткена для вычисления k^p для подстановки в первую формулу Рунге:

$$k^p=(w_{kh}-w_{kh^2})(w_h-w_{kh}).$$

Чтобы использовать этот метод, надо третий раз вычислить w с шагом k^2h.

Метод трапеций получается при замене подынтегральной функции полиномом первой степени $P_1(x)$, для чего приходится использовать обе границы частичного интервала. На каждом элементарном отрезке аргумента x участок кривой интегрирования представляет собой отрезок прямой – две ординаты и отрезок оси абсцисс вместе с этой прямой ограничивают фигуру трапецеидальной формы, что и дает название этому методу кусочно-линейной аппроксимации подынтегральной функции. Приближенное значение интеграла определяется площадью трапеции:

$$\int_{x_i}^{x_i+h} f(x)dx = h[f(x_i)+f(x_i+h)]/2+R.$$



Оценка априорной погрешности равна

$$R_{APRIOR}=-\frac{h^2}{12}\int_{x_0}^{x_n} f^{(2)}(x)dx\,.$$

Метод Симпсона получается при замене подынтегральной функции полиномом 2-й степени, при записи в Ньютоновской форме для 3-х узлов

$$P_2(x)=f_0+f_{01}(x–x_0)+f_{012}(x–x_0)(x–x_1),$$

где разделенные разности

$$f_{01}=(f_0–f_1)/(x_0–x_1)=(f_1–f_0)/h,$$

$$f_{012}=(f_{01}–f_{02})/(x_1–x_2)=(f_0 2f_1+f_2)/2h^2$$

при шаге h между узлами.

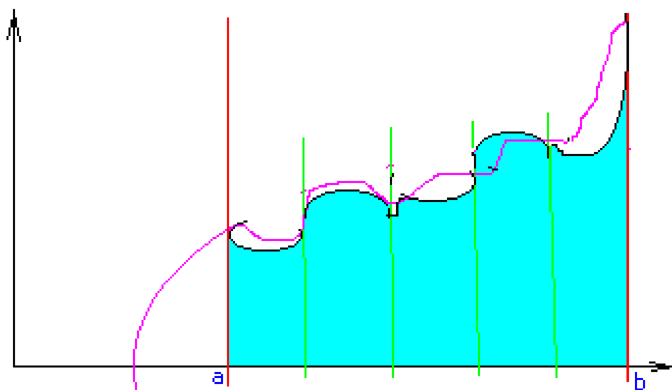
Заменяя $z=x–x_0$ или $x=z–x_0$, получим полином в виде

$$P_2(z)=f_0+(f_{01}–f_{012}h)z+f_{012}z^2$$

Интеграл от полинома

$$\int_{x_0}^{x_2} P_2(x)dx=\int_{0}^{2h} P_2(z)dz=(f_0+4f_1+f_2)h/3.$$

называют *квадратурной формулой Симпсона*.



Полная априорная погрешность оценивается величиной $R_{APRIOR}=-\frac{h^4}{180}\int_{x_0}^{x_n} f^{(4)}(x)dx$, если невелико значение четвертой производной, иначе можно получить большую погрешность, чем у методов второго порядка точности. Например, для функции

$$f(x) = -25x^4 + 45x^2 - 7$$

при $n=2$ для интервала $[-1, 1]$ метод трапеций дает точный результат, равный 4, а формула Симпсона неправильно определяет даже знак (–8/3).

4.1.2. Программная реализация методов численного интегрирования

```
//Нам не обойтись без класса векторов
#include "vector.h"

/*Параметризованный класс для вычисления интеграль-
ных сумм */
template <class YourOwnFloatType>
class Integrate
{
  vector<YourOwnFloatType> x, y; /*Узлы и значения
в узлах*/
 public:
  /*Конструктор, в качестве параметра принимающий
два вектора, содержащих соответственно узлы и зна-
чения в них */
  Integrate(vector<YourOwnFloatType> _x, vector
<YourOwnFloatType> _y): x(_x), y(_y)
  {
    /*Проверяем входные данные на корректность */
    if(x.getm()!=y.getm())
      throw xmsg("Количество узлов не совпадает с
количеством значений в них");
    if(x.getm()<2)
      throw xmsg("Слишком мало точек");
  }
  Integrate(Integrate &_): x(_.x), y(_.y) {}
/*Конструктор копирования */
  //Прототипы методов численного интегрирования
  //Метод прямоугольников
  YourOwnFloatType rectangle_method(),
  //Метод трапеций
                   trapecion_method(),
  /*Метод Симпсона - желательно четное число интер-
валов*/
                   simpson_method();
};
```

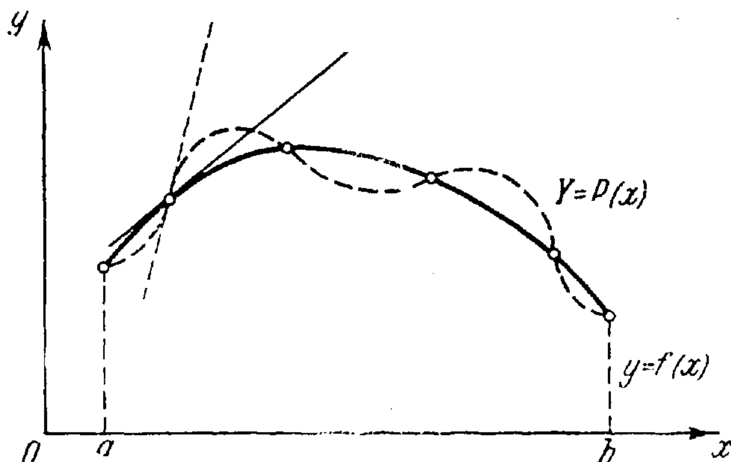
```
//Определения методов численного интегрирования
//Метод прямоугольников
template <class YourOwnFloatType>
YourOwnFloatType
Integrate<YourOwnFloatType>::rectangle_method()
{
  YourOwnFloatType sum=0;/* Интегральная сумма */
/*В связи с возможностью наличия во входных данных
неравноотстоящих узлов шаг интегрирования в этом и
последующих методах будем вычислять как разность
между текущим узлом и ближайшим к нему */
  for(int i=0;i<x.getm()-1;i++)
    sum+=(x[i+1]-x[i])*y[i];
  return sum;/* Возвращаем результат */
}


//Метод трапеций
template <class YourOwnFloatType>
YourOwnFloatType
Integrate<YourOwnFloatType>::trapecion_method()
{
  YourOwnFloatType sum=0;
/*Суммируем предполагаемые значения функции в меж-
доузлиях */
  for(int i=0;i<x.getm()-1;i++)
    sum+=(x[i+1]-x[i])*(y[i]+y[i+1])/2;
  return sum;
}


//Метод Симпсона
template <class YourOwnFloatType>
YourOwnFloatType
Integrate<YourOwnFloatType>::simpson_method()
{
  YourOwnFloatType sum=0;
  /*В этом методе на каждом шаге интегрирования ин-
тегральная сумма вычисляется на двух интервалах (по
трём точкам)*/
  for(int i=0;i<x.getm()-2;i+=2)
```

```
    sum+=(x[i+1]-x[i])*(y[i]+4*y[i+1]+y[i+2])/3;
  return sum;
}

/*Подставляя в шаблон конкретный тип, получаем дей-
ствительный вектор */
typedef vector<double> dvector;

#include <conio.h>

/*Тестирующая функция */
void main()
{
  double limits[2]={.7,1.3};/*Пределы интегрирова-
ния */
  /*Количество интервалов */
  #define POINTCOUNT 20
  /*Вектора, содержащие узлы и значения в них */
  dvector x=POINTCOUNT+1,y=POINTCOUNT+1;
  /*Шаг интегрирования */
  double step=(limits[1]-limits[0])/POINTCOUNT;
  for(int i=0;i<=POINTCOUNT;i++)
    x[i]=limits[0]+i*step,
    y[i]=1/sqrt(2*x[i]*x[i]+.3);
  /*Создаём объект, в качестве параметров используя
только что заполненные векторы */
  Integrate<double> test(x,y);
  /*Выводим на экран результаты интегрирования раз-
личными методами для сравнения и анализа */
  cout<<test.rectangle_method()<<endl
      <<test.trapecion_method()<<endl
      <<test.simpson_method()<<endl;
  getch();
}
```

4.2. Численное дифференцирование

Первое совершенно тривиальное решение, которое приходит в голову при необходимости вычислить производную табличной функции в той или иной точке (в узле или межузловом промежутке) – это выполнить аналитическую замену легко дифференцируемой функцией (напри-

мер, полиномом), продифференцировать полученную аппроксимирующую функцию и вычислить значение функции-производной при заданном значении аргумента. Но мы столкнемся при этом с рядом проблем – «волнистостью» аналитической замены из-за слишком высокой степени интерполяционного полинома, построенного по большому количеству узлов или по узлам со слишком большим шагом, влиянием погрешностей различного происхождения. Это может привести (и, как правило, приводит) к тому, что даже первая производная в заданной точке для исходной и аппроксимирующей функций будут отличаться очень сильно, вплоть до несовпадения их знаков.



Дифференцирование табличных функций – в принципе некорректная задача, так как погрешность вычисления производных может многократно превышать погрешность интерполяции самой функции – погрешность вычисления производной равна производной погрешности интерполяции. Поэтому, если есть возможность избежать численного дифференцирования табличных функций – это надо сделать. Если же выхода нет – применяйте диктуемые здравым смыслом приемы для уменьшения потенциальных погрешностей дифференцирования.

К таким приемам относятся методы фильтрации аппроксимирующей функции – удаления из нее высоких частот незначительной амплитуды, близкой к допустимой погрешности аппроксимации. К такому эффекту осреднения приводит использование метода наименьших квадратов со степенью аппроксимирующего полинома значительно ниже количества используемых узлов. При использовании в качестве базисных функций ортогональных полиномов Чебышева можно постепенно добавлять члены с более высокими степенями с вычислением для каждой степени величины остаточной дисперсии – когда она уменьшится до удовлетворительного значения или темп ее уменьшения станет малым, наращивание степени полинома следует остановить. Попутно можно вычислять производную в заданной точке и, если ее значения подвержены при наращивании степени полинома значительным изменениям, к получаемому результату надо относиться с недоверием.

Другой способ фильтрации высоких частот состоит в том, что полученную тем или иным способом аппроксимирующую функцию пропускают через инерционное звено – математически это означает использование аппроксимирующей функции в качестве правой части неоднородного линейного дифференциального уравнения и получения в результате решения уравнения функции, используемой для последующего дифференцирования. Если программа для решения дифференциальных уравнений снабжена удобным пользовательским интерфейсом, позволяющим плавно менять коэффициенты уравнения и выводить графики исходной функции, аппроксимирующей функции и функции решения дифуравнения, то можно визуально наблюдать результаты аппроксимации и фильтрации и оценивать допустимость производимых при этом искажений. В следующей главе мы рассмотрим методы численного решения дифференциальных уравнений с примером пользовательского интерфейса с выводом графиков функций для операционной

системы Windows и вы сможете воспользоваться ими для проверки возможности использования высказанных рекомендаций.

На этом мы завершим рассмотрение методов численного дифференцирования табличных функций – занятия малоэффективного в части доверия к получаемым результатам.

5. Введение в численные методы решения дифференциальных уравнений

5.1. Обыкновенные дифференциальные уравнения (общие сведения)

Обыкновенным дифференциальным уравнением для функции $f(t)$ называется уравнение вида

$$F(t, f(t), f^{(1)}(t), \ldots, f^{(n)}(t))=0,$$

где F – заданная функция для бесконечного или конечного интервала t. ***Порядок уравнения*** определяется порядком n старшей производной $f^{(n)}(t)$ в этом уравнении. Уравнение называют ***линейным***, если функция F линейно зависит от всех своих аргументов:

$$F(t)=f^{(n)}(t)+a_{n-1}f^{(n-1)}(t)+\ldots+a_1 f(t)+a_0,$$

где $a_0, a_1, \ldots, a_{n-1}$ – либо заданные постоянные коэффициенты, либо заданные функции t. Это уравнение можно рассматривать и в векторном варианте, когда $\boldsymbol{f}$ и $\boldsymbol{F}$ являются вектор-функциями и мы имеем дело не с одним уравнением, а с системой уравнений.

Для рассмотрения методов решения дифференциальных уравнений в принципе достаточно иметь метод решения системы уравнений первого порядка

$$\boldsymbol{f}^{(1)}(t)=\boldsymbol{F}(t, \boldsymbol{f}(t)),$$

где $\boldsymbol{f}$ и $\boldsymbol{F}$ – векторы с n координатами $F_1, F_2, \ldots, F_n, f_1, f_2, \ldots, f_n$, поскольку уравнение n-го порядка сводится к системе n уравнений первого порядка, а систему m уравнений n-го порядка можно свести к системе nm уравнений первого порядка ***методом замены переменных***:

$$f_i(t)=f^{(i-1)}(t),\ i=1, \ldots, n.$$

В случае линейности системы уравнений относительно $\boldsymbol{f}(t)$ она принимает вид

$$\boldsymbol{f}^{(1)}(t)+\mathbf{A}\boldsymbol{f}(t)=\boldsymbol{b}(t),$$

где **A** – заданная матрица размера nхn, $\boldsymbol{b}$ – заданная вектор-функция от t.

Если элементы матрицы **A** не зависят от t и $\boldsymbol{b}$=0, то мы приходим к ***линейной однородной системе с постоянными коэффициентами***:

$$\boldsymbol{f}^{(1)}(t)+\mathbf{A}\boldsymbol{f}(t)=0,$$

решение которой можно получить явно в виде разложения

$$\boldsymbol{f}(t)=\boldsymbol{c}(\mathbf{E}+\mathbf{A}t+\mathbf{A}^2t^2/2+\ldots),$$

где $\boldsymbol{c}$ – произвольный постоянный вектор с n координатами. Разложение в скобках есть экспонента с показателем **A**t и решение можно записать в компактной форме

$$\boldsymbol{f}(t)=e^{\mathbf{A}t}\boldsymbol{c}.$$

Так как общее решение системы зависит от n произвольных постоянных c_i, то для определения конкретного единственного решения необходимо задать n дополнительных условий, которые обычно представляют собой начальные условия вида $\boldsymbol{f}(0)=\boldsymbol{f}_0$ или граничные условия вида $\boldsymbol{f}(a)=\boldsymbol{f}_a$ и, например, решение для системы однородных уравнений первого порядка с постоянными коэффициентами для заданных начальных условий (***задача Коши***) будет

$$\boldsymbol{f}(t)=e^{\mathbf{A}t}\boldsymbol{f}_0.$$

Если дополнительные условия для функции или ее производной заданы не в одной точке диапазона, а в нескольких, то мы имеем дело с так называемой ***краевой задачей***, для которой ключевым является вопрос наличия и единственности решения.

В общем случае уравнения n-го порядка надо задать n условий для функции f и/или ее производных до (n–1)-го порядка, а для системы уравнений первого порядка надо задать p условий для функции в точке a и n–p условий в точке b.

5.2. Процессы как объект исследования и управления

В этом небольшом вступительном разделе мы кратко обсудим, откуда возникают дифференциальные уравнения

и кому необходимо их решение. Этот вопрос не возникает при подготовке специалистов-прикладников, например инженерного профиля, – изучение прикладной области неизбежно сопровождается математическим описанием динамики изучаемых в ней процессов и дифференциальные уравнения воспринимаются как естественный инструмент исследования. Иначе обстоит дело при подготовке специалистов по общей и прикладной математике, так сказать «чистых математиков» – изучение аналитических и численных методов решения систем алгебраических, трансцендентных и дифференциальных уравнений часто воспринимается просто как «упражнения для ума» – преподавание ведется «чистыми» математиками и у студента зачастую остаются весьма смутные представления о прикладном значении изучаемых математических курсов (это особенно заметно в подготовке математиков в педагогических институтах). Если школьные задачи все же включают содержательную часть типа наполняемых жидкостями емкостей или движущихся в разных направлениях автомобилей, то студент-математик чаще всего просто получает задание решить конкретную систему уравнений, не представляя зачем и кому это может понадобиться (кроме необходимости получить оценку). Поэтому мы сочли необходимым хотя бы кратко остановиться на одном из прикладных аспектов изучаемого курса.

Под ***процессом*** понимают изменение некоторой величины во времени и пространстве – это может быть изменение температуры металла, нагреваемого в проходной печи перед прокаткой, изменение химического состава вещества в химическом реакторе, изменение прибыли предприятия, изменение координат движущейся ракеты, изменение численности определенного вида животных в некотором регионе и т.п. Процесс начинается и протекает благодаря внешним воздействиям на реализующий его объект. Эти воздействия могут быть целенаправленными (***управляющими***),

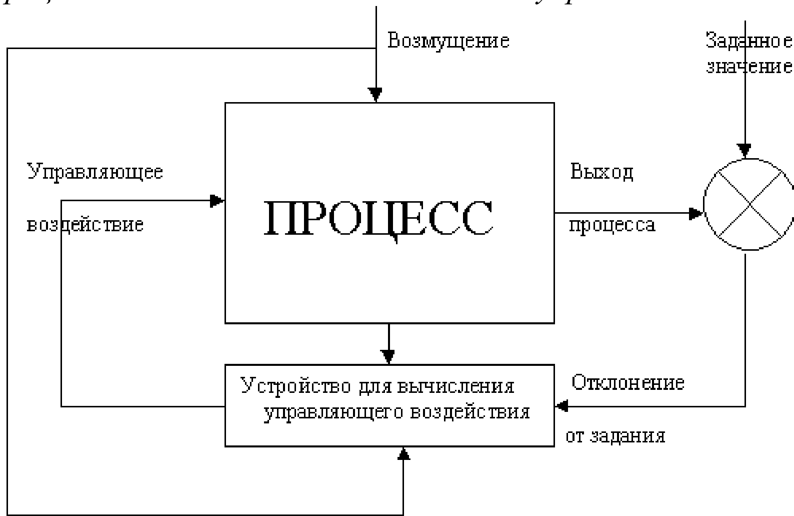
приложенными для обеспечения желаемого характера протекания процесса – изменение подачи топлива в горелки нагревательного устройства, изменение подачи реагентов в химический реактор, изменения объема используемых оборотных средств предприятия, изменение тяги ракетного двигателя или положения рулей самолета, изменение факторов, влияющих на прирост численности животных и т.д.

Другой тип воздействий на процесс называют ***возмущениями*** (нежелательными воздействиями) – колебания силы ветра, отклоняющего летательный аппарат от расчетного курса, изменения теплотворной способности используемого топлива в нагревательных устройствах, колебания спроса на продукцию предприятия и др.

По-видимому, одной из важнейших и труднейших областей интеллектуальной деятельности является управление процессами, которое состоит в определении и последующей реализации таких управляющих воздействий, которые обеспечили бы желательное или по возможности близкое к нему течение процесса, подвергающегося действию нежелательных и часто непредсказуемых и неконтролируемых возмущений. Но для определения необходимых управляющих воздействий необходимо знать, как процесс реагирует на эти воздействия – другими словами, для вычисления управлений необходимо знать их взаимосвязь с ***управляемыми (выходными) величинами*** и эта взаимосвязь должна быть выражена в виде математических соотношений, которые называют ***математической моделью процесса***. Таким образом, ***математическое моделирование процессов*** является неотъемлемой составной частью процесса управления.

На рисунке мы представили примерную общую схему системы управления процессом:
устройство управления содержит сведения о математической модели управляемого процесса, в каждый момент времени получает информацию о действующем возмуще-

нии и отклонении выходной величины от задания, вычисляет управляющее воздействие для уменьшения рассогласования между заданным и фактическим значением выхода процесса и подает его на вход объекта управления.



С общей задачей управления связаны три основные ее составляющие:

1) ***Задача идентификации*** математической модели управляемого процесса возникает, если математические соотношения, связывающие входы и выходы процесса неизвестны. Эти соотношения могут быть определены теоретически с использованием основных законов соответствующей прикладной области или экспериментально (что чаще всего и случается, по крайней мере, в технических системах). Рассмотрим некоторые примеры теоретического решения задачи идентификации.

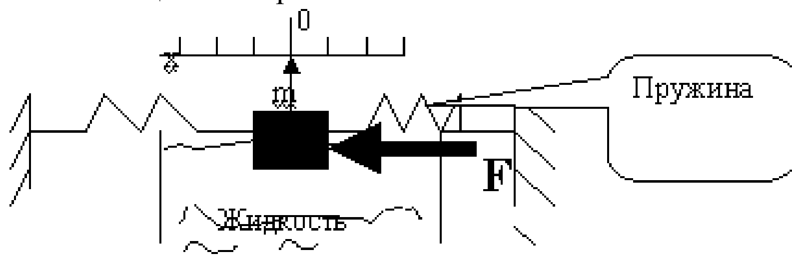
а) Пусть груз массой m перемещается в вязкой жидкости и центрируется в нейтральном положении (x=0) пружинами с линейной характеристикой. Управляющее воздействие – сила $F(t)$**,** приложенная к грузу, выходная – перемещение груза x. В соответствии с законами механики входное воздействие уравновешивается сопротивлением

деформируемых пружин $k_{п}x$, возникающей во время движения силой вязкого трения $r\dfrac{dx}{dt}$ (r – коэффициент вязкого трения) и силой инерции $m\dfrac{d^2x}{dt^2}$ (m – масса тела).

Уравнение процесса движения тела имеет вид:

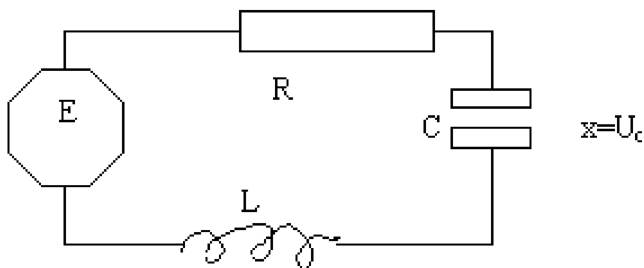
$$m\frac{d^2x}{dt^2}+r\frac{dx}{dt}+k_{п}x=F(t).$$

и представляет собой линейное дифференциальное уравнение с коэффициентами, которые могут быть постоянными или зависящими от времени.



б) Пусть к источнику электрического тока подключена электрическая цепь из включенных последовательно омического сопротивления, конденсатора и катушки индуктивности. Напряжение на конденсаторе будем считать выходом процесса, $E(t)$ – управлением, внутреннее сопротивление источника пренебрежимо мало. В соответствии с законами Кирхгофа сумма падений напряжений в контуре уравновесит ЭДС источников и уравнение процесса формально не отличается от случая механического движения в предыдущем примере:

$$L\frac{d^2x}{dt^2}+R\frac{dx}{dt}+(1/C)\text{x}=E(t).$$



в) Пусть математическая модель строится для определения высоты подъема ракеты при различных углах запуска к горизонту. Обозначим горизонтальную координату полета x, вертикальную y, в момент старта $x(0)=f(0)=0$. Будем считать, что полет происходит в одной вертикальной плоскости (при отсутствии бокового ветра).

Основой для составления математической модели является в этом случае второй закон Ньютона $d(m\boldsymbol{\upsilon})/dt=\boldsymbol{F}$**,** где $m(t)$ – масса ракеты, изменяющаяся во времени из-за расхода топлива, $\boldsymbol{\upsilon}$ – вектор скорости с горизонтальной $x^{(1)}(t)$ и вертикальной $y^{(1)}(t)$ составляющими, модулем

$$\upsilon(t)=[(x^{(1)}(t)^2+y^{(1)}(t)^2]^{1/2}$$

и углом к горизонту

$$\theta(t)=\mathrm{arctg}(y^{(1)}(t)/x^{(1)}(t)).$$

$\boldsymbol{F}$ – сумма действующих на ракету сил: силы тяги $\boldsymbol{T}(t)$, силы гравитации $m\boldsymbol{g}$ и силы сопротивления $c\rho s\upsilon^2$, пропорциональной плотности воздуха ρ, поперечному сечению ракеты s и квадрату скорости. Запишем уравнение движения ракеты через координаты x и y с учетом того, что силы тяги и сопротивления действуют вдоль оси ракеты и в сумме дают $T–c\rho s\upsilon^2$:

$$m^{(1)}x^{(1)}+mx^{(2)}=(T-c\rho s\upsilon^2)\cos\theta,$$
$$m^{(1)}y^{(1)}+my^{(2)}=(T-c\rho s\upsilon^2)\sin\theta-mg.$$

Или:

$$x^{(2)}=(1/m)(T-c\rho s\upsilon^2/2)\cos\theta-(m^{(1)}/m)x^{(1)},$$
$$y^{(2)}=(1/m)(T-c\rho s\upsilon^2/2)\sin\theta-(m^{(1)}/m)y^{(1)}-g.$$

Получили систему двух нелинейных дифференциаль-

ных уравнений второго порядка с начальными условиями $\upsilon(0)=0$, $\theta(0)=0$. В системе только один свободный параметр θ_0 и его изменение будет определять траекторию.

Если моделируется полет простого снаряда с заданной начальной скоростью, отсутствием тяги и изменения массы, то уравнения значительно упрощаются:

$$x^{(2)}=(1/2m)(-c\rho s\upsilon^2/2)\cos\theta,$$

$$y^{(2)}=(1/2m)(-c\rho s\upsilon^2/2)\sin\theta-g \text{ при } \upsilon(0)=\upsilon_0,\ \theta(0)=\theta_0.$$

г) Пусть моделируется задача о популяции двух видов по типу хищник-жертва. Скорости изменения численности жертв $x^{(1)}$ и хищников $y^{(1)}$ определяются их численностями x и y и вероятностью встречи между собой, то есть произведением xy:

$$dx/dt=ax+bxy,\ dy/dt=cy+dxy\ (a>0,\ b<0,\ c<0,\ d>0)$$

с некоторыми начальными численностями $x(0)=x_0$, $y(0)=y_0$. Эти уравнения известны под названием уравнений Лотки-Вольтерра.

2) ***Задача анализа.*** Если математическая модель процесса идентифицирована, она позволяет осуществить имитационные исследования процесса – задаваясь различными функциями управления, можно решением описывающих процесс дифференциальных уравнений получить функции, описывающие реакцию процесса на эти управления. Эта задача является вспомогательной для решения основной задачи – ***задачи управления***.

Задача анализа может рассматриваться в различных постановках.

Если заданы значения выходной величины процесса и всех ее производных до (n–1)-й включительно (n – порядок старшей производной в дифференциальном уравнении процесса) в некоторый момент времени, принимаемый начальным, при отсутствии управляющего воздействия – это ***задача определения свободного движения процесса***. Она рассматривается на полубесконечном интервале времени и имеет смысл, если хотя бы одна из n–1 производных

не равна нулю – в противном случае движение отсутствует и выход процесса постоянен. Эта же задача может решаться и при наличии управляющего воздействия – в этом случае решение дифуравнения представляет собой комбинацию из собственного движения и вынужденного под действием управления. Обе подгруппы задач с начальными условиями носят название ***задачи Коши для дифференциальных уравнений***.

Если процесс анализируется на ограниченном интервале времени и часть условий заданы на левой границе интервала, а часть на правой (общее количество дополнительных условий должно быть равно порядку уравнения), мы имеем дело с так называемой ***граничной задачей*** определения такого решения, которое удовлетворяет заданным условиям на обеих границах.

3) ***Задача управления.*** В задаче управления ставится задача найти управление, переводящее процесс из произвольного (известного) состояния, характеризуемого значениями выходной функции и ее производных в заданное конечное состояние, тоже заданное значением выходной функции и ее производных. Очевидно, что при избыточном количестве дополнительных условий для получения единственного решения необходимо определить, какое из множества возможных управлений надо считать лучшим, чем остальные. В этой постановке мы приходим к классу ***задач оптимального управления***. Например, при управлении движением ракеты можно в одних случаях считать наилучшей такую управляющую функцию, которая обеспечит вывод ракеты в заданную точку с наименьшим количеством израсходованного топлива (управление, оптимальное по затратам ресурсов); в других случаях наилучшим может считаться управление, выводящее ракету в заданную точку за кратчайшее время (управление, оптимальное по быстродействию).

Управляющее устройство и управляемый объект вместе представляют собой систему, описываемую общей систе-

мой дифференциальных уравнений. Параметры объекта (им соответствуют коэффициенты уравнения движения), как правило, не поддаются целенаправленному изменению в процессе управления – они определяются его конструкцией и могут сами дрейфовать в процессе старения объекта во время эксплуатации.

Но параметры другой составляющей системы – управляющего устройства – могут изменяться и это обстоятельство привело к разработке способа управления, основанного на изменении структуры системы в процессе ее движения – это так называемые ***системы с переменной структурой***. Проиллюстрировать это можно следующим примером. Пусть в электрическом контуре зарядки конденсатора с последовательно включенной индуктивностью и омическим сопротивлением у нас есть возможность изменять это сопротивление. Если мы сделаем его маленьким, процесс будет колебательным и его выход на заданный уровень будет длительным. Если сделать его настолько большим, что процесс зарядки станет апериодическим, то длительность процесса тоже будет большой. Но можно применить следующий прием – при большом отклонении напряжения на емкости сделать сопротивление маленьким, что обеспечит движение в автоколебательном режиме с большой скоростью, а при приближении к заданному уровню переключиться на большое сопротивление и небольшое оставшееся рассогласование отработать уже в апериодическом режиме – мы получим процесс зарядки с коротким временем перехода в заданное состояние.

Выводы. Итак, в математике процесс – это функция, описывающая изменение значений управляемых величин процесса в зависимости от управляющих и возмущающих воздействий. Задачи, связанные с решением дифференциальных уравнений, чаще всего возникают при математическом моделировании и исследовании на модели динамики процессов в различных объектах или системах. При этом

входное управляющее или возмущающее воздействие на систему задается в виде некоторой, в общем случае произвольной, функции времени в правой части неоднородного дифференциального уравнения. Вызванное этим воздействием изменение процесса на выходе системы называют ***вынужденным движением***.

Изменение выходной величины во времени при снятом входном воздействии зависит от конструктивных особенностей системы и начальных условий и носит название ***собственного движения***, определяемого как решение однородного уравнения без правой части. Для линейных систем результирующее движение представляет собой сумму собственного и вынужденного движений в предположении, что возмущающее воздействие приложено в момент времени t=0 и отсутствует в предшествующие моменты, а для t=0 заданы значения выходной координаты и ее производных.

5.3. Операционное исчисление и его применение к исследованию динамики линейных систем

5.3.1. Общие сведения

Мы уже отмечали, что наиболее просто вычисляются решения линейных дифференциальных уравнений; для линейного уравнения первого порядка это решение – экспонента, для уравнения n-го порядка – сумма экспонент. Этот тип дифференциальных уравнений и описываемых с их помощью линейных (или искусственно линеаризованных для упрощения) систем широко используется на практике при решении всех перечисленных классов задач – идентификации, анализа и управления.

Еще в конце 19-го века английский физик О. Хевисайд предложил способ вычислений, который назвал ***операционным***. Он рассматривал знак дифференцирования d/dt как оператор p и операцию дифференцирования записывал как $pf(t)$, n-я производная записывалась как $p^n f(t)$ – то есть опе-

ратор p n раз прилагался к функции $f(t)$. Оператор $1/p$ или p^{-1} представлял операцию интегрирования, так как приложение оператора p к p^{-1} снова давало $f(t)$: $pp^{-1}f(t)=f(t)$. При этом формулы с дифференциалами и интегралами приводились к алгебраической форме – Хевисайд называл это алгебраизацией задачи. До работ Карсона и Леви, давших методу фундаментальное математическое основание, обращаться с оператором p как с алгебраическим числом в вычислениях было небезопасно – в этом приеме скрывалось множество скрытых ловушек и только гениальная интуиция Хевисайда спасала его от ошибок в вычислениях.

Современное операционное исчисление рассматривает либо функции, связанные интегральным преобразованием Карсона

$$F_K(p)=p\int_0^{\infty} f(t)e^{-pt}dt,$$

либо преобразованием Лапласа:

$$F_L(p)=\int_0^{\infty} f(t)e^{-pt}dt.$$

В результате обоих преобразований функция $f(t)$ вещественного переменного t преобразуется в функцию $F(p)$ комплексного аргумента $p=\sigma+i\omega$, (σ и ω – вещественные, $i=\sqrt{-1}$).

Между функциями $F_K(p)$ и $F_L(p)$ очевидна взаимосвязь вида

$$pF_L(p)=F_K(p).$$

Преобразование Карсона удобно использовать в анализе электрических цепей, а мы будем использовать преобразование Лапласа (опуская индекс L в обозначении) в соответствии со сложившейся практикой в большинстве прикладных областей.

Приведенное функциональное соотношение записывают в виде $F(p)\subset f(t)$ или $f(t)\supset F(p)$ и говорят, что «$F(p)$ есть

изображение $f(t)$» или «$f(t)$ есть оригинал $F(p)$». Достаточным условием существования изображения функции $f(t)$ является требование ее кусочной непрерывности и существования таких положительных чисел M и σ, чтобы $|f(t)|<Me^{\sigma t}$.

Рассмотрим некоторые

5.3.1.1. Правила операционного исчисления

Сложение. Так как преобразование Лапласа – линейная операция, то *изображение суммы равно сумме изображений* $\sum_i F_i(p) \subset \sum_i f_i(t)$. Справедливо и обратное – оригинал суммы равен сумме оригиналов.

Дифференцирование f(t). Умножим на p обе части преобразования Лапласа для $f(t)$ и проинтегрируем по частям:

$$pF(p)=\int_0^\infty pe^{-pt} f(t)dt = [-e^{-pt} f(t)]_0^\infty + \int_0^\infty e^{-pt} f^{(1)}(t)dt,$$

то есть

$$pF(p)-f(0)\subset f^{(1)}(t).$$

Если $f(0)=0$, то

$$pF(p)\subset f^{(1)}(t).$$

Повторив n раз тот же прием, получим последовательным интегрированием по частям

$$p^n F(p)-p^{n-1}f(0)-p^{n-2}f^{(1)}(0)-\ldots-pf^{(n-2)}(0)-f^{(n-1)}(0)\subset f^{(n)}(t).$$

Если $f(0)=f^{(1)}(0)=\ldots=f^{(n-1)}(0)$, то $p^n F(p)\subset f^{(n)}(t)$.

Интегрирование f(t).

$$\frac{F(p)}{p} \subset \int_0^t f(t)dt \quad \text{и} \quad \frac{F(p)}{p^n} \subset \int_0^t dt \int_0^t dt \ldots \int_0^t f(t)dt,$$

то есть дифференцирование и интегрирование $f(t)$ соответствует соответственно умножению и делению изображения на p.

Теорема смещения. $F(p+\lambda)\subset e^{-\lambda t} f(t)$.

Теорема запаздывания. $e^{-\lambda p}F(p) \subset f(t-\lambda)\Upsilon(t-\lambda)$, где $\Upsilon(t-\lambda)$ – единичная ступенчатая функция. Если $f(t)=\Upsilon(t-\lambda)$, то ее изображение для запаздывания λ будет $\frac{1}{p}e^{-\lambda p} \subset \Upsilon(t-\lambda)$.

Теорема разложения Хевисайда. Теория разложения рациональных функций на простые дроби показывает, что если знаменатель $P(p)$ – полином m-й степени с только простыми корнями a_n, а числитель $Q(p)$ – любой полином более низкой степени, то имеет место тождество

$$\frac{Q(p)}{pP(p)} \equiv \frac{Q(0)}{pP(0)} + \sum \frac{Q(a_n)}{a_n P^{(1)}(a_n)} \cdot \frac{1}{p-a_n}$$

(суммирование по n от 1 до m).

Так как $\frac{1}{p-a_n} \subset e^{a_n t}$, то для функции $f(t)$, оригинал которой соответствует изображению $f(t) \supset \frac{Q(p)}{pP(p)}$, получим:

$$f(t)=\frac{Q(0)}{P(0)} + \sum_{n=1}^{m} \frac{Q(a_n)}{p_n P^{(1)}(a_n)} e^{a_n t} .$$

Если $f(t) \supset \frac{Q(p)}{P(p)}$, то только для простых корней

$$f(t)=\frac{Q(0)}{P(0)} + \sum_{n=1}^{m} \frac{Q(a_n)}{P^{(1)}(a_n)} e^{a_n t} .$$

Для случая кратных корней формула имеет более сложный вид:

$$\frac{Q(p)}{P(p)} \subset \sum_{k=1}^{r} \sum_{j=1}^{n_k} \frac{A_{kj}}{(n_k - j)!} t^{n_k - j} e^{a_k t} \text{ , где}$$

$$A_{kj} = \frac{1}{(j-1)!}\left[\frac{d^{j-1}}{dp^{j-1}} \frac{(p-a_k)^{n_k} Q(p)}{P(p)}\right]_{p=a_k} . (*)$$

r – количество разных корней, n_k – кратность k-го корня, k – текущий номер корня, j – текущая кратность.

Теорема свертывания (Бореля).

Даны две функции – $f_1(t)$ и $f_2(t)$ с изображениями $F_1(p)$ и $F_2(p)$ соответственно. Оригинал произведения изображений $F_1(p)$ и $F_2(p)$ будет равен интегралу произведения функций по параметру τ при смещении аргумента одной из них на величину τ.

Если $F_1(p)\subset f_1(t)$, $F_2(p)\subset f_2(t)$, то

$$F_1(p)F_2(p)\subset \int_0^t f_1(\tau)f_2(t-\tau)d\tau = \int_0^t f_1(t-\tau)f_2(\tau)d\tau .$$

Изображения типовых возмущающих (управляющих) функций.

Ступенчатая функция	$\Upsilon\supset 1/p$
Единичный импульс	$\Upsilon^{(1)}\supset 1$
Синусоида	$\sin\omega t\supset\omega/(p^2+\omega^2)$
Косинусоида	$\cos\omega t\supset p/(p^2+\omega^2)$
Экспонента	$\exp(-\omega t)\supset 1/(p+\omega)$

5.3.2. Решение линейных уравнений с постоянными коэффициентами

Уравнение имеет вид:

$$a_n f^{(n)}(t)+a_{n-1}f^{(n-1)}(t)+\ldots+a_1 f(t)+a_0=u(t).$$

Начальные условия: $f(0)=f_0$, $f^{(i)}(0)=f_i$, $i=1, 2, \ldots, n-1$.

Применим к функциям $f(t)$, $u(t)$ и их производным преобразование Лапласа, обозначив изображения $f(t)$ через $F(p)$, а $u(t)$ через $U(p)$, получим алгебраическое уравнение

$$F(p)[a_n p^n+a_{n-1}p^{n-1}+\ldots+a_1 p+a_0]=U(p)+f(0)[a_n p^{n-1}+$$
$$+a_{n-1}p^{n-2}+\ldots+a_1]+f^{(1)}(0)[a_n p^{n-2}+a_{n-1}p^{n-3}+\ldots+a_2]+\ldots+f^{(n-1)}(0)[a_0]$$

или в свернутом виде

$$F(p)\sum_{i=0}^{n} a_i p^i = U(p)+\sum_{i=0}^{n-1}\left[f^{(i)}(0)\sum_{j=1}^{n-i} a_{n-j+1}p^{n-j-i}\right].$$

Обозначим слагаемое, обусловленное ненулевыми начальными условиями

$$N(p)=\sum_{i=0}^{n-1}\left[f^{(i)}(0)\sum_{j=1}^{n-i}a_{n-j+1}p^{n-j-i}\right],$$

сумму $Q(p)=U(p)+N(p)$, а полином $\sum_{i=0}^{n}a_i p^i=P(p)$ назовём характеристическим полиномом.

Тогда изображение решения дифференциального уравнения

$$F(p)=\frac{Q(p)}{P(p)}.$$

Используя общую формулу разложения, получим решение уравнения:

$$\frac{Q(p)}{P(p)}\subset\sum_{k=1}^{r}\sum_{j=1}^{n_k}\frac{A_{kj}}{(n_k-j)!}t^{n_k-j}e^{a_k t}=f(t),$$

где $A_{kj}=\frac{1}{(j-1)!}\left[\frac{d^{j-1}}{dp^{j-1}}\frac{(p-a_k)^{n_k}Q(p)}{P(p)}\right]_{p=a_k}$.

Здесь мы встретились с одним из положительных качеств символического метода – начальные условия вводятся сразу, еще при постановке задачи и не вызывают никаких осложнений при ее решении. При решении однородного уравнения опускается управляющая функция, а если нас интересует только вынужденное движение, то задаются нулевые начальные условия.

Единственная трудность, которая нас поджидает, состоит в получении изображения управляющей функции $u(t)$, да еще и желательно в виде полинома или рациональной полиномиальной дроби. Если это одна из табулированных функций или ее преобразование Лапласа находится достаточно легко, то проблема решается. Если же это произвольная функция времени, то при компьютерном решении она будет задана последовательностью своих значений на конечном временном интервале. Общее решение ищется в этом случае следующим образом.

Заменим вначале нашу управляющую функцию единичным импульсом, имеющим изображение 1, и найдем общее решение системы в виде реакции на этот импульс. Обозначим это решение через $h^{(1)}(t)$.

Если предположить, что дискретные значения управляющей функции взяты через достаточно малые интервалы аргумента $\Delta\tau$, то эту функцию можно приближенно заменить последовательностью прямоугольных импульсов продолжительностью $\Delta\tau$ и амплитудой $u(i\Delta\tau)$ где i – порядковый номер значения, а реакцию системы на каждый из прямоугольных импульсов заменить реакцией на импульсную функцию $A_i \Upsilon^{(1)}$, где $A_i = u(i\Delta\tau)\Delta\tau$. Если реакция системы на $\Upsilon^{(1)}$ есть $h^{(1)}(t)$, то реакция на i-й прямоугольный импульс будет приближенно равна $h^{(1)}(t-i\Delta\tau)u(i\Delta\tau)\Delta\tau$, причем она будет существовать только для $t \geq i\Delta\tau$, так как реакция не может предшествовать воздействию. Реакция системы в момент времени $t=n\Delta\tau$ будет равна сумме реакций от каждого предшествующего импульса:

$$f(t) \approx \sum_{i=1}^{n} h^{(1)}(t - i\Delta\tau)u(i\Delta\tau)\Delta\tau \ .$$

При желании можно перейти к предельному выражению, считая что $\Delta\tau \rightarrow d\tau$ и прямоугольный импульс стремится к $\Upsilon^{(1)}$, величина $i\Delta\tau$ стремится к непрерывной величине τ, а сумма – к интегралу, дающему точное значение τ:

$$f(t) = \int_0^{\tau} h^{(1)}(t - \tau)u(\tau)d\tau \ .$$

Этот интеграл – свертка функций $h^{(1)}(t)$ и $u(t)$ или ***интеграл Дюамеля*** (мы уже упоминали о нем под названием теоремы свертывания Бореля); функции под интегралом можно поменять местами и представить интеграл в виде

$$f(t) = \int_0^{\tau} h^{(1)}(\tau)u(t - \tau)d\tau \ .$$

5.3.2.1. Передаточные функции линейных динамических систем

Благодаря операционному исчислению стало возможным перейти от классических методов количественного описания динамических свойств линейных систем в виде дифференциальных уравнений к более экономным средствам – передаточным функциям, временным и частотным характеристикам.

Передаточную функцию можно трактовать как комплексный коэффициент преобразования входного воздействия динамической системы в ее реакцию на выходе. Для формирования передаточной функции дифференциальное уравнение системы относительно функций вещественного переменного преобразуют в уравнение для функций комплексного переменного с использованием рассмотренных интегральных преобразований.

Если ввести понятие передаточной функции динамической системы в форме преобразования Лапласа $W(p)$ как отношение изображения выходной функции $F(p)$ к изображению входной (управляющей) $U(p)$ при нулевых начальных условиях $W(p)=F(p)/U(p)$, то оказывается, что изображение интеграла Дюамеля, являющегося изображением выходной функции, равно произведению передаточной функции на изображение управляющей функции.

Сложные динамические объекты редко идентифицируются сразу как единое целое – обычно получают их описание по частям или звеньям, а затем находят общее описание системы как описание соединения отдельных звеньев.

В этом случае передаточные функции оказываются удобным инструментом определения общего описания линейной системы. Вначале рассмотрим примеры передаточных функций некоторых элементарных звеньев.

Рассмотренные нами ранее уравнения 2-го порядка для механической колебательной системы или электрического колебательного контура, записанные в общем виде как

$$a_2\frac{d^2x}{dt^2}+a_1\frac{dx}{dt}+a_0x=ku,$$

приводят к передаточной функции вида

$$W(p)=\frac{k}{a_2p^2+a_1p+a_0}.$$

Если масса механической системы или индуктивность электрической цепи пренебрежимо малы, то приведенное уравнение вырождается в уравнение первого порядка $a_1\frac{dx}{dt}+a_0x=ku$ и его передаточная функция $W(p)=k/(a_1p+a_0)$.

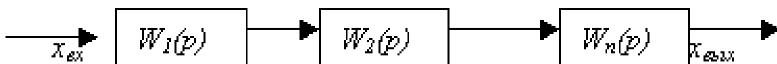
Уравнение интегрирующего звена $Tdx/dt=u$, его решение – $x=\int_0^t udt$, а передаточная функция $W(p)=1/Tp$.

Звено с постоянным запаздыванием (типа, например, конвейера) описывается уравнением $x(t)=u(t–\tau)$ и его передаточная функция $W(p)=e^{-p\tau}$.

В управляющих устройствах систем управления используют звенья, имеющие передаточные функции, обратные передаточным функциям колебательного и инерционного звеньев, например, форсирующее звено первого порядка с передаточной функцией $W(p)=Tp+1$, или форсирующее звено второго порядка с передаточной функцией $W(p)=a_2p^2+a_1p+1$, или дифференцирующее звено $W(p)=ap$.

Звенья в системе могут соединяться ***последовательно***, ***параллельно*** и ***встречно-параллельно (обратной связью)***.

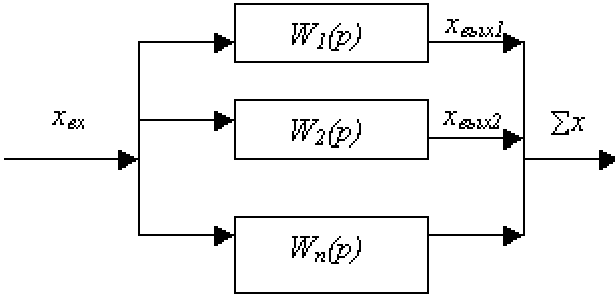
Передаточная функция последовательного соединения звеньев равна произведению входящих в цепочку звеньев:



$$W(p)=\prod_{i=1}^{n}W_i(p).$$

При параллельном соединении звеньев результирующая передаточная функция равна сумме передаточных

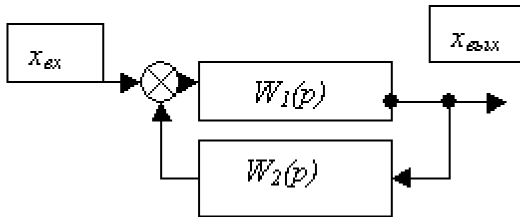
функций входящих в соединение звеньев:



$$W(p)=\sum_{i=1}^{n} W_i(p).$$

И, наконец, соединение двух звеньев по принципу обратной связи, когда

$$x_{вх1}=x_{вх}\pm x_{вых2}$$



может быть получена так:

$$x_{вых}=W_1(p)x_{вх1};\ x_{вых2}=W_2(p)x_{вых}$$

$$W(p)=\frac{W_1(p)}{1\mp W_1(p)W_2(p)}.$$

При этом верхний знак «–» относится к положительной обратной связи, а нижний «+» к отрицательной.

Обратную связь принято называть жесткой, если $W_2(p)$=const, то есть звено в обратной связи есть простой усилитель. Нежесткие обратные связи имеют ряд разновидностей – гибкие, изодромные, скоростные, запаздывающие и пр.

5.3.2.2. Частотные характеристики динамических систем

Эти характеристики – эффективный инструмент исследования свойств системы, позволяющий определить, как подавляются высокие или усиливаются резонансные частоты, как сдвигаются по фазе входные гармоники при прохождении через систему. Для получения частотной характеристики необходимо найти частное решение неоднородного уравнения системы при входном воздействии

$$u(t)=A_{\text{вх}}\,e^{j(\omega t+\varphi)},$$

где $u(t)$ – комплексная величина, которую на комплексной плоскости можно изобразить в виде вектора, образующего с вещественной осью угол $\omega t+\varphi$, линейно возрастающий в функции t; поэтому вектор вращается против часовой стрелки с угловой скоростью ω. Установившееся движение на выходе линейной передающей системы – гармонические колебания с частотой входных и частное решение уравнения системы ищется в форме входного воздействия, то есть $f(t)=A_{\text{вых}}(t)\,e^{j(\omega t+\varphi_{вых})}$ – выходной вектор вращается со скоростью входного, но имеет другой модуль и смещен относительно входного на угол $\varphi=\varphi_{вых}-\varphi_{вх}$. Подстановка указанного решения в уравнение системы и определение отношения $f(t)/u(t)=W(j\omega)=Q(j\omega)/P(j\omega)$ приводят к комплексному коэффициенту передачи или амплитудно-фазовой характеристике системы. Запись последней формулы в показательной форме $W(j\omega)=W(\omega)\,e^{j\varphi(\omega)}$, где $W(\omega)=A_{\text{вых}}/A_{\text{вх}}$ – амплитудно-частотная характеристика, $\varphi(\omega)=\varphi_{\text{вых}}-\varphi_{\text{вх}}$ – фазочастотная характеристика. Так как $W(j\omega)$ – дробно-рациональная функция, то амплитудно-частотная характеристика вычисляется как отношение модулей числителя и знаменателя и определяется просто: $W(\omega)=\dfrac{|Q(j\omega)|}{|P(j\omega)|}$, а фазовая характеристика как разность фазовых углов:

$$\varphi(\omega)=\operatorname{arctg}\frac{I_Q(\omega)}{R_Q(\omega)}-\operatorname{arctg}\frac{I_p(\omega)}{R_p(\omega)}.$$

Наличие базовых математических классов с набором основных операций (в частности класса полиномов и класса матриц) позволяет легко составить компактные программы для реализации приведенных вычислений, основные примеры которых приведены в программной реализации класса дифференциальных уравнений.

5.3.2.3. Фазовые портреты динамических систем

Еще один способ отображения динамических свойств систем состоит в построении годографов в фазовом пространстве. Под ***фазовым пространством*** понимают пространство, образованное совокупностью вектор-функций решения уравнения системы и его производных. Так как графическое отображение возможно только на плоскости (максимум – псевдотрехмерное графическое построение), то обычно ограничиваются парой фазовых координат, например, «функция – ее первая производная». Годографы в такой координатной системе позволяют наглядно видеть взаимосвязь между фазовыми координатами в исследуемом временном интервале – при программной реализации мы построим фазовый портрет системы и рассмотрим его зависимость от различных параметров ее математической модели.

5.3.2.4. Ограничения области применения символического метода

Мы кратко рассмотрели удобный и эффективный метод исследования динамических систем. Это аналитический метод – численные решения мы получаем только для корней характеристического полинома и при вычислении реакции на входное воздействие, заданное в виде дискретной последовательности значений; этот факт показывает, что даже при использовании точных аналитических методов

трудно избежать использования приближенных вычислений.

Следует помнить, что символический метод применим только к линейным системам (с сосредоточенными или распределенными параметрами, т.е. к обыкновенным дифференциальным уравнениям или уравнениям в частных производных), для которых справедлив ***принцип суперпозиции*** – реакция на сумму воздействий может быть вычислена как сумма реакций на отдельные воздействия.

Существенные трудности возникают и при анализе линейных (относительно функций) систем с переменными коэффициентами – приходится аппроксимировать зависимость коэффициентов от времени, например, полиномами, что приводит к появлению производных в уравнениях для изображений со старшей степенью, равной порядку аппроксимирующего полинома, то есть к дифференциальным уравнениям и тоже с переменными коэффициентами – первоначальное намерение алгебраизовать задачу остается неосуществленным.

Кроме того, реальные системы, как правило, нелинейны – мы показали это на очень упрощенных моделях из области экологии и баллистики. Наше обычное стремление привести модель системы к линейной структуре может привести к получению решения, но не для первоначальной задачи. Поэтому разработка эффективных методов для решения нелинейных дифференциальных уравнений будет всегда актуальной вычислительной задачей.

5.3.3. Программная реализация класса символического метода

При составлении алгоритмов символического исчисления мы будем широко использовать объекты таких ранее созданных классов, как полиномы и векторы. Большинство данных и обслуживающих методов сосредоточим в заголовочном файле laplas.h (лучше дать в нем только интер-

фейсную часть, а реализационную перенести в файл laplas.cpp и включить его в состав программного проекта – последний вариант даст возможность использовать однократную предварительную компиляцию заголовочного файла при отладке программы, если не использовать в этом файле процедуры инициализации данных). Если в дальнейшем предполагается создать из файла реализации библиотечный файл, то разделение на интерфейсную и реализационную части обязательно.

Как всегда, в начало программы собираем все понадобившиеся при ее составлении заголовочные файлы интерфейса с библиотеками:

```
//Файл difequat.h

#include "vector.h"
#include "equation.h"
#include <windows.h>
#include <alloc.h>
#include "wlaplasm.rh"
#include "matrix.h"


/*Для упрощения записи конкретизируем типы вектор-
ных и матричных элементов */
typedef vector<double> dvector;
typedef matrix<double> dmatrix;

/*Шаблон структуры для сведений о корнях характери-
стического полинома*/
  struct CA{
  complex ROOT;         //значение корня
  complex A;    //коэффициент для корня в оригинале
  int  J;               //кратность
  int  O;};             //кому кратен

  dmatrix SolMtr; /*Матрица решения и его производ-
ных*/
  dvector userfunc; /*Вектор для произвольного воз-
мущения*/
```

```
  dvector tc;            //Вектор аргумента решений
  dmatrix FreqChMtr; /*Матрица для частотных харак-
теристик*/
  double w;              //Частота среза
  double Cufz;        //Усиление на частоте среза
  /*Класс обыкновенных линейных дифференциальных
уравнений (ОЛДУ),  содержащий методы работы, бази-
рующиеся на операционном исчислении. */

class DifferentialEquation
{
  //приватные данные
  int rngl, rngr;  /*Порядок уравнения - левая и
правая часть*/
 dvector nv;             //Вектор  начальных условий
 double tend, tstep; /*Конечное время решения и шаг
дискретизации по времени*/
 int PointSolCnt;    //Размерность вектора решения

 //Полиномы левой, правой части и начальных условий
  cpolynom PL,PR,PN;

  /*Знаменатель и числитель изображения решения в
виде полиномов*/
  cpolynom PSOL,QSOL;

  CA *R; /*указатель на структуру со сведениями о
корнях*/
  long RootCount;   /*Количество корней общее - при
наличии правой части уравнения оно не совпадает с
его порядком слева*/
  long rcount; /*Количество различных корней - за
вычетом кратных*/
  int cod; double omega, alpha; /*Сведения о стан-
дартных возмущениях*/

  public:     //общедоступные члены класса

  /*Конструктор  должен получить коэффициенты урав-
нения, начальные условия, код стандартного возмуще-
ния и его параметры (частоту гармонической функции,
постоянную экспоненты), конечное время решения и
шаг*/
```

```
  DifferentialEquation(dvector CFNL, dvector CFNR,
dvector NU, int COD, double OMEGA, double ALPHA,
double TEND, double TSTEP);

  //Деструктор
  ~DifferentialEquation()
  {
    if(RootCount>0)
      farfree(R);
  }

/*Функция для вычисления корней характеристического
полинома*/
  void GetRoot();

  /*Функция формирования числителя изображения про-
изводных функции решения*/
  cpolynom Qsol(int der);/*аргумент - порядок про-
изводной*/

/*Функция, вычисляющая коэффициенты оригинала для
всех корней; результат помещается в массив структур
R*/

  void GetCoeffOrigin(void);


/*Функция, возвращающая значение выхода при задан-
ном значении аргумента*/
  double GetValue(double t);
  //Функция вычисления решения дифуравнения
  void Sol();
  //Функция для вычисления частотных характеристик
  void FreqChar();
};
/*Теперь определения подпрограмм.
  Нам понадобятся простые служебные функции - вы-
числения факториала и определения знака числа */
long fact(long x)
{
  long ret=1;
  for(long i=1;i<=x;i++)
```

```
    ret*=i;
  return ret;
}


inline long sign(double x)
{
  return (x>0)?1:((x<0)?-1:0);
}

//Определения методов класса.
//Конструктор по данным пользователя
DifferentialEquation::DifferentialEquation(dvector
CFNL,dvector CFNR,dvector NU, int COD, double
OMEGA,double ALPHA, double TEND, double TSTEP):
cod(COD), omega(OMEGA), alpha(ALPHA), tend(TEND),
tstep(TSTEP)
{
  //Определяем порядки уравнения слева и справа
  rngl=CFNL.getm()-1;rngr=CFNR.getm()-1;
  PL=cpolynom(rngl+1); PR=cpolynom(rngr+1);
  PN=cpolynom(rngl);
  /*Инициализация полиномов с преобразованием коэф-
фициентов в комплексные и реверсированием их после-
довательности - мы предполагаем, что вводятся коэф-
фициенты и начальные условия начиная со старшей
производной*/
  int i;
  for(i=0;i<(rngl+1);i++) PL[i]=complex(CFNL[i],0);
  PL=PL.reverse();
  for(i=0;i<(rngr+1);i++) PR[i]=complex(CFNR[i],0);
  PR=PR.reverse();

  //Создадим также вектор начальных условий
  nv=dvector(rngl);
  for(i=0;i<rngl;i++)
    nv[i]=NU[rngl-i-1];

  /*Формирование полинома начальных условий (числи-
теля изображения решения при свободном движении.
Если будет ненулевая правая часть, то ее изображе-
ние надо прибавить к результату вычисления PN)*/
```

```
  //p-полином первой степени p+0
  cpolynom p(2);
  p[0]=complex(0.0,0.0);p[1]=complex(1.0,0.0);
  cpolynom D=PL,tpn;
  for( i=0;i<rngl;i++)
  {
   D=D/p;//делим полином на p, понижая степень на 1
    /*прибавляем произведение значения производной
i-го порядка в начальной точке на полиномиальный
коэффициент*/
    for(int j=0;j<rngl;j++) D[i]*=nv[i];
    tpn=nv[i]*D;
    PN+=tpn;
  }

  /*Сформируем числитель QSOL и знаменатель PSOL
изображения решения - это полиномиальная дробь*/
  cpolynom rone;
  rone[0]=complex(1.0,0.0);/* веществ. полиномиаль-
ная единица */

  //Если возмущения нет
  if(cod==0) {PSOL=PL; QSOL=PN;}

  /*Если это единичный импульс или произвольная
функция*/
  if(cod==1||cod==6) {PSOL=PL; QSOL=PR+PN;}

  /*Если на входе ступенька - изображение 1/p
  if(cod==2) {QSOL=PN*p+PR; PSOL=PL*p;}

/*Если на входе синусоида - изображение ω/(p²+ω²).*/

  if(cod==3)
  {
    QSOL=PN*((p^2)+((omega*rone)^2))+PR*omega;
    PSOL=PL*((p^2)+((omega*rone)^2));
  }

  //Если косинусоида - изображение p/(p²+ω²).
  if(cod==4)
  {
    QSOL=PN*((p^2)+((omega*rone)^2))+(PR*p);
```

```
    PSOL=PL*((p^2)+((omega*rone)^2));
  }

  //Если экспонента - изображение 1/(p+ω).
  if(cod==5)
  {
    QSOL=PN*(p+(alpha*rone))+PR;
    PSOL=PL*(p+(alpha*rone));
  }

  PointSolCnt=floor(tend/tstep); /*Количество дис-
кретных точек*/
  /*Если объект конструируется для расчета частот-
ных характеристик*/
  if(cod==7)
  {
    FreqChar(); /*Вызываем метод расчета частотных
характеристик*/
    RootCount=0;//Нет необходимости вычислять корни
    return; //Досрочно покидаем конструктор
  }

  /*Определим теперь общее количество корней харак-
теристического полинома*/
  RootCount=PSOL.getm()-1;

  //После определения количества корней выделим па-
мять для хранения сведений о них*/
  R=(CA*)farcalloc(RootCount,sizeof(CA));
  memset(R,0,RootCount*sizeof(CA));

  //Конструируем хранители решений
  double j;

  tc=dvector(PointSolCnt);       //Вектор аргумента
  for(i=0,j=0.0;i<PointSolCnt;i++,j+=tstep)
    tc[i]=j;
  /*Матрица решений на RootCount строк - для функ-
ции и ее производных*/
  SolMtr=dmatrix(RootCount+1,PointSolCnt);
  /*Сразу занесем в матрицу решений начальные зна-
чения функции и ее производных*/
  if(RootCount>0)
```

```
  for(i=0;i<RootCount;i++)  SolMtr[i][0]=nv[i];

  GetRoot(); //Вычисляем корни x-го полинома

}//Конец конструктора


/*Функция для вычисления корней характеристического
полинома*/
void DifferentialEquation::GetRoot()
{
  int i,j;

  //ищем корни характеристического полинома
  cvector polyroot=newton(PSOL);

 //Ограничим точность вычисления корней вблизи нуля
  for(i=0;i<RootCount;i++)
  {
  if(fabs(real(polyroot[i]))<1e-7)
    polyroot[i]=complex(0,imag(polyroot[i]));
  if(fabs(imag(polyroot[i]))<1e-7)
    polyroot[i]=complex(real(polyroot[i]),0);
  }

  //Заносим в структуру значения корней
  for(i=0;i<RootCount;i++) R[i].ROOT=polyroot[i];
  /*определяем кратность каждого корня и заносим в
структуру; признаком -1 пометим корни, уже прове-
ренные на кратность. J=1 будет корень первой крат-
ности корню 0 и т.д.*/
  int repeat;
  rcount=RootCount;/* Вначале предполагаем, что все
корни различны*/
  for(i=0;i<RootCount;i++)
  {
    repeat=1;
    if(R[i].J!=-1) //корень еще не встречался
    {
      R[i].J=1;  //количество повторений i-го корня
      for(j=i+1;j<RootCount;j++)
        if((R[j].J!=-1)&&(R[i].ROOT==R[j].ROOT))
        {
```

```
          repeat++;
          rcount--;R[j].J=-1;R[j].O=i;
          /*устанавливаем признак того, что этот
корень уже учтён*/
        }
      R[i].J=repeat;//кратность корня
    }
  }
}


/*Подпрограмма определения числителя изображения
производной решения*/
cpolynom DifferentialEquation::Qsol(int der)
{
  cpolynom Q=QSOL, p(2),sum;
  //доформируем числитель изображения
  p[0]=complex(0,0);p[1]=complex(1.0,0.0);
  if(der>0 && cod!=6)
  {
    for(int i=0;i<der;i++)
      sum+=PN[RootCount-i-1]*p^(der-i-1);
    Q=Q*(p^der)-PSOL*sum;
  }
  return Q;
}


/*Функция, вычисляющая коэффициенты оригинала для
всех корней; результат помещается в массив структур
R*/

void DifferentialEquation::GetCoeffOrigin()
{
  complex ap, tp;/*Для значений числителя и знаме-
нателя*/
  /*при подстановке значения корня числитель может
оказаться полиномом или просто числом*/
  int qi=QSOL.getm();//Выясняем это
  int pi=PSOL.getm();
  //Если это число
  if(qi==1) ap=QSOL[0];/*Его и используем в каче-
стве значения числителя*/
```

```
  if(pi==1) tp=PSOL[0];
  if(RootCount==1)//Единственный некратный корень
  {
    /*Вычисляем значение производной знаменателя
изображения решения при подстановке в нее значения
единственного корня*/
    if(pi>1) tp=derive(PSOL,1)(R[0].ROOT);
    if(qi>1) ap=QSOL(R[0].ROOT);/*Если числитель -
полином, подставляем в него значение корня*/
    R[0].A=ap/tp;
    return;
  }

  //Если корней больше одного
  long i,j,k,l;
  cpolynom CH=QSOL,ZN=PSOL,RCH,RZN;
  cpolynom pw=2;
  cpolynom A;
  //dcomplex ch;

  for(k=0;k<RootCount;k++)
  {
    if(R[k].J==1)           //Для некратного корня
    { //Формируем полином  pw вида p+0
      cpolynom pw(2);
      pw[0]=-R[k].ROOT;pw[1]=complex(1.0,0.0);
      if(qi>1) ap=QSOL(R[k].ROOT);
      if(pi>1) tp=(PSOL/pw)(R[k].ROOT);
      R[k].A=ap/tp;
    }

    //Если корень кратный
    if(R[k].J>1)
    {
      cpolynom pw(2);pw[0]=-R[k].ROOT;
      pw[1]=complex(1.0,0.0);
      cpolynom pw1=(pw^R[k].J);
      //До кратности этого корня
      for(j=1;j<=R[k].J;)
      {
        if(j==1)
        {
          if(qi>1) ap=QSOL(R[k].ROOT);
```

```
          if(pi>1) tp=(PSOL/pw1)(R[k].ROOT);
          R[k].A=ap/tp;j++;
        }
        else
        {
          for(i=k+1;i<RootCount;i++)
          {
            //Если нашли кратный текущему
            if(R[i].O==k)
            {
              cpolynom tmp=PSOL/pw1;
              //Восстанавливаем знаменатель
              cpolynom chisl=QSOL, znamen=PSOL/pw1,
                       dchisl, dznamen, m1, m2;
              /*Дифференцируем дробь изображения
                решения j-1 раз*/
              for(l=1;l<=(j-1);l++)
              {
                dchisl=derive(chisl,1);
                dznamen=derive(znamen,1);
                m1=dchisl*znamen;
                m2=dznamen*chisl;
                chisl=m1-m2;
                znamen^=2;
              }
              R[i].A=chisl(R[k].ROOT)/
                     (znamen(R[k].ROOT)*fact(j-1));
              j++;
            }
          }
        }
      }
    }
  }
}


/*Подпрограмма, возвращающая значение функции реше-
ния ОЛДУ или ее производной заданного порядка при
заданном t. Она реализует формулу
```

$$\frac{Q(p)}{P(p)} \subset \sum_{k=1}^{r}\sum_{j=1}^{n_k}\frac{A_{kj}}{(n_k-j)!}t^{n_k-j}e^{a_k t}=f(t),$$

где

$$A_{kj} = \frac{1}{(j-1)!}\left[\frac{d^{j-1}}{dp^{j-1}}\frac{(p-a_k)^{n_k} Q(p)}{P(p)}\right]_{p=a_k}$$.*/

```
double DifferentialEquation::GetValue(double t)
{
  int k,i,index;
  double d;
  complex result(0,0);
  if(RootCount>0)
  {
    for(k=0;k<rcount;k++)
    {
      index=0;
      if(R[k].J==1)//Если корень простой
        result+=R[k].A*exp(R[k].ROOT*t);
      if(R[k].J>1)//Если корень кратный
      {
        result+=R[k].A*exp(R[k].ROOT*t);index++;
        for(i=k+1;i<rcount;i++)
        {
          if((R[i].O==k)&&(R[i].J==-1))
          {
            double dpw=R[k].J-index;
            result+=R[k].A*pow(t,dpw)*
              exp(R[k].ROOT*t)/fact(R[k].J-index);
            index++;
          }
        }
      }
    }
    d=real(result);
    return d;
  }
  return 0;
}


/*Подпрограмма решения дифуравнения - она просто
вызывает имеющиеся методы*/
void DifferentialEquation::Sol()
{
```

```
  int i,k;
  cpolynom Q=QSOL;
  if(cod==6)
  {
    GetCoeffOrigin();
    /*Вычисляем реакцию  на импульс и пишем в 0-ю
строку SolMtr*/
    for(k=1;k<PointSolCnt;k++)
      SolMtr[0][k]=GetValue(tc[k]);
    /*Реакцию на произвольное возмущение разместим
в 1-й строке SolMtr*/
    for(i=1;i<PointSolCnt;i++)
    {
      SolMtr[1][i]=0;
      for(k=0;k<i;k++)
        SolMtr[1][i]+=SolMtr[0][i-k]*
                            userfunc[i]*tstep;
    }
  }
  else
    for(i=0;i<RootCount;i++)
    {
      QSOL=Qsol(i);
      GetCoeffOrigin();
      for(k=1;k<PointSolCnt;k++)
        SolMtr[i][k]=GetValue(tc[k]);
      QSOL=Q;
    }
}


/* Подпрограмма вычисления частотных характеристик.
Вначале подбирает частоту среза - то есть определя-
ет верхнюю границу диапазона частот для исследова-
ния - по заданному пользователем коэффициенту сни-
жения коэффициента усиления по отношению к нулевой
частоте. Для этого частота меняется по простому ал-
горитму с переменным по величине и знаку шагом и
вычисляется  выходная амплитуда на этой частоте как
модуль передаточной функции после подстановки в нее
p=jω.*/
void DifferentialEquation::FreqChar()
{
```

```
  FreqChMtr=dmatrix(5,PointSolCnt);
  complex jone=complex(0.0,1.0);// мнимая единица
  //QSOL=PR;PSOL=PL;

  double KU0=fabs(real(PR[0]/PL[0])); /*Усиление на
нулевой частоте*/
  double KUZ=Cufz*KU0;         //Заданное усиление
  double wstep=1.0,    //Начальный шаг по частоте
      QW, //Значение числителя АФХ при частоте ω.
      PW, //Значение знаменателя АФХ при частоте ω.
      KU=KU0, /*Усиление при частоте ω - вначале
равно KU0.*/
      RQ, //Вещественная часть значения числителя
      IQ, //Мнимая часть значения числителя
      RP, //Вещественная часть значения знаменателя
      IP; //Мнимая часть значения знаменателя

  if(PR.getm()==1)
    QW=real(PR[0]); //Если справа - просто число
  //Подбираем частоту среза
  for(w=wstep;;w+=wstep)
  {
    if(PR.getm()>1) /*Если изображение правой части
- полином*/
      QW=sqrt(real(PR(w*jone))*real(PR(w*jone))+
              imag(PR(w*jone))*imag(PR(w*jone)));
    PW=sqrt(real(PL(w*jone))*real(PL(w*jone))+
            imag(PL(w*jone))*imag(PL(w*jone)));
    KU=QW/PW; //Усиление на текущей частоте
    if((KU< 1.1*KUZ) && (KU>0.9*KUZ))
      break;  //Если попали в диапазон
    if((KU<0.9*KUZ) && (wstep>0)) wstep=-0.1*wstep;
    if((KU>1.1*KUZ) && (wstep<0)) wstep=-0.1*wstep;
  }
  /*Заполним вектор частоты в матрице - мы разме-
стим его в 0-й строке*/
  wstep=w/PointSolCnt;
  int i;
  for(i=0;i<PointSolCnt;i++)
    FreqChMtr[0][i]=i*wstep;
  //Вычисляем  частотные характеристики
  if(PR.getm()==1) {RQ=real(PR[0]); IQ=0;}
  for(i=0;i<PointSolCnt;i++)
```

```
  {
    if(PR.getm()>1)
    {
      RQ=real(PR(jone*FreqChMtr[0][i]));
      IQ=imag(PR(jone*FreqChMtr[0][i]));
    }
    RP=real(PL(jone*FreqChMtr[0][i]));
    IP=imag(PL(jone*FreqChMtr[0][i]));
    FreqChMtr[1][i]=(RQ*RP+IQ*IP)/(RP*RP+IP*IP);
    //Вещественная частотная
    FreqChMtr[2][i]=(IQ*RP-RQ*IP)/(RP*RP+IP*IP);
    //Мнимая частотная
    FreqChMtr[3][i]=sqrt(FreqChMtr[1][i]*
                   FreqChMtr[1][i]+FreqChMtr[2][i]*
                   FreqChMtr[2][i]); //Амплитудная
    /*При расчете фазовой х-ки учтем скачек танген-
са при фазе π/2*/
    if(FreqChMtr[1][i]>0)
      FreqChMtr[4][i]=atan(FreqChMtr[2][i]/
                           FreqChMtr[1][i]);
    if(FreqChMtr[1][i]<=0)
      FreqChMtr[4][i]=-M_PI+atan(FreqChMtr[2][i]/
                                 FreqChMtr[1][i]);
    //Если скачек не учитывать, то можно проще:
    /*FreqChMtr[4][i]=atan(FreqChMtr[2][i]/
              FreqChMtr[1][i]);//Фазовая */
  }
}


/*Программа для решения ОЛДУ с постоянными коэффи-
циентами символическим методом - файл
difequat.cpp*/

#include "difequatm.h"
int sc;            //Количество точек решения
char Userfile[80]; /*Для имени файла с произвольным
возмущением*/

/*Для графического отображения функций решения объ-
явим массив структур типа POINT, в который потом
перенесем пересчитанные в координаты значения отоб-
ражаемой функции и значения ее аргумента */
```

```
POINT *pt;
int DeriveNumber=0;  //Порядок производной
/*Переменные для хранения размеров клиентской обла-
сти окна вывода */
int cxClient, cyClient;

int Cod,                        //Код возмущения
Rngl,Rngr;     //Порядок уравнения слева и справа
double Tend,Tstep,  /*Конечное время решения и шаг
по времени*/
  Omega,Alpha; //Параметры гармоник и экспоненты

char* buf;  //Буфер приема строк ввода пользователя
char*tmp; //Для приема указателя на слово от strtok
BOOL err;
//Прототипы функции окна и диалога
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK DlgProc(HWND hwnd, UINT uMsg,
WPARAM wParam, LPARAM lParam);
HINSTANCE hInst; /*Для сохранения идентификатора
приложения*/

//Главная функция программы
#pragma argsused
int PASCAL WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
  hInst=hInstance;
  WNDCLASSEX wc;

  wc.cbSize=sizeof(wc);
  wc.style=CS_HREDRAW|CS_VREDRAW;
  wc.lpfnWndProc=(WNDPROC)WndProc;
  wc.cbClsExtra=0;
  wc.cbWndExtra=0;
  wc.hInstance=hInstance;
  wc.hIcon=(HICON)LoadIcon(0,IDI_WINLOGO);
  wc.hCursor=(HCURSOR)LoadCursor(0,IDC_ARROW);
  wc.hbrBackground=(HBRUSH)GetStockObject
       (COLOR_BACKGROUND);//COLOR_BACKGROUND;
  wc.lpszMenuName="Laplas";
  wc.lpszClassName="Numerical Methods Class";
```

```
  wc.hIconSm=0;

  if(!RegisterClassEx(&wc))
  {
    MessageBox(0,"Не удалось зарегистрировать класс
окна",0, MB_OK|MB_ICONEXCLAMATION);
    return 0;
  }

  HWND hwnd=CreateWindowEx(WS_EX_CONTEXTHELP,
"Numerical Methods Class", "Численные методы. Сим-
волический метод решения ОЛДУ",
                          WS_OVERLAPPEDWINDOW,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          CW_USEDEFAULT,
                          0,
                          0,
                          hInstance,
                          0
                         );

  ShowWindow(hwnd,nCmdShow);
  UpdateWindow(hwnd);

  //userfile= new char[250];
  MSG msg;
  while(GetMessage(&msg,0,0,0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  delete[] pt;
  return 0;
}


/* Массив значений отображаемой функции и ее аргу-
мента нужно преобразовать в массив значений коорди-
нат в окне отображения. Для такого пересчета напи-
шем подпрограмму GetCoord - в качестве аргументов
ей понадобятся массивы значений функции и аргумен-
```

```
та. */

//Структура для смещений координатных осей
struct OFF{long x,y;}off;

void getCoord(dvector y, dvector x)
{
  long i;
  /*Определим наибольшее, наименьшее значение функ-
ции, диапазон изменения- хотя было  бы  неплохо по-
лучать эти значения из класса vector */
  double fymax, fymin, fxmax, fxmin;
  fymax=fymin=y[0];
  fxmax=fxmin=x[0];//Для начала пусть так
  for(i=0;i<sc;i++)
  {
    if(y[i]>fymax) fymax=y[i];
    if(y[i]<fymin) fymin=y[i];
    if(x[i]>fxmax) fxmax=x[i];
    if(x[i]<fxmin) fxmin=x[i];
  }
  //Отцентрируем все данные относительно минимумов
  for(i=0;i<sc;i++)
  {x[i]-=fxmin;y[i]-=fymin;}

  //Теперь заполним массив структур pt
  if(fxmax!=fxmin && fymax!=fymin)
  {
    for(i=0;i<sc;i++)
     pt[i].x=x[i]*(double(cxClient)/(fxmax-fxmin));

    for(i=0;i<sc;i++)
      pt[i].y=cyClient-long(y[i]*
              (double(cyClient)/(fymax-fymin)));
  }
  if((fxmax-fxmin)!=0)
    off.x=fxmin*((double)cxClient/(fxmax-fxmin));
  if((fymax-fymin)!=0)
    off.y=fymin*((double)cyClient/(fymax-fymin));
}


//Оконная процедура
```

```
HPEN hpen[6];
LRESULT CALLBACK WndProc(HWND hwnd, UINT uMsg,
WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  switch(uMsg)
  {
    case WM_SIZE:/* Определяем размеры клиентской
области окна*/
    {
      cxClient=LOWORD(lParam);
      cyClient=HIWORD(lParam);
      return 0;
    }
    case WM_CREATE:
    {
      /*Создаем перья для рисования случайными цве-
тами*/
      randomize();
      for(int i=0;i<6;i++)
        hpen[i]=CreatePen(PS_SOLID, 5,
        RGB(random(55), random(155),random(255)));
      return 0;
    }
  }
  if(uMsg==WM_COMMAND)
    switch(LOWORD(wParam))
    {
      case CM_DATE:
      {
        DialogBox(hInst, MAKEINTRESOURCE(DIALOG_1),
                  hwnd, (DLGPROC)DlgProc);
        return 0;
      }
      case CM_CLEAR:
      {
        InvalidateRect(hwnd,NULL,TRUE);
        return 0;
      }

      //Рисуем заданный переходный процесс
      case CM_GRAPH:
      {
```

```
        hdc=GetDC(hwnd);        //Получаем контекст
        if(Cod==6)
        {
          getCoord(SolMtr[1],tc);
          SelectObject(hdc,hpen[3]);
          PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
        }
        else
        {
          getCoord(SolMtr[0],tc); /*Вычисляем мас-
сив координат точек*/
          SelectObject(hdc,hpen[0]);
          PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);

          /*Вычисляем массив координат точек задан-
ной производной*/
          getCoord(SolMtr[DeriveNumber],tc);
          SelectObject(hdc,hpen[DeriveNumber]);
          PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
        }
        /* Можно нарисовать координатные оси - но
мы этого не делаем, ограничившись качественной кар-
тиной процессов; предоставляем это вам для самосто-
ятельной проработки. Один из вариантов:
         SelectObject(hdc,hpen[8]);
         //Вертикальная ось
         MoveToEx(hdc,fabs(off.x),0,NULL);
         LineTo(hdc,fabs(off.x),cyClient);
         //Горизонтальная ось
         MoveToEx(hdc,0,cyClient-fabs(off.y),NULL);
         LineTo(hdc,cxClient,cyClient-fabs(off.y));
        */
        ReleaseDC(hwnd,hdc);
        return 0;
      }

      //Вещественная частотная
      case CM_U:
      {
        hdc=GetDC(hwnd);//Получаем контекст
        SelectObject(hdc,hpen[1]);
        getCoord(FreqChMtr[1],FreqChMtr[0]);
        //Вычисляем массив координат точек
```

```
  PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
  ReleaseDC(hwnd,hdc);
  return 0;
}

//Мнимая частотная
case CM_V:
{
  hdc=GetDC(hwnd);//Получаем контекст
  SelectObject(hdc,hpen[2]);
  getCoord(FreqChMtr[2],FreqChMtr[0]);
  //Вычисляем массив координат точек
  PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
  ReleaseDC(hwnd,hdc);
  return 0;
}

//Аплитудная частотная характеристика
case CM_AMP:
{
  hdc=GetDC(hwnd);//Получаем контекст
  getCoord(FreqChMtr[3],FreqChMtr[0]);
  //Вычисляем массив координат точек
  SelectObject(hdc,hpen[3]);
  PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
  ReleaseDC(hwnd,hdc);
  return 0;
}

//Фазовая частотная характеристика
case CM_PHAZE:
{
  hdc=GetDC(hwnd);//Получаем контекст
  SelectObject(hdc,hpen[4]);
  getCoord(FreqChMtr[4],FreqChMtr[0]);
  //Вычисляем массив координат точек
  PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
  ReleaseDC(hwnd,hdc);
  return 0;
}

//Амплитудно-фазовая характеристика
case CM_AP:
```

```
      {
        hdc=GetDC(hwnd);//Получаем контекст
        SelectObject(hdc,hpen[5]);
        //Вычисляем массив координат точек
        getCoord(FreqChMtr[2],FreqChMtr[1]);
        PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);
        ReleaseDC(hwnd,hdc);
        return 0;
      }


      //Фазовый портрет функция-производная
      case CM_PORTRET:
      {
        hdc=GetDC(hwnd);//Получаем контекст
        SelectObject(hdc,hpen[4]);
        //Вычисляем массив координат точек
        getCoord(SolMtr[0],SolMtr[DeriveNumber]);
        PolyBezier(hdc,pt,3*(((sc-4)/3)+1)+1);

        ReleaseDC(hwnd,hdc);
        return 0;
      }
    }

  if(uMsg==WM_DESTROY)
  {
    PostQuitMessage(0);
    return 0;
  }
  return DefWindowProc(hwnd, uMsg, wParam, lParam);
}


//Функция обслуживания диалога с пользователем

#pragma argsused
LRESULT CALLBACK DlgProc(HWND hdlg, UINT uMsg,
WPARAM wParam, LPARAM lParam)
{
  int i;
  char buf[80];
```

```
  switch (uMsg)
  {
    case WM_INITDIALOG: return TRUE;

    case WM_COMMAND:
      switch(wParam)
      {
        case IDOK:/* Пользователь закончил ввод
данных о дифуравнении */
        {
       //Получаем и интерпретируем введенные строки
          //Конечное время
          SendDlgItemMessage(hdlg, IDC_EDIT12,
EM_GETLINE, (WPARAM)0, (LPARAM)(LPSTR)buf);
          Tend=atof(buf);
          if(Tend<=0)
          {
            MessageBox(0,"Не введено или ошибочно
конечное время", 0, MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }
          //Шаг по времени
          SendDlgItemMessage(hdlg, IDC_EDIT11,
EM_GETLINE, (WPARAM)0, (LPARAM)(LPSTR)buf);
          Tstep=atof(buf);
          if(Tstep<=0)
          {
            MessageBox(0,"Ошибочен шаг по времени
", 0, MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }

          if(Tstep>Tend)
          {
            MessageBox(0,"Ошибочен шаг по времени -
будет ноль шагов решения ",0,
MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }
          //Количество точек в решении
          sc=floor(Tend/Tstep);
          if(sc<=0)
          {
```

```
            MessageBox(0,"Ноль шагов решения -
ошибка в шаге или конечном времени", 0,
MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }
          /*После получения количества точек опре-
деляются размеры массивов для хранения информации*/
          pt=new POINT[sc];
          userfunc=dvector(sc);

          //Порядок справа
          Rngr=GetDlgItemInt(hdlg, IDC_EDIT4, &err,
FALSE);
          if(Rngr>Rngl || Rngr<0)
          {
            MessageBox(0,"Некорректен код порядка
справа",0, MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }
          dvector Cfnr(Rngr+1);

          //Символьный формат коэффициентов справа
          SendDlgItemMessage(hdlg, IDC_EDIT5,
EM_GETLINE, (WPARAM)0, (LPARAM)(LPSTR)buf);
          tmp=strtok(buf," ");
          for(i=0;tmp!=NULL;i++)
          {
            Cfnr[i]=atof(tmp);
            tmp=strtok(NULL," ");
          }
          if(i<(Rngr+1))
          {
            MessageBox(0,"Количество коэффициентов
должно быть порядок+1",0,
MB_OK|MB_ICONEXCLAMATION);
              return 0;
          }
          //Порядок уравнения слева
          Rngl=GetDlgItemInt(hdlg, IDC_EDIT1, &err,
FALSE);
          if(Rngl<=0)
          {
            MessageBox(0,"Нет смысла решать дифу-
```

```
равнение 0-го порядка",0,
MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }

          /*После определения порядка определяем
вектор коэффициентов */
          dvector Cfnl(Rngl+1);
          /*Получаем символьное представление коэф-
фициентов слева*/
          SendDlgItemMessage(hdlg, IDC_EDIT2,
EM_GETLINE, (WPARAM)0, (LPARAM)(LPSTR)buf);
          /*Выполняем разборку строки и заполнение
вектора Cfnl*/
          tmp=strtok(buf," ");
          for(i=0;tmp!=NULL;i++)
          {
            Cfnl[i]=atof(tmp);
            tmp=strtok(NULL," ");
          }
          if(i<(Rngl+1))
          {
            MessageBox(0,"Количество коэффициентов
должно быть порядок+1",0,
MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }

          //Начальные условия
          dvector Nu(Rngl);
          /*Получаем символьное представление
начальных условий*/
          SendDlgItemMessage(hdlg, IDC_EDIT3,
EM_GETLINE, (WPARAM)0,(LPARAM)(LPSTR)buf);
          tmp=strtok(buf," ");
          for(i=0;tmp!=NULL;i++)
          {Nu[i]=atof(tmp); tmp=strtok(NULL," ");}
          int ncount=i;
          //Частота гармонических возмущений
          SendDlgItemMessage(hdlg, IDC_EDIT7,
EM_GETLINE, (WPARAM)0,(LPARAM)(LPSTR)buf);
          Omega=atof(buf);
```

```
          /*Показатель степени экспоненциального
воздействия*/
          SendDlgItemMessage(hdlg, IDC_EDIT8,
EM_GETLINE, (WPARAM)0,(LPARAM)(LPSTR)buf);
          Alpha=atof(buf);

          //Код возмущения
          Cod=GetDlgItemInt(hdlg, IDC_EDIT6, &err,
FALSE);
          if(Cod<0 || Cod>7)
          {
            MessageBox(0,"Некорректный код возмуще-
ния. Задайте от 0 до 6",0,
MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }
          if((Cod==3 || Cod==4) && Omega==0)
          {
            MessageBox(0,"Для гармоник задайте ча-
стоту",0, MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }

          if(ncount<Rngl && Cod==0)
          {
            MessageBox(0,"Колич. нач. усл. должно
быть равно порядку слева \n или 0 (при ненулевом
коде возмущения)",0,  MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }

          if(ncount>0&&ncount<Rngl&&Cod>0&&Cod<7)
          {
            MessageBox(0,"Колич. нач. усл. должно
быть равно порядку слева \n или 0 (при ненулевом
коде возмущения)",0, MB_OK|MB_ICONEXCLAMATION);
            return 0;
          }

          //Коэффициент усиления на частоте среза
          SendDlgItemMessage(hdlg, IDC_EDIT10,
EM_GETLINE, (WPARAM)0,(LPARAM)(LPSTR)buf);
          Cufz=atof(buf);
```

```
          //Имя файла с произвольным возмущением
          SendDlgItemMessage(hdlg, IDC_EDIT9,
EM_GETLINE, (WPARAM)0,(LPARAM)(LPSTR)Userfile);
          if(Cod==6 && strlen(Userfile))
          {
            ifstream uf(Userfile);
            /*Если файл открыт - читаем из него в
вектор userfunc*/
            if(uf) {uf>>userfunc; uf.close();}
            else
            {
              MessageBox(0,"Неудача при открытии
файла с произвольным возмущением. \n Вычисляем по
имитации",0,MB_OK|MB_ICONEXCLAMATION);
              for(i=0;i<sc;i++)
                userfunc[i]=exp(-0.02*i*Tstep)*
                            cos(i*Tstep*0.05);
            }
          }
          /*Порядок производной, график которой
надо отрисовать рядом с функцией решения*/
          DeriveNumber=GetDlgItemInt(hdlg,
IDC_EDIT13, &err, FALSE);
          if(DeriveNumber<0&&DeriveNumber>(Rngl-1))
          {
            MessageBox(0,"Не корректен порядок
отображаемой производной",0,
MB_OK|MB_ICONEXCLAMATION);
            DeriveNumber=0;
          }

          /*После приема ввода пользователя и его
преобразования конструируем объект и вызываем под-
программу решения для заполнения матрицы решения */
          DifferentialEquation Dequ(Cfnl,Cfnr, Nu,
Cod, Omega, Alpha,Tend, Tstep);
          /*Вызываем метод решения, получая матрицу
решения*/
          if(Cod!=7)
          {
            Dequ.Sol();
            //Сохраним матрицу решений в файле
            ofstream os("solmtr.txt");os<<SolMtr;
```

```
          }
          if(Cod==7)
          {
            Dequ.FreqChar();
            //Сохраним матрицу  в файле
            ofstream os1("freqchar.txt");
            os1<<SolMtr;
          }
          EndDialog(hdlg,0);
          return TRUE;
        }//idOK
      }//switch wparam
  }//switch umsg
  return FALSE;
}


//Файл ресурсов

#include "wlaplasm.rh"

Laplas MENU
{
  MENUITEM "Ввод исходных данных", CM_DATE
  MENUITEM "График решения",CM_GRAPH
  POPUP "Частотные характеристики"
  {
    MENUITEM "Вещественная частотна\377", CM_U
    MENUITEM "Мнимая частотна\377", CM_V
    MENUITEM "Амплитудно-частотна\377", CM_AMP
    MENUITEM "Фазо-частотна\377", CM_PHAZE
    MENUITEM "Амплитудно-фазова\377", CM_AP
   }
   MENUITEM "Фазовый портрет",CM_PORTRET

   MENUITEM "Очистка области вывода", CM_CLEAR
}


DIALOG_1 DIALOG 0, 0, 240, 185
EXSTYLE WS_EX_DLGMODALFRAME | WS_EX_CONTEXTHELP
STYLE DS_MODALFRAME | DS_3DLOOK | DS_CONTEXTHELP |
WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU |
```

```
WS_MINIMIZEBOX | WS_MAXIMIZEBOX
CAPTION "ВВОД ИСХОДНЫХ ДАННЫХ"
FONT 10, "MS Sans Serif"
{
  CONTROL "OK", IDOK, "BUTTON", BS_PUSHBUTTON |
BS_CENTER | WS_CHILD | WS_VISIBLE | WS_TABSTOP, 56,
168, 50, 14
  CONTROL "Порядок левой части уравнения", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 7, 5,
113, 8
  CONTROL "Коэффициенты левой части уравнения", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 6, 15,
138, 8
  CONTROL "Начальные условия", -1, "static",
SS_LEFT | WS_CHILD | WS_VISIBLE, 5, 25, 70, 8
  CONTROL "Порядок правой части уравнения", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 36,
120, 8
  CONTROL "Коэффициенты правой части уравнения", -
1, "static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4,
48, 141, 8
  CONTROL "ЧИСЛА  РАЗДЕЛЯЙТЕ ТОЛЬКО ПРОБЕЛАМИ!", -
1, "static", SS_LEFT | WS_CHILD | WS_VISIBLE, 16,
156, 164, 8
  CONTROL "", IDC_EDIT1, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 4, 20, 9
  CONTROL "", IDC_EDIT2, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 15, 89, 9
  CONTROL "", IDC_EDIT3, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152 , 26, 87, 9
  CONTROL "0", IDC_EDIT4, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 36, 23, 9
  CONTROL "1.0", IDC_EDIT5, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 48, 89, 9
  CONTROL "Частота гармонических возмущений", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 72,
135, 8
  CONTROL "Показатель экспоненты", -1, "static",
```

```
SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 84, 142, 8
  CONTROL "Имя файла с произвольным возмущением", -
1, "static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4,
96, 140, 8
  CONTROL "0.05", IDC_EDIT7, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 72, 88, 9
  CONTROL "0.05", IDC_EDIT8, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 84, 88, 8
  CONTROL "", IDC_EDIT9, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 96, 89, 9
  CONTROL "Усиление на частоте среза", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 108,
100, 7
  CONTROL "0.01", IDC_EDIT10, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 108, 20, 9
  CONTROL "Шаг по аргументу", -1, "static", SS_LEFT
| WS_CHILD | WS_VISIBLE, 4, 120, 115, 8, 0
  CONTROL "", IDC_EDIT11, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 120, 20, 8, 0
  CONTROL "Максимальное значение  аргумента", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 132,
132, 8, 0
  CONTROL "", IDC_EDIT12, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 152, 132, 20, 8, 0
  CONTROL "Код возмущ.: 0-нет, 1-имп, 2-ступ, 3-
sin,4-cos,5-exp, 6-произв. ф-я ", -1, "static",
SS_SIMPLE | WS_CHILD | WS_VISIBLE, 4, 60, 212, 8, 0
  CONTROL "0", IDC_EDIT6, "edit", ES_LEFT |
ES_AUTOHSCROLL | WS_CHILD | WS_VISIBLE | WS_BORDER
| WS_TABSTOP, 220, 60, 20, 8, 0
  CONTROL "Порядок отображаемой производной", -1,
"static", SS_LEFT | WS_CHILD | WS_VISIBLE, 4, 144,
132, 8, 0
  CONTROL "", IDC_EDIT13, "edit", ES_LEFT |
WS_CHILD | WS_VISIBLE | WS_BORDER | WS_TABSTOP,
152, 144, 20, 8, 0
}
```

```
//Файл объявления констант для файла ресурсов
#define DIALOG_1        100
#define IDC_EDIT1       101
#define IDC_EDIT2       102
#define IDC_EDIT3       103
#define IDC_EDIT4       104
#define IDC_EDIT5       105
#define IDC_EDIT6       106
#define IDC_EDIT7       107
#define IDC_EDIT8       108
#define IDC_EDIT9       109
#define IDC_EDIT10      110
#define IDC_EDIT11      111
#define IDC_EDIT12      112
#define IDC_EDIT13      113

#define CM_DATE         201
#define CM_GRAPH        202
#define CM_AMP          203
#define CM_PHAZE        204
#define CM_AP           205
#define CM_CLEAR        206
#define CM_PORTRET      207
#define CM_U            208
#define CM_V            209
```

5.4. Конечно-разностные методы решения задачи Коши для линейных и нелинейных ОДУ

Рассмотренный в предыдущем разделе символический метод решения линейных ОДУ является аналитическим; к численному решению мы прибегали только при вычислении комплексных корней характеристического полинома. В настоящем разделе мы рассмотрим класс численных методов, пригодных для решения как линейных, так и нелинейных дифференциальных уравнений.

Приступая к численному решению задачи Коши для конкретного дифференциального уравнения (в общем слу-

чае – нелинейного относительно искомой функции), мы имеем скудный исходный материал – само уравнение n-го порядка и начальные условия в виде значений функции и n–1 ее производных в момент t=0 (или в более общем случае при $t=a$, если интервал t определен как $[a, b]$).

При этом мы не ставим перед собой задачу отыскания общего решения соответствующего класса уравнений, наша задача – получить частное конкретное решение: найти последовательность значений функции для конечной последовательности значений аргумента, удовлетворяющих заданному уравнению.

По-видимому, решение может быть найдено только в случае его единственности – для корректно поставленных задач. Кроме того, уравнение (или система уравнений) должно быть хорошо обусловленным – малые изменения его коэффициентов или начальных условий должны приводить к малым изменениям решения, в противном случае решение может оказаться неустойчивым. Соответствующее требование устойчивости предъявляется и к методу решения задачи – малым изменениям шага дискретизации по аргументу должны соответствовать малые изменения решения. Теория устойчивости алгоритмов – специальный раздел вычислительной математики, мы коснемся этих вопросов только «вскользь», чтобы предостеречь от возможных недоумений при получении неверных решений.

5.4.1. Одношаговые методы

Систему, заданную одним уравнением n-го порядка или системой таких уравнений, можно привести к системе уравнений первого порядка методом введения новых неизвестных функций – мы обсудили это во вводной части. Чтобы упростить изложение методов решения, ограничимся одним уравнением первого порядка

$$f^{(1)}(t)=F(t, f(t)),\ a \le t \le b,$$

с одной неизвестной функцией и начальным условием

$f(a)=f_a$, а затем распространим эти методы на систему таких уравнений.

Методы, которые мы будем рассматривать, относятся к классу конечно-разностных, поэтому на первом этапе решения разобьем участок $[a, b]$ на конечное число N узловых точек с равным расстоянием $h=(b–a)/N$ между ними, при этом $t_k=a+kh$, $k=1, 2, \ldots, N$. Через f_k обозначим значение точного решения в точке t_k, а через $f(t_k)$ – полученное приближенное значение. Нам хотелось бы, чтобы при $h\to 0$ погрешность нашего решения достаточно быстро уменьшалась.

Так как в нашем распоряжении в момент t_k (в самом начале – в момент t_a) есть значение $f(t_k)$, естественно попытаться выразить значение функции на следующем шаге через значение функции и ее аргумента на текущем:

$$f(t_{k+1})=f(t_k)+h\varphi(t_k, f(t_k))$$

с хорошей конструкцией функции φ.

5.4.1.1. Метод Эйлера

Простейшим решением является взять в качестве φ саму функцию F:

$$f(t_0)=t_a, f(t_{k+1})=f(t_k)+hF(t_k, f(t_k)),\ k=0, 1, \ldots, N\text{-}1.$$

Вывести этот метод просто. Из разложения $f(t)$ в окрестности точки t_k имеем

$$f_{k+1}=f_k+h\, f_k^{(1)}+(h^2/2)\, f_{zk}^{(2)}=f_k+hF(t_k, f_k)+(h^2/2)\, f_{zk}^{(2)}.$$

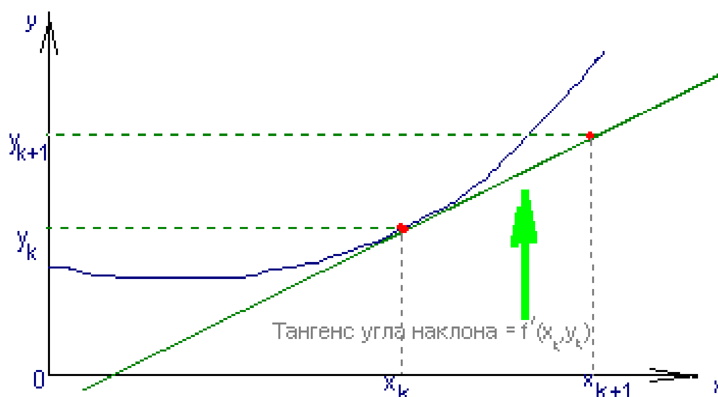
Отбрасывая последний член в предположении ограниченности второй производной и с учетом предполагаемой малости шага h, получаем приближение вида:

$$f_{k+1}\cong f_k+hF(t_k, f_k).$$

Геометрический смысл последнего выражения в аппроксимации решения на отрезке $[t_k, t_{k+1}]$ отрезком касательной к графику решения в точке t_k.

Метод Эйлера очень прост, но характеризуется первым порядком точности, то есть при стремлении шага дискретизации h к нулю приближенное решение будет сходиться к

точному с линейной скоростью по h. Такая медленная сходимость к точному решению препятствует широкому применению метода Эйлера и мы рассмотрим методы, погрешность которых уменьшается быстрее с уменьшением h.



5.4.1.2. Методы Рунге-Кутта 2-го порядка

Для уменьшения погрешности методов, использующих тейлоровское разложение искомого решения, необходимо использовать большее количество членов ряда. Однако при этом возникает необходимость аппроксимации производных от правых частей ОДУ. Основная идея методов Рунге-Кутта заключается в том, что производные аппроксимируются через значения функции $F(t_k, f(t_k))$ в точках на интервале $[t_k, t_k+h]$, которые выбираются из условия наибольшей близости алгоритма к ряду Тейлора.

В зависимости от старшей степени h, с которой учитываются члены ряда, построены схемы Рунге-Кутта разных порядков точности. Так, например, для 2-го порядка получено однопараметрическое семейство схем вида

$$f(t_k+h)=f(t_k+h[(1-L)F_k+LF(t_k+\text{г}h,\ t_k+\text{г}F_k h)]+O(h^3),\ (*)$$

где $0<L\leq 1$ – свободный параметр, $F_k=F(t_k, f(t_k))$, $\text{г}=1/2L$, $O(h^3)$ – остаточная погрешность.

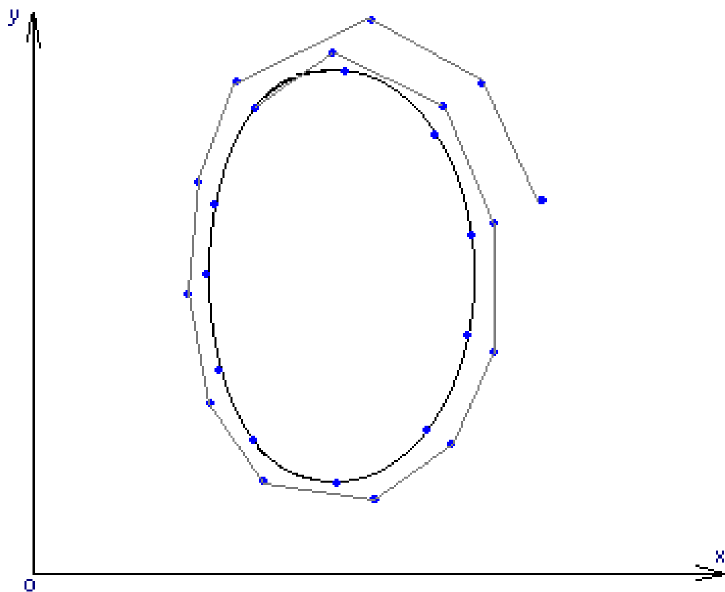
Для параметра L наиболее часто используют значения $L=1/2$ и $L=1$. В первом случае формула (*) приобретает вид

$$f(t_k+h)=f(t_k)+h[F_k+F(t_k+h, f(t_k))+hF_k)]/2$$

Вначале вычисляется приближенное решение ОДУ в точке t_k+h по формуле Эйлера

$$f_э=f_k+hF_k.$$

Затем определяется наклон интегральной кривой в найденной точке $F(t_k+h, f_э)$ и после нахождения среднего наклона на шаге h находится уточненное значение $f_{RK}=f(t_k+h)$.



Схемы подобного типа называют «прогноз – коррекция». С целью экономии памяти при программировании алгоритма, обобщенного на системы ОДУ, изменим его запись с учетом того, что $f_k=f_э-hF_k$:

$$f_i(t_k+h)=f_{iэ}+(1/2)h[F_i(t_k+h, f_{iэ})],$$

где i – номер решения для системы ОДУ.

Теперь не придется держать в памяти массив начальных значений f_{i0} – его можно забыть после вычисления значений эйлеровских приближений $f_{iэ}$, размещаемых на месте массива f_{i0}.

Во втором случае при L=1 от формулы (*) переходим к схеме

$$f(t_k+h)=f_k+hF(t_k+h/2, f_k+hF_k/2).$$

Здесь при прогнозе определяется методом Эйлера решение в точке ($t_k+h/2$):

$$f_{1/2}=f_k+hF_k/2,$$

а после вычисления наклона касательной к интегральной кривой в средней точке решение корректируется по этому наклону.

Таким образом, для методов Рунге-Кутта 2-го порядка функции $\varphi(t, f(t))$ имеют вид:

$$\varphi(t, f(t))=(1/2)[F(t, f(t))+F(t+h), f(t)+hF(t, f(t))]$$

или

$$\varphi(t, f(t))=F(t_k+h/2, f_k+hF_k/2).$$

5.4.1.3. Метод Рунге-Кутта 4-го порядка

Это наиболее популярный из методов Рунге-Кутта классический одношаговый метод четвертого порядка точности.

Для построения вычислительных схем методов Рунге-Кутта 4-го порядка в тейлоровском разложении искомого решения $f(t)$ учитываются члены со степенью шага h до 4-й включительно. После аппроксимации производных правой части ОДУ $F(t, f(t))$ получено семейство схем Рунге-Кутта 4-го порядка, из которых наиболее используемой в вычислительной практике является следующая

$$f(t_k+h)=f(t_k)+\frac{1}{6}(q_1+2q_2+2q_3+q_4)+O(h^5),$$

где

$$q_1=hF(t_k, f(t_k)),$$
$$q_2=hF(t_k+0.5h, f(t_k)+0.5hq_1),$$
$$q_3=hF(t_k+0.5h,\ t_k+0.5q_2),$$
$$q_4=hf(t_k+h,\ t_k+q_3).$$

Эта схема на каждом шаге требует вычисления правой части ОДУ в 4-х точках. Схема обобщается для систем ОДУ, записанных в форме Коши. Для удобства программной реализации формулы рекомендуется преобразовать к

виду:

$$f_i(t_k+h)=f_i(t_k)+\frac{1}{3}(q_{i1}+2q_{i2}+q_{i3}+q_{i4})+O(h^5),\text{ где}$$

$$q_{i1}=h_2F_i(t_k, f_i(t_k)),\ h_2=0.5h$$

$$q_{i2}=h_2F_i(t_k+h_2, f_i(t_k)+q_{i1}),$$

$$q_{i3}=hF_i(t_k+h_2, f_i(t_k)+q_{i2}),$$

$$q_{i4}=h_2F_i(t_k+h, f_i(t_k)+q_{i3}),$$

i=1, 2, ..., n – номер уравнения в системе ОДУ из n уравнений.

5.4.2. Многошаговые методы (методы Адамса)

Перед нами все та же задача Коши

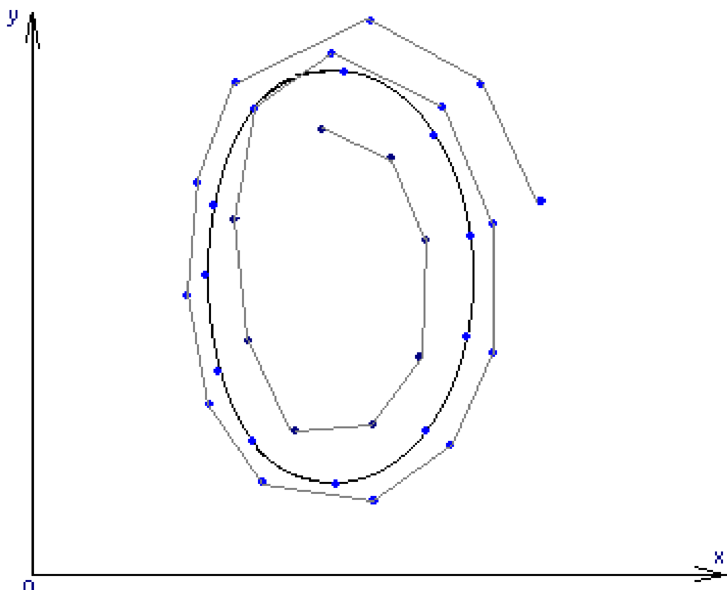
$$f^{(1)}(t)=F(t, f(t)),\ a\le t\le b,\ f(a)=f_a.$$

В одношаговых методах значение $f(t_{k+1})$ определялось только информацией в предыдущей точке t_k. Представляется возможным повысить точность решения, если использовать информацию в нескольких предыдущих точках при ее наличии. Так и поступают в методах, которые называются многошаговыми. С первого взгляда на постановку задачи становится очевидным, что в момент старта $t=t_a$ есть только одно начальное условие и, если мы собираемся работать с двумя, тремя или четырьмя предыдущими точками, то не видно, как получить вторую, кроме использования одношаговых методов. Так и поступают; «комплексный» алгоритм решения может выглядеть так:

на первом шаге одношаговым методом получают вторую точку, на втором получают третью с помощью двухшагового метода, на третьем – четвертую с помощью трехшагового метода и т.д., пока для основного метода, который предполагается использовать, не наберется достаточно предыдущих точек.

Другой вариант состоит в том, что весь стартовый набор точек получается с помощью одношагового метода, например, Рунге-Кутта четвертого порядка. Поскольку

многошаговые методы предполагаются более точными, для стартового одношагового метода используют обычно большее число промежуточных точек, т.е. работают с более коротким шагом.



Многошаговые алгоритмы можно создать так. Учитывая, что

$$f(t_{k+1})=f(t_k)+\int_{t_k}^{t_k+1} F\big(t, f(t)\big)dt,$$

можно численно проинтегрировать стоящую под знаком интеграла правую часть ОДУ. Если использовать метод прямоугольников (интерполяционный полином для интегрируемой функции – константа), получим обычный метод Эйлера. Если использовать 2 точки и интерполяционный полином первого порядка

$$p(x)=\frac{t-t_{k-1}}{h}F_k-\frac{t-t_k}{h}F_{k-1},$$

то интегрирование по методу трапеций от t_k до t_{k+1} даст

следующий алгоритм:

$$f(t_{k+1})=f(t_k)+0.5h(3F_k-F_{k-1}).$$

Аналогично для трех точек будем иметь квадратичный интерполирующий полином по данным (t_{k-2}, F_{k-2}), (t_{k-1}, F_{k-1}), (t_k, F_k) и интегрирование по методу Симпсона даст алгоритм:

$$f(t_{k+1})=f(t_k)+\frac{h}{12}(23F_k-16F_{k-1}+5F_{k-2}).$$

Для 4-х точек полином будет кубическим и его интегрирование даст:

$$f(t_{k+1})=f(t_k)+\frac{h}{24}(55F_k-59F_{k-1}+37F_{k-2}-9F_{k-3}).$$

В принципе мы могли бы продолжать так сколь угодно долго.

Приведенные алгоритмы носят название методов Адамса-Башфорта второго, третьего и четвертого порядков.

Формально мы можем при построении интерполяционного полинома помимо N уже просчитанных точек использовать и еще R будущих t_{k+1}, t_{k+2}; в простейшем случае набор

$$t_{k+1}, t_k, t_{k-1},\ldots, t_{k-N}.$$

При этом порождается класс так называемых методов Адамса-Моултона. В четырехшаговом варианте он оперирует с данными (t_{k+1}, F_{k+1}), (t_k, F_k), (t_{k-1}, F_{k-1}), (t_{k-2}, F_{k-2}) и его алгоритм:

$$f(t_{k+1})=f(t_k)+\frac{h}{24}(9F_{k+1}+19F_k-5F_{k-1}+F_{k-2}).$$

Нельзя, разумеется, вести расчет по отсутствующим данным, поэтому алгоритмы Адамса объединяют в последовательность алгоритмов Адамса-Башфорта и Адамса-Моултона, получая при этом так называемые методы прогноза и коррекции. Например, метод прогноза и коррекции четвертого порядка выглядит так: вначале прогнозируем по алгоритму Адамса-Башфорта с использованием «прошлых» точек

$$f(t_{k+1})=f(t_k)+\frac{h}{24}(55F_k-59F_{k-1}+37F_{k-2}-9F_{k-3}).$$

Затем по вычисляем приближенное значение правой части уравнения

$$F_{k+1}=F(t_{k+1}, f(t_{k+1})).$$

И, наконец, корректируем $f(t_{k+1})$ с использованием его же приближенного значения

$$f(t_{k+1})=f(t_k)+\frac{h}{24}(9F_{k+1}+19F_k-5F_{k-1}+F_{k-2}).$$

Наиболее эффективные из имеющихся компьютерных программ, позволяющих пользователю менять величину шага и порядок метода, основаны на методах Адамса высокого порядка (свыше 10). Опыт эксплуатации этих программ показывает, что различия в их реализации могут оказывать более существенное влияние на точность, чем различия во внутренних свойствах самих методов.

5.4.3. Проблема устойчивости

Одношаговые и многошаговые методы можно объединить в одну обобщенную запись:

$$f(t_{k+1})=\sum_{i=1}^{m}\alpha_i f(t_{k+1-i})+h\varphi(t_{k+1},t_k,...,t_{k+1-m};f(t_{k+1}),f(t_k),...,f(t_{k+1-m})).$$

Для одношаговых методов m=1. Если при этом α_1=1, и φ не зависит от t_{k+1} и $f(t_{k+1})$, то обобщенный алгоритм превращается в одношаговый метод. Если функция φ имеет вид

$$\varphi=\sum_{i=1}^{m}\beta_i F\left(t_{k+1-i}, f\left(t_{k+1-i}\right)\right),$$

то обобщенный алгоритм превращается в линейный многошаговый метод. Обобщенный алгоритм включает все используемые в настоящее время методы.

Описываемый обобщенный метод является *устойчивым*, если все нули полинома

$$\rho(\lambda)\equiv\lambda^m-\alpha_1\lambda^{m-1}-\ldots-\alpha_m$$

удовлетворяют условию $|\lambda_i|\le 1$, а любой нуль, такой что $|\lambda_i|=1$, является простым. Если в дополнение к этому $m-1$ нулей полинома таковы, что $|\lambda_i|<1$, то метод является *строго устойчивым.*

Любой метод, имеющий по крайней мере первый порядок точности, должен удовлетворять условию $\sum_{i=1}^{m}\alpha_i=1$ и, следовательно, единица должна быть нулем соответствующего полинома. В этом случае строго устойчивого метода полином будет иметь один нуль в точке 1, а все остальные – строго меньше 1. Так как методы Рунге-Кутта одношаговые, то для них $\rho(\lambda)=\lambda-1$. Его единственный корень равен 1, поэтому методы Рунге-Кутта всегда строго устойчивы. Для m-шагового метода Адамса $\rho(\lambda)=\lambda^m-\lambda^{m-1}$, так что остальные $m-1$ корней равны нулю и такие методы тоже строго устойчивы.

Рассмотренные результаты теории устойчивости относятся к устойчивости в пределе при $h\to 0$. Даже строго устойчивые методы могут вести себя неустойчиво, если h слишком велик. Попытка обойти эту трудность чрезмерным уменьшением h может привести к недопустимо большим затратам машинного времени, в частности для дифференциальных уравнений, которые называют *жесткими.*

Жесткость есть свойство задачи, а не численного метода решения. Проблемы обеспечения устойчивости для жестких систем выходят за рамки нашего вводного курса; скажем только, что общий подход к решению этой проблемы состоит в использовании неявных методов, то есть использующих, например, значения в точке (t_{k+1}, F_{k+1}). Например, использование формулы

$$f(t_{k+1})=f(t_k)+hF(t_{k+1}, f(t_{k+1})),$$

приводит к неявному методу, который называют *обратным методом Эйлера.*

Он напоминает по форме обычный метод Эйлера, но

использует информацию в точке (t_{k+1}, F_{k+1}) и поэтому является неявным.

5.4.4. Программная реализация численных методов решения задачи Коши

```
//файл включения для векторного и матричного типов
#include "matrix.h"
/*для удобства переопределим шаблонные типы векто-
ров и матриц для работы с числовыми объектами - ве-
щественными числами двойной точности */
typedef vector<double> dvector;
typedef matrix<double> dmatrix;

/*Класс дифференциальных уравнений */

class DEqu
{
  double t0, //начальное время
  t1, //конечное время
  step; //шаг по времени
  dvector x0; //вектор начальных условий
/*Будем рассматривать класс уравнений, которые уда-
ется разрешить относительно старшей производной. В
этом случае при представлении уравнения n-го поряд-
ка в виде системы уравнений первого порядка и реше-
нии этой системы нам нужна только одна функция -
для правой части последнего уравнения в системе,
вычисляющая его правую часть по уже вычисленным до
этого значениям младших производных и самой функ-
ции. Для первого уравнения правая часть - в нулевой
момент времени -это начальное значение первой про-
изводной, для второго уравнения - второй и т.д., а
правая часть для последнего уравнения может вклю-
чать зависимость от времени, значение функции и
всех младших производных. Для унификации ввода этой
функции в интерактивном режиме надо программировать
интерпретатор. Так как нам этого делать не хоте-
лось, то использовать настоящий класс сможет только
программирующий пользователь - ему придется запро-
граммировать функцию правой части и указать ее имя
при вызове соответствующего метода решения системы
```

```
уравнений. Прототип этой функции объявлен через
указатель на функцию, чтобы можно было использовать
в качестве аргумента другой функции */

dvector (*dxbydt)(double t, dvector x);

public:
  /*конструктор принимает в качестве параметров все
вышеперечисленное */
  DEqu(double T0, double Tmax, double Step,
  dvector X0, dvector (*Dxbydt)(double T, dvector
X)): t0(T0), t1(Tmax), step(Step), x0(X0),
dxbydt(Dxbydt) {  }

  //конструктор копирования
  DEqu(DEqu &x): t0(x.t0), t1(x.t1), step(x.step),
                 x0(x.x0), dxbydt(x.dxbydt)
  {
  }
/*Прототипы функций, реализующих методы Эйлера,
Рунге-Кутты второго и четвертого порядка точности и
Адамса. Функции возвращают матрицы с количеством
столбцов, равным порядку системы, то есть в нулевом
столбце - значения функции, а в последующих – зна-
чения производных соответствующего номеру столбца
порядка */

  dmatrix EilerSol();
  dmatrix RK2Sol();
  dmatrix RK4Sol();
  dmatrix Adams4Sol();

  /*набор функций для доступа и модификации пара-
метров системы*/
  double &TStart() { return t0; }//начальное время
  double &TEnd() { return t1; }//конечное время
  dvector &XStart() { return x0; }/*вектор началь-
ных условий*/
  double &StepByT() {return step;}//шаг по времени
};


//Метод Эйлера
```

```
dmatrix DEqu::EilerSol()
{
  /*определяем количество точек, которые нам необ-
ходимо вычислить*/
  long StepCount=(TEnd()-TStart())/StepByT();

/*создаём матрицу, количество строк в которой равно
количеству точек решения с учётом начальной (нуле-
вой) точки */
  dmatrix x(StepCount+1,1);

  x[0]=XStart();/*записываем в нулевую точку
начальные условия*/

  /*итеративно вычисляем все последующие точки ре-
шения*/
  for(long i=0;i<StepCount;i++)
    x[i+1]=x[i]+StepByT()*dxbydt(TStart()+
i*StepByT(), x[i]);
  return x;//возвращаем результирующую матрицу
}

/*Решение по методу Рунге-Кутта второго порядка
(метод трапеций, или метод Хойна)*/
dmatrix DEqu::RK2Sol()
{
  //определяем количество вычисляемых точек
  long StepCount=(TEnd()-TStart())/StepByT();

  //создаём матрицу для хранения решений
  dmatrix x(StepCount+1, 1);

  //заносим начальные условия в 0-й вектор матрицы
  x[0]=XStart();

  for(long i=0;i<StepCount;i++)
  {
    /*вычисляем вектор-коэффициенты следующей точки
решения*/
    dvector k1=dxbydt(TStart()+i*StepByT(),x[i]);
    dvector k2=dxbydt(TStart()+(i+1)*StepByT(),
                  x[i]+StepByT()*k1);
```

```
    //вычисляем вектор решения на следующем шаге
    x[i+1]=x[i]+(StepByT()/2)*(k1+k2);
  }
  return x;//возвращаем матрицу-результат
}


/*Решение по методу Рунге-Кутта 4-го порядка точно-
сти*/
dmatrix DEqu::RK4Sol()
{
  //количество точек решения
  long StepCount=(TEnd()-TStart())/StepByT();

  dmatrix x(StepCount+1, 1);//матрица решения

  x[0]=XStart();//начальные условия

  for(long i=0;i<StepCount;i++)
  {
    //вычисляем вектор-коэффициенты решения
    dvector k1=dxbydt(TStart()+i*StepByT(),x[i]);
    dvector k2=dxbydt(TStart()+(i+0.5)*StepByT(),
                      x[i]+(StepByT()/2)*k1);
    dvector k3=dxbydt(TStart()+(i+0.5)*StepByT(),
                      x[i]+(StepByT()/4)*(k1+k2));
    dvector k4=dxbydt(TStart()+(i+1)*StepByT(),
                      x[i]+StepByT()*(-k2+2*k3));

  x[i+1]=x[i]+(StepByT()/6)*(k1+4*k3+k4);
//следующая точка
  }
  return x;//возвращаем матрицу решений
}

//Метод Адамса
dmatrix DEqu::Adams4Sol()
{
  //определяем количество точек решения
  long StepCount=(TEnd()-TStart())/StepByT();

  dmatrix x(StepCount+1,1);/*матрица решения - мас-
сив векторов*/
```

```
/*преобразуем указатель на данный объект в указа-
тель на объект для решения системы дифференциальных
уравнений методом Рунге-Кутта четвёртого порядка и
получаем объект соответствующего типа вызовом кон-
структора копирования */
  DEqu obj=*this;

/*если количество точек решения таково, что данную
систему нельзя решить методом Адамса-Башфорта, воз-
вращаем решение по методу Рунге-Кутта*/
  if(StepCount<=3) return obj.RK2Sol();

/*В противном случае модифицируем параметры системы
(конечное время) так, чтобы по методу Рунге-Кутта
четвёртого порядка находились только первые четыре
решения */
  obj.TEnd()=TStart()+3*StepByT();

  dmatrix tempx=obj.RK4Sol();/*получаем первые че-
тыре точки*/

  for(long i=0;i<4;i++)/*и переписываем их в векто-
ры результирующей матрицы*/
    x[i]=tempx[i];

  /*начиная с пятой точки, работает собственно ме-
тод Адамса*/
  for(long i=3;i<StepCount;i++)
    x[i+1]=x[i]+(StepByT()/24)*
    (
    55*dxbydt(TStart()+i*StepByT(),x[i])-
    59*dxbydt(TStart()+(i-1)*StepByT(),x[i-1])+
    37*dxbydt(TStart()+(i-2)*StepByT(),x[i-2])-
    9*dxbydt(TStart()+(i-3)*StepByT(),x[i-3])
    );
  return x;//возвращаем результирующую матрицу
}


//Тестовые функции
dvector testfunc(double x, dvector y)
{
```

```
  //Решение уравнения Бесселя
  dvector dy=2;
  dy[0]=y[1];/*начальное условие для первой произ-
водной*/
  dy[1]=-y[0]-y[1]/x;


/*решение уравнений движения
  dvector dy=4;
  dy[0]=y[2];
  dy[1]=y[3];
  dy[2]=-y[0]/hypot(y[0],y[1]);
  dy[3]=-y[1]/hypot(y[0],y[1]);*/
  return dy;
}


void main()
{
  dvector StartCond=2;
  StartCond[0]=0.9384698;
  StartCond[1]=-0.2422685;
  /* dvector StartCond=4;
     StartCond[0]=0.5;
     StartCond[1]=0;
     StartCond[2]=0;
     StartCond[3]=1.63;*/

  DEqu testobj(0.5, 1.1, 0.1,/*0, 20.3, 0.1,*/
               StartCond, testfunc);
  dmatrix resEiler=testobj.EilerSol();
  dmatrix resRK2=testobj.RK2Sol();
  dmatrix resRK4=testobj.RK4Sol();
  dmatrix resAdams4=testobj.Adams4Sol();
  cout<<resEiler<<endl<<resRK2<<endl;
}
```

5.5. Двухточечные краевые задачи

5.5.1. Метод конечных разностей для линейных краевых (граничных) задач

Пусть задано обыкновенное дифференциальное уравнение порядка p или эквивалентная ему система из p уравнений первого порядка. Необходимо найти решение $f(t)$ уравнения на интервале $[t_0, t_n]$, удовлетворяющее граничным условиям функциональной связи между искомой функцией и ее производными

$$\varphi_k(f(t_k), f^{(1)}(t_k), \ldots, f^{(p-1)}(t_k)=0.$$

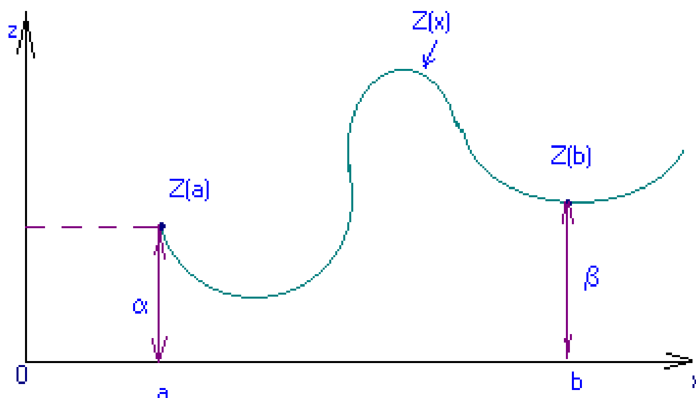
Для существования и единственности решения граничной задачи необходимо, чтобы количество граничных условий k совпадало с порядком заданного дифференциального уравнения. Следовательно, граничную задачу можно поставить для уравнения, порядок которого не ниже второго.

Граничные условия могут быть заданы в любой из точек интервала и, в частности, на границах интервала $[t_0, t_n]$ и в том числе на его границах. Для упрощения выкладок мы и будем рассматривать граничную задачу для линейного дифференциального уравнения второго порядка с переменными коэффициентами

$$f^{(2)}(t)+p(t)f^{(1)}(t)+q(t)f(t)=r(t).$$

Граничные условия зададим в виде линейной комбинации искомой функции $f(t)$ и ее первой производной $f^{(1)}(t)$ в точках t_0 и t_n:

$$g_1f(t_0)+g_2f^{(1)}(t_0)=g_3,\ g_4f(t_n)+g_5f^{(1)}(t_n)=g_6.$$



На первом этапе решения задачи составляются конечно-разностные схемы исходного дифференциального уравнения и граничных условий, в которых производные заменяются их приближенными выражениями через конечные разности.

Интервал $[t_0, t_n]$ разобьем на $n–1$ часть с шагом $h=(t_n–t_0)/n$.

Искомую функцию $f(t)$ в окрестности узла t_i представим в виде разложения в ряд Тейлора

$$f(t)=f(t_i)+(t–t_i)f^{(1)}(t_i)+(1/2)(t–t_i)^2f^{(2)}(t_i)+\ldots$$

В точках t_{i+1} и t_{i-1}, отстоящих на расстоянии h от узла t_i, пользуясь разложением, получим следующие значения функции $f(t)$:

$$f(t_{i+1})=f(t_i)+hf^{(1)}(t_i)+(1/2)h^2f^{(2)}(t_i)+\ldots$$

$$f(t_{i-1})=f(t_i)–hf^{(1)}(t_i)+(1/2)h^2f^{(2)}(t_i)+\ldots$$

Приближенные значения для первой и второй производных функции $f(t)$ в точке t_i получим путем вычитания и сложения левых и правых частей последних выражений:

$$f^{(1)}(t_i)=\frac{f(t_{i+1})-f(t_{i-1})}{2h}+O(h^2),$$

$$f^{(2)}(t_i)=\frac{f\left(t_{i+1}\right)-2f\left(t_i\right)+f\left(t_{i-1}\right)}{h^2}+O(h^2).$$

Заменяя производные в исходном уравнении их приближенными значениями, получим конечно-разностную

схему этого уравнения

$$\frac{f(t_{i+1})-2f(t_i)+f(t_{i-1})}{h^2}+p(t_i)\frac{f(t_{i+1})-f(t_{i-1})}{2h}+q(t_i)f(t_i)=r(t_i),$$

$$i=1, ..., n.$$

Аналогичным образом заменим граничные условия их конечно-разностными представлениями

$$g_1(t_0)f(t_0)+g_2\frac{f(t_1)-f(t_{-1})}{2h}=g_3(t_0),$$

$$g_4(t_n)f(t_n)+g_5\frac{f(t_{n+1})-f(t_{n-1})}{2h}=g_6(t_n).$$

Совокупность разностных схем уравнения и граничных условий представляет собой систему (n+3) линейных алгебраических уравнений относительно неизвестных $f(t_{-1})$, $f(t_0)$, ..., $f(t_{n+1})$. Матрицы этой системы приводятся к трехдиагональному виду, и тогда систему можно решить методом прогонки. Из первого конечно-разностного уравнения для граничных условий выразим неизвестное $f(t_{-1})$ и подставим его в конечно-разностное уравнение системы при i=0, в результате получим линейное соотношение

$f(t_0)=k_0-l_0f(t_1)$, где

$$k_0=\frac{h^2r(t_1)g_2+h(2-hp(t_1))g_3}{(h^2q(t_1)-2)g_2+h(2-hp(t_1))g_1},$$

$$l_0=\frac{2g_2}{(h^2q(t_1)-2)g_2+h(2-hp(t_1))g_1}$$

При i=1 уравнение $f(t_0)=k_0-l_0f(t_1)$ приводится к виду $f(t_1)=k_1-l_1f(t_2)$, а для произвольного значения i получим $f(t_i)=k_i-l_if(t_{i+1})$. Пограничные коэффициенты k_i, l_i вычисляются по рекуррентным формулам:

$$k_i=\frac{2h^2r(t_i)-(2-hp(t_i))k_{i-1}}{2h^2q(t_i)-4-(2-hp(t_i))l_{i-1}},$$

$$l_i=\frac{2+hp(t_i)}{2h^2q(t_i)-4-(2-hp(t_i))l_{i-1}},$$

полученным путем приведения разностного уравнения системы к форме $f(t_i)=k_i-l_if(t_{i+1})$ после замены значений $f(t_{i-1})=k_{i-1}-l_{i-1}f(t_i)$.

Из конечно-разностной схемы для второго граничного условия после подстановки выражений для $f(t_{n+1})$ и $f(t_{n-1})$ из $f(t_i)=k_i-l_if(t_{i+1})$ и $f(t_{i-1})=k_{i-1}-l_{i-1}f(t_i)$ найдем соотношение для вычисления неизвестного $f(t_n)$:

$$f(t_n)=\frac{2hg_6+(k_{n-1}-k_n/l_n)g_5}{2hg_4+(l_{n-1}-1/l_n)g_5}.$$

Затем в процессе обратного хода метода прогонки, полагая последовательно $i=n-1$, $n-2$, …, 0, по формуле $f(t_i)=k_i-l_if(t_{i+1})$ вычисляются значения остальных искомых величин $f(t_{n-1})$, $f(t_{n-2})$, …, $f(t_0)$. Так как погрешности аппроксимации первой и второй производной пропорциональны квадрату шага, этот вариант метода имеет второй порядок.

5.5.2. Метод стрельбы для граничных задач

Метод стрельбы мы рассмотрим на примере граничной задачи для *системы* обыкновенных дифференциальных уравнений второго порядка

$$\begin{cases}\dfrac{df_1(t)}{dt}=F_1\big(t,f_1(t),f_2(t)\big)\\[2mm]\dfrac{df_2(t)}{dt}=F_2\big(t,f_1(t),f_2(t)\big)\end{cases}.$$

Необходимо найти решение этой системы уравнений на интервале $[t_0, t_n]$, удовлетворяющее граничным условиям

$$\varphi(t_0,f_1(t_0),f_2(t_0))=0,$$
$$\psi(t_n,f_1(t_n),f_2(t_n))=0.$$

Сущность метода стрельбы заключается в сведении решения граничной задачи к многократному решению задачи Коши для заданной системы дифференциальных уравнений.

Предположим, что

$$f_1(t_0)=\varsigma,$$

где ς – произвольное число, которое можно задать, используя априорную информацию о характере решения $f_1(t)$. Подставим предполагаемое значение ς в первое граничное условие

$$\varphi(t_0, \varsigma, f_2(t_0))=0.$$

Теперь последнее соотношение является уравнением относительно одного неизвестного $f_2(t)$. Для простых функций φ удается записать аналитическое решение указанного уравнения. В общем случае решение этого уравнения находится одним из численных методов. В результате аналитического или численного решения получим

$$f_2(t_0, \varsigma)=\beta.$$

Таким образом, сформулирована задача Коши для исходной системы дифференциальных уравнений с начальными условиями $f_1(t_0)=\varsigma$ и $f_2(t_0, \varsigma)=\beta$ в точке t_0. Решение задачи Коши можно провести одним из известных методов решения для подобных задач и получить с необходимой точностью значения функций $f_1(t)$ и $f_2(t)$ в точке $t=t_n$: $f_1(t_n, \varsigma)$ и $f_2(t_n, \varsigma)$. Последние результаты поставим во второе граничное условие

$$\psi(t_n, f_1(t_n, \varsigma), f_2(t_n, \varsigma))=0,$$

которое не будет выполняться, так как число ς выбрано нами произвольным.

Последнее соотношение можно рассматривать как уравнение $\psi(\varsigma)=0$ относительно переменной ς. Значение $\varsigma=\varsigma^*$, являющееся корнем этого уравнения, удовлетворяет каждому граничному условию. Следовательно, решениями поставленной граничной задачи будут функции $f_1(t, \varsigma^*)$ и $f_2(t, \varsigma^*)$, определенные на интервале $[t_0, t_n]$ для задачи Коши.

Решение уравнения $\psi(\varsigma)=0$ требует большого объема вычислений, так на каждой итерации необходимо осуществлять интегрирование задачи Коши для исходной системы уравнений. Каждая итерация порождает для функций $f_1(t, \varsigma)$ и $f_2(t, \varsigma)$ траектории, которые в зависимости от

параметра ς либо приводят, либо не приводят к цели – обращению левой части уравнения ψ=0 в нуль. Отсюда и происходит название метода стрельбы. Самым экономичным методом решения таких уравнений является метод Ньютона, но он требует на каждой итерации вычисления не только левой части $\psi(\varsigma)$, но и ее производной по параметру ς, что вызывает необходимость интегрирования лишней пары дифференциальных уравнений. Поэтому для решения уравнения $\psi(t_n, f_1(t_n, \varsigma), f_2(t_n, \varsigma))=0$ чаще используют метод секущих, алгоритм которого в конкретном случае выглядит так:

$$\varsigma_{i+1}=\varsigma_i-\frac{\varsigma_i-\varsigma_{i-1}}{\psi(\varsigma_i)-\psi(\varsigma_{i-1})}\psi(\varsigma_i),$$ где i – номер итерации.

Погрешность решения зависит от выбранного шага и метода интегрирования задачи Коши, а также от погрешности вычисления ς^* методом секущих. Первая из этих погрешностей определит и полосу шума для функции $\psi(\varsigma)$, при попадании в которую вычисляемых значений $\psi(\varsigma_i)$ итерационный процесс уточнения ς приходится прекращать. Такую взаимосвязь погрешностей приходится учитывать, чтобы избежать лишних итераций метода секущих. Чтобы избежать рекурсивного обращения к подпрограмме метода секущих при решении уравнений $\varphi(f_2(t))=0$ и $\psi(\varsigma)=0$, соответствующий блок рекомендуется переписать дважды.

Алгоритм метода стрельбы существенно упрощается для линейных граничных задач, когда правые части системы ОДУ и граничные условия представляют собой линейную комбинацию функций $f_1(t)$ и $f_2(t)$:

$$\frac{df_1(t)}{dt}=f_2(t),$$

$$\frac{df_2(t)}{dt}=r(t)-p(t)f_2(t)-q(t)f_1(t),$$

$$g_1f_1(t_0)+g_2f_2(t_0)=g_3,$$

$$g_4f_1(t_n)+g_5f_2(t_n)=g_6.$$

Начальные условия соответствующей задачи Коши принимают вид:

$$f_1(t_0)=\varsigma, f_2(t_0)=(g_3-g_1\varsigma)/g_2.$$

Решения задачи Коши будут иметь линейную зависимость от ς, поэтому и левая часть уравнения $\psi(\varsigma)=0$ будет линейной функцией аргумента ς. Следовательно, значение ς_2, найденное по формуле секущих, при $i=1$ будет точным корнем уравнения. Таким образом, для линейной граничной задачи достаточно трижды решить задачу Коши.

5.5.3. Программная реализация класса граничных задач

```
#include <values.h>
#include <conio.h>
#include "matrix.h"

/*Для удобства определим пару типов для работы с
вещественными матрицами и векторами */
typedef vector<double> dvector;
typedef matrix<double> dmatrix;

/*Этот класс предназначен для тестирования методов
решения граничных задач */
class GrZadDemo
{
  double min, max, grusl[6];/*минимум, максимум и
массив граничных условий */
 public:
  GrZadDemo()/*Инициализация массива граничных
условий */
  {
    min=1,max=1.6,
    grusl[0]=3,grusl[1]=0.5,grusl[2]=2.075678,
    grusl[3]=2,grusl[4]=0.7,grusl[5]=0.5118773;
  }
  double getmin() {return min;}/*Получение минимума
*/
  double getmax() {return max;}/*Получение максиму-
ма */
```

```
  double getgrusl(int posit)/*получение граничных
условий по номеру */
  {
    if(posit>6||posit<1)
      throw xmsg("Неверные параметры");
    return grusl[posit-1];
  }
  double getrange() {return 2;}/*Порядок уравнения
*/
  /*Вектор значений коэффициентов в данной точке */
  dvector getvalue(double x)
  {
    if(x<min||x>max)
      //возвращает значение функции Бесселя
      throw xmsg("Выход за предела диапазона");
    dvector pqr=3;
    pqr[0]=1.0/x, pqr[1]=1, pqr[2]=0;
    return pqr;
  }
  //вычисляем производную
  dvector getderive(double x, dvector y)
  {
    dvector f=2, pqr=getvalue(x);
    f[0]=y[1];
    f[1]=pqr[2]-pqr[0]*y[1]-pqr[1]*y[0];
    return f;
  }
};


/*Тестирование метода конечных разностей */
void test1()
{
  GrZadDemo test;
  int r=test.getrange();/*порядок системы дифферен-
циальных уравнений */
  int n=100; //число разбиений
  int m=n+r; /*число уравнений в ортогонализируемой
системе*/
  dvector g(r*(r+1));//массив для граничных условий
  double min=test.getmin(), max=test.getmax();
  double h=(max-min)/n; //шаг
  //  ЗАПОЛНИМ МАССИВ ГРАНИЧНЫХ УСЛОВИЙ
```

```
  for(int i=0;i<r*(r+1);i++)
    g[i]=test.getgrusl(i+1);
  dmatrix mtr(m,m),right(m,1);
  // сформируем матрицу СЛАУ
  /* Заполним первую и последнюю строки основной
матрицы СЛАУ*/
  mtr[0][0]=-g[1]/(2*h);
  mtr[0][2]=-mtr[0][0];
  mtr[0][1]=g[0];
  right[0][0]=g[2];
  // всего строк: m :от 0 до m-1, m-1 - последняя
  mtr[m-1][m-2]=g[4]/(2*h);
  mtr[m-1][m-4]=-g[4]/(2*h);
  mtr[m-1][m-3]=g[3];
  right[m-1][0]=g[5];
  /*Далее матрицу заполняем со строки 1 по m-2
включительно*/
  for(long i=1;i<=m-2;i++)
  {
    dvector pqr=test.getvalue(min+(i-1)*h);
    mtr[i][i-1]=1/(h*h)-pqr[0]/(2*h);
    mtr[i][i]=pqr[1]-2/(h*h);
    mtr[i][i+1]=1/(h*h)+pqr[0]/(2*h);
    right[i][0]=pqr[2];
  }
  /*Для тестирования выведем полученную трёхдиаго-
нальную матрицу */
  cout<<"3-diagonal matrix\n"<<mtr<<endl;
  dmatrix result=SLAE_Orto(mtr,right);/*решим
СЛАУ*/
  cout<<"Solution\n";
  /*Решением являются значения функции Бесселя ну-
левого порядка */
  for(long i=0;i<m-1;i++)
    cout<<"X="<<min+i*h<<" Y="<<result[i][0]<<endl;
}


double psi(int l,double a,double h,dvector g,double
min,GrZadDemo test,int n)
{
/*интегратор по числу разбиений с выводом результа-
та */
```

```
  dvector y=2;
  y[0]=a;
  y[1]=(g[2]-g[0]*a)/g[1];
  double x=min;
  if(l==1)
    cout<<x<<" "<<a<<" "<<y[1]<<endl;
  for(int m=0;m<n;m++)
  {
    dvector f=test.getderive(x,y);
    y+=h*f;
    x+=h;
    if(l==1)
      cout<<x<<" "<<y[0]<<" "<<y[1]<<endl;
  }
  if(l==0)
    return g[3]*y[0]+g[4]*y[1]-g[5];
}


/*Тестирование метода стрельбы */
void test2()
{
  GrZadDemo test;
  int r=test.getrange();/*порядок системы дифферен-
циальных уравнений */
  int n=100; //число разбиений
  dvector g(r*(r+1));//массив для граничных условий
  double min=test.getmin(), max=test.getmax();
  double h=(max-min)/n; //шаг
  for(int i=0;i<r*(r+1);i++)
    g[i]=test.getgrusl(i+1);
  double t=psi(0,max,h,g,min,test,n);/*три раза ре-
шаем задачу Коши */
  double k=psi(0,min,h,g,min,test,n);
  psi(1,max-(max-min)/(t-k)*t,h,g,min,test,n);
}


void main()
{
  test1();//Метод конечных разностей
  getch();
  test2();//Метод стрельбы
```

```
  getch();
}
```

6. Численные методы решения систем нелинейных уравнений (СНУ) и поиска экстремумов функций

6.1. Общая постановка задачи

Мы объединили эти два класса задач в одном разделе потому, что они являются «сопряженными» – методы, используемые для вычисления корней, можно использовать для определения экстремумов и наоборот.

Система нелинейных уравнений с p неизвестными может быть записана в виде:

$$f_i(x_1, \ldots, x_p)=0,\ i=1, \ldots, p$$

в которых хотя бы одна функция f_i нелинейна.

С этой системой мы можем связать некоторую неотрицательную функцию $\Phi(x_1, \ldots, x_p)$ такую, что ее нулевой минимум является решением системы нелинейных уравнений, например:

$$\Phi(x_1, \ldots, x_p)=\sum_{i=1}^{p} f_i^2(x_1,\ldots,x_p)$$

и решать задачу поиска корней методами поиска экстремума.

С другой стороны, при заданной для поиска минимума функции многих переменных можно, дифференцируя ее по каждой из переменных и приравнивая частные производные нулю, составить систему уравнений, решение которой даст координаты искомого минимума:

$$\frac{\partial \Phi}{\partial x_i}=0,\ i=1, \ldots, p$$

и выполнить определение координат минимума решением системы уравнений.

Оба класса задач распадаются на подклассы, выделяемые, например, по размерности вектора аргументов – одномерный или многомерный поиск (корней или экстрему-

ма), или по наличию или отсутствию существенных погрешностей в определении значения функций – поиск (корней или экстремума) в условиях помех или при их отсутствии, или по выполнению предварительной локализации искомого объекта – определение экстремума унимодальной или многоэкстремальной функции в заданном диапазоне (с одним или многими корнями в интервале неопределенности в случае вычисления корней).

Мы начнем рассмотрение методов с наиболее простых задач – для функций одной переменной при отсутствии помех в определении значений унимодальной функции в заданных точках.

6.2. Решение нелинейных уравнений

6.2.1. Функция одной переменной при отсутствии помех

Даже этот простейший случай связан с рядом проблем и первая из них – возможность наличия комплексных корней нелинейного или трансцендентного уравнения $f(x)=0$. В разделе, посвященном работе с полиномами, мы привели метод Ньютона для вычисления корней полинома (в том числе комплексных) и не будем к этому возвращаться. Вычисление корней для полиномиальных уравнений связано с большими удобствами – прежде всего, возможностью вычисления всех корней без предварительной процедуры их отделения (локализации корня в некотором диапазоне значений). Эта возможность обеспечивается процедурой последовательного понижения степени полинома делением его на полином первого порядка ($x–x^*$), где x^* – значение уже вычисленного корня. Такое удаление из вычислительной процедуры уже вычисленных корней оказывается невыполнимым для других классов нелинейных функций и мы можем «натыкаться» на уже вычисленный корень мно-

гократно, если не выполним предварительного определения достаточно узких диапазонов размещения всех корней.

К сожалению, кроме табулирования функции и фиксации границ смены знаков или построения графиков функций современная теория ничего не предлагает. Отделение корней – это еще искусство, так как общее количество подлежащих локализации корней, скорее всего, неизвестно. Существенное упрощение задачи для полиномиальных уравнений подсказывает идею полиномиальной аппроксимации нелинейной функции с последующим вычислением корней, но мы не встречали в литературе подобных рекомендаций и не апробировали идею сами – предполагается, что неизбежные при аппроксимации погрешности не позволят получить достоверный результат.

В тех случаях, когда корни вещественны и осуществлено их предварительное отделение в конечном интервале значений (интервале неопределенности), используют описанные ниже методы.

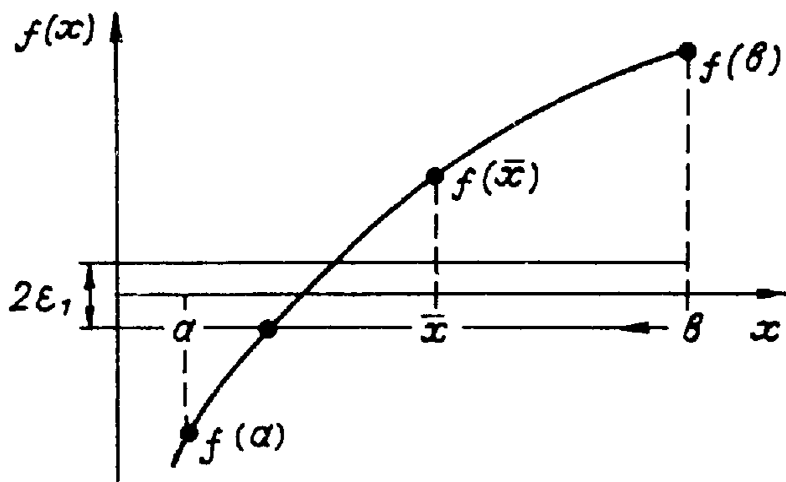
6.2.1.1. Методы последовательного сокращения интервала неопределенности

Все эти методы основаны на том или ином выборе точки (очередного кандидата на уточненное значение корня) внутри текущего интервала неопределенности и только этим методом выбора точки и отличаются друг от друга. После выбора точки определяется значение функции в этой точке и выясняется, на какой из границ интервала неопределенности значение функции совпадает по знаку со значением во внутренней точке – эта граница и переносится на место выбранной внутренней точки, приводя к уменьшению ширины интервала неопределенности. Процесс выбора внутренней точки продолжается до достижения шириной интервала заданной погрешности.

Трудности могут возникнуть для корней, которые образуются касанием функцией оси абсцисс без ее пересечения

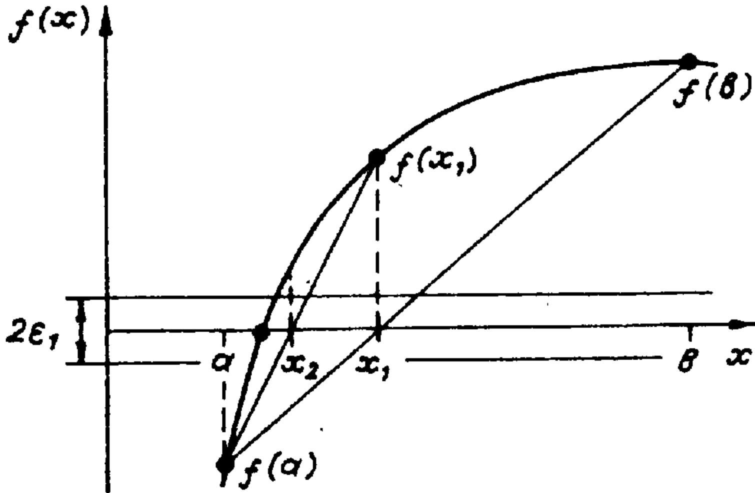
– этот вариант отслеживается либо параллельным контролем значений функции и фиксацией значения корня при входе значения функции в зону, близкую к машинному нулю, либо дополнительным поиском нулевого минимума квадрата функции в том же начальном интервале. Теперь об алгоритмах выбора внутренней точки. Помимо уже разобранного в разделе полиномов метода Ньютона, известны следующие:

Метод дихотомии (половинного деления или метод Больцано) предполагает выбор внутренней точки посредине текущего интервала неопределенности. Его эффективность в среднем трудно оспорить другими предложениями, по крайней мере, при известной начальной ширине интервала и заданной конечной ширине всегда известно, сколько всего потребуется итераций до завершения поиска – метод уменьшает длину интервала вдвое на каждом шаге.



Метод хорд – точка внутри интервала выбирается на пересечении с осью абсцисс линии, соединяющей два крайних (положительное и отрицательное) значения функции. Координаты этой точки вычисляются легко из простых геометрических соображений на подобных треуголь-

никах. По существу, метод основан на линейной интерполяции функции по границам интервала неопределенности.



При желании вы можете использовать параболическую интерполяцию по трем точкам, взяв третью, например, тоже в середине интервала и вычислив координату точки пересечения параболы с осью абсцисс. Если значения аргумента на границах интервала c и d, то координата внутренней точки при линейной аппроксимации

$$x_{вн} = c - \frac{d - c}{f(d) - f(c)} f(c).$$

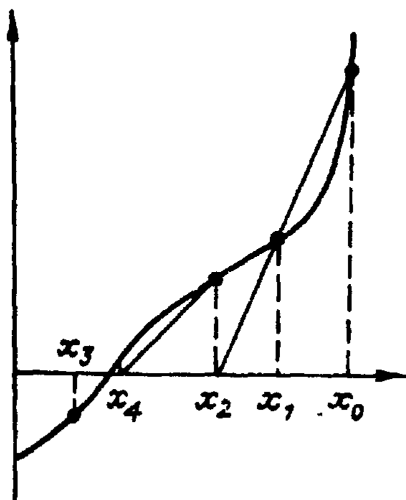
Итерации прекращаются, когда расстояние между соседними приближениями станет меньше заданной погрешности.

Метод случайного поиска – координаты точки внутри текущего интервала неопределенности генерируются с помощью генератора случайных чисел, а операции над границами интервала неопределённости не отличаются от тех, что производятся в описанных выше методах – например, в методе дихотомии.

6.2.1.2. Рекуррентные методы уточнения текущей оценки значения корня

Метод секущих – модификация метода Ньютона, связанная с заменой производной функции отношением приращения функции к приращению аргумента. Для старта процесса уточнения необходимо задать два близких друг к другу приближения x_0 и x_1, а каждое новое приближение получим по формуле:

$$x_{k+1}=x_k-\frac{x_k-x_{k-1}}{f(x_k)-f(x_{k-1})}f(x_k).$$



Скорость сходимости несколько ниже метода Ньютона, но не требуется вычисление производной.

Метод простых итераций может использоваться как для систем уравнений, так и для одиночного уравнения. Дадим для разнообразия его запись в векторной форме.

Если система уравнений $f(\boldsymbol{x})=0$ приведена к виду $\boldsymbol{x}=\varphi(\boldsymbol{x})$ и выбрано начальное приближение $\boldsymbol{x}_0$, то последующие приближения методом простой итерации находятся по формуле

$$\boldsymbol{x}_{m+1}=\varphi(\boldsymbol{x}_m),\ m=0,\ 1,\ \ldots$$

Если последовательность векторов $\boldsymbol{x}_m$ сходится к вектору $\boldsymbol{x}^*$, а функции $\varphi_i(\boldsymbol{x})$ непрерывны и задают сжимающее отображение, то вектор $\boldsymbol{x}^*$ является приближенным решением системы.

6.2.2. Уточнение корня функции одной переменной в условиях помех

Если значения функции при заданных значениях аргумента определяются с погрешностями, которыми нельзя пренебречь и которые существенно искажают результаты поиска корня в заданном интервале, используют так называемые

6.2.2.1. Методы стохастической аппроксимации

В приложении к задаче поиска корня этот метод реализуется известной ***процедурой Роббинса-Монро:***

$$x_{n+1}=x_n-a_n z(x_n),$$

где x_n, x_{n+1} – значения аргумента на соответствующих индексам шагах, $z(x_n)$ – полученная с погрешностью оценка функции $f(x)$ на n-м шаге, a_n – некоторый член последовательности положительных чисел, удовлетворяющих *условиям сходимости Дворецкого*:

1. Предел членов последовательности сходится: $\lim a_n=0$ при $n\to\infty$.
2. Сумма членов последовательности расходится: $\sum_{n=1}^{\infty} a_n = \infty$.
3. Сумма квадратов членов – сходится: $\sum_{n=1}^{\infty} a_n^2 < \infty$.

Этим условиям удовлетворяет, например, гармоническая последовательность $1/n$: 1, 1/2, 1/3, …, обеспечивающая наиболее быстрое сокращение шага по сравнению с другими последовательностями вида n^{-p}.

Роббинс и Монро показали, что при выполнении этих требований их процедура сходится в среднеквадратическом смысле – математическое ожидание квадрата отклонения от истинного значения корня стремится к нулю:

$$\lim \{E[(x_n - x^*)^2]\} = 0 \text{ при } n \to \infty.$$

Впоследствии Блюм показал, что при $n \to \infty$ последовательность x_n сходится к x^* с вероятностью единица:

$$p\{\lim x_n = x^*\} = 1.$$

6.3. Методы поиска экстремума функций

6.3.1. Унимодальные функции одного аргумента при отсутствии помех

6.3.1.1. Метод дихотомии

Нетрудно доказать, что если в нашем распоряжении только 2 опыта, то лучшее, что можно сделать для получения информации с целью максимального приближения к экстремуму – это провести эти 2 опыта в середине интервала по возможности ближе друг к другу. После переноса одной из границ в более удаленную от экстремума точку длина интервала будет больше половины первоначального как раз на расстояние между опытами. Следующие опыты можно осуществить снова в середине интервала и т. д. – эта простейшая и достаточно эффективная стратегия активного эксперимента называется *методом половинного деления (дихотомии)* – аналог метода Больцано в задаче поиска корня.

6.3.1.2. Метод Фибоначчи

Существует более совершенный метод, чем метод дихотомии – это предложенный в 1953 г. Кифером ε-минимаксный алгоритм, основанный на использовании чисел Фибоначчи для определения коэффициента деления текущего интервала неопределенности. Пусть в нашем рас-

поряжении n экспериментов для исследования на экстремум исходного интервала единичной длины. Предположим, что все эксперименты, кроме одного, уже проведены и оставшийся интервал имеет длину L_{n-1}, а внутри его находится оставшийся опыт, давший лучшее приближение к экстремуму и поэтому не ставший новой границей интервала. Так как у нас остался единственный опыт в запасе, то не видно лучшего решения, как провести его симметрично уже имеющемуся относительно середины интервала. Чтобы эта симметрия соблюдалась и на более ранних опытах, длины интервалов должны удовлетворять условию $L_{j-1}=L_j+L_{j+1}$, что возможно, если первый опыт провести в 2-х точках, отстоящих от противоположных концов исходного интервала в доле его длины, равной F_{n-1}/F_n, где F_{n-1}, F_n – числа Фибоначчи соответствующих номеров. Одна из внутренних точек станет новой его границей, а вторая окажется внутри интервала и следующий опыт необходимо провести просто симметрично этой оставшейся точке относительно середины нового интервала.

6.3.1.3. Метод золотого сечения

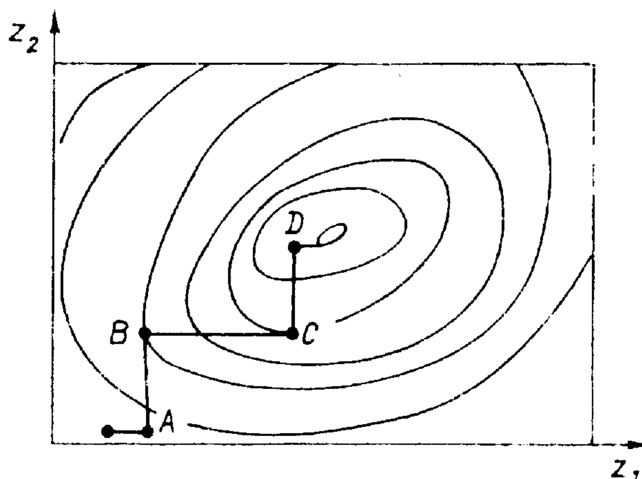
Начиная поиск экстремума, экспериментатор часто не знает, сколько опытов ему для этого понадобится – обычно он собирается проводить свои эксперименты до достижения поставленной цели, то есть пока интересующий его критерий не будет удовлетворен. Однако каждый хотел бы использовать быстро сходящийся метод, экономящий время и средства на экспериментирование. Как мы уже знаем, наиболее эффективным теоретически является метод Фибоначчи, но воспользоваться им можно, только заранее зная число испытаний, так как первый опыт рассчитывается по числам Фибоначчи, номера которых зависят от общего числа опытов. Существует метод, почти столь же эффективный, как и метод Фибоначчи и не зависящий от количества запланированных опытов, то есть со свободным за-

вершением. По этому методу предлагается выдерживать постоянным отношение длин последовательных интервалов неопределенности $L_{j-1}/L_j=L_j/L_{j+1}=t$, откуда приходим к квадратному уравнению $t^2=t+1$, имеющему 1 положительный корень 1.618033989... – *золотое сечение*. По результатам 2-х экспериментов устанавливается, в каком из сегментов интервала опыты будут продолжены. В этом оставшемся интервале уже есть 1 предыдущий эксперимент и для продолжения достаточно провести симметричный ему опыт. После n испытаний приходим к интервалу неопределенности $L_n=1/t^{n-1}$. Метод предложен Кифером и Джонсоном и при большом числе опытов практически совпадает по эффективности с методом Фибоначчи.

6.3.2. Многомерный поиск экстремума

6.3.2.1. Метод координатного спуска

Алгоритм координатного спуска используется в многомерных задачах экспериментальной оптимизации и заключается в сведении многомерной задачи к последовательности одномерных, решаемых методами минимизации функции одной переменной, например золотого сечения.



Вначале в заданной области определения функции всем координатам, кроме одной, присваиваются фиксированные значения и целевая функция делается зависимой только от одной переменной. Далее ищется условный минимум функции, вариацией одной свободной переменной. Полученная координата условного минимума фиксируется в найденном значении и ищется условный минимум вариацией следующей переменной. После использования всех координат процесс снова продолжается с первоначальной переменной и т. д.

Если в области минимума функция цели достаточно гладкая, то процесс спуска по координатам будет линейно сходиться к минимуму. В сходящемся процессе расстояния между соседними точками однокоординатных минимумов будут стремиться к 0, что можно использовать для формулировки условия завершения итерационного процесса. Недостатком метода является чрезмерно большой объем экспериментов и опасность группирования опытов вокруг ложного экстремума при сложном рельефе исследуемой поверхности.

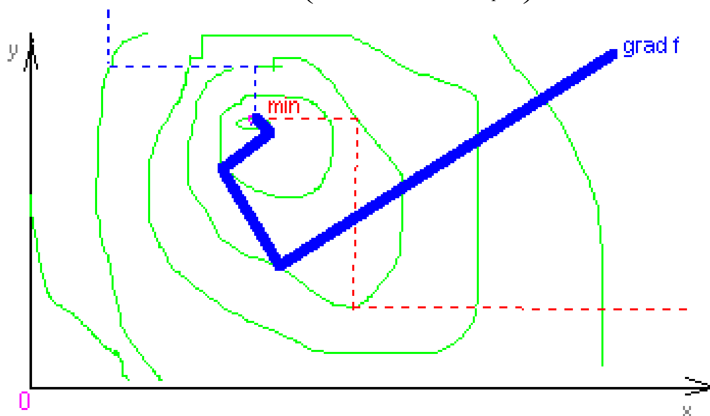
6.3.2.2. Метод градиента (наискорейшего спуска или крутого восхождения или метод Бокса-Уилсона)

Турист, желающий подняться на вершину холма, достигнет цели, если будет непрерывно подниматься вверх, а если он торопится, то ему придется двигаться по самым крутым направлениям.

Реализация этого интуитивно понятного метода носит название метода крутого восхождения или ***метода градиента***. Помимо направления, в математической реализации метода необходимо иметь и алгоритм выбора шага поиска – при слишком большом шаге неизбежны потери на рыскание вокруг экстремума, а при слишком маленьком движение к цели будет медленным. Градиент функции $f(x_1, x_2, ...,$

x_n) в каждой точке направлен в сторону наибольшего локального возрастания этой функции:

$$\operatorname{grad} f(x) = \left(\frac{\partial f(x)}{\partial x_1}, \ldots, \frac{\partial f(x)}{\partial x_p} \right)^T .$$



Если выбрано начальное приближение $x=x_0$, то каждое очередное вычисляется так:

$$x_{m+1}=x_m-a_m \operatorname{grad} f(x_m),$$

где значение a_m определяет величину шага и может быть вычислено как наименьший положительный корень скалярного уравнения

$$\frac{\partial}{\partial a}(x_m-a_m \operatorname{grad} f(x_m))=0.$$

Для поиска максимума необходимо двигаться в направлении градиента, а для поиска минимума – в противоположном. Для определения направления движения (приращений текущих координат независимых переменных) придется вычислять производные целевой функции по каждой координате (составляющие градиента) и нормировать их по модулю вектора градиента, чтобы выделить в чистом виде направление (направляющие косинусы вектора градиента или, что то же самое, составляющие вектора единичной длины с тем же направлением, что и вектор градиента).

Теперь еще о величине шага. *При наличии погрешностей* в определении значения целевой функции шаг должен убывать по мере приближения к цели, т.е. его величина должна быть членом последовательности беззнаковых чисел, удовлетворяющей условию $\lim a_n=0$ при $n\to\infty$. Сумма членов этой последовательности должна расходиться $\sum_{n=1}^{\infty} a_n = \infty$, а сумма квадратов – сходиться $\sum_{n=1}^{\infty} a_n^2 < \infty$. Мы уже рассматривали эти условия Дворецкого при изучении методов поиска корня в условиях помех. Кестен показал, что асимптотическая сходимость может быть заменена затухающим колебательным процессом, если длину шага уменьшать по гармоническому закону только после того, как соответствующая составляющая градиента поменяет знак.

Но шаг целесообразно делать переменным и в процессе поиска в отсутствии помех – вдали от экстремума можно двигаться с большим шагом, а по мере приближения к цели его следует уменьшать для повышения точности в определении экстремума.

6.3.2.3. Последовательный симплексный поиск (ПСМ) субоптимальной области в многомерном пространстве

6.3.2.3.1. Вводные замечания

Метод был предложен в 1962 году Спиндлеем, Херстом и Химсуорсом [см. А.П. Дамбраускас. Симплексный поиск. – М.: Энергия, 1979]. Оригинальность метода состоит в том, что движение к оптимуму в многомерном пространстве независимых переменных осуществляется последовательным отражением вершин симплекса. В k-мерном евклидовом пространстве *симплексом* называют фигуру, образованную k+1 точками (вершинами), не принадлежащими одновременно ни одному пространству меньшей размерно-

сти. В одномерном пространстве симплекс есть отрезок прямой, в двумерном – треугольник, в трёхмерном – тетраэдр и т. д. Симплекс называется *регулярным*, если расстояния между его вершинами равны. В ПСМ используются регулярные симплексы.

Из любого симплекса, отбросив одну его вершину, можно получить новый симплекс, если к оставшимся вершинам добавить всего одну точку. Это замечательное свойство и было использовано авторами метода при построении алгоритма движения симплекса в сторону искомой цели.

Для оценки направления движения во всех вершинах симплекса необходимо определить значения целевой функции Y_j. При поиске максимума наиболее целесообразным будет движение от вершины v_s с наименьшим значением Y_s к противоположной грани симплекса. Шаг поиска выполняется переходом из некоторого симплекса S_{n-1} в новый симплекс S_n путем исключения вершины v_s и построения ее зеркального отображения $v_{нs}$ относительно грани, общей обоим симплексам. Многократное отражение худших вер-

шин приводит к шаговому движению центра симплекса к цели по траектории некоторой ломаной линии. Если не учитывать эксперименты в вершинах исходного симплекса, то на каждый шаг поиска требуется всего одно определение целевой функции.

6.3.2.3.2. Алгоритм последовательного симплексного метода

Исходные данные:

- количество независимых переменных k;
- предельные значения каждой i-й независимой переменной x_{imin}, x_{imax}. Эти предельные значения для реальных объектов исследования определяются априорными сведениями, условиями безопасности при проведении экспериментов и т. д.
- допустимая ошибка в определении координат оптимума ε;
- предполагается также возможность определения значения целевой функции $Y(x_{1j}, x_{2j}, ..., x_{ij}, ..., x_{kj})$ для каждой j-й вершины симплекса.

6.3.2.3.3. Подготовительные операции

1. Прежде всего необходимо определить стартовую точку для начала поисковых процедур; при отсутствии дополнительных априорных данных естественно расположить ее в центре области, ограниченной предельными значениями независимых переменных:

$$x_{1c}=(x_{1max}–x_{min})/2,\ x_{2c}=(x_{2max}–x_{2min})/2,\ ...,\ x_{kc}=(x_{kmax}–x_{kmin})/2.$$

Перенос в эту точку начала координат облегчит последующие вычислительные процедуры (достигается вычитанием).

2. Определяются координаты вершин начального симплекса. Из множества возможных ориентаций начального симплекса на практике используют 2 варианта:

а) Центр симплекса располагается в начале координатной системы, а одна из вершин ($(n+1)$-я) – на оси x_n.

Остальные вершины при этом расположатся симметрично относительно координатных осей.

Координаты вершин вычисляются в этом варианте с помощью матрицы:

Номер вершины	Координаты вершин					
	x_1	x_2	x_3	...	x_{n-1}	x_n
1	$-r_1$	$-r_2$	$-r_3$	...	$-r_{n-1}$	$-r_n$
2	R_1	$-r_2$	$-r_3$	...	$-r_{n-1}$	$-r_n$
3	0	R_2	$-r_3$	...	$-r_{n-1}$	$-r_n$
...	...	...	...	...	...	…
n	0	0	0	...	R_{n-1}	$-r_n$
n+1	0	0	0	...	0	R_n

где при единичной длине ребра симплекса $r_i=\dfrac{1}{\sqrt{2i(i+1)}}$,

$R_i=\dfrac{1}{\sqrt{2(i+1)}}$, i=1, 2, ..., n.

б) Во втором варианте одна из вершин симплекса размещается в начале координат, а исходящие из нее ребра образуют одинаковые углы с соответствующими осями. Вспомогательная расчетная матрица для координат вершин начального симплекса имеет вид:

Номер вершины	Координаты вершин					
	x_1	x_2	x_3	...	x_{n-1}	x_n
1	0	0	0	...	0	0
2	p	q	q	...	q	q
3	q	p	q	...	q	q
4	q	q	p	...	q	q
...	...	...	...	...	...	…
n+1	q	q	q	...	q	p

где при единичной длине ребра $p=n-1+\frac{\sqrt{n+1}}{n\sqrt{2}}$,

$q=\frac{\sqrt{n+1}-1}{n\sqrt{2}}$.

2. Получаем значения функции отклика в вершинах исходного симплекса и на этом завершаем подготовительные операции.

6.3.2.3.4. Алгоритм поиска

В цикле с выходом по заданному условию выполняем:

Отбрасываем вершину с наихудшим значением критерия оптимальности (наименьшим при поиске максимума или наибольшим при поиске минимума).

Вычисляем координаты вершины, зеркально отображаемой отброшенной относительно противоположной ей грани симплекса:

$$x_i=x_i(2(x_{1i}+x_{2i}+\ldots+x_{j-1,\,i}+x_{j+1,\,i}+\ldots+x_{n+1,\,i})/n-x_{ji},$$

где j – номер отбрасываемой вершины, i – номер координаты.

Проводим эксперимент в новой точке для получения значения целевой функции.

Условием выхода из цикла может быть малое приращение целевой функции на протяжении заданного числа опытов или при сохранении одной из вершин своего присутствия в симплексе заданное число раз и т. п.

При приближении к области оптимума точность может быть повышена уменьшением размера симплекса, но при наличии погрешностей в определении значений целевой функции необходимо ограничить размер симплекса снизу, чтобы избежать блужданий под действием случайных шумов измерений.

6.3.2.3.5. Преимущества метода

Число необходимых опытов для определения направления движения мало по сравнению с другими методами.

Легко учитываются ограничения на область изменения варьируемых при поиске факторов.

Эффективность метода растет с увеличением мерности пространства поиска.

Малый объем вычислений на каждом шаге.

Отсутствие высоких требований к точности оценки значения целевой функции – достаточно возможности проранжировать значения качественно по принципу «больше-меньше».

Метод пригоден для преследования дрейфующей цели (максимума или минимума), что делает его применимым в адаптивных алгоритмах.

Возможность изменять мерность пространства на ходу изменением количества вершин симплекса.

6.3.2.3.6. Недостатки метода

Отсутствие данных о влиянии каждого фактора на целевую функцию.

Трудность интерпретации характера поверхности отклика по данным реализуемых в методе опытов.

6.4. Программная реализация класса подпрограмм для поиска экстремума унимодальных в заданном интервале функций одной переменной

```
#include<conio.h>
#include "matrix.h"

typedef matrix<double> dmatrix;
typedef vector<double> dvector;

/*Определим вначале некоторые вспомогательные под-
программы*/
//Для вычисления числа Фибоначчи с заданным номером
long Fib(long n)
{
  long F[3]={0,1};
  if(n<0)
```

```
    throw xmsg("Некорректный номер числа");
  if(n<2)
    return F[n];
  for(long i=2;i<=n;i++)
  {
    F[2]=F[0]+F[1];
    F[0]=F[1];
    F[1]=F[2];
  }
  return F[2];
}


/*Простая парабола для тестирования одномерного по-
иска*/
double gv(double x)
{
  return (4*x*x+5*x+1);
}


/*Функциональный класс объектов для поиска экстре-
мума; мы включили в него и некоторые функции опре-
деления значения вещественного корня*/
class MonoExtremum
{
  double c, d;//Границы для одномерного поиска
  double eps;//Для допустимой погрешности поиска
/*Указатель на подпрограмму, возвращающую значение
исследуемой функции*/
  double (*getv)(double x);
 public:
  //Конструктор для одномерного поиска
  MonoExtremum(double C, double D, double EPS,
double (*GETV)(double X)): c(C), d(D), eps(EPS),
getv(GETV) { }
  //Прототипы функций поиска вещественного корня
  double RealRoot(long cod); /*сокращением интерва-
ла неопределенности*/
  double DNewton(); //Метод секущих
  double Stoh(); //Стохастическая аппроксимация
  /*Прототипы функций решения задачи условной опти-
мизации*/
```

```
  double Dihot(); //метод дихотомии
  double Fibon(long cnt); //метод Фибоначчи
  double Gold(); //метод золотого сечения
  dvector Coord(); //метод покоординатного спуска
  dvector Grad(); //метод градиента
  dvector PSM(); //последовательный симплекс-метод
};


/*Обобщенная функция уточнения корня последователь-
ным сокращением интервала неопределенности*/
double MonoExtremum::RealRoot(long cod=0)
{
  double ct=c,dt=d,x,vc,vd,vx,k;
  for(;;)
  {
    switch(cod)
    {
       case 0: //Больцано
         k=0.5;
         vc=getv(ct);
         break;
       case 1: //Хорды
         vc=getv(ct);
         vd=getv(dt);
         k=vc/(vd+vc);
         if(k<0)
           k=-k;
         break;
       case 2: //Случайный поиск
         randomize();
         k=double(1+random(1111))/1111.0;
         vc=getv(ct);
         break;
    }
    x=ct+k*(dt-ct);
    vx=getv(x);
    if(fabs(vx)<eps)
      return x;
    if(vx*vc>0)
      ct=x;
    else
      dt=x;
```

```
  }
}


//Метод секущих - дискретный аналог метода Ньютона
double MonoExtremum::Dnewton()
{
  double sx[3];
  sx[0]=(c+d)/2;
  sx[1]=sx[0]+1e-4;
  sx[2]=sx[1]-(getv(sx[1]))*(sx[1]-sx[0])/
            (getv(sx[1])-getv(sx[0]));
  while(1)
  {
    if(getv(sx[2])<eps)
      return sx[2];
    sx[0]=sx[2];
    sx[1]=sx[0]+1e-4;
    sx[2]=sx[1]-(getv(sx[1]))*(sx[1]-sx[0])/
            (getv(sx[1])-getv(sx[0]));
  }
}


//Стохастическая аппроксимация
double MonoExtremum::Stoh()
{
  double scnt=1;
  //Для определения знака приращения
  double sx[3],fa[3];
  sx[0]=(c+d)/2.0;
  fa[0]=fabs(getv(sx[0]));
  if(fa[0]<eps)
    return sx[0];
  sx[1]=sx[0]+1e-2;
  fa[1]=fabs(getv(sx[1]));
  if(fa[1]<eps)
    return sx[1];
  for(scnt=1;;scnt++)
  {
    sx[2]=sx[1]-(20.0/scnt)*fa[1]*(sx[1]-sx[0])/
(fa[1]-fa[0]);
    fa[2]=fabs(getv(sx[2]));
```

```
    if(fa[2]<eps)
      return sx[2];
    sx[0]=sx[1];
    sx[1]=sx[2];
    fa[0]=fa[1];
    fa[1]=fa[2];
  }
}


//Реализация функций одномерного поиска экстремума
//Метод дихотомии
double MonoExtremum::Dihot()
{
  double xl,xr,yl,yr,dt=d,ct=c;
  for(;(dt-ct)>eps;)
  {
   /*задаем координаты 2-х точек эксперимента вбли-
зи средины интервала*/
    xl=(ct+dt)/2-0.01*(dt-ct);
    xr=(ct+dt)/2+0.01*(dt-ct);
    //Вычисляем значения функции в этих точках
    yl=getv(xl);
    yr=getv(xr);
    if(sign(yl-yr)>0)
    /*Бывшая правая - теперь левая, а правая - сим-
метрично*/
    {
      ct=xl;
      xl=xr;
      xr=ct+dt-xl;
    }
    else
    {
      dt=xr;
      xr=xl;
      xl=ct+dt-xr;
    }
  }
  return (ct+dt)/2;
}
```

```
//Метод Фибоначчи для поиска экстремума
double MonoExtremum::Fibon(long cnt)
{
  double k=(double)Fib(cnt-1)/(double)Fib(cnt);
  /*вычисляем координаты 2-х первых точек экспери-
мента*/
  double ct=c, dt=d, xl=d-k*(d-c), xr=c+k*(d-c),
yl, yr;
/*Далее в цикле в худшую точку переносим ближайшую
границу, а добавочную точку определяем симметрично
оставшейся относительно середины интервала */
  for(long i=cnt;i>1;i--)
  {
    yl=getv(xl);
    yr=getv(xr);
    if(sign(yl-yr)>0)
    {
      ct=xl;
      xl=xr;
      xr=ct+dt-xl;
    }
    else
    {
      dt=xr;
      xr=xl;
      xl=ct+dt-xr;
    }
  }
  return (ct+dt)/2;
}


//Метод золотого сечения
double MonoExtremum::Gold()
{
  double k=0.618033989;
  /*вычисляем координаты 2-х первых точек экспери-
мента*/
  double ct=c, dt=d, xl=d-k*(d-c), xr=c+k*(d-c),
yl, yr;
/*Далее в цикле в худшую точку переносим ближайшую
границу, а добавочную точку определяем симметрично
оставшейся относительно середины интервала*/
```

```
  for(;(dt-ct)>eps;)
  {
    yl=getv(xl);
    yr=getv(xr);
    if(sign(yl-yr)>0)
    {
      ct=xl;
      xl=xr;
      xr=ct+dt-xl;
    }
    else
    {
      dt=xr;
      xr=xl;
      xl=ct+dt-xr;
    }
  }
  return (ct+dt)/2;
}


//Программа тестирования методов уточнения корней и
//одномерного поиска экстремума
main()
{
  /*Тестирование методов уточнения корня последова-
тельным сужением интервала неопределенности*/
  MonoExtremum root1(-0.5,0.5,1e-3,gv);
//Сконструируем объект
  double res0=root1.RealRoot(0); //0-метод Больцано
  cout<<endl<<"Bolzano: "<<res0<<" Func : "<<
gv(res0);
  double res1=root1.RealRoot(1); //1-метод хорд
  cout<<endl<<"Chord : "<<res1<<" Func : "<<
gv(res1);
  double res2=root1.RealRoot(2); /*2-метод Монте _
Карло*/
  cout<<endl<<"Rand :"<<res2<<" Func : "<<gv(res2);

  //Тестирование поиска корня методом секущих
  double resDNewton=root1.DNewton();
  cout<<endl<<"DNewton : "<<resDNewton<<" Func : "
      <<gv(resDNewton);
```

```
  /*Тестирование поиска корня методом стохастиче-
ской аппроксимации*/
  double resStoh=root1.Stoh();
  cout<<endl<<"Stoh : "<<resStoh<<" Func : "
      <<gv(resStoh);

  //Тестирование методов поиска экстремума
  MonoExtremum extr1(-3.0,+5.0,1e-
3,gv);/*Сконструируем объект*/
  //Вызываем методы решения
  double resDihot=extr1.Dihot(), //Дихотомии
         resFib=extr1.Fibon(25), //Фибоначчи
         resGold=extr1.Gold(); //Золотого сечения
  cprintf("\r\nDihot : %lf Fibon : %lf Gold : %lf
", resDihot,resFib,resGold);
  getch();
}
```

6.5. Программная реализация класса подпрограмм для многомерного поиска экстремума унимодальных функций

```
#include <conio.h>
#include <iostream.h>
#include "matrix.h"

typedef matrix<double> dmatrix;
typedef vector<double> dvector;


//Определим некоторые вспомогательные подпрограммы
//Возвращает код знака числа
inline long sign(double x)
{
  return (x>0)?1:((x<0)?-1:0);
}


/*Простой параболоид для тестирования многомерного
поиска*/
double gv(dvector x)
```

```
{
  double sum=0;
  int i;
  int r=x.getm();
  for(i=0;i<r;i++)
    sum+=10.0*(x[i]-4)*(x[i]-4);
  return sum;
}


/*Простейшая функция для численного дифференцирова-
ния, возвращающая значение вектора градиента в за-
данной точке при заданной матрице ограничений на
составляющие аргумента*/
dvector GetGrad(dvector x, dmatrix rm)
{
  int r=x.getm();
  dvector grad(r);
  double _yold=gv(x), //Текущее значение функции
         _ynew, dx;
  for(int i=0;i<r;i++)
  {
/*Приращение аргумента возьмем в малую долю его ин-
тервала*/
    dx=fabs(rm[i][1]-rm[i][0])*1e-10;
    x[i]+=dx;
    _ynew=(gv(x)); //Новое значение функции
    /*Составляющая вектора градиента*/
    grad[i]=(_ynew-_yold)/(dx);
    x[i]-=dx;
  }
  return (~grad);//Возвращаем, нормировав по модулю
}


/*Функциональный класс объектов для многомерного
поиска экстремума*/
class MultiExtremum
{
  /*Размерность вектора аргументов исследуемой
функции*/
  long range;
```

```
  /*матрица ограничений на значения составляющих
вектора аргументов*/
  dmatrix rm;
  dvector veps;/*Вектор допустимых погрешностей по
аргументу*/
  /*Указатель на подпрограмму, возвращающую значе-
ние исследуемой функции при заданном векторе ее ар-
гументов*/
  double (*getv)(dvector x);
 public:
/*Конструктор для многомерного поиска примет аргу-
менты - мерность пространства, матрицу ограничений,
вектор допустимых погрешностей, указатель на под-
программу вычисления значений функции*/
  MultiExtremum(long RANGE, dmatrix RM, dvector
VEPS, double (*GETV)(dvector XM)): range(RANGE),
rm(RM), veps(VEPS), getv(GETV) {}
  /*Прототипы функций решения задачи условной опти-
мизации*/
  void MultiDihot(dvector& x,int n);
/*вспомогательный метод дихотомии*/
  dvector Coord(); //метод покоординатного спуска
  dvector Grad(); //метод градиента
  dvector PSM(); //последовательный симплекс-метод
};


/*Реализация локального поиска экстремума по одной
координате методом дихотомии*/
void MultiExtremum::MultiDihot(dvector& x,int n)
{
  double xl,xr,yl,yr,dt=rm[n][1],ct=rm[n][0];
  for(;(dt-ct)>veps[n];)
  {
    x[n]=xl=(ct+dt)/2-0.01*fabs(dt-ct);
    yl=getv(x);
    x[n]=xr=(ct+dt)/2+0.01*fabs(dt-ct);
    yr=getv(x);
    if(sign(yl-yr)>0)
    {
      ct=xl;
      xl=xr;
      xr=ct+dt-xl;
```

```
    }
    else
    {
      dt=xr;
      xr=xl;
      xl=ct+dt-xr;
    }
  }
  x[n]=(ct+dt)/2;
}


//Методы многомерного поиска
//Координатный спуск
dvector MultiExtremum ::Coord()
{
  dvector xtp(range),xtn(range), xeps(range);
  int i,flag;
  for(i=0;i<range;i++)
    xtp[i]=xtn[i]=0.5*(rm[i][0]+rm[i][1]);
  for(;;)
  {
    flag=0;
    for(i=0;i<range;i++)
      MultiDihot(xtn,i);
    xeps=xtn-xtp;
    for(i=0;i<range;i++)
      if(fabs(xeps[i])>fabs(veps[i]))
        flag++;
    if(!flag)
      break;
    else
      xtp=xtn;
  }
  return xtn;
}


//Метод градиента
dvector MultiExtremum::Grad()
{
  double k=5.0,vn,vp;
```

```
  dvector xtp(range), xtn(range), xeps(range),
gr(range);
  /*Вначале станем в средину допустимой области по-
иска*/
  for(int i=0;i<range;i++)
    xtp[i]=(0.5*(rm[i][0]+rm[i][1]));
  for(;;)
  {
    //Вычислим градиент
    gr=GetGrad(xtp,rm);
   /*Корректируем аргумент в направлении, противо-
положном градиенту*/
    xtn=xtp-k*gr;
    vn=getv(xtn);
    vp=getv(xtp);
    //Если шаг слишком большой - уменьшаем его
    for(;vn>vp;)
    {
      k/=1.2;
      xtn=xtp-k*gr;
      vn=getv(xtn);
      vp=getv(xtp);
    }
    /*Если шаг по градиенту не изменяет существенно
значение функции - прекращаем поиск*/
    if(fabs(getv(xtn)-getv(xtp))<1e-8)
      break;
    xtp=xtn;
  }
  return xtn;
}


/*Последовательный симплекс-метод.
Этот метод достаточно громоздок в программном ис-
полнении, особенно при использовании переменного
размера поискового симплекса, защиты от вращения
симплекса вокруг одной из вершин и других манипуля-
ций, необходимых для его эффективной работы. Мы
приводим упрощенный, а, следовательно, не очень
точный в вычислениях пример программы, демонстриру-
ющий принцип работы алгоритма */
dvector MultiExtremum::PSM()
```

```
{
  double sum;
  int i,j;

  dvector yv(range+1);/*Вектор значений функции в
вершинах симплекса*/
/*Формируем исходный регулярный симплекс в про-
странстве размерности range с центром в начале ко-
ординат */
  dmatrix simp(range+1,range);/*матрица для вершин
симплекса*/
  dvector stmp(range);
  const double L=2.0L;
  //заполняем матрицу симплекса
  //Сначала нулевую строку
  for(j=0;j<range;j++)
    simp[0][j]=0.0;
  //Затем остальные
  double pk=(L/(range*sqrt(2)))*
            (sqrt(range+1)+double(range)-1.0);
  double qk=L*(pk-0.5*sqrt(2));
  for(i=1;i<(range+1);i++)
    for(j=0;j<range;j++)
      simp[i][j]=(j==(i-1))?pk:qk;
  int pmax=-1,//предыдущая плохая
      cicl;
  int vmin=0,vmax=0;
  //Бесконечный цикл поиска
  for(cicl=1;/*cicl<100000*/;cicl++)
  {
    //Вычисляем значения функции во всех вершинах
    for(i=0;i<(range+1);i++)
      yv[i]=getv(simp[i]);
    /*Отыскиваем номера наихудшей и наилучшей вер-
шин*/
    for(i=1;i<(range+1);i++)
    {
      if(yv[i]>yv[vmax])
        vmax=i;
      if(yv[i]<yv[vmin])
        vmin=i;
    }
```

```
    /*Если наилучшая нас устраивает - прекращаем
поиск*/
    if(fabs(yv[vmin])<0.05)
      break;/*Грубая оценка попадания в зону экс-
тремума. В этом месте вместо break можно было бы
уменьшить размер симплекса и условие достижения
экстремума*/
    /*Если плохая не сменила номер - ищем ближайшую
плохую*/
    if((cicl>1)&&(vmax==pmax))
      for(i=1;i<(range+1);i++)
      {
        if((yv[i]>yv[vmax])&&(i!=pmax))
          vmax=i;
      }
    pmax=vmax;//Запоминаем как предыдущую плохую
    //меняем плохую вершину на зеркально отраженную
    for(j=0;j<range;j++)//j-номер координаты
    {
      /*Для каждой координаты вычислим ее среднее
значение по всем вершинам*/
      sum=0;
      for(int k=0;k<(range+1);k++)
        sum+=simp[k][j];//k- номер вершины
      sum*=(2.0/(range+1));
      /*Из среднего вычитаем плохое значение и ре-
зультат сохраняем в промежуточном векторе*/
      stmp[j]=sum-simp[vmax][j];
    }
    /*Новый вектор координат переносим на место
наихудшей вершины*/
    simp[vmax]=stmp;
  }//бесконечный цикл
  return simp[vmin];
}


//Программа тестирования методов поиска экстремума
main()
{
  //Тестируем многомерные методы поиска экстремума
  /*Формируем размерность, матрицу ограничений и
вектор погрешностей*/
```

```
  int r=5;
  dmatrix m(r,2);dvector e(R);
  for(int i=0;i<r;i++)
  {
    m[i][0]=-8.0;
    m[i][1]=15.0;
    e[i]=0.0001;
  }
  MultiExtremum extM(r,m,e,gv);/*Конструируем объ-
ект*/
  /*Вызываем для этого объекта методы поиска мини-
мума*/
  dvector resCoord=extM.Coord(); /*Метод покоорди-
натного спуска*/
  cout<<endl<<gv(resCoord);
  cout<<endl<<resCoord;
  dvector resGrad=extM.Grad(); /*Метод градиента
(наискорейшего спуска)*/
  cout<<endl<<gv(resGrad);
  cout<<endl<<resGrad;

  dvector resPSM=extM.PSM(); /*Последовательный
симплекс-метод*/
  cout<<endl<<gv(resPSM);
  cout<<endl<<resPSM;
  getch();
}
```

Приложение. Как переносить данные в Excel и отображать графики зависимостей

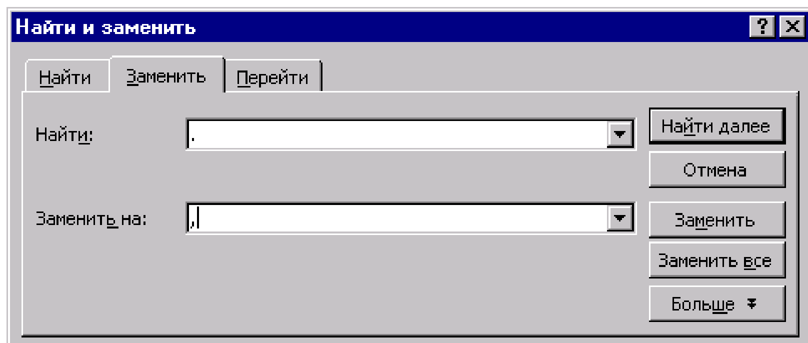
Пусть Вам необходимо отобразить в виде графика данные, которые находятся в файле. Вначале Вам необходимо скопировать эти данные в буфер обмена, используя, к примеру, Far. Для этого выберите режим редактирования файла, нажав клавишу F4 на имени файла, выделите необходимые Вам данные клавишами управления курсором (с прижатой Shift), нажатием Ctrl+Ins скопируйте помеченный блок в буфер обмена. Далее определите, какой вид у этих данных:

1. одна строка произвольной длины
2. один столбец произвольной длины
3. два столбца произвольной длины, первый из которых содержит независимую переменную, а второй – зависимую.
4. две строки произвольной длины, первая из которых содержит независимую переменную, а вторая – зависимую.
5. две строки длины до 64 чисел каждая, первая из которых содержит независимую переменную, а вторая – зависимую.

Случаи большего числа строк (столбцов) сводятся к этим пяти.

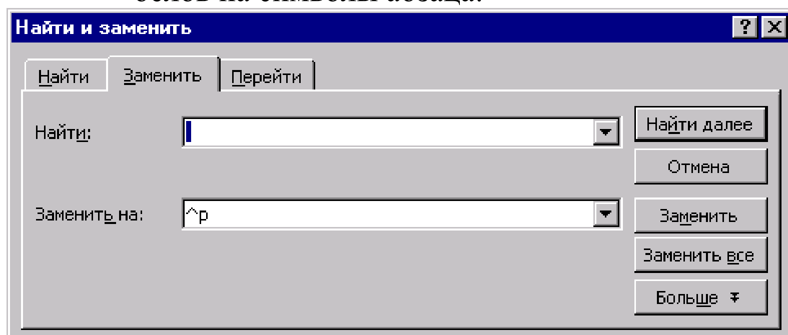
Запустите редактор Word и комбинацией Shift+Ins вставьте содержимое буфера обмена. Произведите замену точек на запятые: нажатием Ctrl+H вызовите диалоговое окно **Найти и заменить**, в поле **Найти** ввести точку, в по-

ле **Заменить на** – запятую и нажать на клавишу **Заменить все**.

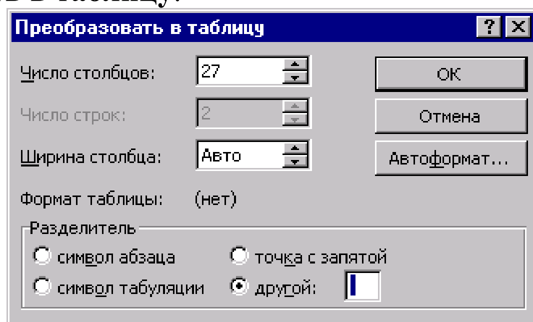


Рассмотрим возможные случаи:

1. Этот случай сводится ко второму заменой пробелов на символы абзаца:



3, 5. Комбинацией Ctrl+A выделите весь текст в окне. Затем в пункте меню **Таблица** выберите пункт **Преобразовать в таблицу**.

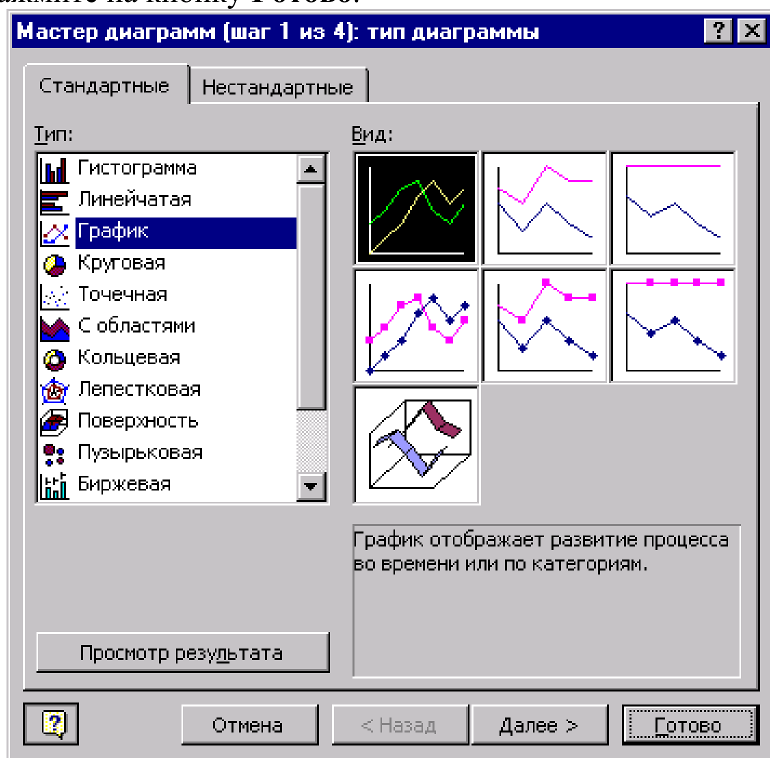


В появившемся диалоговом окне в качестве символа разделителя введите пробел (поле **другой**) и нажмите на клавишу **ОК**.

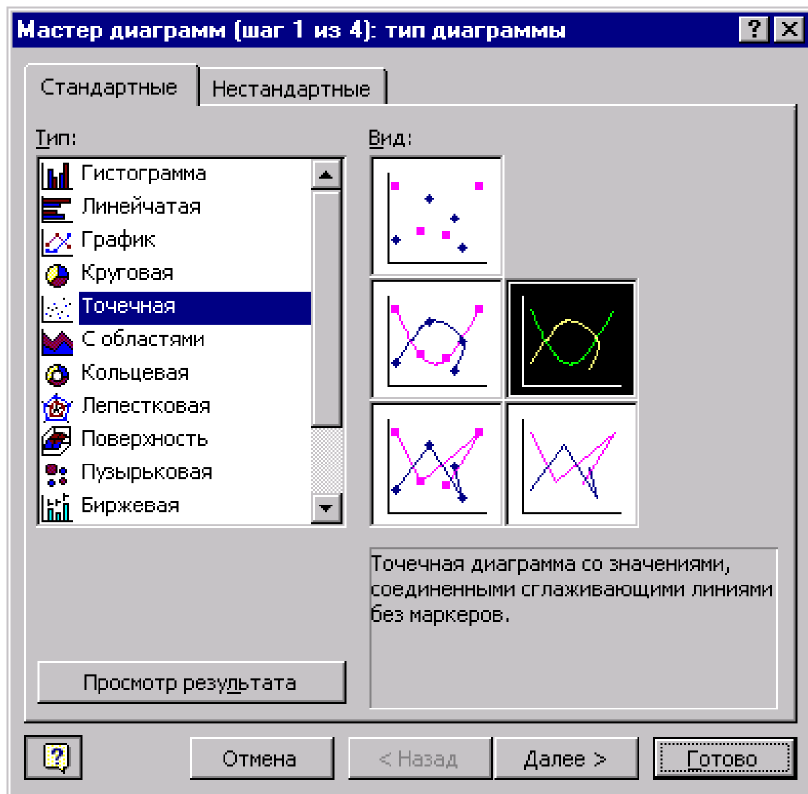
4. А не транспонировать ли Вам данные перед выводом в файл? Сводится к случаю 3.

Следующий этап – выделение всего содержимого окна (Ctrl+A) и копирование выделенного фрагмента в буфер обмена (Ctrl+Ins), затем – запуск табличного процессора Excel и вставка содержимого буфера обмена нажатием Shift+Ins. Не снимая пометки, выберите пункт меню **Вставка**, а в нём – подпункт **Диаграмма**:

1, 2. В списке **Тип** выберите пункт **График**, а в разделе **Вид** – первый из предложенных видов графиков и нажмите на кнопку **Готово**.



3, 4, 5. В списке **Тип** выберите пункт **Точечная**, а в разделе **Вид** – третий из предлагаемых видов графиков и нажмите на кнопку **Готово**.



Литература

1. Алгоритмы и программы восстановления зависимостей. – М.: Наука, 1984.
2. Анго А. Математика для электро- и радиоинженеров. – М.: Наука, 1964.
3. Бабак В.П., Хандецький В.С., Шрюфер Е. Обробка сигналів. – К.: Либідь, 1996.
4. Боглаев Ю.П. Вычислительная математика и программирование: Учебное пособие для студентов втузов. – М.: Высшая школа, 1990.
5. Вайнер Р., Пинсон Л. С++ изнутри. – Киев: ДиаСофт, 1993.
6. Вентцель Д.А., Вентцель Е.С. Элементы теории приближенных вычислений. – М.: Издательство ВВИА им. Н.Е. Жуковского, 1949.
7. Вентцель Е.С. Исследование операций. – М.: Радио и связь, 1970.
8. Вентцель Е.С. Теория вероятностей. – М.: Наука, 1969.
9. Гавурин М.К. Лекции по методам вычислений. – М.: Наука, 1971.
10. Гантмахер Ф.Р. Теория матриц. – М.: Государственное издательство технико-теоретической литературы, 1953.
11. Гринчишин Я.Т., Ефимов В.И., Ломакович А.Н. Алгоритмы и программы на Бейсике. – М.: Просвещение, 1988.
12. Гринчишин Я.Т. TURBO PASCAL: Чисельні методи в фізиці та математиці: Навч. посібник – Тернопіль, 1994.
13. Гулд Х., Тобочник Я. Компьютерное моделирование в физике: В 2-х частях. Часть 2: Пер. с англ. – М.: Мир, 1990.
14. Гутер Р.С., Овчинский Б.В. Элементы численного анализа и математической обработки результатов опыта. – М.: Наука, 1970.

15. Дамбраускас А.П. Симплексный поиск. – М.: Энергия, 1979.
16. Демидович Б.П., Марон И.А. Основы вычислительной математики. – М.: Физматгиз, 1963.
17. Демидович Б.П., Марон И.А., Шувалова Э.З. Численные методы анализа. – М.: Наука, 1967.
18. Джордж А., Лю Дж. Численное решение больших разряжённых систем уравнений. – М.: Мир, 1984.
19. Дьяконов В.П. Справочник по алгоритмам и программам на языке Бейсик для персональных ЭВМ. – М.: Наука, 1987.
20. Иванов В. В. Методы вычислений на ЭВМ: Справочное пособие. – К.: Наукова думка, 1986.
21. Кантор И.Л., Солодовников А.С. Гиперкомплексные числа. – М. Наука, 1973.
22. Лукас П. С++ под рукой. – Киев: ДиаСофт, 1993.
23. Лукомский Я.И. Теория корреляции и её применение к анализу производства. – М.: Госстатиздат ЦСУ СССР, 1961.
24. Мак-Кракен Д., Дорн У. Численные методы и программирование на Фортране. – М.: Мир, 1977.
25. Маликов В.Т., Кветный Р.Н. Вычислительные методы и применение ЭВМ: Учебное пособие. – К.: Выща школа, 1989.
26. Мудров А.Е. Численные методы для ПЭВМ на языках Бейсик, Фортран и Паскаль. – Томск: Раско, 1991.
27. Ортега Дж., Пул У. Введение в численные методы решения дифференциальных уравнений. – М.: Наука, 1986.
28. Плис А.И., Сливина Н.А. Лабораторный практикум по высшей математике. – М.: Высшая школа, 1983.
29. Полищук А.П. Персональный компьютер и его программирование (С, С++, Паскаль). Учебно-справочное пособие. – Кривой Рог: КГПИ, 1997.

30. Попов В.С., Мансуров Н.Н., Николаев С.А. Электротехника. – М.: Издательство Министерства коммунального хозяйства РСФСР, 1958.
31. Смирнов В.И. Курс высшей математики. Т. 1-5. – М.: Наука, 1974.
32. Статистическая обработка результатов экспериментов на микро-ЭВМ и программируемых калькуляторах. – Л.: Энергоатомиздат, 1991.
33. Страуструп Б. Язык программирования С++: В 2-х частях. – К.: Диасофт, 1993.
34. Тихонов А.Н., Арсенин В.Я. Методы решения некорректных задач. Учебное пособие для вузов. – М.: Наука, 1986.
35. Тихонов А.Н., Самарский А.А. Уравнения математической физики. – М.-Л.: Государственное издательство технико-теоретической литературы, 1951.
36. Уилкинсон Дж., Райнш К. Справочник алгоритмов на языке Алгол. Линейная алебра. – М.: Машиностроение, 1976.
37. Уткіна С.В., Наришкіна Л.С. Алгебра і числові системи: Навч. посібник. – К.: Вища школа, 1995.
38. Фаддеева В.Н. Вычислительные методы линейной алгебры. – М.: Гостехиздат, 1950.
39. Фейсон Т. Объектно-ориентированное программирование на Borland C++ 4.5. – Киев: Диалектика, 1996.
40. Форсайт Д., Малькольм М., Моллер К. Машинные методы математических вычислений. – М.: Мир, 1980.
41. Форсайт Д., Моллер К. Численное решение систем линейных алгебраических уравнений. – М.: Мир, 1968.
42. Хемминг Р.В. Численные методы. – М.: Наука, 1968.
43. Шуп Т.Е. Прикладные численные методы в физике и технике. – М.: Высшая школа, 1990.
44. Шуп Т.Е. Решение инженерных задач на ЭВМ. Практическое руководство. – М.: Мир, 1982.

Учебное пособие

Александр Павлович Полищук

Сергей Алексеевич Семериков

Методы вычислений в классах языка C++

Подп. к печати 28.12.98	Формат 80x84 1/16.
Бумага офсетная №1	Усл. печ. л. 18,38
Усл. кр.-от. 18,38	Уч.-изд. л. 22,29
Тираж 500	Зак. №12-2483

КГПИ, 324086, Кривой Рог-86, пр. Гагарина, 54

Криворожская городская типография
324050, Кривой Рог-50, пр. Металлургов, 28.