Криворожский государственный педагогический институт
Кафедра информатики и прикладной математики

А.П. Полищук, С.А. Семериков

ЧИСЛЕННЫЕ МЕТОДЫ В ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ МЕТОДОЛОГИИ

Учебное пособие

Раздел 2: Матрицы и задачи линейной алгебры

Общие сведения о матрицах и матричных операциях
Методы решения систем линейных алгебраических уравнений
Методы вычисления собственных значений и векторов матриц
Метод наименьших квадратов
Введение в линейное программирование

Кривой Рог

1998

Полищук А.П., Семериков С.А. Численные методы в объектно-ориентированной методологии. Раздел «Матрицы и задачи линейной алгебры». Учебное пособие. – Кривой Рог: КГПИ, 1998. – 98 с.

Авторы:

Полищук А.П.	к. т. н., с. н. с., доцент кафедры информатики и прикладной математики.
Семериков С.А.	магистр математики.

Рецензенты:

Рашевский Н.А.	к. ф.-м. н., доцент кафедры математики (КГПИ)
Теплицкий И.А.	учитель-методист, зам. директора по научной работе (Центрально-Городская гимназия)

Под общей редакцией доктора физико-математических наук, профессора В.Н. Соловьёва.

Рекомендовано к печати на заседании кафедры информатики КГПИ, протокол №1 от 31.08.98 г.

Підп. до друку 14.09.98
Друк №3. Друк офсетний
Умовн. фарбо-відб. 3,87
Тираж 300

Формат 80x84 1/16.
Умовн. друк. арк. 4,0
Зам. №9-1473

КДПІ, 324086, Кривий Ріг-86, пр. Гагаріна, 54

Криворізька міська друкарня
324050, Кривий Ріг-50, пр. Металургів, 28.

Оглавление

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, 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:

$$y_i=a_{i1}x_1+a_ix_2+\ldots+a_{in}x_n,\ (i=1,2,\ldots n).$$

Обозначим эту операцию буквой **A** - очевидно, она может быть записана как система n векторов

$$a_i(a_1, a_2, \ldots, a_n),$$

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 **A** преобразует различные векторы в различные, а это соответствует тому, что определитель матрицы **A** отличен от нуля, то операция и обозначающая ее матрица 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 (α_1, α_2, …, α_n) элементов 1, 2, …, n и, следовательно, содержит $n!$ слагаемых, причем χ=0, если перестановка четная, и χ=l, 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:

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

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 вектора вдоль координатных осей

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

то оно выражается ***диагональной матрицей***

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

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

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

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

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 $x\neq 0$), то все ее собственные значения положительны.

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

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 1x*n* называется ***вектор-строкой***, а матрица типа *m*x1 - ***вектор-столбцом***.

Число (скаляр) можно рассматривать как матрицу типа 1x1.

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 действиям 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.

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.

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, слишком неэффективен – из-за слишком большого количества арифметических операций даже для небольших матриц (с порядком в несколько десятков) 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.

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 из школьного курса 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} ($j=k, k+1, \ldots, n$),

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 $\mathbf{A}\mathbf{A}^{-1}=\mathbf{E}$ вытекает, 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 nxn. Прямой ход Гаусса при решении полученных систем можно осуществлять для всех систем одновременно.

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

Алгоритм Гауссовых исключений можно использовать для факторизации матриц. Если сконструировать нижнюю диагональную 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 **A**=**LU**, что и является треугольной факторизацией матрицы **A**. Если теперь, используя прямую подстановку, решить треугольную систему **L*y***=***b*** относительно ***y***, то получим вектор правой части для системы **U*x***=***y*** и решение исходной системы **A*x***=***b*** можно выполнить в три этапа: факторизации **A**=**LU**, решения **L*y***=***b*** и решения **U*x***=***y***. Этот подход используется в некоторых вычислительных вариантах процесса исключения.

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 **L** – нижняя треугольная матрица, у которой на главной диагонали не обязательно стоят единицы, как было в **LU**-разложении. При положительности диагональных элементов **L** разложение будет единственным. Правило получения элементов матрицы **L** через элементы **A** получим, приравнивая элементы в левой и правой частях **A**=**LL**$^{\mathrm{T}}$. 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 **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 **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-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-p), $l_{ij}=(a_{ij}-\sum_{k=r}^{j-1} l_{ik}l_{jk})/l_{jj}$.

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.

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*.

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

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

Максимальная из сумм модулей элементов столбцов

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

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

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

Методы вычисления собственных значений матриц

Проблема вычисления 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 определенность 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 собственных значений, поэтому мы ограничимся кратким обзором наиболее характерных подходов к решению этой задачи.

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 соответ-

ствующую 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. Недостатки метода очевидны – применимость только в случае вещественных собственных значений, трудность с предварительным определением области собственных значений для задания близкой к этой области последовательности λ_i, необходимость многократного вычисления значений определителя и т.д.

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 содержит значения коэффициентов характеристического полинома:

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

характеристический полином которой имеет вид:

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

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

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

где i=1, 2, …, 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:

$\mathbf{M}_1=$

$$\begin{vmatrix} 1 & 0 & \ldots & 0 & 0 & 0 \\ 0 & 1 & \ldots & 0 & 0 & 0 \\ \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\ \dfrac{-\mathbf{A}[n,1]}{\mathbf{A}[n,n-1]} & \dfrac{-\mathbf{A}[n,2]}{\mathbf{A}[n,n-1]} & \ldots & \dfrac{-\mathbf{A}[n,n-2]}{\mathbf{A}[n,n-1]} & \dfrac{1}{\mathbf{A}[n,n-1]} & \dfrac{-\mathbf{A}[n,n]}{\mathbf{A}[n,n-1]} \\ 0 & 0 & \ldots & 0 & 0 & 1 \end{vmatrix}$$

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

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

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

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

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

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 $\mathbf{A}$: $\mathbf{A}_k=\mathbf{Q}_k\mathbf{R}_k$, $\mathbf{A}_{k+1}=\mathbf{R}_k\mathbf{Q}_k$, k=0, 1, 2, …, где каждая из матриц $\mathbf{Q}_k$ является ортогональной (если мтрица $\mathbf{A}$ вещественна) или унитарной (если А невещественна). $\mathbf{R}_k$ - верхние треугольные матрицы.

Примечание: унитарной называется матрица $\mathbf{U}$, удовлетворяющая соотношению $\mathbf{U}^*\mathbf{U}=\mathbf{E}$, где $\mathbf{U}^*$ – матрица, сопряженная к $\mathbf{U}$, т.е. получающаяся из $\mathbf{U}$ заменой элементов на комплексно-сопряженные с последующим транспонированием, $\mathbf{E}$ - 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 $\mathbf{A}_2=(\mathbf{E}\text{-}2\boldsymbol{w}_1\boldsymbol{w}_1^T)\mathbf{A}_0=\begin{vmatrix} s & * & * & * \\ 0 & * & * & * \\ \dots & * & * & * \\ 0 & * & * & * \end{vmatrix}$, где * обозначает в общем случае ненулевой элемент. Выберем теперь вектор $\boldsymbol{w}_2$ так, что

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

$$s_2 = \begin{matrix} + \\ - \end{matrix} \sqrt{\sum_{j=2}^{n} \mathbf{A}_2[j,2]},$$

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

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

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

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 $\mathbf{Q}^{-1}=\mathbf{Q}^{\mathrm{T}}$, то можем записать $\mathbf{A}=\mathbf{QR}$, 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.

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

……..

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

$$\boldsymbol{x}_0=\boldsymbol{e};\ x_i^k=\lambda_k x_{i-1}^k+b_i^k\ \ i=1, 2, \ldots, n\text{-}1$$

где e – столбец единичной матрицы, b_i^k – одноименный столбец матрицы $\mathbf{B}_k$, а собственный вектор x_{n-1}^k соответствует λ_k.

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, $c\neq 0$, 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 $\mathbf{v}_1$; их модули будут либо стремится к нулю (если $|\lambda_1|<1$), либо к бесконечности ($|\lambda_1|>1$). Если текущие значе-

ния 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 прцедуре 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.

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:

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

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 $(\lambda-\lambda_i)$.

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 значений некоторых функций от экспериментальных данных) составляющих вектора входа, $\boldsymbol{y}$ – соответствующие строкам $\mathbf{X}$ 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 кроэффициентов $\mathbf{X}^T\mathbf{X}$ (носит название матрицы Грамма), столбцом свободных членов $\mathbf{X}^T\boldsymbol{y}$ 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 $\boldsymbol{a}$:

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

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

$$N>(\text{rank}(\mathbf{X})=m)$$ (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 $\mathbf{X}^T\mathbf{X}$ являются скалярные произведения 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 задан-

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

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, до-

страиваем 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 практически 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.

Программная реализация матричного класса

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 под-

ставляемыми могут быть типы, для которых определены стандартные арифметические 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:

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

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 *m*x*n*. 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 параметр. 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 a и e одинаковые, но в разных областях памяти
matrix &c=a;//матрицы c и a 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 <YourOwn-
FloatType> &, matrix<YourOwnFloatType> &);
```

Дружественная функция сложения матриц принимает два параметра - матрицы одинакового типа (одинаковой размерности), а матрицу-сумму возвращаset 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:

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

```
friend matrix<YourOwnFloatType> operator-(matrix <YourOwn-
FloatType> &, 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:

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

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

```
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 $\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \ldots & \ldots & \ldots & \ldots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix}$ и $\mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1q} \\ b_{21} & b_{22} & \ldots & b_{2q} \\ \ldots & \ldots & \ldots & \ldots \\ b_{p1} & b_{p2} & \ldots & b_{pq} \end{pmatrix}$ - матрицы типов соответственно *m*x*n* и *p*x*q*. Если число столбцов мат-

рицы **A** равно числу строк матрицы **B**, т. е. $n=p$, то для этих матриц определена матрица **C** типа mxq, 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 $c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{in}b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$ (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*(YourOwnFloatType,
matrix<YourOwnFloatType> &);
friend matrix<YourOwnFloatType> operator*(matrix <YourOwn-
FloatType> &, 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 &,matrix<YourOwnFloatType>
&);
friend istream &operator>>(istream &,matrix<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 <YourOwn-
FloatType> &, 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:

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

Из элементов этой системы составим три матрицы:

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

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

$$\mathbf{AX=B}.$$

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:

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

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 некоторый 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 <YourOwn-
FloatType> &, 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 (***) к системе с треугольной матрицей, из которой затем последовательно (обратным ходом) получаются значения всех неизвестных.

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

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):

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

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 процесс, вместо системы (***) получим равносильную ей систему с треугольной матрицей коэффициентов при неизвестных:

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

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

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, невырожденна, 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, поскольку 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 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, нужных 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 (***) 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 (***) нужно получить произведение ведущих элементов, используемых на каждом шаге метода Гаусса.

```
matrix<YourOwnFloatType> operator=(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+=(matrix<YourOwnFloatType>
&x);
matrix<YourOwnFloatType> operator-=(matrix<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 *m*x*n* 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 *n*x*m*.

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 **A**. Матрица **A** называется симметричной, если она совпадает со своей транспонированной, т. е. если $\mathbf{A}^T$=**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 $\mathbf{A}^T$ - nxm, размерность матрицы **X** - nx1, размерность матрицы **B** - mx1. Размерность произведения $\mathbf{A}^T\mathbf{A}$, согласно правилу умножения матриц, будет nxn, размерность произведения $\mathbf{A}^T\mathbf{B}$ - 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:

XA=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:

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

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 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 вектор-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.

```
};
```

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 формате 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*( YourOwn-
FloatType, 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 заданных строки и столбца
  matrix<YourOwnFloatType> operator=(matrix <YourOwn-
FloatType> &);//присвоение
  matrix<YourOwnFloatType> operator*=(matrix
<YourOwnFloatType> &x);//набор сокращённых операций умноже-
ния,
  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 <YourOwn-
FloatType> &,long);
//степень матрицы как дружественная функция
  matrix<YourOwnFloatType> operator~();
//транспонирование
  matrix<YourOwnFloatType> operator!(); /*аналитическое
обращение матрицы*/
  matrix<YourOwnFloatType> operator*(); /*численное об-
ращение матрицы*/
  YourOwnFloatType operator&();/*численный детерминант
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 равенство
```

```
  friend long operator!=(matrix<YourOwnFloatType> &,
matrix<YourOwnFloatType> &);//проверка на неравенство
  vector<YourOwnFloatType>    &operator[](long    a);
//индексирование матрицы
  long getm() { return m; }//число строк
  long getn() { return n; }//число столбцов
};
#endif
```

Файл реализации матричного класса matrix.cpp

```
#include "matrix.h"
```

/*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 арифметические 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 данных. 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 матрице, а второе - размерность каждого из этих векторов. В файловых операциях с векторами нам поможет определённая в векторном классе операция ввода вектора 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 расширяется до нужной размерности*/
    mtr[i]=vector<YourOwnFloatType>(n);
  for(i=0;i<m&&fp>>mtr[i];i++);//при вводе используем
  //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 соответствующими 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 выде-
ления 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 достаточно просто - созданием матрицы из одного единственного 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 выде-
ления 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 векторов
}
```

/* Как и векторы, матрицы тоже бывает необходимым проиндексировать; результатом этого действия, естественно, будет соответствуюший 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 матриц, к тому же эта операция не является коммутативной!
*/

```
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 является единичной, положительная степень определяется через произведение, отрицательная - через обращение матрицы и произведение. Последний случай можно красиво реализовать с использованием рекурсии, как, впрочем, и предыдущий.
*/

```
template <class YourOwnFloatType> matrix <YourOwn-
FloatType> matrix<YourOwnFloatType>:: operator^(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>:: operator^=(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 имеющуюся операцию умножения матрицы на число для сохранения коммутативности
*/

```
template <class YourOwnFloatType> inline matrix
<YourOwnFloatType> operator*(YourOwnFloatType f, ma-
trix<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 совпадают. В противном случае приходится уничтожать текущую матрицу, заново её создавать с новыми размерами и лишь тогда производить повекторное копирование
*/

```
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;//устанавливваем 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>:: opera-
tor+=(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 сделанное сокращённое сложение*/
}


/*
Очень часто бывает нужна унарная операция транспонирования:
*/
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, одинаковой ли они размерности, а потом в случае необходимости сравниваем векторы:
*/
template <class YourOwnFloatType> long operator==
(matrix<YourOwnFloatType> &f,matrix<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 opera-
tor!=(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
&operator<<(ostream &os,matrix<YourOwnFloatType> &x)
{
  for(long i=0;i<x.m;i++)//построчный вывод векторов матрицы
    os<<x[i]<<"\n";
  return os;
}
```

/*
Ввод матрицы из потока целиком перекладываем на векторный класс, используя его метод для ввода
*/

```
template <class YourOwnFloatType> istream
&operator>>(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 обратными 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 решения. Для небольших систем (<200) можно использовать прямые методы (например, метод Гаусса); для реальных расчётов мы рекомендуем метод ортогонализации, в основе которого лежат ортогональные преобразования 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 минора 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 поряд-
ка
    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 квадратной 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 невозможно\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 невозможно\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 преобразования матрицы в
число
*/
template <class YourOwnFloatType> inline ma-
trix<YourOwnFloatType>::operator YourOwnFloatType()
{
  return det(*this);//используем быстрый детерминант
}
```

Программная реализация методов вычисления собственных значений и собственных векторов матриц

Метод неопределенных коэффициентов

Под стандартной матричной проблемой собственных значений мы понимаем задачу отыскания комплексных чисел $\lambda_1, \lambda_2, \ldots, \lambda_n$ и соответствующих ненулевых векторов $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ удовлетворяющих уравнению $\mathbf{A}\boldsymbol{X}=\lambda\boldsymbol{X}$, где **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

$$D(\lambda)=\lambda^n+p_1\lambda^{n-1}+\ldots+p_n \quad (******)$$

есть вековой определитель матрицы **A**, т.е. $\det(\mathbf{A}-\lambda\mathbf{E})=0$. Если в 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 заменой 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 произвольной матрицы по методу неопределенных коэффициентов */
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 коэффициентом. Для того, чтобы найти эти
коэффициенты, используем следующий способ: характеристический по-
лином прямыми методами получается развёрткой детерминанта, и значе-
ние 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 уравне-
ний, линейных относительно коэффициентов характеристического поли-
нома. Результатом решения этой системы и будут искомые коэффиценты,
зная которые, мы можем найти корни полинома, и являющиеся искомыми
собственными значениями */
  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
      a[i][j]=pow(i,a.getm()-j-1);//коэффициентах
    cmatrix temp=x;
    for(long j=0;j<temp.getm();j++)
      temp[j][j]-=i;//вычитаем из элементов главной диагонали
    b[i][0]=det(temp);//текущую точку и находим детерминант
  }
  cmatrix result=SLAE_Orto(a,b);//решаем СЛАУ
  cpolynom lambdaeq=result.getm();
  for(long i=0;i<result.getm();i++)/*формируем характери-
стический*/
    lambdaeq[i]=result[result.getm()-i-1][0];/*полином
и находим*/
  return newton(lambdaeq);/*его корни модифицированным ме-
тодом Ньютона*/
}


/*Подпрограмма вычисления собственных векторов при известных мат-
рице 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 размерностей
    throw xmsg("Собственные значения не соответствуют
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 значени-
ям
  {
    cmatrix temp=x;
    for(long j=0;j<temp.getm();j++)/*вычитаем единичную
матрицу, умноженную */
      temp[j][j]-=e[i];  //на собственное значение
/*Для нахождения ненулевого решения однородной системы уравнений
проделаем следующее: примем одну из компонент вектора равной 1. За-
тем перенесём в системе уравнений в правую часть с противоположным
знаком столбец, соответствующий этой компоненте. Решив такую СЛАУ,
мы найдём все остальные компоненты вектора. Для того, чтобы вектора
не совпадали, для каждого из них "объединичим" свою компоненту - сна-
чала 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
          b[k][0]=-temp[k][j];//значения, сделаем равным 1,
//а соответствующий столбец
        else //переносим в правую часть; все остальное
          a[k][l++]=temp[k][j];//остаётся в левой части
    cmatrix res=SLAE_Orto(a,b);//решаем СЛАУ относительно
    //неизвестных составляющих
    for(long j=0,l=0;j<ev.getm();j++)/*компонуем соб-
ственный 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
}
```

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 нехватке памяти
  catch(xalloc)
  {
```
/*не пытаясь исправить эту ситуацию, выбрасываем, в свою очередь, стандартное исключение с диагностическим 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, заданным 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 соответствующие 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 случайных чисел
/*процесс построения повторяем до тех пор, пока не выполнится условие
линейной независимости векторов - неравенство нулю детерминанта мат-
рицы, составленной из построенных векторов*/
  }while(det(test)==(complex)0);
  //разлагаем последний вектор по предыдущим (в силу их линейной
  //независимости) как по базису и находим 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) модифицированным
методом Ньютона*/
  return newton(p);
}
```

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 матрицы,
  //однако соответствующая проверка не выполняется
  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 полинома
  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();
  }
}
```

Элементы линейного программирования

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 независимых 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

$$\min L=c_1x_1+c_2x_2+...+c_nx_n$$

Примеры задач линейного программирования

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. 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 1-й 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, …, 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:

$$a_{11}x_1+a_{12}x_2+...+a_{1n}x_n=b_1$$
$$a_{21}x_1+a_{22}x_2+...+a_{2n}x_n=b_2$$
$$\text{……..}$$
$$a_{m1}x_1+a_{m2}x_2+...+a_{mn}x_n=b_m$$

Себестоимость единицы товара T_j равна $S_j=a_{1j}D_1+a_{2j}D_2+...+a_{mj}D_m$.

Прибыль Q_j от реализации единицы товара $T_jQ_j=C_j-S_j$.

Общая прибыль $L=Q_1x_1+Q_2x_2+...+Q_nx_n$.

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.

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 пункта-получателя (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:

$$\sum_{j=1}^{p} x_{ij} \le k_j,\ j=1,\ \ldots,\ p.$$

и линейный критерий оптимальности решения - минимум общей стоимости перевозок:

$$L = \sum_{i=1}^{n} \left(\sum_{j=1}^{p} c_{ij} \right) x_{ij} \ \rightarrow \min$$

Эта задача носит название *транспортной задачи линейного программирования* - она значительно проще общей задачи, так как все коэффициенты основной системы уравнений равны 1.

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.

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 *n*x*n* 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.

Алгоритм замены базисных переменных

Процедура переразрешения системы ОЗЛП относительно новых базисных переменных может быть формализована и сведена к алгоритму (последовательности допустимых 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;

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.

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.

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.

Транспортная задача линейного программирования

Общие сведения

Классическая формулировка транспортной задачи линейного программирования выглядит так:

- имеется m пунктов отправления A_1, A_2, …, A_m, в которых есть запасы однородного товара в количествах соответственно a_1, a_2, …, a_m единиц;
- имеется также n пунктов назначения B_1, B_2, …, B_n, желающих получить соответственно b_1, b_2, …, 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 симплекс-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 записывются 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.

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.

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.

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 $\tilde{c}_{ij} = \alpha_i + \beta_j$; $i = 1, 2, \ldots, m$; $j = 1, 2, \ldots, n$. 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
>
> $$\sum_{i=1}^{n}\sum_{j=1}^{m}\tilde{c}_{ij}x_{ij} = \text{const}$$

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: $\tilde{c}_{ij} = \alpha_i + \beta_j = c_{ij}$ при $x_{ij} > 0$. Оказывается, что соотношение между псевдостоимостями и стоимостями в свобод-

ных 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 ($x_{ij}=0$) $\widetilde{c}_{ij}=\alpha_i+\beta_j\le c_{ij}$, 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 $\widetilde{c}_{ij}=c_{ij}$ для всех базисных клеток, для каждой свободной клетки цена цикла переноса равна разности между стоимостью и псевдостоимостью в данной клетке $c_{ij}-\widetilde{c}_{ij}$.

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 $\widetilde{c}_{ij}=\alpha_i+\beta_j$;
- 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.

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.

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 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 использования запрещен-

ных 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.

Программная реализация задач линейного программирования

Симплекс-метод решения ОЗЛП

```
//файл включения для векторных и матричных объектов
#include "matrix.h"

//определяем типы "действительная матрица" и "действительный вектор"
typedef matrix<double> dmatrix;
typedef vector<double> dvector;


//процедура перестановки векторов матрицы для выбора 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 соотвествующей компонентой
 //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 диагональным 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 - перестановки местами переменных с попутным переразрешением полученной системы. Параметрами её являются матрица ОЗЛП, вектор линейной формы, номера обмениваемых переменных и вектор, в котором учитывается производимая 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, соответствующий 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 находятся 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 свободные члены неотрицательны */
  /*в строке, где мы нашли отрицательный свободный член, среди коэффициентов пытаемся найти положительный*/
```

```
  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 переменных, что недопустимо*/
```
  if(j==x.getn()-1)
    throw xmsg("Допустимых решений нет");
```
/*среди строк, в которых в определённом (разрешающем) столбце отрицательному свободному члену соответствует 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 строка.
  //Меняем в нашей матрице столбцы с соответствующими 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 переменных */

```
  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, оптимального решения ОЗЛП не существует*/

```
  if(j==x.getm())
    throw xmsg("Линейная форма не ограничена снизу");
```

/*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 коэффициенте 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 матрицей (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 совпадать");
  if(a.getn()!=c.getm())
    throw xmsg("Количество неизвестных должно совпадать");
/*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 основной 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 линейно-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 линейную 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;
}
```

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 "действительный 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 заданном 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 координаты
  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 макро. Например:
- в этом фрагменте проверяется, превышает ли число неудачных (тупико-
вых) попыток продвинуться в одном из четырёх возможных направлений
количество этих направлений. Если это так, то пункт, из которого мы пы-
таемся пройти дальше, объявляется тупиковым (помечается нулём), затем
изымается из массива  пунктов цикла - пути перевозки груза, общее коли-
чество пунктов перевозки уменьшается на единицу и количество неудач-
ных 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 определен следующий
пункт, куда можно перевезти груз. При этом вначале количество неудач-
ных попыток увеличивается на единицу - а вдруг идти в этот пункт нель-
```

зя? Затем выполняется проверка координат пункта на допустимость и наличие пункта в цикле (не провозили ли мы уже здесь груз в текущем цикле перевозок). Если всё Ок, пункт добавляется в цикл перевозок, количество неудачных попыток устанавливается в ноль, а количество пунктов перевозки в цикле увеличивается. 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 нечего
    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 пунктов, 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, вычёркиваем 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 угла", 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 заявки
    }
    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 отрицательных 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 увеличиваем, в отрица-
тельных – 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) перевозки; возвра-
щаемое значение - матрица, первые два столбца в которой указывают но-
мер 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 положение дел,*/
        {
          movegruz(x,way);//перевезя по нему груз
          j=x.getn();/*перезапускаем после перевозки цикл про-
смотра 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 которую заносим
  //направление перевозки и количество груза
  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 принимает 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;
}
```